A Flexible Framework for Implementing Software Transactional Memory

Maurice Herlihy
Brown University
Providence, RI
mph@cs.brown.edu

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

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

ABSTRACT

We describe DSTM2, a Java$^{\text{TM}}$ software library that provides a flexible framework for implementing object-based software transactional memory (STM). The library uses *transactional factories* to transform sequential (unsynchronized) classes into atomic (transactionally synchronized) ones, providing a substantial improvement over the awkward programming interface of our previous DSTM library. Furthermore, researchers can experiment with alternative STM mechanisms by providing their own factories. We demonstrate this flexibility by presenting two factories: one that uses essentially the same mechanisms as DSTM (with some enhancements), and another that uses a completely different approach.

Because DSTM2 is packaged as a Java library, a wide range of programmers can easily try it out, and the community can begin to gain experience with transactional programming. Furthermore, researchers will be able to use the body of transactional programs that arises from this community experience to test and evaluate different STM mechanisms, simply by supplying new transactional factories. We believe that this flexible approach will help to build consensus about the best ways to implement transactions, and will avoid the premature "lock-in" that may arise if STM mechanisms are baked into compilers before such experimentation is done.

1. INTRODUCTION

The major chip manufacturers have, for the time being, given up trying to make processors run faster. Moore's law has not been repealed: each year, more and more transistors fit into the same space, but their clock speed cannot be increased without overheating. Instead, attention is quickly turning toward *chip multiprocessing* (CMP), in which multiple computing cores are included on each processor chip. These trends mean that, in the medium term, advances in technology will provide increased parallelism, but not increased single-thread performance. As a result, system designers and software engineers can no longer rely on increasing clock speed to hide software bloat. Instead, they must learn to make effective use of increasing parallelism.

This adaptation will not be easy. In today's programming practices, programmers typically rely on combinations of locks and conditions, such as monitors, to prevent concurrent access by different threads to the same shared data. While this approach allows programmers to treat sections of code as "atomic", and thus simplifies reasoning about interactions, it suffers from a number of severe shortcomings.

First, programmers must decide between *coarse-grained* locking, in which a large data structure is protected by a single lock, and *fine-grained* locking, in which a lock is associated with each component of the data structure. Coarse-grained locking is simple, but permits little or no concurrency, thereby preventing the program from exploiting multiple processing cores. By contrast, fine-grained locking is substantially more complicated because of the need to ensure that threads acquire all necessary locks (and only those, for good performance), and because of the need to avoid deadlock when acquiring multiple locks. The decision is further complicated by the fact that the best engineering solution may be platform-dependent, varying with different machine sizes, workloads, and so on, making it difficult to write code that is both scalable and portable.

Second, conventional locking provides poor support for code composition and reuse. For example, consider a lock-based hash table that provides atomic `insert` and `delete` methods. Ideally, it should be easy to move an element atomically from one table to another, but this kind of composition simply does not work. If the table methods synchronize internally, then there is no way to acquire and hold both locks simultaneously. If the tables export their locks, then modularity and safety are compromised.

Finally, such basic issues as the mapping from locks to data, that is, which locks protect which data, and the order in which locks must be acquired and released, are all based on convention, and violations are notoriously difficult to detect and debug. For these and other reasons, today's software practices make concurrent programs too difficult to develop, debug, understand, and maintain.

In reaction to these problems, the *transactional* model of synchronization has received attention as an alternative programming model. In transactional programming, code that accesses shared memory is divided into *transactions*, which are intended to be executed atomically: operations of two different transactions should not appear to be interleaved. A transaction may *commit*, in which case all its operations appear to take place atomically, or *abort*, in which case its operations appear not to have taken place at all. If two transactions conflict, that is, they access the same objects, and at least one of them writes it, then one must wait until the other either commits or aborts. In a transaction aborts, it is typically retried until it commits, usually after taking some measures to reduce contention and avoid further conflicts. This approach has been investigated in hardware [1, 6, 12, 19, 20], in software [7, 8, 11, 14, 15, 17, 25], and in schemes that mix hardware and software [18, 23]. Here we focus on software schemes.

