Software Transactional Memory for Dynamic-Sized Data Structures

Maurice Herlihy
Computer Science Dept.
Brown University
Providence, RI 02912, USA
mph@cs.brown.edu

Victor Luchangco
Sun Microsystems Laboratories
1 Network Drive
Burlington, MA 01803, USA
victor.luchangco@sun.com

Mark Moir
Sun Microsystems Laboratories
1 Network Drive
Burlington, MA 01803, USA
mark.moir@sun.com

William N. Scherer III
Computer Science Dept.
University of Rochester
Rochester, NY 14620, USA
scherer@cs.rochester.edu

ABSTRACT

We propose a new form of software transactional memory (STM) designed to support dynamic-sized data structures, and we describe a novel non-blocking implementation. The non-blocking property we consider is *obstruction-freedom*. Obstruction-freedom is weaker than lock-freedom; as a result, it admits substantially simpler and more efficient implementations. A novel feature of our obstruction-free STM is its use of modular contention managers to guarantee progress in practice.

We illustrate the utility of our dynamic STM with a straightforward implementation of an obstruction-free red-black tree, thereby demonstrating a sophisticated non-blocking dynamic data structure that would be difficult to implement by other means. We also present the results of simple preliminary performance experiments that demonstrate that an "early release" feature of our STM is useful for reducing contention, and that our STM lends itself nicely to the effective use of modular contention managers.

1. INTRODUCTION

Locking in shared-memory multiprocessors has well-known software engineering problems. Coarse-grained locks, which protect relatively large amounts of data, generally do not scale: threads block one another even when they do not really interfere, and the lock becomes a source of contention. Fine-grained locks can reduce these scalability problems, but they introduce software engineering problems as the locking conventions for guaranteeing correctness and avoiding deadlock become complex and error-prone. Locks also cause vulnerability to thread failures and delays. For example, a thread preempted while holding a lock will obstruct other threads.

Dynamic Software Transactional Memory (DSTM) is a low-level application programming interface (API) for synchronizing shared data without using locks. A *transaction* is a sequence of steps executed by a single thread. Transactions are *atomic*: each transaction either *commits* (it takes effect) or *aborts* (its effects are discarded). Transactions are *linearizable* [9]: they appear to take effect in a one-at-a-time order. Transactional memory supports a computational model in which each thread announces the start of a transaction, executes a sequence of operations on shared objects, and then tries to commit the transaction. If the commit succeeds, the transaction's operations take effect; otherwise, they are discarded. Although transactional memory was originally proposed as a hardware architecture [8], there have been several proposals for non-blocking [1] software transactional memory (STM) and similar constructs [3, 4, 10, 13, 14, 15].

We present the first *dynamic* STM implementation. Prior STM designs required both the memory usage and the transactions to be defined statically in advance. In contrast, our new DSTM allows transactions and transactional objects to be created dynamically, and transactions may determine the sequence of objects to access based on the values observed in objects accessed earlier in the same transaction. As a result, DSTM is much better suited to the implementation of dynamic-sized data structures, such as lists and trees.

We have developed prototype implementations of DSTM in the C++ and Java™ programming languages. In this paper, we focus on the Java version, which is considerably simpler because there is no need for explicit memory management. Our Java implementation uses an experimental prototype of Doug Lea's `java.util.concurrent` package [1]

[1] We use "non-blocking" broadly to include all progress conditions requiring that the failure or indefinite delay of a thread cannot prevent other threads from making progress, rather than as a synonym for "lock-free", as some authors prefer.

PODC 2003, July 13–16, 2003, Boston, Massachusetts, USA.

1-58113-661-7/03/0006 ...$5.00.

to call native compare-and-swap (CAS) operations.

Much of the simplicity of our implementation is due to our choice of non-blocking progress condition. A synchronization mechanism is *obstruction-free* [7] if any thread that runs by itself for long enough makes progress (which implies that a thread makes progress if it runs for long enough without encountering a synchronization conflict from a concurrent thread). Like stronger non-blocking progress conditions, such as lock-freedom and wait-freedom, obstruction-freedom ensures that a halted thread cannot prevent other threads from making progress.

Unlike lock-free mechanisms, obstruction-free mechanisms do not rule out *livelock*; interfering concurrent threads may repeatedly prevent one another from making progress. As demonstrated here and elsewhere [7, 11], the obstruction-free property admits substantially simpler implementations that are more efficient in the absence of synchronization conflicts among concurrent threads.

Livelock is, of course, unacceptable. Nevertheless, we believe that there is great benefit to treating the mechanisms that ensure progress as a matter of policy, evaluated by their empirical effectiveness for a given application and execution environment.

Our obstruction-free DSTM implementation lends itself nicely to simple, open-ended mechanisms for prioritizing transactions. In particular, our implementation can detect that one transaction is about to abort another, and can consult a *contention manager* whether it should do so immediately or wait for some time to allow the other transaction a chance to complete. Thus we can design and verify an obstruction-free data structure once, and then "plug in" modular contention management schemes. These schemes can exploit information about time, operating systems services, scheduling, hardware environments, and other practical sources of information that have largely been neglected in the lock-free literature. We believe that this approach will yield simpler and more efficient concurrent data structures, which will help to accelerate their widespread acceptance and deployment.

