# Transaction Safe Nonblocking Data Structures *

Virendra J. Marathe and Michael L. Scott

Technical Report #924

Department of Computer Science, University of Rochester
{vmarathe, scott}@cs.rochester.edu

September 2007

**Abstract**

This paper addresses interoperability of software transactions and ad hoc nonblocking algorithms. Specifically, we explain how to modify arbitrary nonblocking methods so that (1) they can be used both inside and outside transactions, (2) external uses serialize with transactions, and (3) internal uses succeed if, only if, and when the surrounding transaction commits. Interoperability has two important benefits. First, it allows nonblocking methods to play the role of fast, closed nested transactions, with potentially significant performance benefits. Second, it allows programmers to safely mix transactions and nonblocking methods, e.g., to update legacy code, call nonblocking libraries, or atomically compose nonblocking methods.

We demonstrate our ideas in the context of the Java-based ASTM system on several lock-free data structures. Our findings are encouraging: Although performance of transaction-safe nonblocking objects does not match that of the original nonblocking objects, the degradation is not unacceptably high (particularly after application of an optimization we call *lazy logging*). It is, moreover, significantly better than that of analogous transactional objects. We conclude that transaction safe nonblocking objects can be a significant enhancement to software transactional memory.

**Keywords:** Transactional Memory, Nonblocking Synchronization, ASTM, Lazy Logging

## 1 Introduction

Transactional Memory (TM) [6, 7] is an abstraction for parallel programming that promises to simplify the task of writing correct multithreaded code. As typically conceived, transactions take the place of mutual exclusion locks: The programmer identifies blocks of code that should execute as transactions. The underlying system ensures that these transactions are atomic, isolated, and consistent, while running them in parallel whenever possible. The typical implementation achieves this concurrency via speculation: running transactions optimistically in parallel, and aborting and "rolling back" in the event of conflict.

TM advocates expect that with some mix of hardware and high-quality runtime software, transactions will combine performance comparable to that of fine-grain locking protocols with programmer effort comparable to that of coarse-grain locking. Most researchers acknowledge, however, that locks will continue to be needed, for interoperability with legacy code and for protection of operations (e.g., I/O) that cannot safely

*This work was supported in part by NSF grants CNS-0411127 and CNS-0615139, equipment support from Sun Microsystems Laboratories, and financial support from Intel and Microsoft.

be rolled back. We posit that the other traditional alternative to locks—nonblocking concurrent objects—will also continue to be needed, again for interoperability with legacy code, and also for performance, at least on traditional hardware: for certain important data structures, nonblocking implementations are almost certain to be faster than equivalent code based on software transactional memory (STM).

In this paper we propose a construction for *transaction-safe* nonblocking objects. Specifically, we explain how to modify arbitrary nonblocking operations so that (1) they can be used both inside and outside transactions, (2) external uses serialize with transactions, and (3) internal uses succeed if, only if, and when the surrounding transaction commits.

Interoperability has two important benefits. First, it allows operations on nonblocking data structures to play the role of fast, closed nested transactions, with potentially significant performance benefits. Second, it allows programmers to safely mix transactions and nonblocking operations, e.g., to update legacy code, call nonblocking libraries, or atomically compose nonblocking operations.

Our approach is to base nonblocking objects on "transaction aware" versions of references and other primitive types. These provide special read, write, and compare-and-swap operations, which the programmer uses in place of conventional accesses. The special operations guarantee that any realizable history of the nonblocking object is equivalent to a history in which nonblocking non-transactional operations are non-overlapping not only with each other and with transactional nonblocking operations, but with the *entire* transactions in which those transactional calls appear. Given correct nonblocking object implementations and transaction aware versions of all primitive types used in the object, the changes required to create a transaction-safe version are local and mechanical.

Section 2 describes transaction-aware types in more detail, and presents, as an example, the AtomicReference type from the Java concurrency library in the context of the Java-based ASTM system [8]. In Section 3 we use this type to build transaction safe versions of Michael and Scott's lock-free queue [10], Harris's lock-free linked list [3], and Michael's lock-free double-ended queue [9]. Experimental results suggest that while transaction safety makes nonblocking data structures somewhat slower, the resulting constructs interoperate smoothly with transactions, and can significantly outperform the natural "fully transactional" alternatives.

Our evaluation also demonstrates some extraneous overhead in the straightforward construction of transaction safe nonblocking objects. In Section 4 we introduce a technique we call *lazy logging* to eliminate a significant portion of the extra overhead. Lazy logging is in a sense complementary to the *early release* heuristic introduced by Herlihy et al. in the DSTM system [5]: it is an application-specific (potentially unsafe) optimization that allows the programmer to avoid the overhead of transaction safety on reads that do not contribute to the serializability of the transaction calling a nonblocking operation. It significantly improves the performance of our example algorithms.

## 2 Transaction Safety

Given a correct nonblocking object, the key to transaction safety is to ensure that memory accesses of methods called from inside a transaction occur (or appear to occur) if, only if, and when the surrounding transaction commits. We do this by making writes manifestly speculative, with their fate tied to that of the transaction, and by logging reads for validation immediately before the transaction commits. Because correct nonblocking code is designed to tolerate races, validation in the middle of nonblocking methods is not required. (We discuss the problem of transaction consistency in greater detail in Section 2.3.)

When called from outside a transaction, methods behave as they did in the original nonblocking code, except that they aggressively abort any transaction that stands in their way. Methods inside a transaction may use a contention management protocol [12, 13] to handle conflicts with transactional peers. They are unaware of nontransactional peers.