As an application programming interface (API), software transactional memory (STM) promises to alleviate the difficulty of programming using conventional methods such as locks and condition variables. One of the principal challenges in understanding how to make STM effective is how to design and evaluate effective run-time mechanisms for transactional synchronization, including how to establish and restore checkpoints, how to detect synchronization conflicts, and how to guarantee progress. There is no shortage of potential mechanisms, but much remains to be learned about how such mechanisms interact, and especially how they are affected by characteristics of applications. The community needs more experience with applications written in a transactional style before we can reach consensus on the best ways to support this API.

This paper describes the *DSTM2* software transactional memory library, a collection of Java^{TM} packages that supports a transactional API. The goal of this project is to provide a simple, self-contained open-source system that can be downloaded and used by researchers who would like to implement transactional applications, and to experiment with alternative run-time implementations of key transactional mechanisms[1]. As the name suggests, DSTM2 is our second software transactional memory package. An earlier design, DSTM [11], focused on the basic model of computation and on run-time techniques, with little attention to providing a safe, convenient, and flexible API for application programmers. DSTM2 incorporates the lessons we learned from DSTM.

Designing a software transactional memory library is more complex than designing conventional libraries. The effect of transactional synchronization is pervasive: the need to synchronize access to shared objects affects how classes are defined, and the need to commit and abort transactions affects normal control flow. An STM library for a standard language must do more than simply provide a collection of useful objects and algorithms. The challenge is also to provide a nearly-seamless view of transaction synchronization and recovery, with a minimum of pitfalls.

A central principle in the design of DSTM2 is that under current circumstances, it is premature to commit to specific implementation techniques. While existing STM systems provide roughly similar functionality, there is little or no consensus on the best ways to implement key runtime features such as conflict detection and resolution, or transaction synchronization and recovery. A key aspect of DSTM2 is that it provides users the ability to "plug in" their own synchronization and recovery mechanisms in the form of *transactional factories* that transform stylized sequential interfaces into transactionally synchronized data structures. Like its predecessor, DSTM2 also allows users to customize their techniques for managing contention.

This modular decomposition is intended to make it as easy as possible to modify or replace those modules most likely to be the targets of experiment. It allows users to customize the underlying run-time executive to match the characteristics of the application, a valuable property if, as experience suggests, no single run-time mechanism is ideal for all kinds of applications.

We focus on a Java library, instead of a compiler, because a library is easier to distribute and provides greater freedom to experiment. (For examples of language-based STM systems, see Harris et al. [8] or Bratin et al. [24]). Eventually, if the transactional API wins wide acceptance, we can expect language and compiler support for the model. A language is safer and easier to use than a library, and a compiler can perform many non-local optimizations that would be difficult or impossible in a library. Nevertheless, a library for a standard, widely-used language has the advantages that it can be used right now, without installing proprietary software or waiting for language design committees, and that it can be integrated more easily into other research projects.

The remainder of the paper is organized as follows. In Section 2, we give an overview of the factory approach to implementing STM, and present the DSTM2 API through a simple example,. In Section 3, we describe the factory approach in more detail, and present a transactional factory that uses essentially the same mechanisms as the original DSTM, as well as an alternative transactional factory that demonstrates the flexibility of our approach. We present the results of preliminary performance experiments in Section 4. We discuss related work in Section 5 and conclude in Section 6.

2. THE DSTM2 LIBRARY

The DSTM2 library assumes that multiple concurrent *threads* share data *objects*. The DSTM2 library provides a new kind of thread that can execute *transactions*, which access shared *atomic objects*. DSTM2 threads provide methods for creat-

[1]The DSTM2 source will be available for download soon, well before OOPSLA.

ing new atomic classes and executing transactions.