DSTM provides a simple and effective mechanism for constructing non-blocking implementations of complex concurrent data structures. Section 2 illustrates the use of DSTM through a series of simple examples. To evaluate the utility of DSTM for implementing complex data structures, we have also used it to implement an obstruction-free red-black tree. As far as we are aware, this red-black tree is the most complex non-blocking data structure achieved to date. Our implementation is a reasonably straightforward transformation of a sequential implementation [6], but it would be very difficult to construct such a non-blocking implementation from first principles. Indeed, it would be difficult to implement even a lock-based red-black tree that allows operations accessing different parts of the tree to proceed in parallel.

Section 3 describes how our STM detects synchronization conflicts and how transactions commit and abort, with an emphasis on how the obstruction-free property simplifies the underlying algorithm. In Section 4, we describe how our implementation interfaces with contention managers, which are responsible for ensuring progress. Section 5 describes some simple experiments conducted with our prototype DSTM implementation. Concluding remarks appear in Section 6. Code for our DSTM implementation, contention managers, and related experiments is publically available [2].

2. OVERVIEW AND EXAMPLES

In this section, we illustrate the use of DSTM through a series of simple examples. DSTM manages a collection of *transactional objects*, which are accessed by *transactions*. A transaction is a short-lived, single-threaded computation that either *commits* or *aborts*. A transactional object is a container for a regular Java object. A transaction can access the contained object by *opening* the transactional object, and then reading or modifying the regular object. Changes to objects opened by a transaction are not seen outside the transaction until the transaction commits. If the transaction commits, then these changes take effect; otherwise, they are discarded.

Transactional objects can be created dynamically at any time. The creation and initialization of a transactional object is not performed as part of any transaction.

Concretely, the basic unit of parallel computation is the `TMThread` class, which extends regular Java threads. Like a regular Java thread, it provides a `run()` method that does all the work. In addition, the `TMThread` class provides additional methods for starting, committing or aborting transactions, and for checking on the status of a transaction. Threads can be created and destroyed dynamically.

Transactional objects are implemented by the class `TMObject`. To implement an atomic counter, one would create a new instance of a `Counter` class (not shown), and then create a `TMObject` to hold it:

```
Counter counter   = new Counter(0);
TMObject tmObject = new TMObject(counter);
```

Any class whose objects may be encapsulated within a transactional object must implement the `TMCloneable` interface. This interface requires the object to export a public `clone()` method that returns a new, logically disjoint copy of the object. DSTM uses this method when opening transactional objects, as described below. (DSTM guarantees that the object being cloned does not change during the cloning, so no synchronization is necessary in the `clone()` method.)

A thread calls `beginTransaction()` to start a transaction. Once it is started, a transaction is *active* until it is either committed or aborted.

While the transaction is active, a transaction can access the encapsulated counter by calling `open()`:

```
Counter counter =
  (Counter) tmObject.open(WRITE);
counter.inc(); // increment the counter
```

The argument to `open()` is a constant indicating that the caller may modify the object. The `open()` method returns a *copy* of the encapsulated regular Java object[2] created using that object's `clone()` method; we call this copy the transaction's *version*.

The thread can manipulate its version of an object by calling its methods in the usual way. DSTM guarantees that no other thread can access this version, so there is no need for further synchronization.

Note that a transaction's version is meaningful only during the lifetime of the transaction. References to versions

[2]The `open()` method actually returns an object of class `java.lang.Object`, which we must explicitly cast back to class `Counter`.

should not be stored in other objects; only references to transactional objects are meaningful across transactions.

A thread attempts to commit its transaction by invoking `commitTransaction()`, which returns *true* if and only if the commit is successful. A thread may also abort its transaction by invoking `abortTransaction()`.

Transactions that successfully commit are *linearizable*: they appear to execute in a one-at-a-time order. But what kind of consistency guarantee should we make for a transaction that eventually aborts? One might argue that it does not matter, as the transaction's changes to transactional objects are discarded anyway. However, synchronization conflicts could cause a transaction to observe inconsistencies among objects before it aborts. For example, while a transaction T is executing, another transaction might modify objects that T has already accessed as well as objects that T will subsequently access. In this case, T will see only partial effects of that transaction, which might cause T to have unexpected side-effects, such as dereferencing a null pointer, array bounds violations, and so on.

DSTM addresses this problem by *validating* a transaction whenever it opens a transactional object. Validation consists of checking for synchronization conflicts, that is, whether any object opened by the transaction has since been opened in a conflicting mode by another transaction. If a synchronization conflict has occurred, `open()` throws a `Denied` exception without returning a value, indicating to the transaction that it cannot successfully commit in the future. The set of transactional objects opened before the first such exception is guaranteed to be consistent: `open()` returns the actual states of the objects at some recent instant. (Throwing an exception also allows the thread to avoid wasting effort by continuing the transaction.)

Ultimately, we would like DSTM to support nested transactions, so that a class whose methods use transactions can invoke from within a transaction methods of other classes that also use transactions. However, we have not acquired sufficient programming experience to decide on the appropriate nesting semantics for DSTM, so we do not specify this behavior for now.[3]

2.1 Extended Example

Consider a linked list whose values are stored in increasing order. We will use this list to implement an integer set (class `IntSet`) that provides `insert()`, `delete()`, and `member()` methods. Relevant code excerpts are shown in Figure 1.