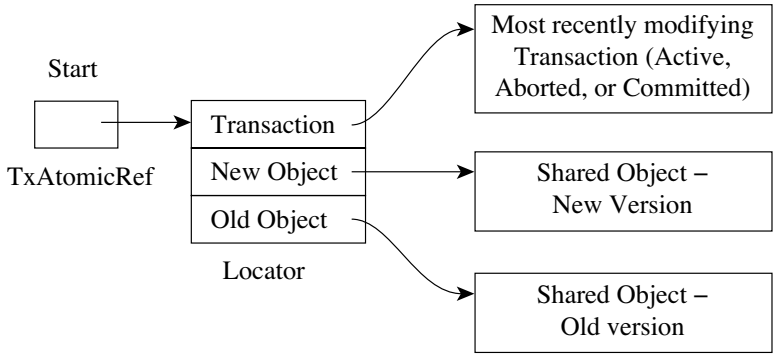


Figure 1: The TxAtomicRef when modified by a transaction.

We assume that the state of nonblocking objects is disjoint from other data. Aggressive aborts in conjunction with commit-time verification of transaction-safe reads effectively ensure that any realizable history is equivalent to one in which nonblocking nontransactional methods are non-overlapping not only with each other and with transactional nonblocking methods, but with the *entire* transactions in which those transactional calls appear.

## 2.1 Transaction Aware Atomic Primitives

Our construction of transaction safe nonblocking objects relies entirely on transaction aware atomic primitives (such as atomic references, integers, longs, etc.) that the nonblocking objects are composed of. To make a primitive type transaction aware, we must be able to distinguish between real and speculative values. For some types (e.g., pointers in C) we may be able to claim an otherwise unused bit to make the distinction. For others (such as integer and long types) we may use a sentinel value to trigger address-based lookup in a separate transactional metadata table. In this paper we use the TxAtomicRef—a transaction aware version of the Java concurrency library class AtomicReference—as a representative example. The data structure examples of Section 3 leverage this primitive to guarantee transaction safety.

In more detail, we provide transaction aware Get, Set, and CAS (compare-and-swap) operations, which the programmer uses instead of conventional operations to access atomic primitives. If called inside a transaction, Get logs the current value of the target type instance for later validation; Set and CAS modify the instance to point to an *indirection object* which in turn contains information about the speculative and non-speculative versions of the instance, along with a reference to the transaction. Changes do not become permanent until transaction commit time. If called inside a transaction, an operation that finds an indirection object invokes the TM system's native contention management system to resolve the implied conflict. If called outside a transaction, all three operations "clean up" any instance that points to an indirection object, aborting the corresponding transaction if it was still active.

As an example, Figure 1 depicts the indirection object (called the *locator* in DSTM [5] terminology) inserted by a transaction in a TxAtomicRef during a Set or CAS operation (this closely resembles the standard implementation of transactional objects in DSTM and ASTM, our implementation platform). The locator enables an operation to determine the most recent owner of the TxAtomicRef and its current consistent version (the new version is consistent if the transaction *committed*, otherwise the old one is). The locator also enables rollback when transactions abort. To update a TxAtomicRef, $A$, a transaction $T$ first allocates a locator, with the old version pointing to $A$'s current version, the new version pointing to the new value, and

```
V Get()
begin
top:
  if (current is Locator)
    if (in transaction)
      if (Active owner)
        call contention manager
          // may cause me to pause or abort
        abort owner;  goto top
      endif
      consistent = getConsistent(current)
      simpleCAS(this, current, consistent)
      goto top
    else
      if (Active owner)
        abort owner;  goto top
      endif
      consistent = getConsistent(current)
      if (simpleCAS(this, current, consistent))
        return consistent
      endif
      goto top
    endif
  else
    if (in transaction)
      log(this, current)
    endif
    return current
  endif
end

V getConsistent(Locator loc)
begin
  if (loc.owner is Committed)
    return loc.new
  else
    return loc.old
  endif
end
```

Figure 2: Pseudocode for transaction aware Get in TxAtomicRef.

the transaction pointer pointing to $T$. $T$ then CASes the locator into $A$. Figure 3 depicts the pseudocode of TxAtomicRef's CAS operation.

Set (also in Figure 3) is much like the CAS with the difference that it always succeeds. Get (Figure 2) ensures that the TxAtomicRef is not being concurrently modified by another transaction, and logs the current version of the TxAtomicRef if invoked by a transaction. In the presence of a conflict between transactions, a contention manager may be invoked to decide which transaction gets to proceed; nontransactional invocations always get to proceed, aborting conflicting transactions if necessary.

## 2.2 Transaction Safe Nonblocking Objects

Given a correct (linearizable) nonblocking object $O$, our construction of a transaction-safe version replaces all instances of primitive types in $O$ with their transaction aware counterparts. We assume that $O$'s data are accessed only by $O$'s methods, and in particular that $O$ is disjoint from all other data accessed by

```
boolean CAS (V expect, V update)
begin
top:
  if (current is Locator)
    if (in a transaction)
      if (Active owner)
        call contention manager
        // may cause me to pause or abort
        abort owner;  goto top
      endif
      consistent = getConsistent(current)
      if (consistent == expect)
        return simpleCAS(this, current,
                         new Locator(transaction,
                             consistent, update))
      endif
      return false
    else
      if (Active owner)
        abort owner;  goto top
      endif
      consistent = getConsistent(current)
      if (consistent == expect)
        return simpleCAS(this, current, update)
      endif
      return false
    endif
  else
    if (in a transaction)
      return simpleCAS(this, expect,
                       new Locator(transaction,
                           expect, update))
    else
      return simpleCAS(expect, update)
    endif
  endif
end

void Set(V update)
begin
  while (CAS(Get(), update) == false) do
    continue
  enddo
end
```

Figure 3: Pseudocode for transaction aware CAS and Set in TxAtomicRef. `current` is the object to which TxAtomicRef currently points. Note that a transactional CAS always inserts a locator in the TxAtomicRef, whereas the nontransactional CAS does the cleanup if necessary.

transactions. We also assume that the TM implementation is correct—i.e., that in the absence of nonblocking objects it ensures the atomicity, isolation, and consistency of transactions. More specifically, we assume that transactions are strictly serializable—that their serialization order is consistent with any dependences among operations that occur outside transactions.

