NZTM: Nonblocking Zero-Indirection Transactional Memory *

Fuad Tabba Cong Wang
James R. Goodman
University of Auckland
{fuad,wcon006,goodman}@cs.auckland.ac.nz

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

Abstract

This workshop paper reports work in progress on NZTM, a nonblocking, zero-indirection object-based hybrid transactional memory system. NZTM can execute transactions using best-effort hardware transactional memory if it is available and effective, but can execute transactions using NZSTM, our compatible software transactional memory system otherwise.

Previous nonblocking software and hybrid 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", thus avoiding the costly levels of indirection in previous nonblocking STMs, and improves cache performance by collocating object metadata with the data it controls.

1. Introduction

With the computer industry increasingly focussing on building multicore processor chips, rather than making individual processors faster, it is increasingly important for everyday applications to be able to exploit multiple cores concurrently. But concurrent programming using traditional mechanisms such as locks and condition variables is subject to a number of serious pitfalls, including troublesome tradeoffs between performance, scalability, and software complexity.

Transactional memory [11] is widely considered to be the most promising avenue for alleviating this situation. With transactional memory, programmers specify *what* should be executed atomically, leaving the system to determine *how* this is achieved.

Numerous research groups are investigating techniques for providing system support for transactional memory. Despite significant progress, Software Transactional Memory (STM) [29] implementations are generally one to two orders of magnitude slower than what Hardware Transactional Memory (HTM) [11] can reasonably be expected to achieve. But 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.

Damron et al. [3] proposed Hybrid Transactional Memory (HyTM), in which transactions can be attempted using HTM support, but executed entirely in software in case they fail. Because HyTM can execute any transaction in software, it can exploit *best effort* HTM support that does not necessarily guarantee to be able to commit all transactions. Best effort HTM can be substantially simpler than *unbounded* HTM [1, 8, 24, 26], which supports all transactions in hardware. With the HyTM approach, we can develop and test transactional programs in existing systems today, and we can exploit best effort HTM support as it becomes available to boost performance. As HTM support improves over time, applications developed using HyTM will automatically benefit from the improvements. Thus HyTM supports and encourages an incremental approach to the adoption of HTM support.

In this paper, we focus on *object based* HyTM implementations, in which we assume data objects have headers that can be easily located whenever the application accesses an object. Herlihy et al. [10] introduced the first object-based dynamic software transactional memory system, DSTM. In DSTM, in order to access the current value of an object, a thread must first read a `Start` pointer associated with the object in order to determine a current `Locator`, then read the contents of the locator to determine the last transaction to open the object, then read the status of that transaction to determine which of two copies of the object indicated by the locator is current, and finally access that copy.

Recently, several research groups [4, 5, 6, 28] have proposed STM implementations that store object data "in place" and (optionally, in some cases) collocate object metadata with objects, thus avoiding costly cache misses caused by levels of indirection to reach the data, as well as separate metadata. The performance experiments presented by these groups confirm the intuition that this approach to structuring object data results in significantly better performance than the above-mentioned approaches that involve at least one level of indirection in all cases. However, all of these implementations have sacrificed the nonblocking progress proper-

* This work is based on work described in [32], where it was called GSTM.

ties provided by the earlier implementations, 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.

As proponents of blocking STMs argue [4, 6], in many cases it is possible to mostly avoid the disadvantages of a blocking implementation in practice. For example, the Solaris™ `schedctl` function can discourage (not prevent) the scheduler from preempting a thread during the blocking part of a transaction. Nonetheless, without an implementation that is truly nonblocking, we can still occasionally experience the disadvantages of blocking implementations. For example, if one transaction experiences a long delay due to a page fault or being preempted, this can cause many other transactions to have to also wait for a long time.

Blocking is more than "merely" a performance concern, however. For example, as pointed out by Ramadan et al. [27], it is *unacceptable* for an interrupt handler to be blocked by the thread it has interrupted. 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 transactional memory implementations, despite the appeal of simpler blocking ones.

In this paper, we show that in fact it is *not* necessary to sacrifice nonblocking progress properties in STMs in order to store data in place in the common case, and to collocate metadata with data objects. Specifically, we present Nonblocking Zero-Indirection Software Transactional Memory (NZSTM), which stores object data in place in the common case, resorting to a level of indirection only when a thread encounters a conflict with a thread that is unresponsive, for example because it is preempted. 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 be executed using best-effort HTM support, but executed using NZSTM if this is not successful. The design we present is optimized for the case in which HTM support is available and is able to commit most transactions. Because hardware transactions usually access objects in place, object metadata is collocated with the objects themselves, and transactions executed using HTM do not need to copy objects they modify, this arrangement should allow us to achieve near-optimal cache behavior in the common case.

A key difficulty in designing a nonblocking STM that stores object data in place is the uncertainty that arises when one transaction $T1$ is updating an object and another transaction $T2$ wishes to access the object. $T2$ cannot simply wait for to complete because this would result in a blocking implementation. $T2$ can attempt to inform $T1$ that it should stop modifying the object, but until $T2$ can determine that $T1$ has become aware that it should stop, it is not safe for $T2$ or other transactions to update the object data in place, because $T1$ may still overwrite the data. Therefore, it is hard to see any 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 $T1$ 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 it does so, then the uncertainty is resolved and we can continue to access the data in place. Thus, we can generally avoid the overhead of introducing levels of indirection except when the conflicting transaction is unresponsive (again, blocking implementations will simply freeze up in this case). We expect to be able to largely eliminate the performance gap between previous blocking and nonblocking object-based transactional memory implementations by eliminating their levels of indirection.

