NZTM: Nonblocking Zero-indirection Transactional Memory

Fuad Tabba
The University of Auckland
fuad@cs.auckland.ac.nz

Mark Moir
Sun Microsystems
mark.moir@sun.com

James R. Goodman
The University of Auckland
goodman@cs.auckland.ac.nz

Andrew W. Hay
The University of Auckland
andrewh@cs.auckland.ac.nz

Cong Wang
The University of Wisconsin
wang@cs.wisc.edu

ABSTRACT

This paper introduces NZTM, a nonblocking, zero-indirection, object-based, hybrid transactional memory system. NZTM comprises a nonblocking software transactional memory (STM) system that can exploit best-effort hardware transactional memory (HTM) if available to improve performance.

Most previous nonblocking software transactional memory implementations pay a significant performance cost in the common case, as compared to simpler, blocking ones. However, blocking is problematic in some cases and unacceptable in others. NZTM is nonblocking, but shares the advantages of recent blocking STM proposals in the common case: it stores object data "in place", avoiding the costly levels of indirection of previous nonblocking STMs, and improves cache performance by collocating object metadata with the data it controls.

We also explain how our nonblocking NZSTM algorithm can be substantially simplified using very simple hardware transactions, and evaluate its performance on Sun's forthcoming Rock processor. Our results show that nonblocking support introduces little overhead when compared with blocking STMs, and that NZTM is competitive with LogTM-SE, an unbounded HTM.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming

General Terms

Algorithms, Design, Performance

Keywords

Transactional Memory, Nonblocking Synchronization, Hardware

1. INTRODUCTION

Transactional Memory (TM) [20] is widely considered to be a promising model for concurrent programming, an increasingly important area as multiprocessing becomes ubiquitous [13, 14, 16, 22, 41]. TM promises to ease the burden of concurrent programming: With TM, programmers specify *what* should be executed atomically, leaving the system to determine *how* this is achieved.

Numerous research groups are investigating techniques of system support for transactional memory. Despite significant progress, Software Transactional Memory (STM) [39] implementations are slower than what Hardware Transactional Memory (HTM) can be expected to achieve [4, 23]. However, because most HTM proposals are complicated and leave tricky issues unresolved, it will be difficult to include them in commercial systems in the near future.

To bridge the gap, Damron et al. [7] propose Hybrid Transactional Memory (HyTM), which attempts transactions using HTM support, and can execute entirely in software in case transactions fail. Because HyTM can execute any transaction in software, it can exploit *best-effort* HTM support that may not guarantee all transactions to commit. Best-effort HTM can be substantially simpler than *unbounded* HTM [1, 15, 34, 35, 44], which supports all transactions in hardware. With the HyTM approach, we can develop and test transactional programs in *existing* systems today, and exploit best-effort HTM support as it becomes available, such as Sun's forthcoming Rock processor [5, 8, 33]. As HTM support improves, applications developed using HyTM automatically benefit from the improvements. Thus HyTM encourages an incremental approach to the adoption of HTM.

In this work, we focus on *object-based* TM, where data objects have headers that can be easily located whenever the application accesses an object. Herlihy et al. [19] introduced DSTM, the first object-based dynamic software transactional memory system. DSTM, a nonblocking algorithm, requires two levels of indirection to access the data; each level of indirection is a potential cache miss, which can have a negative impact on performance. RSTM [28] and OSTM [12] reduce the indirection to one level in the common case, while remaining nonblocking.

Other research groups [10, 11, 37] have proposed STM implementations that store object data "in place", avoiding cache misses caused by indirection to reach the data. The performance experiments presented by these groups confirm the intuition that this approach to structuring object data results in significantly better performance than those involving at least one level of indirection in all cases. However, all of these implementations sacrifice the nonblocking progress properties provided by the earlier ones, and most have implied or argued directly that this is fundamentally necessary in order to store object data in place and collocate metadata with object data.

Proponents of blocking STMs argue [10, 11] that it is sometimes possible to avoid the disadvantages of blocking algorithms. For example, the Solaris™ `schedctl` function can *discourage* (not prevent) the scheduler from preempting a thread during the blocking

SPAA'09, August 11–13, 2009, Calgary, Alberta, Canada.

Copyright 2009 Sun Microsystems, Inc. and ACM

978-1-60558-606-9/09/08 ...$10.00.

part of a transaction. Nonetheless, without an implementation that is truly nonblocking, we can still experience the disadvantages of blocking implementations. For example, if a transaction experiences a long delay due to a page fault or being preempted, this can require many other transactions to also wait for a long time.

Blocking is more than "merely" a performance concern. For example, it is *unacceptable* for an interrupt handler to be blocked by the thread it has interrupted [36]; the design of interrupt handlers is often significantly complicated by this restriction. TM can help, but *only* if it is nonblocking. It is therefore important to continue research on nonblocking TM, despite the appeal of simpler blocking implementations.

We present NZTM, and show that it is *not* necessary to sacrifice nonblocking progress guarantees in order to store data in place in the common case. Specifically, we present Nonblocking Zero-indirection Software Transactional Memory (NZSTM), which stores object data in place in the common case, resorting to indirection only when a thread encounters a conflict with an unresponsive thread. Blocking STMs *must* block in such cases, so claims that excessive overhead is introduced by using indirection to avoid blocking are unconvincing.

We have designed a HyTM system, NZTM, in which transactions can run using best-effort HTM support, and fall back onto NZSTM if this is not successful. NZTM is optimized for the case in which HTM support is available and able to commit most transactions. An object's metadata is collocated with the object in the common case, and transactions executed using HTM do not need to copy the data they modify. Thus our design achieves near-optimal cache behavior in the common case.

Some of the initial ideas for NZTM were presented earlier at the TRANSACT'07 workshop [43]. This work expands on these ideas, and presents experimental data gathered using a variety of benchmarks. Our main contributions in this paper are:

- We describe NZSTM, the first nonblocking object-based STM that does not require indirection to access data in the common case, and does not rely on special hardware support. We also show how it can be substantially simplified using simple hardware transactions, if available, and demonstrate the effectiveness of this technique on a Rock system.
- We describe NZTM, a HyTM based on NZSTM. NZSTM is particularly suited for HyTM as it requires no indirection to access data in the common case.
- We present performance evaluation for NZTM with best-effort HTM using Sun's ATMTP simulator [33]. We compare NZTM against LogTM-SE [44], an unbounded hardware transactional memory. Our results show that NZTM is competitive with LogTM-SE in many benchmarks.
- We present performance evaluation for NZSTM using a variety of benchmarks, and compare it with the DSTM2 Shadow Factory (DSTM2-SF) [18], a blocking object-based STM that never requires indirection to access data. Our results show that NZSTM's performance closely tracks DSTM2-SF's.