To establish the correctness of our transaction-safe construction, we must argue that (1) methods of $O$, invoked within transactions, linearize in the order in which their surrounding transactions serialize, and (2)

when invoked *outside* transactions, methods of $O$ serialize not only with methods invoked inside transactions, but with the transactions *surrounding* those invocations.

Our construction ensures property (1) directly: reads within a nonblocking method are re-validated at commit time, and writes become effective by virtue of the CAS that commits the surrounding transaction. If $O_1$ and $O_2$ (two distinct versions of $O$), embedded in transactions $T_1$ and $T_2$, conflict with one another, the TM system's contention manager (invoked when the conflict is detected) ensures that at most one of $T_1$ and $T_2$ commits. If $O_1$ and $O_2$ do *not* conflict, and the TM system commits $T_1$ before $T_2$, then by construction $O_1$'s reads and writes take effect before those of $O_2$ as well.

Property (2) is ensured by aborting any transaction that conflicts with a nontransactional nonblocking method. If nonblocking method $O$ overlaps with $T$, one of three cases must hold: (a) $O$ does not conflict with $T$. In this case $O$ may be said to serialize before or after $T$; it does not matter. (b) $O$ conflicts with some primitive operation $P$ contained in the prefix of $T$ that precedes $O$'s linearization point. In this case $O$ aborts $T$, and $T$ never appears in the serialization order. (c) $O$ conflicts with some primitive operation $P$ in the suffix of $T$ that follows $O$'s linearization point, but not with anything in $T$'s prefix. In this case, if $T$ commits, it can safely serialize after $O$.

### 2.3 Transaction Consistency

Since nonblocking methods tolerate transient inconsistencies, execution of such methods by aborted (or to-be-aborted) transactions is guaranteed to complete. However, this robustness does not extend to transactional code outside nonblocking methods.

As an illustration, consider transaction $T$ that invokes a dequeue operation on a transaction safe nonblocking queue $Q$. The dequeue returns object $O$. Thereafter, transaction $T$ does some computation based on the current state of $O$, and attempts to commit. (Note that $T$'s dequeue is speculative until $T$ commits.) Now consider a case where a concurrent nontransactional dequeue happens on $Q$, before $T$ examines $O$'s current state, returning $O$ and dooming $T$ to abort. The thread invoking the nontransactional dequeue may subsequently manipulate $O$ nontransactionally, unaware of the fact that $T$ is about to access $O$. If $T$ has not yet detected that it is doomed, it may observe an inconsistent state of $O$ and perform arbitrary operations.

This problem is essentially the *doomed transaction* half of the *privatization problem* [14, 16]—the nontransactional nonblocking operation privatizes some data that is being accessed speculatively by a concurrent transaction, potentially leading to arbitrary faults in the transaction.

Several solutions of the privatization problem are applicable to our problem. The thread invoking the nontransactional nonblocking operation can be forced by the runtime system to wait (*quiesce*) until the conflicting transaction recognizes (and acknowledges) that it is doomed. With architectural support for *immediate aborts* [15] (where the hardware guarantees that doomed transactions will not take another step), we can avoid the wait and preserve nonblocking progress in the nontransactional thread.

Another solution requires a transaction to *post-validate* every subsequent read that has a data or control dependence on the outcome of a nonblocking method. (Alternatively, the language might disallow such dependences, but this seems quite restrictive.) Post-validation guarantees that a doomed transaction will always access consistent data (or notice it is doomed). In managed languages [1, 4] the runtime system may be able to catch or contain any failures (stray memory references, system calls, infinite loops, etc.) that occur in doomed transactions.

Application specific knowledge may also be used to determine whether a thread must wait after a nontransactional nonblocking operation for conflicting transactions to complete execution. The experiments in Sections 3 and 4 take this approach.

# 3 Examples

In this section we present transaction safe constructions of Michael and Scott's queue [10], Harris's linked list [3], and Michael's lock-free double ended queue (deque for short) [9]. All constructions were straightforward. We conjecture that many more would be equally simple. We measured performance of the transaction safe versions of these algorithms with varying mixes of transactional and nontransactional calls. The results suggest that our construction can provide significant performance benefits relative to equivalent all-transactional code. This is in addition to the qualitative benefit of interoperability with nontransactional code and being able to compose arbitrary operations on nonblocking objects using transactions.

## 3.1 The Algorithms

Michael and Scott's lock-free queue [10] is conceptually simple: The queue is maintained as a linked list, with a dummy node at the head. An enqueue CASes a new node in the tail node's next field, and then CASes the Tail pointer to refer to the new node. Failure in any CAS means that a concurrent operation succeeded, guaranteeing lock-freedom. The dequeue simply CASes the Head pointer to the second node in the list.

Harris's lock-free algorithm for linked lists [3] avoids the race between conflicting inserts and deletes around the same node (which might lead to a lost node) in the list by including a *delete bit* in every node's next pointer. The delete operation first marks the target node for deletion by atomically (via a CAS) setting its delete bit, and then cleans the node out of the list. A concurrent operation that intends to insert a node immediately after the deleted node observes that the node is deleted. During list traversal (to search for target nodes), a thread cleans up any nodes that have been marked for deletion.