Several STM proposals [7, 18, 20] improve upon DSTM's performance by eliminating a level of indirection in the common case in which the object is not being modified. Nonetheless, all of these proposals still involve at least one level of indirection, even in the best case. Similarly, to our knowledge, all previous proposals for using hardware support to accelerate nonblocking object-based STM implementations still require at least one level of indirection [12, 15, 30].

The remainder of this paper is organized as follows. In Section 2, we describe our NZSTM and NZTM designs. In Section 3, we present preliminary simulation experiments. Section 4 reports on the current status of our ongoing work and discusses future work. We conclude in Section 5.

2. NZTM

In this section, we describe NZTM. We first describe the basic data structures used by NZTM, and then present a simple object-based STM that stores object data in place and collocates metadata with objects, but is blocking. Next we describe two ways to extend this STM to achieve our nonblocking NZSTM, one using HTM support, and one that can be used in existing systems today. Finally we describe how to use HTM support to execute transactions in a way that is compatible with transactions executed in the STM, yielding our hybrid NZTM design.

2.1 NZTM Data Structures

The following description will be best understood by readers already familiar with DSTM [10]. The programming model for NZTM is the similar to the one for DSTM, and data structures such as the `TMThread` and `ContentionManager` structures are mostly the same as for DSTM. The `Transaction` structure is also similar to the one in DSTM—it contains a `Status` that can be `Active`, `Aborted`, or `Committed`. However, NZTM's `Transaction` objects additionally contain an `AbortNowPlease` flag, which is used to request that the transaction abort itself (see Figure 1). This flag is stored to-

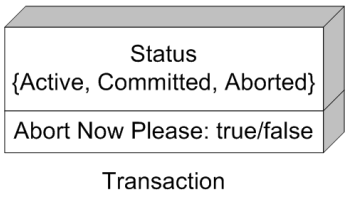

Figure 1. The Transaction data structure used by NZTM. The `AbortNowPlease` field is initially false.

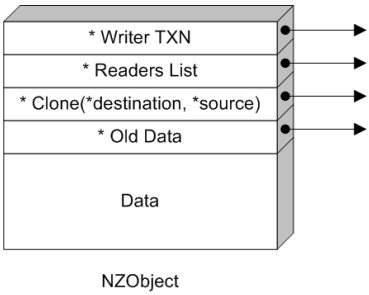

Figure 2. The structure of an `NZObject`. The Data field is the actual data and can have any size or structure.

gether with the `Status` field, so that both may be accessed atomically using a *Compare&Swap* (CAS) instruction.

The `NZObject` structure is analogous to DSTM's `TMObject` structure in that it encapsulates a program object that can be accessed by NZTM transactions. The `NZObject` structure contains the following fields (Figure 2): The `WriterTXN` field, if non-NULL, points to the last transaction to open this object for writing. The `ReaderList` field points to a linked list of readers of this object. The `Clone` field points to a function that can be used for creating a copy of the object. The `OldData` 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, there is no level of indirection required to access it.

After an `NZObject` is initialized, `Data` contains the initial value of the object, and all other fields (except `Clone`) are NULL.

2.2 Simple Blocking STM Algorithm

The simple blocking STM described in this section provides a basis for our non-blocking NZSTM and for our hybrid variant NZTM that can execute transactions using HTM support if available. Our primary objective to date has been to show that we can implement a non-blocking STM that can interoperate with transactions executed using HTM, with object data stored in place in the common case and object metadata collocated with the object data, and to evaluate the performance that can be achieved with such a system when HTM is usually effective. We have not experimented with the numerous alternative design decisions or with different contention management schemes in the STM.

As in DSTM, a thread begins a transaction by creating a new `Transaction` object with its status set to `Active`. It then proceeds to execute its transaction, "opening" each object it accesses, either for reading or for writing. When it completes execution of the transaction, it attempts to change its status from `Active` to `Committed`. During execution, another transaction that detects a conflict with this one may decide to wait for it or to attempt to abort it, according to the decision of a contention manager.

Unlike DSTM and other STMs, a transaction does not explicitly abort a conflicting transaction, but instead *requests* that it abort itself; this request is made by changing the transaction's `AbortNowPlease` field to true (while confirming that its `Status` is still `Active`). When the transaction observes that its `AbortNowPlease` flag is true, it sets its `Status` field to `Aborted`. The requesting transaction waits for this to occur before proceeding to open the object on which the conflict occurred. Because of this waiting, the simple STM is blocking.

Normally, the `Data` field of an `NZObject` contains the current value of the object. A transaction that wishes to modify the object acquires exclusive ownership of the object by placing a pointer to its `Transaction` in the object's `WriterTXN` field. Before modifying the `Data`, the transaction creates a copy of it (using the object's `Clone` function), and makes the object's `OldData` field point to the copy. This copy is needed in case the transaction aborts, in which case it can be copied back to `Data`, thus undoing the transaction's effects. Because the copy is not stored in place with the `NZObject` (as in the Shadow Factory of DSTM2 [9], which incurs 100% space overhead as a result), creating the copy may incur additional cache misses, as compared to a pure blocking STM, though we have some ideas for reducing this effect. Nonetheless, our approach does eliminate the level of indirection present in previous nonblocking STMs for transactions executed using HTM as well as for read operations of NZSTM in the common case. Because we are concentrating for now on the case in which HTM support is available and usually effective, and because reads are usually much more common than writes [2], this overhead is imposed only in uncommon cases. We have yet to evaluate a version of NZSTM that is optimized for existing systems (without HTM support), but when we do, we may find it preferable to use incremental backup logs, rather than copying entire objects.