The remainder of this paper is organized as follows. Section 2 describes the design of NZSTM and NZTM. Section 3 discusses our use of a model checker to increase our confidence in the correctness of NZSTM. Section 4 presents performance results. Section 5 concludes this paper.

2. NZSTM AND NZTM

NZSTM and NZTM are written in C and use a programming model derived from the Java™-based DSTM [19]. The DSTM programming model was designed to facilitate experimentation, and is not intended for production use; rather it is envisaged that future language support might target such an interface, with programmers writing to a higher-level programming interface [6]. For now, the DSTM interface, and the variant we use in C, is sufficient for experimenting with alternative implementations.

Our main goal in the design of NZSTM and NZTM is a nonblocking object-based hybrid TM system that provides performance competitive with similar blocking designs. Our approach is to store data "in place" in the common case in order to eliminate the costly indirection used in previous nonblocking STMs.

A key difficulty in designing a nonblocking STM that stores object data in place is the uncertainty that arises when one transaction T_1 is updating an object, and another transaction T_2 wishes to access the same object. T_2 cannot simply wait for T_1 to complete as this would be blocking. T_2 can attempt to inform T_1 that it should stop modifying the object, but until T_2 can determine that T_1 has become aware that it should stop, it is not safe for T_2 or other transactions to update the object data in place, because T_1 may still overwrite the data. Therefore, it is hard to see an alternative to storing the correct data somewhere other than its natural home in this case. This leads to indirection and associated overhead during the period that T_1 is unresponsive.

NZSTM differs from previous nonblocking STMs in that, rather than actively aborting a conflicting transaction, we can "request" that the transaction abort itself, and wait a short time until it does. If the transaction does abort, then the uncertainty is resolved and we can continue to access the data in place. Thus, we can generally avoid the overhead of introducing indirection except when the conflicting transaction is unresponsive. This approach largely eliminates the performance gap between previous blocking and nonblocking object-based transactional memory implementations by eliminating their levels of indirection.

This section describes the NZSTM and NZTM algorithms. We begin by presenting a *blocking* object-based STM that stores object data in place and collocates metadata with objects. We then extend this STM to the nonblocking NZSTM. To simplify the discussion, we present an algorithm that acquires objects exclusively and does not support read sharing. However, the algorithm we describe can handle read sharing with little modification [42], for both visible and invisible readers [19, 24, 28].

2.1 NZSTM Data Structures

The basic data structure used in NZSTM is shown in Figure 1.

`NZObject` encapsulates a program object that can be accessed by NZSTM transactions, and serves as a container for its data and metadata. It is analogous to other containers such as DSTM's `TMObject` and RSTM's `Object Header` [19, 28]. The `NZObject` structure contains the following fields: The `Owner` field, which if non-`NULL`, points to the last transaction to acquire this object. The `Clone` field points to the function used for creating a copy of the object. The `Backup Data` field points to a backup copy of the object while a transaction that modifies the object is in progress. Finally, the `Data` field contains the actual object data. Because the object data is stored at a fixed offset from the start of the object, no indirection is required to access it.

NZSTM's `Transaction` is similar to the transaction descriptors used by DSTM and RSTM. NZSTM's `Transaction`, however, adds an `AbortNowPlease` flag, which when set is interpreted as a request for a transaction to abort itself. This flag is stored together with the `Status` field, so both may be accessed atomically using a *Compare&Swap* (CAS) instruction.

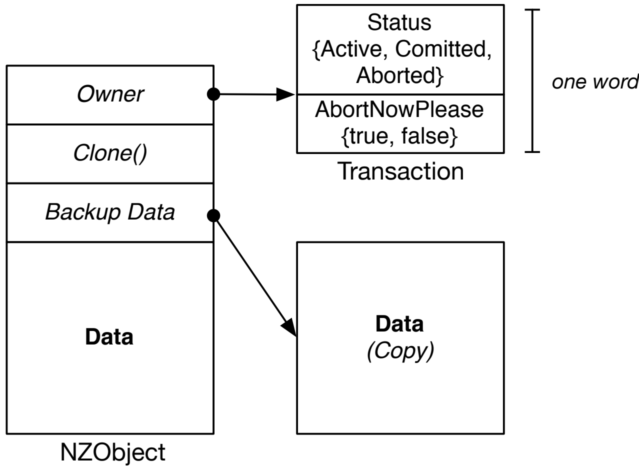

Figure 1: The structure of an `NZObject`. The `Data` field is the actual data and can have any size or structure.

At initialization, the `NZObject Data` field contains the initial value of the object data, `Clone` points to a function that creates a copy of the data, and the remaining fields are `NULL`.

2.2 A Blocking STM Algorithm

In our blocking STM, a thread begins a transaction by creating a new `Transaction` object with its status set to `Active` and `AbortNowPlease` not set. The thread then executes its transaction, *acquiring* each object it accesses. When it completes the execution of a transaction, it attempts to atomically change its transaction's status from `Active` to `Committed` while ensuring that `AbortNowPlease` is not set. During execution, another transaction that detects a conflict with this one may wait or to attempt to abort it; this decision is made by a *contention manager* [19].

Unlike DSTM and other STMs, a transaction T_1 does *not* explicitly abort a conflicting transaction T_2. Instead T_1 *requests* that T_2 abort itself; this request is made by atomically setting T_2's `AbortNowPlease`, and confirming afterwards that T_1 itself has not been asked to abort (i.e., T_1's `AbortNowPlease` is not set). When a transaction observes that its own `AbortNowPlease` flag is set, it must abort and set its own `Status` field to `Aborted` as an acknowledgement. The requesting transaction waits for this acknowledgement before proceeding to acquire the object on which the conflict occurred; because of this waiting this STM is blocking.

Normally, the `Data` field of an `NZObject` contains the object's current data. A transaction that wishes to access the object acquires ownership by atomically placing a pointer to its `Transaction` in the object's `Owner` field. Before modifying the `Data` field, the acquiring transaction creates a copy of the data (using the object's `Clone` function), and points the object's `Backup Data` field to that copy. If the transaction aborts, the backup copy can be lazily restored, undoing the transaction's effects.

The backup copy is not stored in place with the `NZObject`, as in DSTM2-SF [18], which incurs 100% space overhead as a result. Instead, the memory for the backup data is allocated from a thread-local memory pool. Unless a transaction aborts, this backup data is not accessed by other threads. Thus good cache locality can be achieved by reusing thread local data for these backups.