The `IntSet` class uses two types of objects: *nodes* and *list elements*; nodes are transactional objects (class `TMObject`) that contain list elements (class `List`), which are regular Java objects. The `List` class has the following fields: `value` is the integer value, and `next` is the `TMObject` containing the next list element. We emphasize that `next` is a `TMObject`, not a list element, because this field must be meaningful across transactions.

The `IntSet` constructor allocates two sentinel nodes, containing list elements holding the minimum and maximum integer values (which we assume are never inserted or deleted). For brevity, we focus on `insert()`. This method takes an integer value; it returns *true* if the insertion takes place, and *false* if the value was already in the set. It first creates a new list element to hold the integer argument, and a new node to hold that list element. It then repeatedly retries the transaction until it succeeds. The transaction traverses the list, maintaining a "current" node and a "previous" node. At the end of the traversal, the current node contains the smallest value greater than or equal to the value being inserted, so the method can detect a duplicate or insert the new node between the previous and current nodes. The transaction then tries to commit. If the commit succeeds, the method returns; otherwise, it resumes the loop.

An attractive feature of DSTM is that we can reason about this code almost as if it were sequential. The principal differences are the need to catch `Denied` exceptions and retry failed transactions, and the need to distinguish between transactional nodes and non-transactional list elements.

```
public class List {
  int value;
  TMObject next;
}

public class IntSet {
  private TMObject first;

  public IntSet() {
    List firstList =
           new List(Integer.MIN_VAL);
    this.first = new TMObject(firstList);
    firstList.next =
     new TMObject(new List(Integer.MAX_VAL));
  }

  public boolean insert(int v)
                      throws TMException {
    List newList = new List(v);
    TMObject newNode = new TMObject(newList);
    TMThread thread =
            (TMThread)Thread.currentThread();
    while (true) {
      thread.beginTransaction();
      boolean result = true;
      try {
        List prevList =
          (List)this.first.open(WRITE);
        List currList =
          (List)prevList.next.open(WRITE);
        while (currList.value < v) {
          prevList = currList;
          currList =
            (List)currList.next.open(WRITE);
        }
        if (currList.value == v) {
          result = false;
        } else {
          result = true;
          newList.next = prevList.next;
          prevList.next = newNode;
        }
      } catch (Denied d){}
      if (thread.commitTransaction())
        return result;
    }
  }
  ...
}
```

Figure 1: Integer Set Example

[3] Our implementation does support a rudimentary form of nested transactions, but we do not use it in any of the examples discussed in this paper.

2.2 Conflict Reduction

A transaction A will typically fail to commit if a concurrent transaction B opens an object already opened by A. Ultimately, it is the responsibility of the contention manager (discussed in Section 4) to ensure that conflicting transactions eventually do not overlap. Even so, the `IntSet` implementation just described introduces a number of unnecessary conflicts. For example, consider a transaction that calls `member()` to test whether a particular value is in the set, running concurrently with a transaction that calls `insert()` to insert a larger value. One transaction will cause the other to abort, since they will conflict opening the first node of the list. Such a conflict is unnecessary, however, because the transaction inserting the value does not modify any of the nodes traversed by the other transaction. Designing the operations to avoid such conflicts reduces the need for contention management, and thereby generally improves performance and scalability.

DSTM provides several mechanisms for eliminating unneeded conflicts. One conventional method is to allow transactions to open nodes in read-only mode, indicating that the transaction will not modify the object. Concurrent transactions that open the same transactional object for reading do not conflict.

```
List list = (List) node.open(READ);
```

The revised `insert()` (not shown) method navigates through the list in read-only mode until it identifies which nodes to modify. It then "upgrades" its access from read-only to regular access by reopening that transactional object in `WRITE` mode. Read-only access is particularly useful for navigating through tree-like data structures where all transactions pass through a common root, but most do not modify the root.

DSTM also provides a novel and more powerful (and more dangerous!) way to reduce conflicts. By invoking the `release()` method, a transaction may *release* an object that it has opened in `READ` mode before it commits. Once an object has been released, other transactions accessing that object do not conflict with the releasing transaction over the released object. The programmer must ensure that subsequent changes by other transactions to released objects will not violate the linearizability of the releasing transaction. (This is similar to our earlier point about the need for validation; releasing objects from a transaction causes future validations of that transaction to ignore the released objects. Therefore, as before, a transaction can observe inconsistent state. The effects in this case are potentially even worse because that transaction can actually commit.)

In our `IntSet` example, releasing nodes is useful for navigating through the list with a minimum of conflicts, as shown in Figure 2. As a transaction traverses the list, opening each node in `READ` mode, it releases every node before its `prev` node.[4] A transaction that adds an element to the list "upgrades" its access to the node to be modified by reopening that node in `WRITE` mode. A transaction that removes an element from the list opens in `WRITE` mode both the node to be modified and the node to be removed. It is easy to check that these steps preserve linearizability.

Because it is often difficult, especially in the face of aliasing, for a transaction to keep track of the objects it has opened, and in what mode each was opened, an object may be opened several times, and in different modes, by a single transaction. Therefore, for each object, DSTM matches invocations of `release()` with invocations of `open(READ)`; an object is not actually released until `release()` has been invoked as many times as `open(READ)` for that object. Objects opened in `WRITE` mode by a transaction cannot be released before the transaction commits; if a transaction opens an object in `READ` mode and then "upgrades" to `WRITE` mode, subsequent requests to release the object are silently ignored.