Michael's deque [9] gives lock-free progress guarantees with a two-word wide CAS instruction by sacrificing concurrency among operations at the two ends of the deque. (A single-word CAS-based version is also possible, with bounds on the maximum size of the deque.) The data structure consists of a two-word wide Header and a doubly-linked list. The Header consists of a Head and Tail pointer, and a "stability" flag. The flag can have three values: stable, lpush, and rpush. An enqueue Head operation (similarly for Tail) consists of first CASing the Head pointer to the new node (this also switches the Header's flag to lpush mode, indicating that an enqueue at the Head is in progress). The operation then CASes the last Head node's previous pointer to point to the new Head node, followed by another CAS at the Header to set its state back to stable. A dequeue operation is far simpler, requiring just a single CAS on the Header to change the Head or Tail, as the case may be. To guarantee lock-freedom, concurrent operations help an in-progress operation if the Header's flag is not in stable mode.

## 3.2 Implementation

For the lock-free queue, all AtomicReferences (for the Head, Tail, and next nodes) in the original algorithm were replaced with TxAtomicRefs. For the linked list, the next pointer in each node was implemented using an AtomicMarkableReference, which emulates the delete bit in Java using an extra level of indirection. For the lock-free deque, due to Java typing (which precludes using an AtomicLong for the 2-word wide Header), the Header was also implemented using indirection through an AtomicReference. The previous and next pointers in each node were implemented with AtomicReferences. We simply replaced these AtomicReferences with TxAtomicRefs.

Our implementations are based on our prior ASTM system [8]. While slower than recent indirection-free STM systems (all of which require compiler support or sacrifice type safety), ASTM is, to our knowledge, the fastest library-based STM for Java. We believe that performance results would be qualitatively similar for other STMs. ASTM is also obstruction-free, with out-of-band contention management [5, 12, 13] for forward progress in the presence of conflicts. This in turn makes our transaction-safe objects obstruction

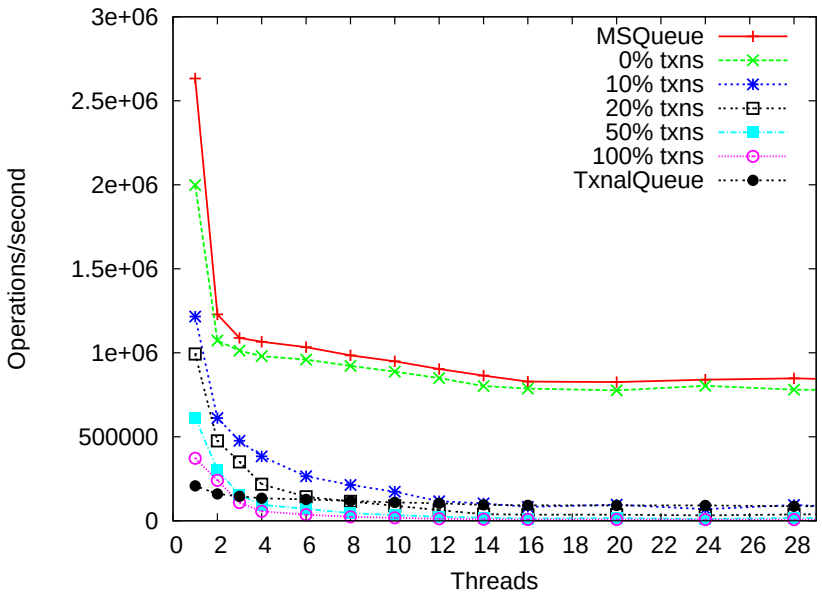


Figure 4: Performance of transaction safe version of Michael and Scott's lock-free queue (MSQueue) with varying percentage of transactional and nontransactional invocations (50% inserts and 50% deletes).

free. (Stronger contention managers [2] may be used to obtain lock-free [or possibly wait-free] progress guarantees.)

### 3.3 Performance Evaluation

Performance results of test runs on the three data structures appear in Figures 4 through 6. Throughput was averaged over 3 test runs. Experiments were conducted on a 16-processor SunFire 6800, a cache-coherent multiprocessor with 1.2GHz UltraSPARC III processors. The testing environment was Sun's Java 1.5 HotSpot JVM. Threading level was varied from 1 to 32. We used the Polite contention manager [13], which basically does finely tuned exponentially backoff before deciding to abort a contending writer.

The top curve in each graph is the standard nonblocking algorithm, running in an entirely nontransactional program. The bottom curve is the natural implementation of an equivalent concurrent object in ASTM. In between are curves for our transaction-safe version of the nonblocking algorithm, with transactional and nontransactional calls accounting for different percentages of the computational mix.

As one might expect, throughput degrades with increasing percentage of transactions. The bookkeeping and validation overhead of transactional accesses contributes to this performance degradation. In the next Section we introduce a *lazy logging* optimization that significantly reduces this overhead with modest (but careful) modifications to each algorithm's source.

Notice that there is a significant drop in scalability for Harris's lock-free linked list when we move from 0% transactions to 10% transactions in test runs. We guarantee isolation by logging all a transaction's TxAtomicRef reads, and validating them at commit time. This leads to some unnecessary aborts, and these turn out to be our primary source of overhead. For example, if a transaction inserts a node after node 200, it does not matter (for correctness) if a concurrent operation deletes node 1 just before the transaction validates its nonblocking readset. However, ASTM's conservative validation strategy will force the transaction to abort. Our lazy bookkeeping optimization also helps to minimize these "spurious" aborts.