We use a *visible readers* [13, 19] mechanism, in which a transaction opening an object for reading inserts a pointer to itself into the object's `ReaderList`; this way, transactions opening the object for writing can detect conflicts with transactions reading the object, and can decide to abort them if necessary to resolve the conflict. (The tradeoffs between invisible, semi-visible, and visible read mechanisms are much the same for NZTM as for other STM and HyTM systems; we have made this choice to optimize for systems in which

HTM is available and usually effective. In this case, the benefit of making hardware transactions faster should outweigh the cost of using visible readers for software transactions.)

With this overview in mind, we now explain in more detail how transactions open objects for reading and for writing.

Open for write If a transaction T that is opening an object for writing finds that the object's `WriterTXN` field points to T's `Transaction`, then T has previously opened this object for writing so it simply returns a pointer to the `Data` field. Otherwise, T must ensure that there are no conflicts with other transactions before acquiring ownership of the object. If the `WriterTXN` field is NULL, or points to a committed or an aborted transaction, there is no conflict with writing transactions, so T can atomically change the `WriterTXN` field to point to its `Transaction` (using CAS to ensure it has not changed in the meantime). If `WriterTXN` points to an active transaction, then the contention manager is consulted, and depending on the outcome, T either waits or requests that the active transaction aborts itself, as described above.

At this point, there are no conflicts with other writing transactions, but T still needs to check for conflicts with reading transactions. To do so, it traverses the `ReaderList` looking for active transactions. For each active transaction it finds (other than itself), T consults the contention manager to decide whether to wait or to request the other transaction to abort. If T is in the `ReaderList`, then it is upgrading its access to the object from read to write. In this case, because T already owns the object during this traversal, there is no risk that another transaction modified the object after T opened the object for read and before T acquired it for write unless that transaction has requested T to abort. Therefore, T can ensure the upgrade is consistent by checking its `AbortNowPlease` flag after acquiring ownership of the object. If it finds the flag set, it changes its status to `Aborted`, and returns a NULL pointer, indicating to the caller that the transaction has aborted. (In our prototype, objects are opened using macros that check whether the returned pointer is NULL, and control is diverted to an abort handler if it is.)

When all potential conflicts have been resolved, if the object was owned by an aborted transaction immediately before T acquired it, T restores the backup copy (indicated by `OldData`) if there is one. Otherwise, T creates a new backup copy using the `Clone` function and stores it under the `OldData` pointer. Finally, T returns a pointer to the data.

Open for read Next we describe how transaction T opens an object for reading. First, if T already owns the object for writing (i.e., `WriterTXN` points to T's `Transaction`), T simply returns a pointer to the `Data` field. Otherwise, T checks for conflicts with writing transactions and resolves them if necessary, as described above.

T then temporarily acquires exclusive ownership of the object by swapping the `WriterTXN` to point to itself. Next, T adds itself to the `ReaderList`, checks whether the previous writer transaction it replaced (if any) was aborted, and if so restores the backup if one exists. After restoring the backup, T stores NULL to the `OldData` field. Finally, T relinquishes exclusive ownership of the object by swapping `WriterTXN` to NULL, and returns a pointer to the `Data` field.

The reason for temporarily acquiring exclusive ownership of the object while adding an entry into the `ReaderList` is *not* to protect the `ReaderList`, which is updated using standard nonblocking mechanisms in preparation for the modifications we describe later to make our STM nonblocking. Instead, it is to avoid a race condition in which another transaction acquiring the object for write access does not encounter T on the `ReaderList` and thus fails to resolve the conflict between itself and T. Other techniques are possible, but this simple approach suffices for our purposes so far.

Before T returns a pointer to the `Data` field, it checks its `AbortNowPlease` field, and aborts and returns NULL (relinquishing ownership of the object first if necessary) if it is set. This is necessary to ensure that the set of objects the transaction has opened for read are consistent: if T has been aborted due to a conflict with another transaction on another object while attempting to open this object for read, the reads may not be consistent, in which case it is not safe to return to user code.[1] Validating at other times, while not strictly necessary, may be desirable for performance reasons. For example, validating before asking another transaction to abort and/or before waiting for another transaction that we have requested to abort may avoid unnecessary aborts and waiting.

2.3 NZSTM: Making the STM Non-Blocking

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

Using Short Hardware Transactions

In the simple blocking STM described above, the reason a transaction that requests another to abort must wait for it to confirm that it has aborted is that it is not safe to restore the backup to the `Data` field and continue to access the object in place if there is still a possibility that the victim transaction might store to the `Data` field. Such "late stores" can be avoided by making them conditional upon the storing transaction's `AbortNowPlease` flag not being set. This can be achieved using short HTM transactions that read the `AbortNowPlease` flag, confirm it is false, and perform the store if so. This method can be used provided the HTM support guarantees that such simple transactions will

[1] Because many object-based systems execute in managed runtime environments that can catch and mask exceptions, this may be unnecessary in some cases. But given the complication involved in ensuring that all possible bad behavior that could arise from inconsistent reads (including infinite loops that do not raise any exceptions), and the low cost of validating in our system, we prefer to avoid relying on such mechanisms for now.

always (eventually) succeed, i.e., that no such transaction will deterministically fail. This idea was first proposed in [23]. Next we explain how to make the simple blocking STM nonblocking, resulting in our NZSTM, even if no HTM support is available or the HTM support does not make such a guarantee.