Clearly, the release facility must be used with care; casual or careless use may violate transaction linearizability. Nevertheless, we have found it very useful for designing shared pointer-based data structures such as lists and trees in which a transaction reads its way through a complex structure.

```
public boolean delete(int v)
                      throws TMException {
  TMThread thread =
          (TMThread)Thread.currentThread();
  while (true) {
    thread.beginTransaction();
    boolean result = true;
    try {
      TMObject lastNode = null;
      TMObject prevNode = this.first;
      List prevList =
        (List)prevNode.open(READ);
      List currList =
        (List)prevList.next.open(READ);
      while (currList.value < v) {
        if (lastNode != null)
          lastNode.release();
        lastNode = prevNode;
        prevNode = prevList.next;
        prevList = currList;
        currList =
          (List)currList.next.open(READ);
      }
      if (currList.value != v) {
        result = false;
      } else {
        result = true;
        prevList = (List)prevNode.open(WRITE);
        prevList.next.open(WRITE);
        prevList.next = currList.next;
      }
    } catch (Denied d){}
    if (thread.commitTransaction())
      return result;
  }
}
```

Figure 2: Delete method with early release

3. IMPLEMENTATION

We now describe our DSTM implementation. A *transaction* object (class `Transaction`) has a `status` field that is initialized to be `ACTIVE`, and is later set to either `COMMITTED` or `ABORTED` by a `CAS` [5] call. (`CAS` functionality is provided by the `AtomicRef` class in the experimental prototype of Doug

[4] This is analogous to the technique of lock coupling (see [5], e.g.), but of course does not use any locks.

[5] A `CAS(a,e,n)` instruction takes three parameters: an address `a`, an expected value `e`, and a new value `n`. If the value currently stored at address `a` matches the expected value `e`, then CAS stores the new value `n` at address `a` and returns *true*; we say that the CAS *succeeds* in this case. Otherwise, CAS returns *false* and does not modify the memory; we say that the CAS *fails* in this case.

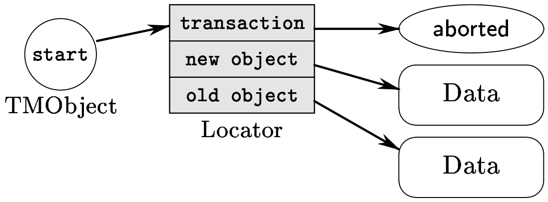

Figure 3: Transactional object structure

Lea's `java.util.concurrent` package [1].)

3.1 Opening a Transactional Object

Recall that a transactional object (class `TMObject`) is a container for a regular Java object, which we call a *version*. Logically, each transactional object has three fields: (1) `transaction` points to the most recent transaction to open the transactional object in `WRITE` mode; (2) `oldObject` points to an *old object version*; and (3) `newObject` points to a *new object version*. The *current* (i.e., most recently committed) version of a transactional object is determined by the status of the transaction that most recently opened the object in `WRITE` mode. If that transaction is committed, then the new object is the current version and the old object is meaningless. If the transaction is aborted, then the old object is the current version and the new object is meaningless. If the transaction is active, then the old object is the current version, and the new object is the active transaction's tentative version. This version will become current if the transaction commits; otherwise, it will be discarded. Observe that, if several transactional objects have most recently been opened in `WRITE` mode by the same active transaction, then changing the status of that transaction from `ACTIVE` to `COMMITTED` atomically changes the current version of each respective object from its old version to its new version; this is the essence of how atomic transactions are achieved in our implementation.

The interesting part of our implementation is how a transaction can safely open a transactional object, without changing its current version (which should occur only when the transaction successfully commits). To achieve this, we need to atomically access the three fields mentioned above. However, current architectures do not generally provide hardware support for such atomic updates. Therefore, we introduce a level of indirection, whereby each `TMObject` has a single reference field `start`, that points to a `Locator` object (Figure 3). The `Locator` object contains the three fields mentioned above: `transaction` points to the transaction that created the `Locator`, and `oldObject` and `newObject` point to the old and new object versions. This indirection allows us to change the three fields atomically by calling `CAS` to swing the `start` pointer from one `Locator` object to another.

We now explain in more detail how transaction A opens a `TMObject` in `WRITE` mode. Let B be the transaction that most recently opened the object in `WRITE` mode. A prepares a new `Locator` object with `transaction` set to A. Suppose B is committed. A sets the new locator's `oldObject` field to the current `newObject`, and the new `newObject` field to a copy of the the current `newObject` (Figure 4). (Recall that every class that can be encapsulated by a transactional object must export a public `clone()` method.) A then calls `CAS` to change the object's `start` field from B's old locator