Perhaps the most novel aspect of DSTM2 is the way atomic classes are defined. First, the programmer defines a stylized sequential *interface*, a named collection of method signatures satisfying certain simple consistency conditions. Second, the programmer (or the DSTM2 library) provides a *transactional factory*, which uses a combination of reflection and run-time class synthesis to create an anonymous class implementing the original interface, whose methods are transactionally synchronized. This transactional factory returns an *atomic-object factory*, which creates instances of this anonymous class. As a result, the choice of how to implement atomic objects is a run-time decision, which is intended to facilitate experimentation with different synchronization and recovery techniques (for example, nonblocking versus blocking, undo versus redo).

It is worth emphasizing that reflection and run-time class synthesis occurs once per atomic class (that is, when its factory is first created), and is not invoked each time an atomic object is created using that factory.

DSTM2 provides direct support for simple atomic classes that provide transactional `get` and `set` methods for "virtual fields" (as well as optional "escape" methods described below). As illustrated in Section 4, it is straightforward to implement more complex data structures, such as lists or skip-lists, on top of such simple atomic types.

DSTM2 also provides the ability to register methods (in the form of `Callable<Boolean>` objects) to be called when important transactional events occur. For example, one can register a method that can veto a transaction's attempt to commit (the so-called *validation* step). One can also register methods to run immediately after a transaction commits or aborts (useful for cleaning up data structures). We have found this service, which as far as we know is not supported by any other STM library, to be essential to giving factories the flexibility needed to implement a range of strategies. As explained later, it is also key to supporting certain built-in transactional libraries, such as transactional arrays.

Finally, DSTM2 also provides support for user-defined contention managers. This was a focus of our previous version of DSTM [11], and also of work by other researchers (e.g., [5, 13]), so we do not focus on it in this paper.

The initial release of DSTM2 does not support certain advanced features such as nested transactions [21] or conditional waiting [8, 9]. We think these features are important in the long run, and expect to support them in the near future.

The DSTM2 library requires the ability to construct classes at run-time. We currently use a combination of reflection, class loaders, and the *byte-code engineering library* (BCEL) [3]), a rich collection of packages for synthesizing code and classes at run-time under program control. This approach may not be well-suited to languages such as C or C++ that lack support for reflection or run-time code generation.

In the rest of this section, we introduce the basic features by example, and discuss some of the design decisions we made based on our experience with the original DSTM library.

```
@atomic public interface INode {
  int getValue();
  void setValue(int value);
  INode getNext();
  void setNext(INode value);
}
```

Figure 1: Interface for an atomic class

2.1 The DSTM2 API

Perhaps the easiest way to introduce the DSTM2 API is by example. We will walk through the construction of a simple linked-list benchmark, in which transactions add, remove, and look up entries in a sorted list of integers.

The first step is to define the class implementing list nodes. Because list nodes will be read and modified by concurrent transactions, they must be explicitly declared to be atomic by providing an *interface* for the object to satisfy. Figure 1 shows the interface we would use for a list node. The interface declaration is prefaced by the `@atomic` attribute, which is an explicit declaration that objects satisfying this interface should be safe to share among concurrent transactions.

The atomic interface defines one or more *properties*, where a property of type `T` is a pair of matching method signatures:

```
T getField{};
void setField(T value);
```

Think of a property as defining a *virtual field*, in this case, named `field` of type `T`. The interface in Figure 1 defines two properties: `value` of type `int`, and `next` of type `INode`. As mentioned earlier, and for reasons explained later, each property must have a type that is either scalar or an interface satisfying the `@atomic` attribute. (In addition to defining a set of properties, an `@atomic` interface may define certain other specialized methods, which we discuss later.)