Displacing data

We can also make the simple blocking STM described above nonblocking by temporarily displacing the logical data into a different location than the `Data` field of the `NZObject`. Because previous nonblocking object-based STMs [7, 10, 18, 20] involve at least one level of indirection in all cases, we can "inflate" an object and use techniques like existing object-based STMs use when an aborted owner is unresponsive.

For now our design uses a DSTM-like approach when an object needs to be inflated. Because DSTM's `Start` object is just a pointer, we can integrate it into NZSTM by using the `WriterTXN` field as a `Start` pointer, and indicating that it should be treated as such using the low order bit of the pointer. We expect the need to inflate objects to be rare, so the overhead of using a DSTM-like inflated object is probably acceptable in general; nevertheless, we may investigate reducing this cost in the future using approaches like those in the literature that aim to improve on DSTM.

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 `Locator`; points the locator's `Transaction` field to T's `Transaction`; points the locator's `Old` field to the backup copy created by the unresponsive transaction or to the `Data` field if there isn't one; and points the locator's `New` field to a private copy of this backup created using the `Clone` function. NZSTM's `Locator` object has an additional field `AbortedTXN`, which points to the `Transaction` of the unresponsive transaction.

Next, T attempts to swap `WriterTXN` from the unresponsive transaction's `Transaction` to the newly created locator, setting the low-order bit to 1 to indicate that the object now points to a DSTM-like locator, rather than to an NZSTM `Transaction`. This results in a state like the one illustrated in Figure 3. Henceforth, the object is treated like a DSTM object (with the addition that each new `Locator` introduced contains the `AbortedTXN` field from the replaced `Locator`, thus preserving the identity of the unresponsive transaction).

Once the unresponsive transaction finally aborts itself, so we know it is no longer modifying the `Data` field of the object, it is desirable to restore the object to being a normal `NZObject` so that subsequent transactions can again enjoy the performance benefits of accessing the object in place. This can be achieved by a transaction T as follows. T opens the object for writing according to the normal

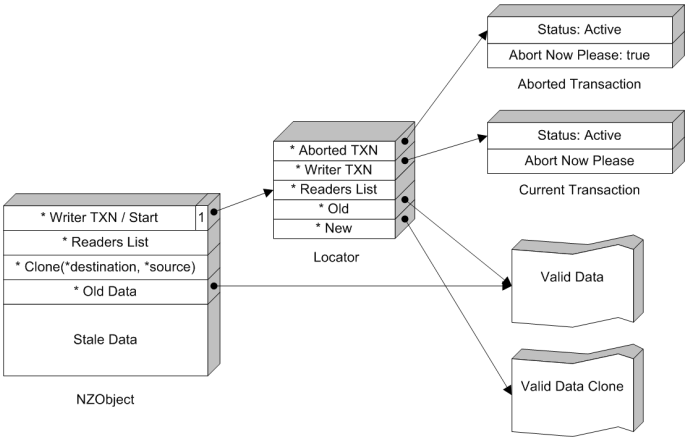

Figure 3. An `NZObject` immediately after being transformed to behave in the same manner as a DSTM `TMObject`. The LSB of the `WriterTXN` indicates how the object should be interpreted.

DSTM method, except that it stores a pointer to its own `Transaction` in the `AbortedTXN` field of the new locator it installs. This establishes T as the transaction that must be aborted and respond before a subsequent attempt to restore the object can begin, in case T does not complete the restoration first.

Next T stores into `OldData` a pointer to the current object value (indicated by the `Old` pointer in the newly installed `Locator`), so that it will serve as a backup in case T aborts. Then T attempts to replace the locator it just installed in `WriterTXN` with a pointer to its own `Transaction` (setting the low-order bit to 0 to indicate that the object is now back in normal mode). If this succeeds, T simply copies the current object value to the `Data` field and proceeds as normal.

Before T swaps its `Transaction` into `WriterTXN`, other transactions may wish to access the object, in which case they can proceed as usual, opening the object according to the modified DSTM method described above. If a transaction is successful in doing so before T swaps its `Transaction` into `WriterTXN`, then T's CAS on `WriterTXN` will fail, so T will know that it did not succeed in restoring the object. In this case, it aborts itself in order to allow a subsequent restoration attempt to succeed (because it is now the transaction identified by `AbortedTXN`).

One subtle point with this mechanism is that a slow transaction that has started to restore an object might store an out-of-date value into `OldData`. However, this is not a problem, because subsequent attempts to restore the object will not begin until after this transaction acknowledges that it has aborted. Until then `OldData` is irrelevant because the object remains inflated.

2.4 NZTM

NZTM is a HyTM system that uses NZSTM for software transactions. Like the HyTM system presented in [3], transactions can be attempted using HTM and if (repeatedly) unsuccessful, are eventually retried using NZSTM software transactions.

When a transaction executed using HTM opens an object, it checks for conflicts with software transactions, and explicitly aborts itself if any are discovered; it can then retry either in hardware or in software, according to decisions made by the contention manager. If no conflict is discovered, the transaction can safely proceed because a subsequent conflict that arises with a software transaction will modify data that the hardware transaction has read, thus preventing it from committing successfully.