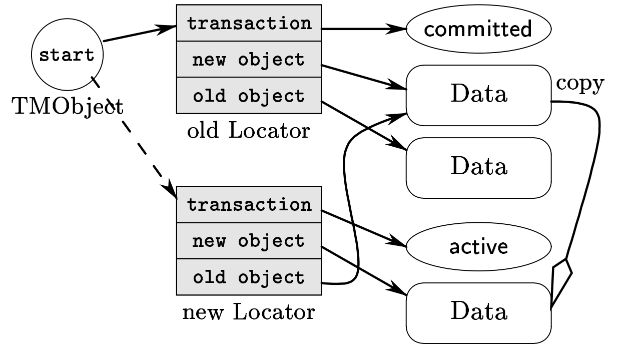

Figure 4: Opening transactional object after recent commit

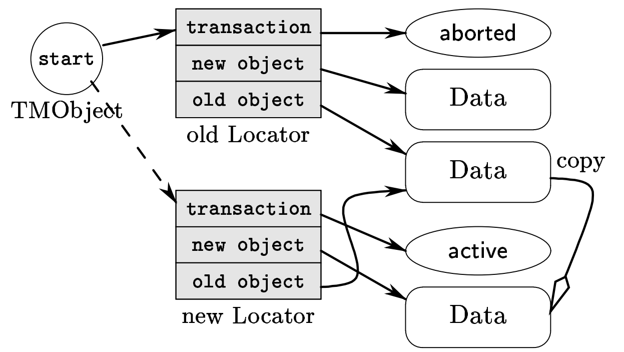

Figure 5: Opening transactional object after recent abort

to A's new locator.[6] If the `CAS` suceeds, the `open()` method returns the new version, which is now the tentative version for this object. A can update that version without further synchronization. If the `CAS` fails, the transaction rereads the object's `start` field and retries. Suppose, instead, that B is aborted. A follows the same procedure, except that it sets the new locator's `oldObject` field to the current `oldObject` (Figure 5).

Finally, suppose B is still active. Because B may commit or abort before A changes the object's `start`, A cannot determine which version is current at the moment its `CAS` succeeds. Thus, A cannot safely choose a version to store in the `oldObject` field of its `Locator`. The beauty of obstruction-freedom is that A does not need to guarantee progress to B, and can therefore resolve this dilemma by attempting to abort B (by using `CAS` to change B's `status` field from `ACTIVE` to `ABORTED`) and ensuring that B's `status` field is either `ABORTED` or `COMMITTED` before proceeding (the change may have been effected by the action of some other transaction). This resolution also highlights an important property of our algorithm with respect to the integration of

[6]Readers familiar with the use of `CAS` may be concerned about the ABA problem [12], in which a `CAS` operation fails to notice that the location it accesses has changed to a new value and then back to the original value, causing the `CAS` to succeed when it should have failed. This problem does not arise in our Java implementation, because garbage collection (GC) ensures that a `Locator` object does not get recycled until no thread has a pointer to it. While GC eliminates the ABA problem in this case, we caution the reader against assuming that the ABA problem can *never* occur in environments that support GC.

contention managers: Because A can determine in advance that it will interfere with B, it can decide, based on the policy implemented by its contention manager (discussed in the next section), whether to abort B or to give B a chance to finish.

Read-only access is implemented in a slightly different way. When A opens a transactional object o for reading, it extracts the last committed version v (possibly by aborting an active transaction) exactly as for write access. Instead of installing a new `Locator` object, however, A adds the pair (o, v) to a thread-local *read-only table.*

To match invocations of `open(READ)` and `release()`, the transaction also maintains a counter for each pair in its read-only table. If an object is opened in `READ` mode when it already has an entry in the table, the transaction increments the corresponding counter instead of inserting a new pair. This counter is decremented by the `release()` method, and the pair is removed when the counter is reduced to zero.

3.2 Validating and Committing a Transaction

After `open()` has determined which version of an object to return, and before it actually returns that version, the DSTM must validate the calling transaction in order to ensure that the user transaction code can never observe an inconsistent state. Validation requires two steps:

1. For each pair (o, v) in the calling thread's read-only table, verify that v is still the most recently committed version of o.

2. Check that the `status` field of the `Transaction` object remains `ACTIVE`.

Committing a transaction requires two steps: validating the entries in the read-only table as described above, and calling `CAS` to attempt to change the status of the `Transaction` field from `ACTIVE` to `COMMITTED`. [7]

3.3 Costs

In the absence of synchronization conflicts, a transaction that opens W objects for writing requires $W + 1$ `CAS` operations: one for each `open()` call, and one to commit the transaction. Synchronization conflicts may require `CAS` calls to abort other transactions. These are the only strong synchronization operations needed by our DSTM implementation: once `open()` returns an object version, there is no need for further synchronization to access that version. A transaction also incurs the cost of cloning objects opened for write; this is achieved using simple load and store instructions.

Validating a transaction that has opened W objects for writing and R objects for reading (that have not been released) requires $O(R)$ work. Because validation must be performed whenever an object is opened and when the transaction commits, the total overhead due to the DSTM implementation for a transaction that opens R for reading and W objects for writing is $O((R + W)R)$ plus the cost of copying each of the W objects opened for writing once. Note that, in addition to reducing the potential for conflict, releasing objects opened for reading also reduces the overhead due to validation: released objects do not need to be validated. Thus, if at most K objects are open for reading at any time, then the total overhead for a transaction is only $O((R + W)K)$ plus the cost of cloning the W objects.