The atomic interface is passed to a transactional factory, which returns an atomic-object factory capable of creating list nodes. The DSTM2 library provides several transactional factories, and researchers are encouraged to write their own to investigate alternative STM implementations. Each DSTM2 thread maintains a transactional factory (which can be, but typically is not, changed at run-time), which can be applied to the `INode` interface as follows:

```
Factory<INode> factory =
                    dstm2.Thread.makeFactory(INode.class);
```

The factory itself provides a single method:

```
INode node = factory.create();
```

This method creates an object of an anonymous class implementing the `INode` interface. The object is then initialized by calling the properties' `set` methods.

```
public class List extends IntSetBenchmark {

  static Factory<INode> factory =
      Thread.makeFactory(INode.class);

  public T insert(T v) {
    INode newNode = factory.create();
    newNode.setValue(v);
    INode prevNode = this.root;
    INode currNode = prevNode.getNext();
    while (currNode.getValue() < v) {
      prevNode = currNode;
      currNode = prevNode.getNext();
    }
    if (currNode.getValue() == v) {
      return false;
    }
    else {
      newNode.setNext(prevNode.getNext());
      prevNode.setNext(newNode);
      return true;
    }
  }
  ...
}
```

Figure 2: A transaction

Transactions are written as methods. Figure 2 shows an example transaction that inserts a value in a list. This example illustrates an important property of DSTM2: a transaction is almost identical to a sequential program for inserting an object in a list. The principal differences are that nodes are created by calling a factory `create()` method, not by calling `new`, and that property `get` and `set` methods are used instead of field accesses.

Transactions (or methods used within transactions) should ensure that all shared objects are atomic, since the DSTM2 library does not provide synchronization and recovery for non-atomic objects. This restriction is analogous to the conventional lock-based discipline that requires that all objects shared by concurrent threads be thread-safe (that is, locked while in use). Because DSTM2 manages synchronization for atomic objects, there is no need to avoid deadlocks or to associate locks with objects.

The transaction itself is invoked by creating a `Callable<T>` object (that is, one that provides a `T call()` method), and passing it to the `Thread.doIt(...)` method.

```
  result = Thread.doIt(new Callable<Boolean>() {
    public Boolean call() {
      return intSet.insert(value);
      }
   });
```

The `doIt` method, shown in Figure 3 (slightly simplified for legibility), ensures that transactions are properly nested. It replaces an earlier API in which programmers explicitly called methods to begin and end transactions. The `doIt` method repeatedly retries a transaction until it either commits or throws an unexpected exception.

```
public static <T> T doIt(Callable<T> xaction) {
  T result = null;
  while (!Thread.stop) {
    beginTransaction();
    try {
      result = xaction.call();
    } catch (AbortedException d) {
    } catch (Exception e) {
      throw new
            PanicException("Unhandled exception " + e);
    }
    if (commitTransaction()) {
      return result;
    }
  }
}
```

Figure 3: The transaction retry loop

The private `beginTransaction` method called by `doIt` allocates a new *transaction descriptor*, which has a `status` field that indicates whether the transaction is committed, aborted, or still active. To commit a transaction, a thread attempts to change its descriptor's `status` field from `active` to `committed` using the atomic `compareAndSet` method (of `AtomicReference` from `java.util.concurrent.atomic`). The `dstm2.Thread` package exports the descriptor for the currently active transaction (if any), and factories typically use references to transaction descriptors to keep track of which transactions are reading or writing a particular object.

Property calls that occur outside a transaction have the same effect as regular, unsynchronized method calls. Such calls are not thread-safe, but are useful for initializing data objects before running benchmarks and running correctness checks afterwards.

2.2 Lessons

The design of this API was influenced by the lessons learned from the earlier version of DSTM. DSTM used a *wrapper* API, in which a `Node` object would be kept inside a transactional container, called a `TMObject<Node>` object. One could create an atomic `Node` object by calling

```
  TMObject<Node> tmNode =
                    new TMObject<Node>(new Node());
```

To open the object for reading or writing:

```
  Node rNode = tmNode.openRead(); // for reading
   ...
  Node wNode = tmNode.openWrite(); // for writing
```