A variety of approaches to checking for conflicts with software transactions are possible. Generally, more conservative approaches are simpler, but are more likely to revert to software, which is harmful to performance. Different schemes can be used together. The simplest and most conservative scheme is as follows. A hardware transaction opening an object for reading checks that the `WriterTXN` field is NULL, aborting if not, and a hardware transaction opening an object for writing checks that both `WriterTXN` and `ReaderList` are NULL, aborting if not.

This scheme is conservative because if `WriterTXN` points to an aborted or committed transaction, there is no conflict with writers. Similarly, it may be that the `ReaderList` contains pointers to only aborted and committed transactions, in which case there is no conflict with readers. Therefore, in either case, hardware transactions could look in more detail to determine if there is a real conflict, checking the status of the transaction identified by `WriterTXN` to check for conflicts with writers and/or traversing the `ReaderList` checking the status of each transaction in the list to see if any (other than the executing transaction) are active. In case we do check the status of the `WriterTXN`, we must return a pointer to `Data` if the identified transaction is committed and return `OldData` if it is aborted. In the simple scheme described so far, if the low-order bit of `WriterTXN` is set, we would simply abort the hardware transaction.

Hardware transactions can also improve the performance of subsequent transactions that access the object (either in hardware or in software) by setting `WriterTXN` and `ReaderList` to NULL in some cases. Specifically, if a hardware transaction determines that there are no active transactions in the `ReaderList`, it can set `ReaderList` to NULL, eliminating the need for subsequent hardware transactions to traverse the list. Similarly, if `WriterTXN` identifies a committed transaction, the hardware transaction can set `WriterTXN` back to NULL. If `WriterTXN` identifies an aborted transaction, this is *not* safe, because the current value is in `OldData`; it would be safe to set `WriterTXN` to NULL if it also copied `OldData` back to `Data`.

Finally, we note that hardware transactions can even access objects that are inflated to DSTM-like objects. Hardware transactions can modify the current object version (identified by the `Old` or `New` field of the current `Locator`, depending on the status of the transaction identified by the locator, provided it is committed or aborted) in place, without replacing the locator, because any software transaction that attempts to open the object will modify `WriterTXN`, thereby preventing the hardware transaction from committing successfully. Similarly, hardware transactions can participate in resetting the object to a normal `NZObject` by following the same code as a software transaction would. However, if a hardware transaction encounters a situation in which a software transaction would request another to abort, then it must abort itself, because, without more sophisticated HTM support than we assume, it is not useful to request the transaction to abort and then wait for it to abort itself because when the software transaction aborts itself, this causes the hardware transaction to abort anyway.

3. Evaluation

In this section we report on an environment created to allow us to evaluate NZTM and NZSTM.

3.1 Experimental Environment

We used a simulation framework based on Virtutech Simics [16] in conjunction with customized memory models built on the University of Wisconsin GEMS [22] and based on the model used for LogTM [24]. The simulator models processors have best-effort HTM support as well as LogTM support, with instructions for begin-transaction, commit-transaction, and abort-transaction.

For best-effort HTM, begin-transaction includes an address where execution is resumed when the transaction is aborted. These instructions were simulated using Simics magic instructions (special no-op instructions Simics can catch and pass to the underlying memory model). This simulation for the best-effort HTM uses L1 and L2 cache as transactional cache; therefore hardware transactions are limited by the size of the L2 cache for their read/write-sets.

The simulated system for all the simulation runs is a SPARC®/Solaris™ Sun Fire™ server, and the parameters used were based on those used in LogTM (Table 1). This model assumes a traditional SMP, where each processor is single-threaded (single strand) with no shared caches.

3.2 Benchmarks

We utilized four microbenchmarks with varying workloads and bottlenecks to test the Transactional Memory implementations developed. These benchmarks have been used by other Transactional Memory implementations such as DSTM and RSTM. These benchmarks are only our first step towards understanding the behavior of Transactional Memory programs and developing benchmarks that are more representative. Following is a description of the benchmarks:

Parameter	System Model Settings
Processors	16, 1 GHz, single-issue, in-order, non-memory Instructions per Cycle (IPC) = 1
L1 Instruction Cache	16 Kilobyte 4-way, 1-cycle latency
L1 Data Cache	16 Kilobyte 4-way, 1-cycle latency
L2 Cache	4 Megabyte 4-way unified, 12-cycle latency
Memory	8 Gigabyte, 80-cycle latency
Directory	Full-bit vector sharer list; migratory sharing optimization; Directory cache, 6-cycle latency
Interconnection Network	Hierarchical switch topology, 14-cycle link latency

Table 1. Simulated system model parameters. Based on [24].

LinkedList 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 0..255, with the distribution of operations being 10:10:80, respectively. Based on the code that comes with DSTM [10].

RBTree Same benchmark as LinkedList, with the concurrent set instead implemented by a red-black tree. More potential for concurrent operations than LinkedList, although re-balancing the tree when a node is inserted or deleted sometimes results in changes near the root, thus conflicting with most ongoing operations. Based on the code that comes with DSTM [10].

Hash Same benchmark again, this time using a chained hash table in which each bucket consists of a sorted linked list. The hash table has 256 buckets. Thus, because values are chosen from 0..255 in the benchmark, there are no hash collisions. Thus parallelism for this benchmark should be high as conflicts only occur when two threads choose the same value at around the same time. Based on the code that comes with RSTM [21].