4. CONTENTION MANAGEMENT

Our advocacy of obstruction-free synchronization does *not* mean that we expect progress to take care of itself. On the contrary, we have found that explicit measures are often necessary to avoid cyclic restart and starvation. Obstruction-free synchronization encourages a clean modular distinction between the obstruction-free mechanisms that ensure correctness (such as conflict detection and recovery) and additional mechanisms that ensure progress (such as adaptive backoff or queuing).

In our transactional memory implementation, progress is the responsibility of the *contention manager*. Each thread has its own contention manager instance, which it consults to decide whether to force a conflicting thread to abort. In addition, contention managers of different threads may consult one another to compare priorities and other attributes.

The correctness requirement for contention managers is simple and quite weak. Informally, any active transaction that asks sufficiently many times must eventually get permission to abort a conflicting transaction. More precisely, every call to a contention manager method eventually returns (unless the invoking thread stops taking steps for some reason), and in any infinite sequence of requests by a single transaction to abort another transaction, such permission is granted infinitely often. This requirement is needed to preserve the obstruction-free property: A transaction A that is forever denied permission to abort a conflicting transaction will never commit even if it runs by itself.[8] If the conflicting transaction is also continually requesting permission to abort A, and incorrectly being denied this permission, the situation is akin to deadlock. Conversely, if A is eventually allowed to abort any conflicting transaction, then A will eventually commit if it runs by itself for long enough.

The correctness requirement for contention managers does *not* guarantee progress in the presence of conflicts. Whether a particular contention manager should provide such guarantees — and under what assumptions and system models it should do so — is a policy question that may depend on applications, environments, and other factors. The problem of avoiding livelock is thus delegated to the contention manager.

Rather than mandate a specific contention-reduction policy, our DSTM implementation defines a `ContentionManager` interface that every contention manager must implement. This interface specifies two kinds of methods, *notification* methods and *feedback* methods. Notification methods simply inform a contention manager of relevant events in the DSTM and do not return any value. For example, the `commitTransactionSucceeded` method is invoked by the DTSM whenever a transaction commits successfuly, and the `commitTransactionFailed` method is invoked whenever an attempt to commit a transaction fails. The `openReadAttempt`

[7] A further optional step can reduce space overhead by storing null to whichever of the object pointers in a locator becomes obsolete after its transaction either commits or aborts, thereby allowing the garbage collector to claim it. (Straightforward changes would be required in order to avoid dereferencing these null pointers.)

[8] Here and elsewhere "runs by itself" means that no concurrent transaction takes a step, *not* that no concurrent transaction exists.

method is called to notify a contention manager before any attempt to open for reading an object that is not already open; similarly, the `openWriteAttempt` method is called before any attempt to open an object for writing.

Feedback methods are called by the DSTM to determine what action should be taken in various circumstances. One important feedback method is `shouldAbort`, which the DSTM invokes when it detects a conflicting transaction during an attempt to open an object. The `shouldAbort` method is passed the object being opened and the manager of the conflicting transaction, and it returns a boolean indicating whether to try to abort the conflicting transaction.

In addition to their explicit purposes, the contention manager's methods may implement other measures to reduce contention, for example, by backoff or queuing. We have done only preliminary work using these methods to implement some simple contention management strategies, and we expect the `ContentionManager` interface to evolve as we gain more experience with what methods—especially notification methods—are useful for implementing more sophisticated strategies.

4.1 Examples

As a baseline for the experimental results reported in Section 5, we implemented a trivial `Aggressive` contention manager that always and immediately grants permission to abort any conflicting transaction. We also implemented a simple `Polite` contention manager, that adapatively backs off a few times when it encounters a conflict. Specifically, when a transaction first invokes `shouldAbort` for an object, the method sleeps for a random duration before returning *false*, refusing permission to abort the other thread. Each subsequent call to `shouldAbort` for the same object doubles the expected sleep time, until a threshold is reached. Beyond that threshold, `shouldAbort` returns immediately and returns *true*, granting the caller permission to abort the conflicting transaction.

One can imagine many variations on this strategy, as well as different strategies based on queueing rather than backoff combined with spinning. Discovering which strategies work best remains an open area of research.

5. RESULTS

In this section, we briefly present the results of some simple performance experiments we conducted on a Sun Fire™ 15K server with 72 1050MHz SPARC® processors.

In each experiment, we implemented an integer set and measured how many operations completed on the integer set in 20 seconds, varying the number of participating threads between 1 and 576 (i.e., a multiprogramming level of 8). For each operation, we randomly choose a value between 0 and 255 and randomly choose whether to insert or delete the value. The restricted range ensures significant contention among concurrent threads, and thus exercises the contention managers. In each experiment, each thread executes operations repeatedly with no delay between them in order to examine how the implementations scale with increased contention.

The results of our experiments are presented in Figure 6. The graphs show results as throughput in operations per millisecond. Each point represents the average of at least ten runs of the relevant experiment. The upper graph shows the results for the various experiments, running from 1 to 576 threads. The lower graph presents a more detailed look at the experiments in which the number of threads does not exceed the number of processors (72). Of course, many more experiments can and will be conducted to test various implementation approaches at the transaction, contention manager, and STM levels. The simple experiments presented here are intended only to demonstrate some broad principles.

We first implemented a simple linked list synchronized with a single lock (see the "Simple Locking" line in Figure 6). Due to its simplicity, this implementation yielded a higher throughput than any other configuration in the single-threaded case (736 operations per millisecond). However, as the number of threads increases, the throughput of this implementation quickly falls off; in particular, when there are more threads than processors, this implementation performs very badly due to preemption while holding the lock.