To acquire an object, a transaction T first determines if it has already acquired the object by examining it `Owner` field. If not, T must ensure that there are no conflicts with other transactions before acquiring ownership of the object. If the `Owner` field is `NULL`, or points to a committed or an aborted transaction, there is no conflict, and T can atomically change the `Owner` field to point to its `Transaction`. If the `Owner` field points to an active transaction, then as in DSTM and RSTM, a contention manager is consulted, and depending on the outcome, T either waits or requests that the active transaction abort itself as described above.

When all conflicts have been resolved and T has pointed the `Owner` field to its own `Transaction`, if the previous owner of the object was an aborted transaction, T restores the backup copy (indicated by the `Backup Data` field) if there is one. Otherwise, T creates a new backup copy as described earlier. T finally *validates* by checking that its own `AbortNowPlease` flag is not set — if it is then it must abort and set its own `Status` field to `Aborted` as an acknowledgement. If `AbortNowPlease` is not set, then T has successfully acquired the object and can now access the data.

Validating at other times, by checking that the `AbortNowPlease` flag is not set, is not strictly necessary. However, it may be desirable for performance reasons; e.g., validating before asking another transaction to abort, or before waiting for another transaction to abort, may avoid unnecessary aborts and waiting.

2.3 NZSTM: Making the STM Nonblocking

In this section, we describe two ways to make the STM described above nonblocking, thereby ensuring that a transaction can always make progress, even in the face of conflicts with unresponsive transactions.

2.3.1 *Inflating the object and displacing data*

NZSTM can "inflate" an object and use techniques like existing object-based STMs to temporarily displace the logical data into a different location than the `Data` field of the `NZObject`. NZSTM resorts to this approach only when an aborted transaction is unresponsive, which is relatively rare. In contrast, previous nonblocking object-based STMs [12, 19, 27, 28] involve at least one level of indirection in *all* cases.

Our design uses a DSTM-like approach when an object needs to be inflated that introduces two levels of indirection *only* when needed. The pointer to the first level is integrated into NZSTM by overloading the `Owner` field. We indicate that the `Owner` field should be treated as such by setting its low order bit, effectively changing the meaning of the `Owner` field to an `Inflated Object` field (Figure 2).

The cost of accessing an inflated object is considerably higher than accessing a non-inflated one. However, we expect (and our experiments confirm) that the need to inflate objects is rare. Moreover, NZSTM inflates objects only in cases in which blocking STMs would have no choice but to wait for an unresponsive transaction, which is likely to be considerably more expensive.

We now describe this approach in more detail. When a transaction T has requested the current owner of an object to abort, and the owner has not responded, T may decide not to wait any longer, and to inflate the object into a DSTM-like object. To do so, T creates a DSTM-like `Locator` [19], a data structure used to track the object's metadata; points the `Locator`'s `Owner` field to T's `Transaction`; points the locator's `old data` field to the backup copy created by the unresponsive transaction, or to a new copy of the data if there is no backup[1]; and points the locator's `new data` field to a copy of this backup created using the `Clone` function. NZSTM's `Locator` object also has an `Aborted Transaction`

[1]This can happen only if the unresponsive transaction became unresponsive in the process of acquiring the object, in which case it has not yet modified the object.

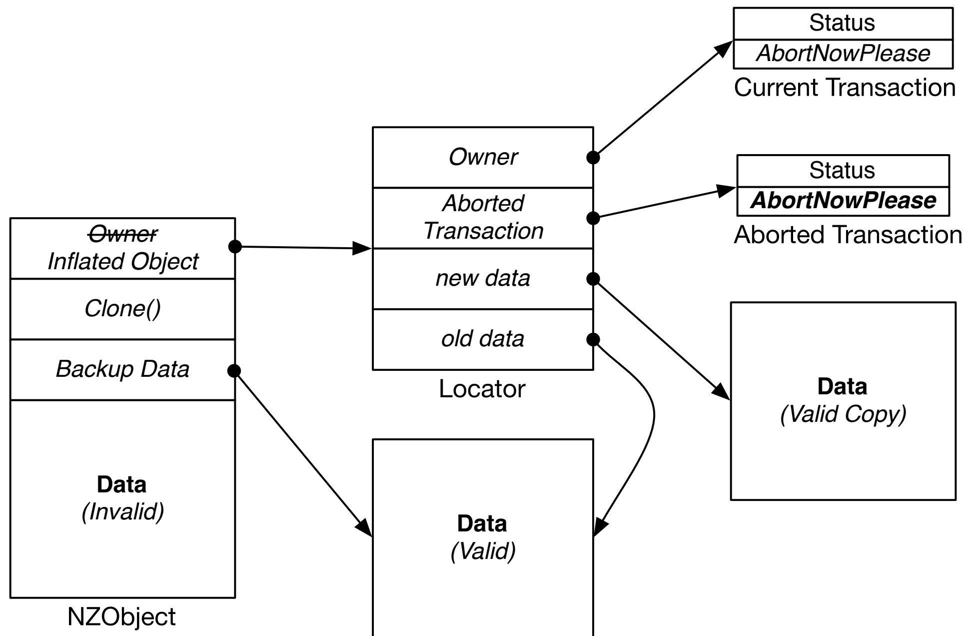

Figure 2: An `NZObject` immediately after being inflated. The `Owner`'s low order bit indicates how the object is interpreted.

field, which points to the unresponsive thread's `Transaction` (Figure 2).

After creating a new `Locator`, T checks that T itself is still `Active` without any pending abort requests, that the unresponsive transaction is still unresponsive, and that the object has not been acquired or inflated by another transaction. If any of these assumptions do not hold, T tries to acquire the object again. In the absence of interference from other threads, this is guaranteed to eventually succeed; hence it is obstruction free [17].

If these checks succeed, then T attempts to atomically swap the `Owner` field from pointing to the unresponsive transaction's `Transaction` object to the newly created locator, in the process setting the `Owner` field's least significant bit to indicate that the object now points to a DSTM-like `Locator` rather than an NZSTM `Transaction`. This results in a state like the one illustrated in Figure 2. Henceforth, the object is treated like a DSTM object and the nonblocking DSTM algorithm applies [19], with the addition that each new `Locator` introduced contains the `Aborted Transaction` field from the replaced `Locator`, thus preserving the identity of the unresponsive transaction.

Once the unresponsive transaction finally aborts itself, we know it is no longer modifying the `Data` field of the object, and it is desirable to restore the object to a normal `NZObject` so subsequent transactions can again enjoy the performance benefits of accessing the object data in place. This can be done by a transaction T as follows: T acquires the object exclusively according to the normal DSTM method [19]. T then verifies that the unresponsive transaction has indeed aborted and that T itself is still active, with no pending abort requests. If that is the case, T atomically swaps `NZObject`'s `Backup Data` field to point to the valid data. If successful, T swaps the `Transaction` field to point to itself and copies the backup data back to the `Data` field. This ensures that if T were to abort at any time the valid data is readily accessible. If any of the preceding steps fail, T tries again.