LFUCache A web cache simulation with least-frequently-used page replacement. It uses a large array-based index of 2048 entries, and a smaller binary-tree-based priority queue of 255 entries to track the pages that are accessed the most in the simulation. Worker threads repeatedly access a page, and a real web cache workload is approximated by randomly choosing pages from a Zipf distribution with exponent 2. Based on the code that comes with RSTM [21].

3.3 Experiments

Each benchmark was run for 1, 2, 4, 8 and 16 threads each running on its own processor. We first initialize the relevant data structures and then each thread traverses these structures in order to load them into the cache memory. Then we begin measurements, recording the simulated machine's elapsed clock cycles required for all threads to complete 2,000 operations each.

We evaluated four different mechanisms for executing transactions:

Simple Lock The system does not take advantage of HTM support. A single global lock is used. A thread must acquire the lock before beginning a transaction and release it on committing.

LogTM The system takes advantage of HTM support by using the same mechanism described in LogTM [24]. To simulate LogTM we used the code which comes with Wisconsin GEMS 1.4 [22]. These transactions, unlike the best-effort hardware support we employ for NZTM, are unbounded and not limited by the size of L2 cache. Moreover, LogTM transactions do not impose any software overhead on the benchmarks.

NZSTM The system assumes no hardware support and runs by using pure NZSTM software transactions. Performance seems to be poor in this area as we have optimized NZSTM for hardware support. However, we are aware of software optimizations and different design points that can be implemented to enhance performance, and are planning on including them in future work.

NZTM The system takes advantage of best-effort HTM support, built on modified LogTM (GEMS 1.4 [22]) code. However, it utilizes a First-In-Fist-Out (FIFO) contention management scheme similar to the one employed by TLR [25], and as mentioned earlier, limits the size of the transactions by the size of the L2 cache for their read/write-sets. Transactions executed using this best-effort HTM are augmented to check on a per-object basis for conflicts with transactions executed using NZSTM. When a hardware transaction aborts, the thread retries the execution zero or more times, then switches to NZSTM to execute the transaction. NZTM/HTM/3 indicates that the system attempts the transaction three times in hardware and finally falls back onto software (NZSTM) transactions, while NZTM/HTM/16 tries sixteen times instead of three before falling back onto software.

We chose to compare the performance of NZTM against Simple Lock and LogTM for two reasons. First, Simple Lock demonstrates the performance that can be achieved in existing systems today (with no HTM support) with the same level of programming complexity as using transactions. This provides a baseline scenario in a multithreaded environment,

because such schemes are known to scale very poorly. On the other hand, the near-minimal overhead imposed on hardware transactions by LogTM sets a performance standard when there are no conflicts, so this allows us to evaluate the performance and scalability of our system under nearly ideal conditions.

Because we paid careful attention to cache performance when designing NZTM, we expected that it should perform similarly to LogTM in the absence of failure of the best-effort hardware transactions. When the best-effort HTM transactions fail due to conflicts, however, we can expect different performance. If that happens, NZTM invokes NZSTM, switching to a software-based scheme that avoids the serialization of a lock, but brings the overhead of executing transactions using STM as compared to HTM. Thus the failure of best-effort hardware transactions could cause a significant penalty. Our goal is to compare these schemes as a first step in evaluating the effectiveness of the hybrid approach, though we note that the software for NZSTM is not currently well tuned.

None of the benchmarks push the resource limits of the best-effort HTM, so the only reason for transactions to abort is cache-coherence conflicts. We have not had time to experiment extensively with the optimal number of retries, but we did find that retrying several times generally resulted in better performance.

We have not yet implemented the non-blocking capability of NZTM. We are currently working on implementing the code for this, though implementing it would have a trivial effect on our reported results: the additional overhead incurred is merely confirming that the low-order bit of the `WriterTXN` pointer (which must be read anyway) is zero. Because the bit is one only when one software transaction has conflicted with another that is unresponsive, in the common case execution of hardware transactions will otherwise be the same as in our current implementation.

As noted in Section 2.4, HTM-supported transactions may use more or less sophisticated methods for determining when to abort. Because NZSTM is currently untuned, we wanted to minimize the frequency of aborting, so we implemented the most sophisticated scheme that is compatible with our current implementation. Thus, hardware transactions check the status of the transaction identified by `WriterTXN` (if any), and also traverse the `ReaderList`, checking that there is no conflict with software readers (if any). We have not yet experimented with the other alternatives.

3.4 Results

Figure 4 shows the rate at which transactions are completed for each of the four benchmarks. In all cases Simple Lock performance is comparable to NZTM for a single thread, but degrades rapidly for additional threads.

For Hash, the transaction execution rates for the LogTM and NZTM are close, with LogTM slightly better than NZTM in all cases. NZSTM's performance is significantly worse than either of these two schemes; however we note that when compared to Simple Lock, it scales better and outperforms it at 4 processors and above.

RBTree tree shows similar results to Hash, with LogTM outperforming NZTM at all threading levels. The larger margin than for Hash probably reflects the fact that RBTree opens multiple objects, increasing the overhead for NZTM but not for LogTM. However, RBTree performance does not scale for NZSTM, possibly due to the contention at the root node of the tree.

LFUcache similarly shows nearly the same results, with LogTM and NZTM showing roughly a 25% improvement in performance going from one to two processors, but throughput actually declining by approximately the same amount as more processors are added.