There are also specialized optimistic locking algorithms that exploit the simple semantics of linked lists to substantially improve performance. However, these algorithms involve unsynchronized reads of shared data, and thus require careful reasoning about concurrency to ensure correctness and avoid deadlock. Furthermore, they do not generalize straightforwardly to more complex data structures. Because our purpose here is to illustrate the implications of different implementation approaches, not to construct the best implementation of integer sets, we do not consider such algorithms in this paper.

Next, we used DSTM to implement the simple transactional integer set shown in Figure 1, and composed it with the trivial `Aggressive` contention manager (IntSetSimple/Aggressive in Figure 6). This configuration immediately livelocks as soon as there is more than one thread. However, when we compose the same implementation with the slightly more sophisticated `Polite` contention manager, it performs much better. In fact, it outperforms the simple lock based implementation with more than about 20 threads. These results demonstrate the necessity and effectiveness of contention management.

Although we can *manage* contention, it is often preferable to simply *avoid* contention, as discussed in Section 2. We therefore also tested the linked list implementation with early release shown in Figure 2. As seen in Figure 6, this implementation does not livelock even when used with the `Aggressive` contention manager, which demonstrates that this programming technique is an effective way of reducing contention. Also, because this implementation gives rise to less contention, the effect of contention management is less pronounced. This is because the number of objects opened by the transaction at any time is much smaller in this implementation.

In the context of sequential algorithms, it is standard practice to design more complex algorithms that outperform simpler ones (for example, by implementing a balanced tree instead of a list). For non-blocking algorithms, however, implementing more complex data structures has been prohibitively difficult. Our work on DSTM makes a significant step towards overcoming this problem. To demonstrate, we have used DSTM to implement a non-blocking red-black tree using a straightforward translation from sequential code [6].

As can be seen from Figure 6, our red-black tree significantly outperforms the other non-blocking implementations

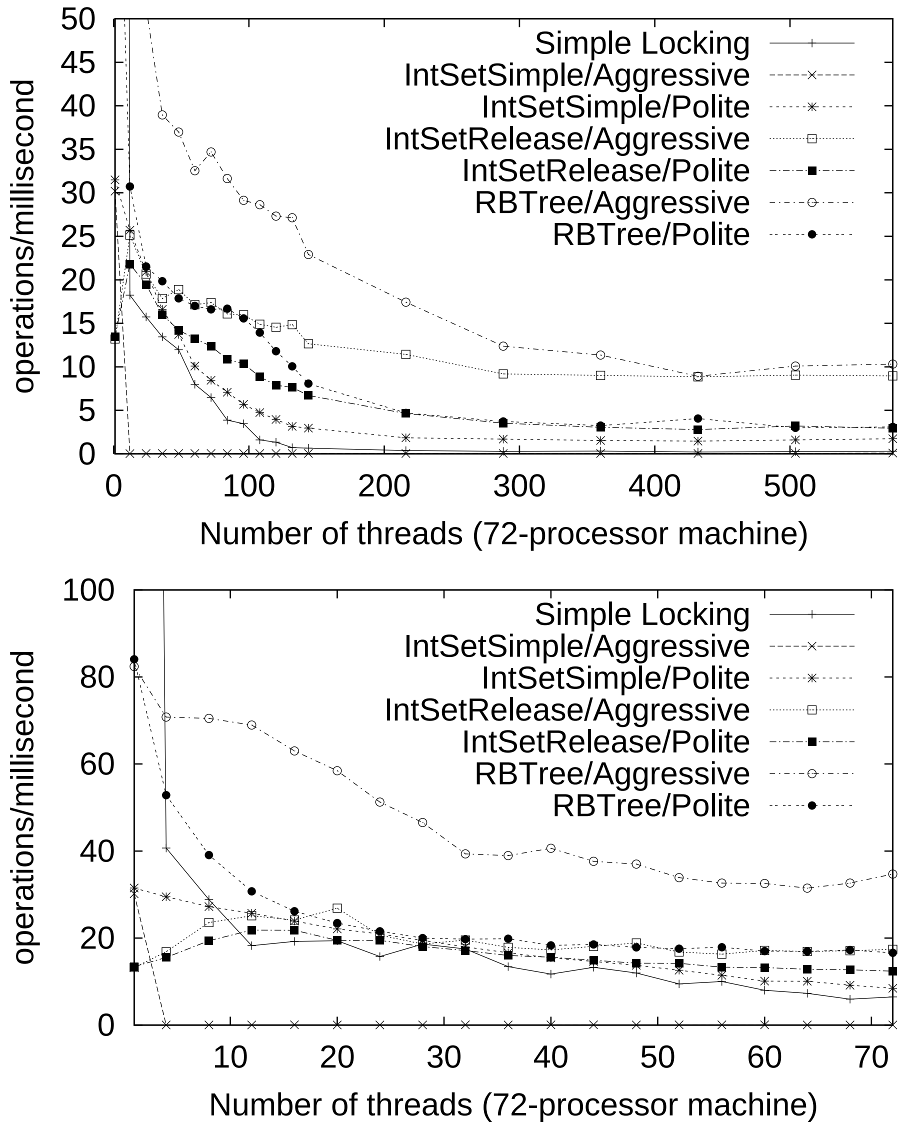

Figure 6: Experimental results on 72-processor Sun Fire 15K server. The bottom figure shows a detailed view of the top one for up to 72 threads.

at low levels of contention. This is because its time complexity is logarithmic in the size of the set, in contrast to the linear time complexity of the list. Note that this effect would be even more pronounced if we chose values to insert from a larger range, which would result in larger sets in steady state.