Once the object is deflated, T and any subsequent transactions can again access this object as a normal `NZObject`.

2.3.2 *Using an atomic Single-Compare Single-Store*

The need to wait for an unresponsive transaction in the blocking STM arises because it is not safe to restore the backup to the `Data` field and continue to access the object in place: the aborting transaction might write to the `Data` field. A transaction can avoid such "late writes" by atomically pairing each write with a check of the `AbortNowPlease` flag, using an operation we call *Single-Compare Single-Store* (SCSS) [32].

SCSS can be implemented using a short hardware transaction that modifies a location only if another location's value matches a specified expected value. A conservative SCSS implementation that can sometimes fail even if the expected value is in the second location is acceptable; however, if it could fail deterministically forever, a software alternative would be needed, which would reintroduce much of the complexity we aim to eliminate. Thus, this approach assumes an SCSS implementation that makes sufficient guarantees for SCSS operations that a software alternative is not necessary. We make no statement here as to whether Rock provides such a guarantee. In practice, however, SCSS experiments running on Rock (Section 4) all terminated, despite the lack of a software alternative for the hardware transactions used to implement SCSS, allowing us to evaluate the algorithm.

This simple operation eliminates more than a PODC paper worth of complexity from NZSTM: we no longer require code that essentially implements the DSTM algorithm [19] to access inflated objects, and we no longer need code for inflating and deflating objects to switch between the two styles.

Depending on the HTM implementation, using short hardware transactions in place of simple writes would likely result in at least a small slowdown in the common case. However, we believe the dramatic code simplification this technique offers will often justify a small performance overhead. We hope that this observation will encourage hardware designers to recognize the value of supporting even very small and simple low-latency hardware transactions.

2.4 NZTM: The Hybrid Approach

NZTM is a HyTM system that uses NZSTM for software transactions. Like the HyTM system presented by Damron et al. [7], NZTM attempts transactions using HTM and if (repeatedly) unsuccessful, transactions are run using NZSTM software transactions.

When a hardware transaction acquires an object, it checks for conflicts with software transactions, and explicitly aborts itself if any are discovered; it can then try again either in hardware or in software, depending on the policy. If no conflicts are discovered, the transaction can safely proceed because a subsequent conflict that arises with a software transaction will modify data that the hardware transaction has accessed, thereby aborting the hardware transaction.

A variety of approaches to checking for conflicts with software transactions are possible. More conservative approaches are simpler, but are more likely to revert to software, which is harmful to performance. In the simplest and most conservative scheme, a hardware transaction accessing an object aborts if the `Owner` field is non-`NULL`, and proceeds otherwise. This scheme is conservative because if the `Owner` field is not `NULL`, but points to an aborted or a committed transaction, there is no conflict.

Therefore, in our implementation, hardware transactions look in more detail to determine if there is a real conflict, by checking the status of the transaction identified by `Owner`. By checking the status of the `Owner` field, NZTM can access the data directly if the identified transaction is committed, but first restores the backup data if the previous software owner aborted. On the other hand, if the object accessed by a hardware transaction is inflated, i.e., the `Owner` field's low-order bit is set, NZTM first attempts to deflate an inflated object, and then accesses the data in place.

Once a hardware transaction has determined that the `Owner` field of an object no longer points to an active transaction, it sets the object's `Owner` field to `NULL`. This eliminates the need for subsequent hardware transactions to perform similar checks.

While it is possible for hardware transactions to access the backup data or even the inflated data directly, our implementation attempts to restore the object to its original "in place" state to make what we believe to be the common case fast.

It is important to note that these techniques are achieved by controlling what *code* is executed within a hardware transaction, *not* by assuming any special support in the hardware. Our NZTM implementation works with best-effort HTM, such as Sun's forthcoming Rock processor.

Our NZTM implementation also supports software read sharing, and checks for active software readers of an object. A description of our read sharing algorithm is available online [42].

3. CORRECTNESS EVALUATION

In developing NZSTM and NZTM, we have used model checking to increase our confidence in the correctness of our algorithm. We briefly discuss this aspect of our work in this section.

Inspired by the work of Ananian and Rinard [2], in the process of developing NZSTM and NZTM, we created a model of the algorithm in PROMELA and mechanically checked various useful properties of it using SPIN 4.3 [21]. This work was helpful in developing the algorithm, resolving some subtle bugs that did not manifest in normal testing, and increasing our confidence in the algorithms.

PROMELA is a C-like language used to describe models to be checked using SPIN, which is a mechanical verification system developed at Bell Labs and designed to check high-level models of concurrent systems. SPIN can perform exhaustive searches of all possible executions of a given model; applying sanity checks; and finding unreachable code, deadlocks, and cycles (livelocks).

Due to the complexity of TM algorithms and the amount of state they require, it is not generally feasible to establish their correctness using model checking. Instead, we use these tools to check the correctness of limited cases in order to increase our confidence in the algorithms. Using SPIN, we were able to perform complete state-space searches for up to three concurrent threads, each thread accessing up to three objects for either writing or reading using our read-sharing algorithm. Going up to four threads increases the search space so that complete searches were not possible due to memory limitations. Instead, verification using the non-exhaustive bitstate hashing [21] was performed for up to four threads.

SPIN's cycle detection, deadlock detection, and unreachable code reporting were used. For the models we tested, all code paths are taken at least once, no deadlocks occur, and no cycles (livelock) occur. We note that NZSTM is only obstruction-free [17] and not lock-free, and thus livelock can in fact occur in the algorithm; it is the role of the contention manager to ensure this does not happen in practice. The reason SPIN reports that the model is livelock-free is that a retried transaction allocates a new `Transaction` object, so even when the algorithm exhibits livelock, no state is repeated. For the interested reader, the PROMELA model source code for NZSTM is available online [42].

4. PERFORMANCE EVALUATION

This section reports on our performance evaluation of NZTM and NZSTM.

4.1 Experimental Environments

We use a simulation framework based on Virtutech Simics [25] in conjunction with customized memory models built on the University of Wisconsin GEMS [29]. The simulator models processors that have best-effort HTM support using Sun's ATMTP simulator [33], as well as unbounded HTM support, LogTM-SE [44].

The simulated system is a SPARC®/Solaris™ Sun Fire™ server; the simulated environment parameters, unless noted otherwise, are the ones used in LogTM-SE and ATMTP. This model assumes a traditional CMP, where each processor is single-threaded, with a shared L2 cache and a private L1 cache per core. The size of the L1 cache is set to 256 KB and the size of the ATMTP write buffer to 256 entries; each entry represents a single store and is typically one word. These parameters are larger than the ATMTP defaults because we are interested in evaluating the algorithm and understanding the support it needs to be successful, rather than evaluating a particular best-effort HTM.