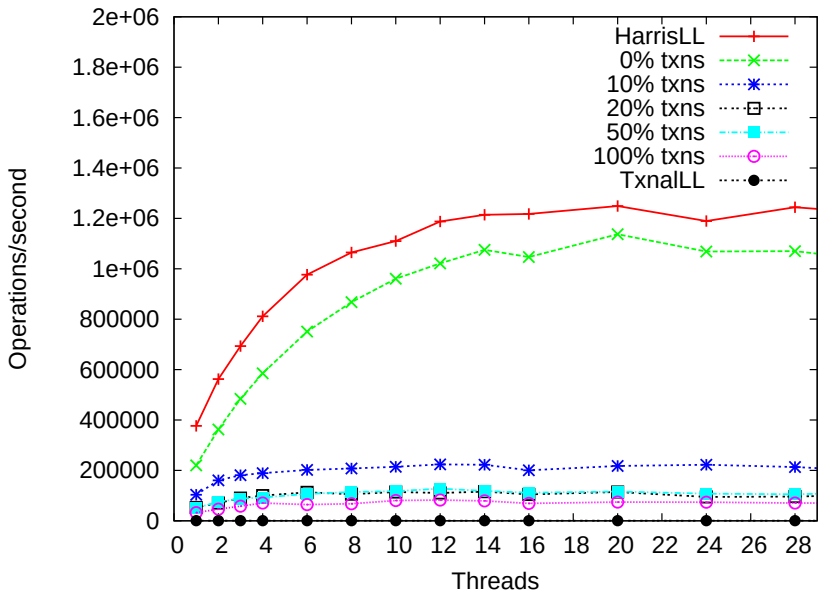


Figure 5: Performance of transaction safe version of Harris's lock-free linked list (LinkedList) with varying percentage of transactional and nontransactional invocations (50% inserts and 50% deletes).

Figure 7 demonstrates the impact of contention on the behavior of transaction safe objects. The bar chart shows the average percentage of time spent by a thread in transactional and nontransactional execution of operations on the MSQueue and LinkedList. The actual percentage of transactions executed in all test runs here is a mere 10%. In both data structures, the amount of time spent in doing transactional work increases significantly with increasing numbers of threads.

In LinkedList, the ratio of time spent between transactional and nontransactional work flips from 1:3 to 3:1 when we increase concurrency from 1 to 16 threads, and remains constant thereafter since we reach the maximum permitted concurrency in the machine. In case of the MSQueue however, the change in ratio is more drastic, flipping from 1:2 to 47:1. This immense increase is the result of the "helping" that happens in the MSQueue to guarantee lock-freedom. This helping further exacerbates contention by causing transactions to bump into even more transactions and nontransactional methods, and subsequently abort. While the MSQueue does not scale with concurrency in any event, it suffers a major drop in performance between 0% and 10% transactions.

Results for the Deque are qualitatively similar to those for the MSQueue, though with a smaller gap between 0% and 10% transactions. Since helping is comparatively rare in LinkedList, we do not see similar degradation there.

# 4 Lazy Logging

As we observed in the previous section, the sources of overhead in our construction of transaction safe nonblocking objects are quite significant, particularly during heavy contention. In the experiments we further observed that most transactions were aborted due to validation failures.

Our next logical step is to reduce such validation failures. We do so by allowing the programmer to avoid bookkeeping, and hence subsequent validation, of certain carefully chosen reads in nonblocking methods contained within transactions. Our intuition here is that not all reads by a transaction in a nonblocking

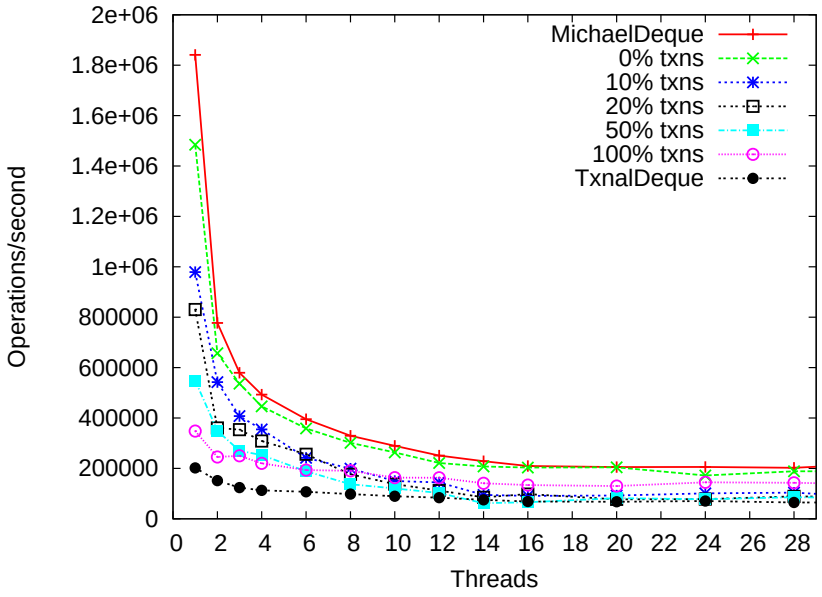


Figure 6: Performance of transaction safe version of Michael's lock-free double-ended queue (Deque) with varying percentage of transactional and nontransactional invocations (50% inserts and 50% deletes).