Even with this limited value range, the red-black tree (using the `Polite` contention manager) remains competitive with all of the other configurations shown while we have at most one thread per processor, and is significantly better than most of them. However, there is a marked degradation in its performance with increasing numbers of threads, and it does not perform as well as the other configurations when we have more threads than processors. We believe that we can improve on this in two ways. First, by judicious use of the early release mechanism, it should be possible to reduce the size of transactions and thereby reduce contention, just as we did with the list-based set. Furthermore, we believe that the more complex nature of the red-black tree algorithm requires more sophisticated contention management. We have already started work in both of these directions.

One shortcoming of our current DSTM implementation is that there is no way for one transaction to detect that it is about to abort another transaction via an object that the second transaction has opened in `READ` mode. Clearly there is a tradeoff between the amount of synchronization needed to open an object in one of these modes in a "visible" way in order to allow competing transactions to "be polite" and the benefit derived from doing so. We are currently working on some ideas in this direction.

6. CONCLUDING REMARKS

We have proposed a new form of dynamic software transactional memory (DSTM), which supports relatively straightforward programming of a wide variety of dynamic-sized data structures. For example, we have used it to implement a non-blocking red-black tree, by far the most sophisticated non-blocking data structure achieved to date. We have implemented an obstruction-free prototype of our DTSM in the Java programming language. Obstruction-freedom is a new non-blocking progress condition we proposed recently; it is weaker than previous such conditions, and as a result, admits substantially simpler implementations.

An attractive feature of our implementation is the ability for a transaction to detect that it will cause another to abort before it does so, and therefore decide whether or not to proceed or to give the other a transaction to complete first. Such policy decisions are made by modular contention managers that can be "plugged in" without affecting the transaction code or its correctness. Preliminary performance experiments show that non-trivial contention management schemes are necessary in order to avoid livelock, and that quite simple schemes can be effective.

We have only begun to explore the range of possible contention manager designs. Ultimately, we think that designing, testing, and reasoning about modular contention managers will be a rich source of research problems. It is interesting to note that it is possible to design contention managers that make provable progress guarantees in the presence of certain weak but reasonable assumptions about the underlying system (and certain reasonableness assumptions about transaction code). Whether such managers are practical is a matter for future research.

Another interesting and novel feature of our DSTM is the ability to "release" objects from a transaction before it commits. This feature puts significantly more burden on the transaction programmer in reasoning about correctness, but can also provide considerable performance improvements when used with care.

There also remain a number of interesting issues regarding interface and semantics. In many cases, there are tradeoffs between efficiency of implementation and usability and simplicity of interface; we have yet to explore these tradeoffs in detail.

Acknowledgments: Thanks to Ron Larson for getting us access to the Sun Fire 15K computer, to Doug Lea for his experimental `java.util.concurrent` package, and to Steve Heller for useful discussions. Thanks also to Guy Steele and Jan-Willem Maessen for useful feedback, especially about the DSTM interface. (Jan also suggested nulling out the extra pointer of locators whose transactions are aborted or commited to allow garbage collection of the obsolete version.) We are also grateful to Yossi Lev for feedback and useful suggestions for future improvements.

7. REFERENCES

[1] *Java Specification Request for Concurrent Utilities (JSR166).* `http://jcp.org`.

[2] *Sun Microsystems Laboratories Scalable Synchronization Research Group publications page.* `http://research.sun.com/scalable/pubs`.

[3] Y. Afek, D. Dauber, and D. Touitou. Wait-free made fast. In *Proceedings of the 27th Annual ACM Symposium on Theory of Computing*, pages 538–547, 1995.

[4] G. Barnes. A method for implementing lock-free shared data structures. In *Proceedings of the Fifth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 261–270, 1993.

[5] R. Bayer and M. Schkolnick. Concurrency of operations on B-trees. *Acta Informatica*, 9:1–21, 1977.

[6] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* McGraw-Hill, 1990.

[7] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, 2003.

[8] M. Herlihy and J. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th International Symposium in Computer Architecture*, pages 289–300, 1993.

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

[10] A. Israeli and L. Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*, pages 151–160, 1994.

[11] V. Luchangco, M. Moir, and N. Shavit. Nonblocking k-compare-single-swap. In *Proceedings of the 15th Annual ACM Sympoium on Parallel Architecures and Algorithms*, 2003.

[12] M. Michael and M. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *Journal of Parallel and Distributed Computing*, 51(1):1–26, 1998.

[13] M. Moir. Transparent support for wait-free transactions. In *Proceedings of the 11th International Workshop on Distributed Algorithms*, pages 305–319, 1997.

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

[15] J. Turek, D. Shasha, and S. Prakash. Locking without blocking: making lock based concurrent data structure algorithms nonblocking. In *Proceedings of the eleventh ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 212–222, 1992.