We set the parameters for ATMTP to allow function calls inside transactions. Hardware transactions in ATMTP abort on events such as TLB misses, context switches, interrupts, and resource exhaustion. ATMTP transactions are limited by the size and associativity of the L1 cache for their read sets, and by the size of the store buffer for their write sets.

We also evaluated our algorithms on a pre-production revision of Sun's forthcoming Rock chip, which was configured with 16 cores in SSE mode, running at 1.5 GHz; see Dice et al. [9] for details, where the revision we used is referred to as "R1".

4.2 Benchmarks

We ran three microbenchmarks and three STAMP benchmarks [30] with varying workloads to compare the systems. These benchmarks have been used to evaluate many other TMs [3, 19, 28, 33].

`linkedlist` is a concurrent set implemented using a single sorted linked list. Each thread randomly chooses to insert, delete, or look up a value in the range of 0 to 255, with the low contention

distribution of operations being 1:1:8 (insert:delete:lookup) and the high contention distribution of operations being 1:1:1. `redblack` and `hashtable` are also concurrent sets, implemented using a red-black tree and a chained hash-table.

We also use the `kmeans`, `genome`, and `vacation` STAMP benchmarks, with the same parameters used by Minh et al. [31] for both *low* and *high* contention tests.

4.3 Experiments

On the simulator, each benchmark was tested using 1, 3, 7, 15 threads[2] each running on its own processor. We initialize the relevant data structures, and then begin taking measurements, recording the simulated machine's elapsed clock cycles to complete the benchmark. On the Rock machine, we ran 1, 2, 4, 8, 16 threads, each on a separate core.

We evaluate different mechanisms for executing transactions:

NZSTM The system assumes no special hardware support and runs using NZSTM software transactions with visible reads [42].

NZTM/ATMTP The system takes advantage of best-effort HTM support using ATMTP. When a hardware transaction aborts, NZTM retries the transaction in hardware a number of times proportional to the total number of running threads, only if the reason for aborting was due to a transactional (coherence) conflict as determined by the CPS register [33]. After all attempts are exhausted, or if the reason for aborting was something other than a coherence conflict, NZTM falls back onto NZSTM. We have not yet experimented extensively with improving on this policy.

LogTM-SE The system takes advantage of unbounded HTM support using LogTM-SE with *perfect* filters. Though such filters are not implementable in real hardware [44], they represent an upper bound of how well LogTM-SE can perform. In contrast to ATMTP, transactions in LogTM-SE are not limited by the size and geometry of hardware resources such as caches and store buffers. Moreover, LogTM-SE transactions do not impose software overheads unless they abort, in which case a software abort handler is invoked.

BZSTM The system assumes no special hardware support and runs using *blocking* NZSTM software transactions. Objects never get inflated, so it is not necessary to check if the objects are inflated.

DSTM2-SF The system assumes no special hardware support and runs using DSTM2 Shadow Factory software transactions. We use the Shadow Factory because it is a blocking object-based STM designed from the ground up as a blocking algorithm. Moreover, the experimental results of Herlihy et. al. [18] show that DSTM2-SF significantly outperforms the nonblocking DSTM. Our implementation of DSTM2-SF uses the same visible reads and contention management extensions as NZSTM.

SCSS The system runs using NZSTM software transactions, and assumes an HTM with sufficient guarantees to support an SCSS implementation. This support obviates the need for inflating objects, as described in Section 2.3.2.

[2]Due to limitations of early versions of the simulator, we leave one free processor to handle interrupts, based on advice from the GEMS developers.

We present results of evaluating NZSTM, NZTM, and LogTM-SE on the simulator; and NZSTM, BZSTM, DSTM2-SF, and SCSS on the Rock machine. We also discuss our preliminary efforts to evaluate NZTM on Rock.

We use ATMTP to model best-effort HTM as it is based on the same memory models as LogTM-SE, has many limitations that we expect would exist in a real best-effort HTM [8], and is binary compatible with the forthcoming Rock processor. ATMTP uses a "requester wins" policy [33], while LogTM-SE uses built-in deadlock detection, and avoids aborts unless potential deadlock is detected.

The contention management policy used for software transactions is a variant of Karma [38], in which each transaction's priority is proportional to the number of objects it has already acquired in this transaction attempt. We combine Karma with a deadlock detection scheme; when a conflict is detected, a transaction is not aborted even if it is of a low priority, unless a deadlock has been detected or a timeout is triggered.

The deadlock detection scheme used in NZSTM is based on the one in LogTM [34]. By default, whenever a conflict is detected, transactions do *not* abort the other transaction unless a timeout is triggered. Whenever a transaction T_L detects a conflict with a high priority transaction T_H, T_L raises a flag and it waits until T_H is done, i.e., T_H commits or aborts. When a transaction T_H detects a conflict with a low priority transaction T_L whose flag is raised, T_H infers that there is a potential cycle and aborts T_L.

Because of the careful attention to cache performance in NZTM, we expect it should perform similarly to LogTM-SE in the absence of failure of the best-effort hardware transactions. If hardware transactions abort repeatedly, however, we expect NZTM's performance to be noticeably worse than that of LogTM-SE, because transactions must be executed in software.

Most of the benchmarks we use do not push the resource limits of ATMTP (configured as described above), and those that do only do so occasionally, so the main reason for transactions aborting is due to conflicts.

4.4 Results and Discussion

Figure 3 shows the completion rate of transactions (throughput) on the simulator, normalized to the throughput of LogTM-SE running on a single processor.

Figure 4 shows the completion rate of transactions (throughput) on the Rock machine, normalized to the throughput of a single global lock (not shown) running on a single processor. The data is normalized to the throughput of a single global lock as it demonstrates the performance that can be achieved in systems with no HTM support, with the same level of programming complexity as using transactions.

We note that the results obtained using the Rock machine are consistent with the simulator results and exhibit similar trends.

4.4.1 Results on Simulator

In general, LogTM-SE, has the best throughput. This is unsurprising because LogTM-SE is unbounded, uses perfect filters, which are not implementable in hardware, and has a better contention management policy than ATMTP's "requester wins". Moreover, NZTM hardware transactions have additional instrumentation to detect conflicts with software transactions, which is not necessary with an unbounded HTM such as LogTM-SE.

From these results we can make some general observations. In experiments with an overall low number of conflicts, as in `hashtable`, `genome` and `kmeans`, NZTM's throughput is within 10–15% of LogTM-SE's.