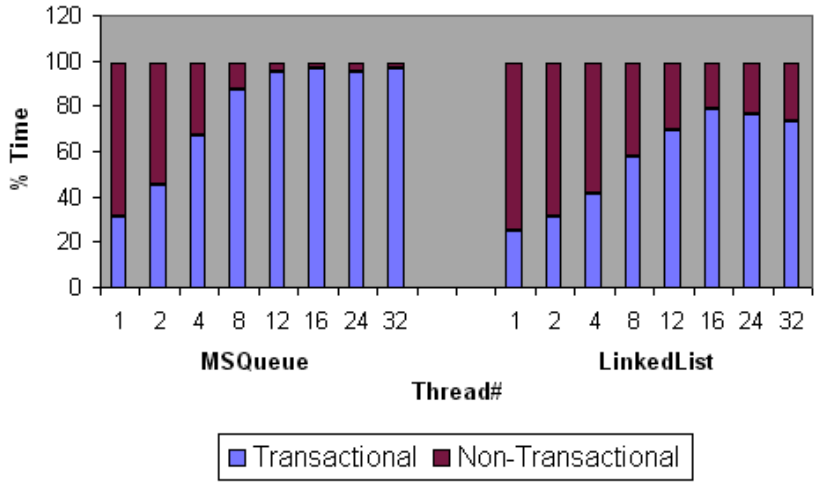


Figure 7: Distribution of time spent in transactional and nontransactional execution of nonblocking operations of MSQueue and LinkedList. The ratio of transactional invocations to nontransactional invocations is 10:90 in all tests.

method need to be logged for later validation to guarantee that transactions serialize with other transactions and with nontransactional nonblocking methods. Only the minimal set of reads necessary for serialization need to be logged for commit time validation. We call these the *critical read set*.

To enable transactional reads *without* bookkeeping in nonblocking methods of our transaction safe constructions, we provide the programmer with a FastGet operation on a transaction aware atomic primitive.

```
V FastGet()
begin
top:
  if (current is Locator)
    if (in transaction)
      if (Active owner)
        call contention manager
        // may cause me to pause or abort
        abort owner; goto top
      endif
      consistent = getConsistent(current)
      simpleCAS(this,
                current, consistent)
      goto top
    else
      if (Active owner)
        abort owner; goto top
      endif
      consistent = getConsistent(current)
      if (simpleCAS(this,
                    current, consistent))
        return consistent
      endif
      goto top
    endif
  else
    return current
  endif
end

void LogRead(V value)
begin
  log(value)
end
```

Figure 8: Pseudocode for a transaction aware FastGet and LogRead methods in TxAtomicRef.

Figure 8 depicts the pseudocode for FastGet. Notice that FastGet differs from Get (Figure 2) only in the read logging part—the former does not have it.

The LogRead method in the transaction aware atomic primitive logs earlier "fast reads" in the transaction's read set. The programmer must explicitly hand-code calls to log critical reads in nonblocking methods. Pseudocode for LogRead appears in Figure 8.

To add the lazy logging optimization, the nonblocking object designer must carefully identify non-critical reads, and points at which such reads become critical and insert the FastGet and LogRead calls in the respective places in the nonblocking methods. For example consider a delete method in Harris's lock-free linked list. The method reads each node in the list (beginning from the head node) and compares the node's key with the target key. Assuming that the list is sorted by key, the search for the target key will stop when a node with a larger or the same key is encountered. For a transactional version of delete to serialize with other methods, it is sufficient to simply log reads of the last node accessed in the search. Logging the last node guarantees that any change to that node by a concurrent nonblocking method will result in failure of the transaction.

This approach is viable because nonblocking algorithms are designed to tolerate inconsistencies at each step in a method. Moreover, the decision of what constitutes the most general critical read set of a trans-

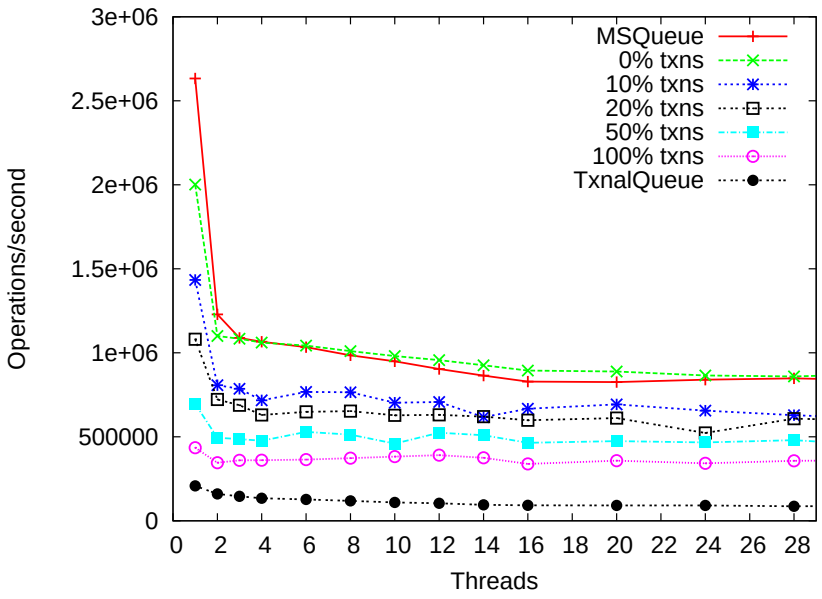


Figure 9: Performance of MSQueue with lazy logging optimization.

actional nonblocking method is completely *local*—the critical read set depends only on the internal implementation of the nonblocking object and not on the context in which the operation is invoked. The LogRead operation simply serves as a hook to enforce serializability of *entire* transactions with respect to other transactions and nontransactional nonblocking methods.

Lazy logging is semantically similar to the *early release* optimization suggested in the DSTM system [5]. Early release enables the programmer to remove logged entries of reads (from a transaction's read set) that are semantically not required for the transaction's serialization. Open nested transactions [11] can also be used in such situations.

The key difference between lazy logging and early release (and likewise, open nesting) is that the transactional bookkeeping of reads, if required, is postponed in lazy logging, whereas the bookkeeping happens immediately for early release (it is "undone" when correctness no longer requires it). A significant semantic side effect of this difference is that early release, if carefully used, guarantees that new object accesses based on the "unreleased" reads are always consistent on these reads; such consistency is not guaranteed with lazy logging. Lazy logging is attractive for transaction safe nonblocking object implementations because such algorithms are inherently tolerant of certain inconsistencies.