LinkedList shows interesting behavior. While LogTM shows superior performance in all cases up to 8 threads, NZTM shows nearly linear improvement in performance up to 16 processors, while LogTM exhibits much smaller improvements, and actually declines beyond 8 processors. Although we have not yet investigated the reason for this, a possible explanation is the different ways NZTM and LogTM manage contention.

3.5 Discussion

The single-threaded results shed some light on the overhead of the various schemes in the absence of concurrency. Almost in all cases, LogTM outperforms the other schemes because its only overhead is in the hardware itself, and could be presented as a small one-time overhead *per transaction*. Its advantage over Simple Lock probably results from the fact that Simple Lock instead performs a CAS operation per transaction to acquire the lock. While the overhead for NZTM is also read-only, this overhead is incurred *per object*.

As the number of processors running the benchmarks increases, performance for coarse-grained locking degrades dramatically, while LogTM and NZTM both show significant improvement for all but LFUCache, which shows no significant improvement for any scheme beyond two threads.

We note that the number of times we attempt a transaction in hardware, when using NZTM, seems to only matter after exceeding a certain threshold. For example, we observe in the LinkedList benchmark that NZTM/HTM/3 and NZTM/HTM/16 have similar performance except for 16 processors. This could be explained by the fact that at 16 processors many more transactions abort and fallback onto software transactions. This might imply that a conservative approach when setting the number of tries in hardware, favoring a higher number, is preferable.

We also note that Simple Lock uses a single traditional lock, thus serializing *all* operations. While using a Reader-Writer lock would allow us to execute lookups in parallel, this entails at least some additional programming complex-

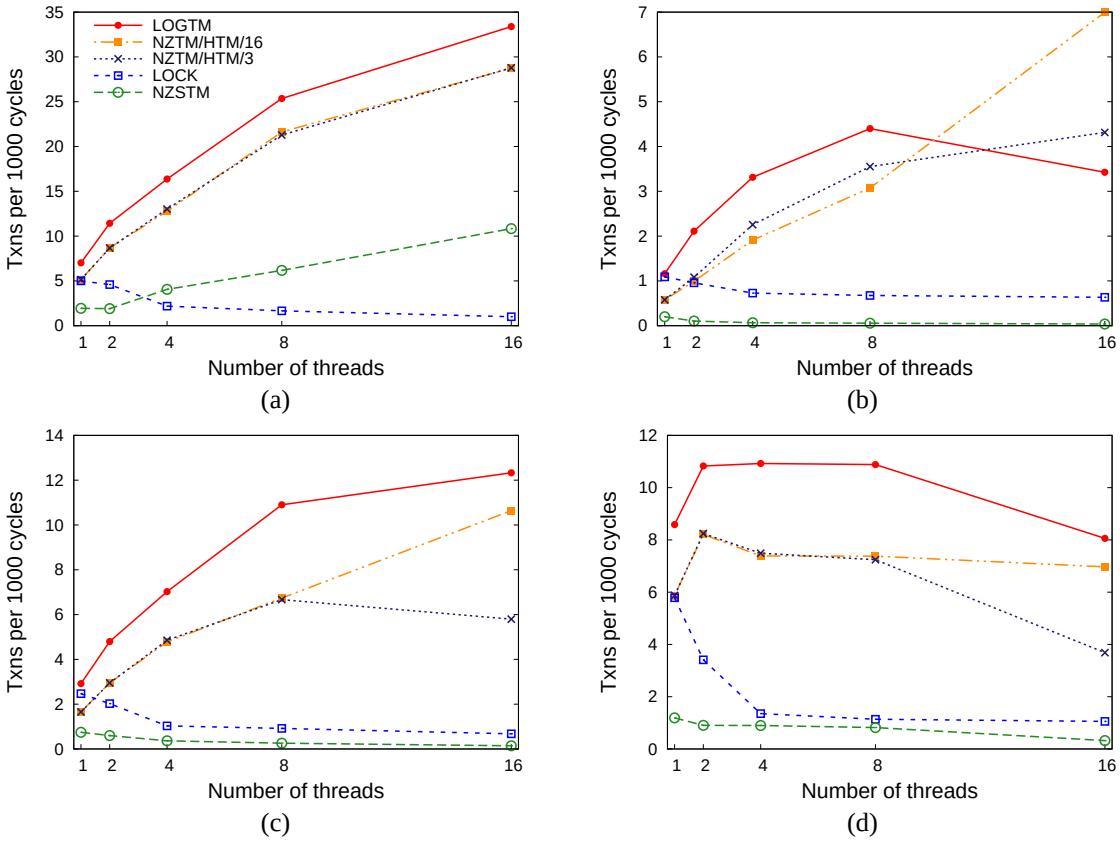

Figure 4. Total throughput: (a) Hash (b) LinkedList (c) RBTree (d) LFUCache.

ity over a transactional programming model, and a significant complexity in medium to large systems with multiple such locks, because in this case it is the programmer's responsibility to ensure that the right locks are held in the right modes for correctness, and that deadlock is not possible.

The results seem to confirm our expectations that when running more than one thread, NZTM provides dramatic improvements over the software-only methods using hardware support of modest complexity (compared to LogTM). Furthermore, when conflicts become frequent, the switch to the software-based NZSTM does not seem to impose much of an overhead on performance. While admittedly preliminary, we believe these results make a strong case for further investigation of this approach.