NZTM is not as competitive with LogTM-SE in experiments

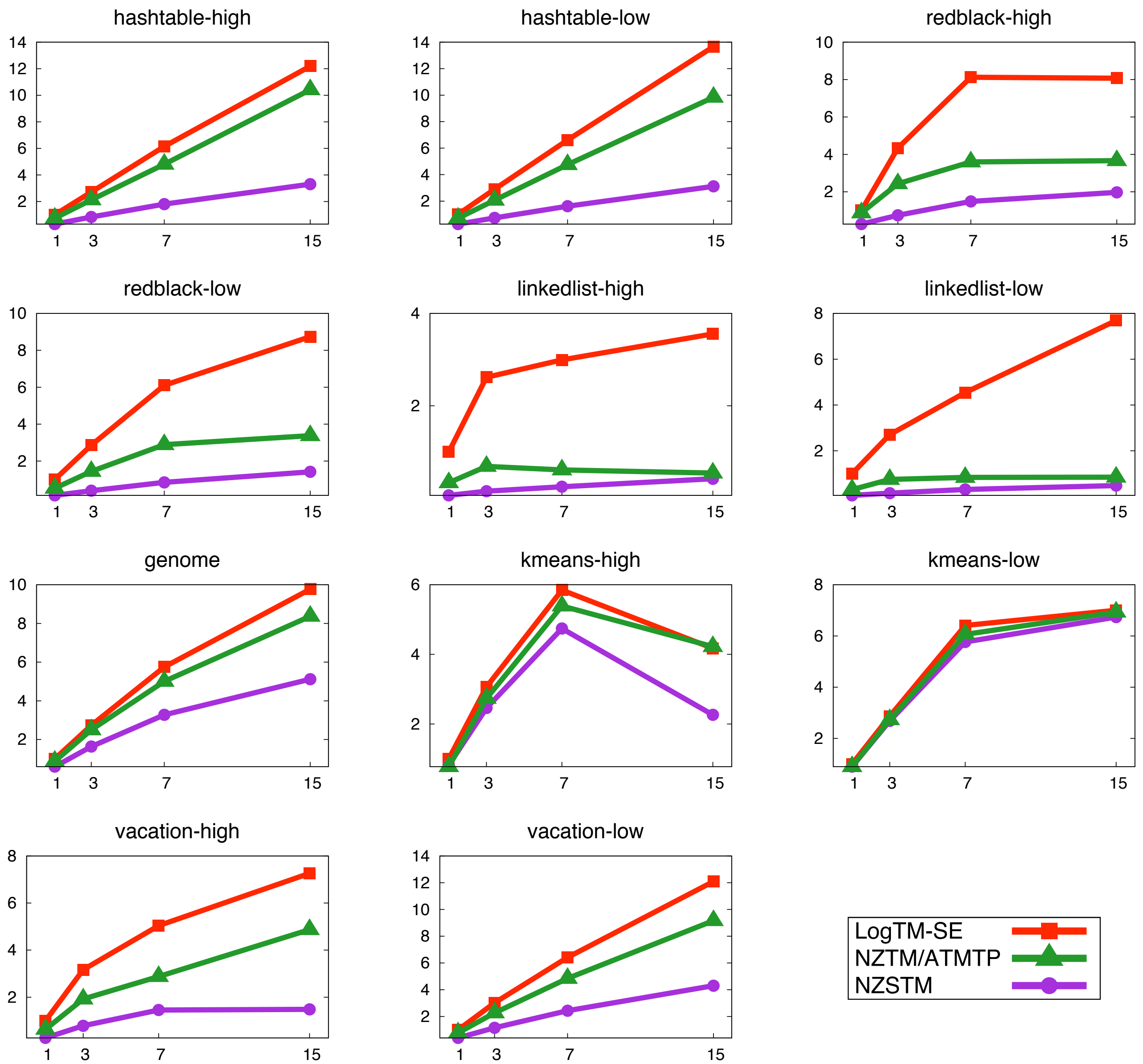

Figure 3: Results of running the benchmarks on the simulator. The x-axis represents the number of threads, the y-axis represents the throughput normalized to a single thread of LogTM-SE.

with higher conflict rates. We believe this is due to LogTM-SE's contention management, which only aborts conflicting transactions on potential deadlocks. ATMTP's contention management, on the other hand, uses a simple "requester wins" policy. When ATMTP detects a conflict, one transaction always aborts. This could lead to livelock, which results in many transactions falling back onto software [8, 33]. Improvements to the underlying HTM's contention management should reflect positively on NZTM.

`hashtable` has a small number of conflicts; our results show that at 15 processors, less than 1% of NZTM transactions abort — most transactions succeed in hardware. Therefore, this benchmark is a good indicator of the inherent overhead imposed by NZTM to detect conflicts with software transactions.

`linkedlist` and `redblack` have a high number of conflicts, leading to a larger gap in throughput between NZTM and LogTM-SE. `linkedlist` has more conflicts than `redblack`; at 15 processors, about 19% of `linkedlist`'s transactions abort, compared to 14% for `redblack`. This explains why the gap is bigger in `linkedlist` than it is in `redblack`. We expect a better hardware contention management policy would improve NZTM's throughput for such benchmarks.

In `kmeans`, only about 10% of the workload is transactional, and there are few conflicts. This results in the throughput of NZTM and LogTM-SE being closer than in most other benchmarks. An important observation is that the additional metadata and instrumentation overhead attached to transactional objects does not noticeably impact performance outside transactions.

`vacation`, a benchmark that uses linked list and red-black tree data structures, has throughput similar to the two microbenchmarks that are based on these structures. Additionally, `vacation`'s transactions are significantly bigger, in terms of runtime and size of the read and write sets, than all other benchmarks. At 15 processors, about 25% of all NZTM hardware transactions abort to due resource limitations. This is true even when using write-buffer and cache memory sizes that are larger than the ATMTP default, as mentioned earlier.

`genome` does not have many conflicting transactions; therefore, its behavior is similar to `hashtable`.

From these results, it seems that the biggest overheads in NZTM using ATMTP are contention management and instrumentation overheads. A better hardware contention management policy might mitigate the contention overheads. Hardware support that puts the burden of checking for conflicts between hardware and software transactions on software transactions, rather than hardware transactions, would mitigate instrumentation overheads for the hopefully common case; such support was proposed by Baugh et al. [3].

4.4.2 Results on Rock

The four TM algorithms we tested on Rock performed mostly within 10% of each other, except in the write-dominated `kmeans`. In most cases, NZSTM slightly lags BZSTM, by about 2–5%. This is due to overhead for checking for inflated objects; it is not due to any actual object inflation, which was not observed in our experiments (we did induce inflation in testing).

SCSS was virtually identical with NZSTM in all benchmarks except for `kmeans`. `kmeans` is a write-dominated benchmark where the overhead of wrapping every store with a short hardware transaction significantly impacts its performance.