Early release is the obvious choice for transactions in the general case. On the other hand, early release requires extra work to first add and then remove an entry from the read set of the transaction for each noncritical read; lazy logging avoids this work. Thus, lazy logging can deliver significantly better performance in operations that have a small critical read set.

## 4.1 Revisiting Data Structures

Incorporating lazy logging in our example nonblocking objects was quite straightforward. An operation in MSQueue and Deque is either an enqueue or a dequeue. The operation always terminates with a successful update to the data structure. (As suggested above, the algorithms are naturally tolerant of intermediate inconsistencies in the logged variables.) This update in turn serves as the "hook" for transactions to se-

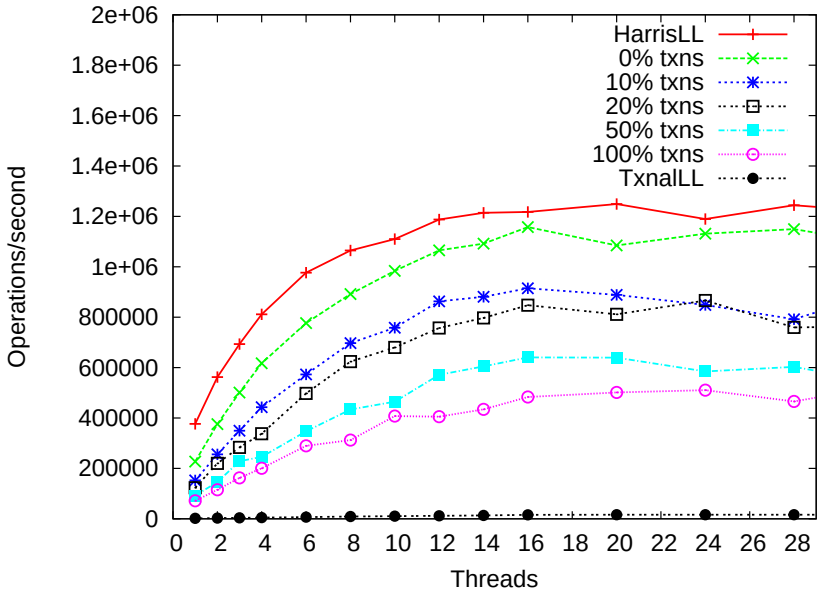


Figure 10: Performance of LinkedList with lazy logging optimization.

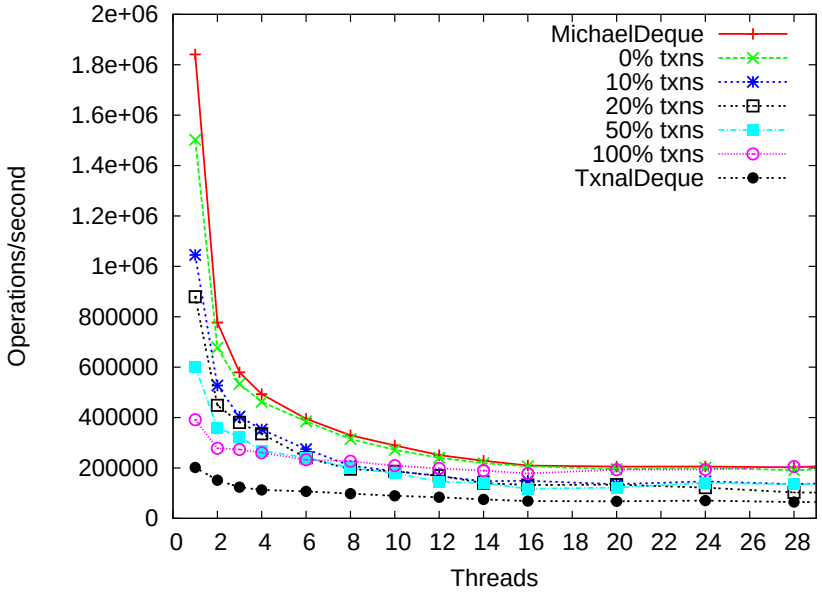


Figure 11: Performance of Deque with lazy logging optimization.

rialize with other transactions and nontransactional operations. As a result, logging of intermediate reads (in transactional invocations of these operations) to guarantee serializability of transactional operations is superfluous, and can be eliminated safely. We consequently replaced all Gets from the original transaction safe implementations of MSQueue and Deque operations with FastGets.

In case of the LinkedList, any of the operations (insert, delete, lookup) may fail, in which case the operation would be read-only most of the time (in the absence of cleanup helping). It is thus necessary to

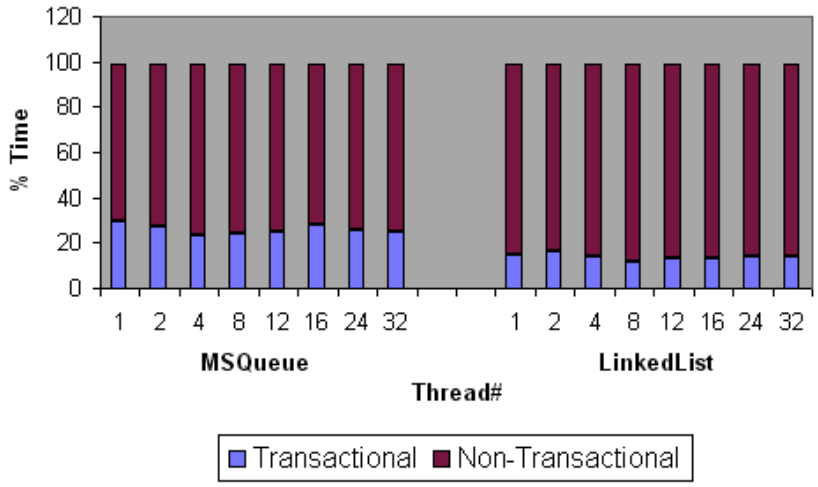


Figure 12: Distribution of time spent in transactional and nontransactional execution of nonblocking operations of MSQueue and LinkedList with the lazy logging optimization. The ratio of transactional invocations to nontransactional invocations is 10:90 in all tests.

log reads of at least those nodes that surround the position where the target node is expected in the list. In all three operations, logging the node just before the target node, plus the target node itself, is sufficient to guarantee that the invoking transaction serializes with other transactions and nontransactional operations.