4. Status and Ongoing Work

We still have plenty of work to do in developing and fully evaluating our system. Nonetheless, as described in this workshop paper, we have already developed a prototype implementation as well as a simulator that allows us to evaluate the performance of NZTM in systems with support for hardware transactional memory. Our preliminary performance results, together with planned improvements and optimizations, give us confidence that our careful attention to cache behavior will allow us to achieve implementations that are very competitive with previous blocking implementations, while eliminating their need to wait for unresponsive transactions.

While we have introduced a novel approach to structuring transactional data and metadata in order to significantly improve performance over previous nonblocking STM and Hybrid TM systems, NZSTM and NZTM are subject to many of the design alternatives and tradeoffs encountered in previous work. We have yet to apply the lessons learned from this other work in the context of NZTM. We are particularly interested in the observation due to Lev et al. [14] that, by supporting the execution of transactions by different implementations in different "modes", we can not only reduce overhead on transactions executed in hardware, but also allow more flexibility in the implementation of software transactions.

We are currently optimizing NZSTM to work better as a software-only scheme. This is being done by allowing for invisible reads and investigating the integration of invisible reads with hardware transactions. Moreover, since memory allocation is an integral part of the operation of NZSTM, we are also investigating the broad area of scalable mem-

ory allocators and garbage collectors and how those can be leveraged for our advantage.

We are also currently implementing and testing the compotent of NZSTM that makes it nonblocking. Moreover, most of the testing performed so far has been on simulated environments, therefore we are planning to conduct a thorough test of NZSTM on a real multi-processor machine.

Benchmarking is also an important aspect of this endeavor. Currently we are only using four simple benchmarks. We are investigating which other benchmarks might be suitable for testing transactional memory schemes and preparing to implement these benchmarks and apply our schemes to them.

5. Concluding Remarks

We have introduced NZSTM, an object-based software transactional memory that is nonblocking, and eliminates the expensive levels of indirection in previous such STMs in all but the uncommon case of a conflict with an unresponsive thread. Furthermore, we have shown how transactions can be executed using best effort hardware transactional memory if it is available, yielding our hybrid transactional memory system NZTM.

We note that researchers at the University of Rochester [31] have had similar insights about the importance of eliminating indirection to improve on the performance of previous nonblocking STM. 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 special HTM support to achieve similar benefits, our proposal nonetheless includes a nonblocking, zero-indirection STM that can be used in existing systems today, without additional hardware support.

Marathe and Moir [17] showed that it is possible to implement 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 unresponsive transaction. The design philosophy for NZ(S)TM was inspired in part by work of Marathe and Moir, but the details are quite different because they addressed 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 Dan Nussbaum for useful conversations related to our simulation work, and especially to Kevin Moore for his help getting us started with the Wisconsin simulation tools. We thank Jayaram Bobba for his comments on the implementation of the Wisconsin simulation tools, and Virtutech AB for the Simics academic site 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, Feb. 2005.

[2] J. Chung, H. Chafi, A. McDonald, C. C. Minh, B. D. Carlstrom, C. Kozyrakis, , and K. Olukotun. The common case transactional behavior of multithreaded programs. In *Proc. 12th Annual International Symposium on High Performance Computer Architecture*, 2006.

[3] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 336–346, New York, NY, USA, 2006. ACM Press.

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

[5] D. Dice and N. Shavit. What really makes transactions faster? In *TRANSACT Workshop*, June 2006. http://research.sun.com/scalable/pubs/TRANSACT2006-TL.pdf.

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

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

[8] 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.

[9] 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. ACM Press.

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

[11] 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.

[12] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid transactional memory. In *Proc. ACM SIGPLAN Sym-*

posium on Principles and Practice of Parallel Programming, Mar. 2006.

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

[14] Y. Lev, M. Moir, and D. Nussbaum. PhTM: Phased Transactional Memory. Transact 2007 workshop, Aug. 2007. http://research.sun.com/scalable/pubs/TRANSACT2007-PhTM.pdf.

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

[16] P. Magnusson, F. Dahlgren, H. Grahn, M. Karlsson, F. Larsson, F. Lundholm, A. Moestedt, J. Nilsson, P. Stenstrom, and B. Werner. SimICS/sun4m: A virtual workstation. In *Proceedings of the USENIX 1998 Annual Technical Conference (USENIX '98)*, June 1998.

[17] 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. ACM Press.

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

[19] V. J. Marathe and M. L. Scott. A qualitative survey of modern software transactional memory systems. Technical Report TR 839, Computer Science Department, University of Rochester, June 2004.

[20] 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. Transact 2006 workshop, June 2006. http://www.cs.rochester.edu/u/scott/papers/2006_TRANSACT_RSTM.pdf.

[21] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer III, and M. L. Scott. The rochester software transactional memory runtime, 2006. http://www.cs.rochester.edu/research/synchronization/rstm/.

[22] 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.

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

[24] 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.

[25] R. Rajwar and J. R. Goodman. Transactional lock-free execution of lock-based programs. *SIGARCH Comput. Archit. News*, 30(5):5–17, 2002.

[26] 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.

[27] 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.

[28] 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. ACM Press.

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

[30] A. Shriraman, V. J. Marathe, S. Dwarkadas, M. L. Scott, D. Eisenstat, C. Heriot, W. N. Scherer III, and M. F. Spear. Hardware acceleration of software transactional memory. Transact 2006 workshop, June 2006. http://www.cs.rochester.edu/u/scott/papers/2006_TRANSACT_RTM.pdf.

[31] 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.

[32] F. Tabba. Practical transactional memory: The hybrid approach. Master's thesis, University of Auckland, Auckland, New Zealand, Feb. 2007.