The nonblocking NZSTM is competitive with the blocking DSTM2-SF, sometimes lagging behind it slightly, and at other times outperforming it slightly. In `kmeans`, NZSTM significantly outperforms DSTM2-SF. This is likely due to the additional space overhead in DSTM-SF. The size of the main transactional object in `kmeans`, without metadata, is 100 bytes, requiring two cache lines. Objects in most other benchmarks, including metadata, fit comfortably in a single cache line.

To reduce the effects of false sharing, we have padded these objects, making the object size for DSTM2-SF and NZSTM the same for most benchmarks. In `kmeans`, however, a padded object requires two cache lines when using NZSTM and four cache lines when using DSTM2-SF. Therefore, for DSTM2-SF, if an object is accessed that has not been accessed recently, then there are additional cache misses because DSTM2-SF allocates data for backup in place with the object. In contrast, as explained earlier, NZSTM uses thread-local memory for backups, which is reused after successful transactions, thus improving cache locality.

We have begun to experiment with NTZM using Rock, and have successful results for some cases. For example, in `hashtable` with low contention running on 16 processors, about 75% of all transactions run successfully in hardware, and throughput is over 60% higher than that of NZSTM running on Rock. However, in other cases, such as `redblack`, we were unsuccessful because hardware transactions failed repeatedly, even if retried many times.

An investigation of this transaction failure revealed similar behavior as reported by Dice et al. [9], in which code that handles transaction failures and decides how to react interferes with micro-DTLB state and prevents a subsequent retry from succeeding. This problem can be mitigated by instrumenting loads and stores in transactions using special instructions that Rock provides for this purpose [9]. However, in our context we have no specific compiler support for such instrumentation, and manual instrumentation is too time consuming and error prone to produce reliable results.

5. CONCLUDING REMARKS

Blocking is unacceptable in some cases, but previous nonblocking object-based STMs use expensive indirection in the common case, resulting in worse performance than blocking counterparts. We have introduced NZSTM, a nonblocking object-based STM that eliminates such indirection in the common case, resulting in performance competitive with blocking STMs. NZSTM supports our hybrid NZTM, which can exploit best-effort hardware transactional memory when available. We have also shown that the algorithm can be dramatically simplified using only small, simple hardware transactions if support for them is available, which has important implications for hardware designers.

Researchers at the University of Rochester [40] have had similar insights about the importance of eliminating indirection to improve the performance of nonblocking STMs. Their work was concurrent with and independent of ours, and they concentrate on a different design point, namely the use of special "Alert On Update" hardware to make nonblocking progress properties possible. While our approach is designed explicitly to be able to take advantage of HTM support to achieve similar benefits, our proposal also includes a nonblocking, zero-indirection STM that can run on existing systems today without additional hardware support.

Ananian and Rinard [2] eliminate indirection for hardware transactions, but software transactions that perform writes still require indirection. Moreover, their algorithm employs a *LoadLinked-StoreConditional* variant that is not supported in any modern system. NZSTM requires only *Compare&Swap*.

Marathe and Moir [26] implement a nonblocking word-based STM that eliminates much of the overhead of previous nonblocking word-based STMs by storing data in place in the common case, resorting to more complicated and expensive techniques to displace data only when necessary because of a conflict with an unrespon-

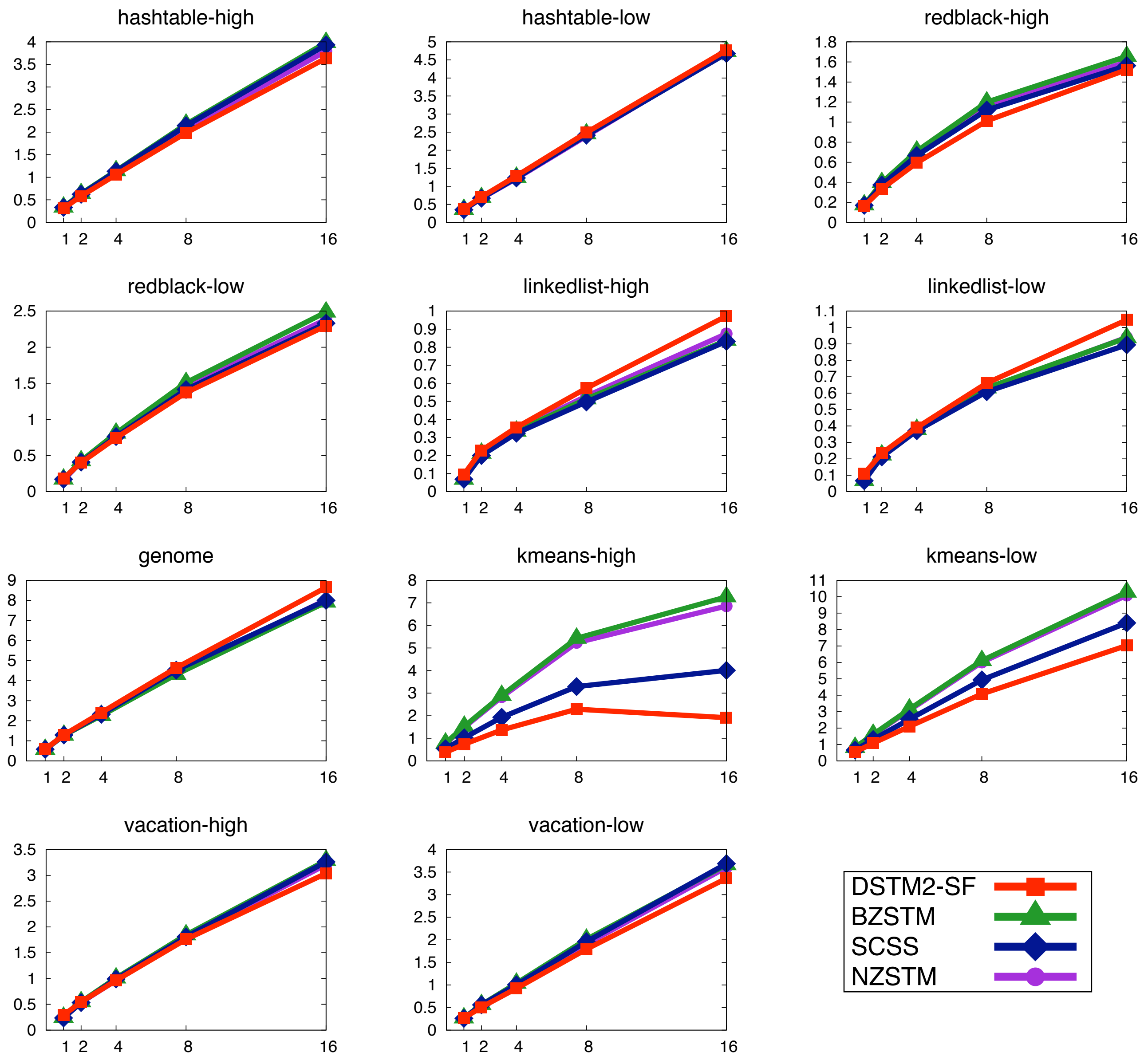

Figure 4: Results of running the benchmarks on the Rock machine. The x-axis represents the number of threads, the y-axis represents the throughput normalized to a single thread using a single global lock.

sive transaction. The design philosophy for NZ(S)TM was inspired in part by their work, but the details are quite different because they address word-based STMs, which cannot employ object headers and cannot collocate metadata with data (at least if they are to be integrated into compilers for languages such as C and C++, where we cannot dictate how data is laid out in memory).