Figures 9 through 11 demonstrate the effectiveness of lazy logging as compared to the straightforward constructions (Figures 4 through 6). Performance degradation with increasing percentage of transactions is more graceful. Additionally, scaling in LinkedList is far better in the lazy logging version, and degradation due to contention is also significantly mitigated in MSQueue.

Distribution of time spent in transactional and nontransactional executions of operations on the MSQueue and LinkedList appears in Figure 12. The bar chart clearly shows that increasing contention does not adversely affect success rate of transactions with the lazy logging optimization. These examples make a strong case for lazy logging or similar optimizations, particularly for transaction safe nonblocking constructions.

# 5 Conclusion

In this paper we presented a technique to modify arbitrary ad hoc nonblocking objects to permit concurrent transactional and nontransactional accesses. Our construction relies on systematic conversion of the primitive types in nonblocking objects into their transaction aware equivalents. Interoperability permits nonblocking operations to act as fast closed nested transactions, and also allows composability of arbitrary nonblocking operations.

We demonstrated our ideas in the context of the Java-based ASTM system by developing transaction safe versions of several lock-free algorithms. Our empirical analysis led us to a novel optimization called *lazy logging*, which significantly reduces transactional bookkeeping and extraneous contention instances. Performance results are quite encouraging. We conclude that transaction safe nonblocking objects can be a significant enhancement to software transactional memory, particularly for objects that lend themselves to lazy logging.

# References


[1] A.-R. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and Runtime Support for Efficient Software Transactional Memory. In *Proc. of the SIGPLAN 2006 Conf. on Programming Language Design and Implementation*, Ottawa, ON, Canada, June 2006.

[2] R. Guerraoui, M. Herlihy, and B. Pochon. Toward a Theory of Transactional Contention Managers. In *Proc. of the 24th ACM Symp. on Principles of Distributed Computing*, Las Vegas, NV, Aug. 2005.

[3] T. L. Harris. A Pragmatic Implementation of Non-Blocking Linked-Lists. In *Proc. of the 15th Intl. Symp. on Distributed Computing*, pages 300-314, Lisboa, Portugal, Oct. 2001.

[4] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing Memory Transactions. In *Proc. of the SIGPLAN 2006 Conf. on Programming Language Design and Implementation*, Ottawa, ON, Canada, June 2006.

[5] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software Transactional Memory for Dynamic-sized Data Structures. In *Proc. of the 22nd ACM Symp. on Principles of Distributed Computing*, pages 92-101, Boston, MA, July 2003.

[6] M. Herlihy and J. E. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proc. of the 20th Intl. Symp. on Computer Architecture*, pages 289-300, San Diego, CA, May 1993. Expanded version available as CRL 92/07, DEC Cambridge Research Laboratory, Dec. 1992.

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

[8] V. J. Marathe, W. N. Scherer III, and M. L. Scott. Adaptive Software Transactional Memory. In *Proc. of the 19th Intl. Symp. on Distributed Computing*, Cracow, Poland, Sept. 2005.

[9] M. M. Michael. CAS-Based Lock-Free Algorithm for Shared Deques. *Proc. of the 9th European Conf. on Parallel Processing (EURO- PAR)*, 2790:651-660, Springer-Verlag, Aug. 2003.

[10] M. M. Michael and M. L. Scott. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *Proc. of the 15th ACM Symp. on Principles of Distributed Computing*, pages 267-275, Philadelphia, PA, May 1996.

[11] E. Moss and T. Hosking. Nested Transactional Memory: Model and Preliminary Architecture Sketches. In *Proc., Workshop on Synchronization and Concurrency in Object- Oriented Languages*, San Diego, CA, Oct. 2005. In conjunction with OOPSLA '05.

[12] W. N. Scherer III and M. L. Scott. Contention Management in Dynamic Software Transactional Memory. In *Proc. of the ACM PODC Workshop on Concurrency and Synchronization in Java Programs*, St. John's, NL, Canada, July 2004.

[13] W. N. Scherer III and M. L. Scott. Advanced Contention Management for Dynamic Software Transactional Memory. In *Proc. of the 24th ACM Symp. on Principles of Distributed Computing*, Las Vegas, NV, July 2005.

[14] T. Shpeisman, V. Menon, A.-R. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. L. Hudson, K. F. Moore, and B. Saha. Enforcing Isolation and Ordering in STM. In *Proc. of the SIGPLAN 2007 Conf. on Programming Language Design and Implementation*, San Diego, CA, June 2007.

[15] M. F. Spear, A. Shriraman, H. Hossain, S. Dwarkadas, and M. L. Scott. Alert-on-Update: A Communication Aid for Shared Memory Multiprocessors (poster paper). In *Proc. of the 12th ACM Symp. on Principles and Practice of Parallel Programming*, San Jose, CA, Mar. 2007.

[16] M. F. Spear, V. J. Marathe, L. Dalessandro, and M. L. Scott. Privatization Techniques for Software Transactional Memory. TR 915, Dept. of Computer Science, Univ. of Rochester, Feb. 2007.