Acknowledgments

We are grateful to Jayaram Bobba, Dave Dice, Kevin Moore, Daniel Nussbaum, and the anonymous reviewers for their valuable help and feedback on this work. We would also like to thank Sun Microsystems for funding this research and granting us access to Niagara and Rock servers, and Virtutech for the Simics license provided for the University of Auckland.

References

[1] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *Proc. 11th International Symposium on High-Performance Computer Architecture*, pages 316–327, 2005.

[2] C. S. Ananian and M. Rinard. Efficient object-based software transactions. *Synchronization and Concurrency in Object-Oriented Languages*, 2005.

[3] L. Baugh, N. Neelakantam, and C. Zilles. Using hardware memory protection to build a high-performance, strongly-atomic hybrid transactional memory. *ISCA*, 2008.

[4] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software transactional memory: why is it only a research toy? *Queue*, 6(5), Sep 2008.

[5] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay. Rock: A high-performance sparc cmt processor. *IEEE Micro*, 29(2):6–16, 2009.

[6] L. Dalessandro, V. Marathe, M. Spear, and M. L. Scott. Capabilities and limitations of library-based software transactional memory in C++. *TRANSACT*, 2007.

[7] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *ASPLOS*, pages 336–346, 2006.

[8] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. *ASPLOS*, 2009.

[9] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. Technical Report TR-2009-180, Sun Microsystems Laboratories, 2009.

[10] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Proc. 20th Intl. Symp. on Distributed Computing*, September 2006.

[11] R. Ennals. Software transactional memory should not be obstruction-free, 2005. `http://berkeley.intel-research.net/rennals/pubs/052RobEnnals.pdf`.

[12] K. Fraser. *Practical Lock-Freedom*. PhD thesis, Cambridge University Technical Report UCAM-CL-TR-579, Cambridge, England, Feb. 2004.

[13] D. Geer. Chip makers turn to multicore processors. *IEEE Computer*, 2005.

[14] A. Ghuloum. Unwelcome advice, 2008. `http://blogs.intel.com/research/2008/06/unwelcome_advice.php`.

[15] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *Proc. 31st Annual International Symposium on Computer Architecture*, June 2004.

[16] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 4th edition, 2006.

[17] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proc. 23rd International Conference on Distributed Computing Systems*, pages 522–529, 2003.

[18] M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 253–262, New York, NY, USA, 2006.

[19] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software transactional memory for supporting dynamic-sized data structures. In *PODC*, pages 92–101, 2003.

[20] M. Herlihy and J. E. 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.

[21] G. J. Holzmann. *The SPIN Model Checker*. Addison-Wesley Professional, 2003.

[22] G. Koch. Discovering multi-core: Extending the benefits of Moore's law. *Technology*, 2005.

[23] J. R. Larus and R. Rajwar. *Transactional Memory*. Synthesis Lectures on Computer Architecture. Morgan and Claypool Publishers, 2007.

[24] Y. Lev and M. Moir. Fast read sharing mechanism for software transactional memory, 2004. `http://research.sun.com/scalable/pubs/PODC04-Poster.pdf`.

[25] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, 35(2):50 – 58, 2002.

[26] V. Marathe and M. Moir. Efficient nonblocking software transactional memory (poster paper). In *PPoPP '07: Proceedings of the ACM SIGPLAN symposium on Principles and practice of parallel programming*, New York, NY, USA, 2007.

[27] V. Marathe, W. N. Scherer III, and M. L. Scott. Adaptive software transactional memory. In *Proc. 19th Intl. Symp. on Distributed Computing*, September 2005.

[28] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer III, and M. L. Scott. Lowering the overhead of nonblocking software transactional memory. 2006.

[29] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, 2005. 1105747.

[30] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. *IEEE International Symposium on Workload Characterization, 2008. IISWC 2008.*, pages 35–46, 2008.

[31] C. C. Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, June 2007.

[32] M. Moir. Hybrid transactional memory, July 2005. `http://research.sun.com/scalable/pubs/Moir-Hybrid-2005.pdf`.

[33] M. Moir, K. E. Moore, and D. Nussbaum. The adaptive transactional memory test platform: A tool for experimenting with transactional code for Rock. *The third annual SIGPLAN Workshop on Transactional Memory*, 2008.

[34] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based transactional memory. In *Proc. 12th Annual International Symposium on High Performance Computer Architecture*, 2006.

[35] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *Proc. 32nd Annual International Symposium on Computer Architecture*, pages 494–505, Washington, DC, USA, 2005.

[36] H. E. Ramadan, C. J. Rossbach, D. E. Porter, O. S. Hofmann, A. Bhandari, and E. Witchel. MetaTM/txLinux: Transactional memory for an operating system. In *Proc. 34th Annual International Symposium on Computer Architecture*, 2007.

[37] 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 *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 187–197, New York, NY, USA, 2006.

[38] W. Scherer, III, and M. Scott. Advanced contention management for dynamic software transactional memory. *PODC '05: Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, Jul 2005.

[39] N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, Special Issue(10):99–116, 1997.

[40] M. F. Spear, A. Shriraman, L. Dalessandro, S. Dwarkadas, and M. L. Scott. Nonblocking transactions without indirection using alert-on-update. In *Proceedings of the 19th ACM Symposium on Parallelism in Algorithms and Architectures*, June 2007.

[41] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 2005.

[42] F. Tabba, M. Moir, J. R. Goodman, A. W. Hay, and C. Wang. Appendix to NZTM. `http://www.cs.auckland.ac.nz/~fuad/nztm-appendix.pdf`.

[43] F. Tabba, C. Wang, J. R. Goodman, and M. Moir. NZTM: Nonblocking zero-indirection transactional memory. *TRANSACT*, 2007.

[44] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood. LogTM-SE: Decoupling hardware transactional memory from caches. *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 261–272, 2007.