While this API was adequate for experimental purposes, it has several pitfalls that make it less than ideal for widespread use. First, the programmer must not modify the object referenced by `rNode`. Second, if `wNode` is opened before `rNode`, then changes to `wNode` are visible through `rNode`, but not if they are opened in the opposite order, which can cause confusion. Third, the `rNode` and `wNode` references shown above are meaningful only during the lifetime of the transaction in which the calls to `open` occurred, so the programmer must be careful that these references do not linger. Furthermore, programmers must be aware of the container-based STM implementation when declaring classes. For example, if `Node` is a list node, then the reference to the next node must be a container, not another node:

```
class Node {
  int value;
  TMObject<Node> next; // not Node
}
```

Finally, the implementor of the `Node` class must provide a `clone()` method that does a "shallow copy" of the object for creating checkpoints.

The list example shows that DSTM2 substantially improves on the programming interface of DSTM. The key to improving transparency lies in the factory approach to transactional synchronization, which allows a straightforward transformation from field accesses to method calls. Another advantage is that there is no need to define a class-specific `clone()` method, because such methods, if needed, can be synthesized by the factory.

Encapsulating transactions in `Callable<T>` objects may seem awkward at first, but it has substantial advantages over earlier approaches. In the earlier DSTM API, transactions were delimited by explicit calls to methods to begin and end transactions. The call to end a transaction would return a condition indicating whether the transaction had committed. The principal disadvantage of this approach is that the application programmer was in charge of retrying aborted transactions, so syntactically, every transaction was a `try/finally` block within an unbounded loop. Moreover, this structure contains a subtle pitfall: if transactions can be nested, then a transaction might repeatedly abort because its parent transaction has already aborted. It follows that ending a transaction must return one of *three* conditions: (1) the transaction committed, or (2) the transaction failed, but can be retried, or (3) the transaction failed, and there is no point in retrying. Programming in this style is tiresome and error-prone.

3. ATOMIC-OBJECT FACTORIES

In this section, we describe in more detail the factory approach at the heart of the DSTM2 library.

We place transactional synchronization within method calls because there is no easy way to intercept field accesses directly. Defining atomic types by an interface has the advantage that each factory is free to provide its own implementations for the methods declared. Putting property definitions (that is, matching stylized `set` and `get` methods) in the interface has the advantage that the intended semantics of property `set` and `get` methods are clear, while the implementation of this semantics is left up to the factory, and necessary syntactic consistency conditions are easily checked (via reflection). Because all property types are required to be scalar or `@atomic`, a factory can checkpoint any atomic object's state simply by performing a "shallow copy", that is, copying field values but not the objects to which they refer.

The `Factory<T>` interface can be implemented by more than one class. In fact, it is possible to mix and match different factory implementations, even within the same program, as long as they agree on the order in which transactions appear to take effect. The factories provided by DSTM2 are compatible in this sense.

This flexibility is useful for experimentation. For example, one could run the same program repeatedly with different factories to test their relative performance (see Section 4). One could also use different factories for different classes, or switch dynamically among factories in response to changes in load or environment. To pick an extreme example, one could construct a linked list in which different nodes, all satisfying the same `@atomic` interface, were created by different compatible factories.

As noted in earlier work [2, 11], some applications may benefit from certain "escape mechanisms". For example, applications that scan through a list can reduce the level of synchronization conflict by "releasing" list nodes whose contents have been read, but whose values do not affect the application [11]. In a similar way, some applications can benefit from a transactional "snapshot" that returns the value that *would* have been returned had the object been read, allowing transactions to avoid reading unnecessary data [2]. The details of these mechanisms do not concern us here, but they illustrate an important point: some applications are likely to require escape mechanisms, and the nature of these mechanisms cannot be foreseen.

To address this issue, we split the methods declared by `@atomic` interfaces into two classes: property declarations, which have a well-known syntax and semantics, and other methods, whose interpretations are specific to the factory. For example, if your application requires a release or snapshot method, then you are free to declare one in your interface, but you must also provide a factory that can implement such a method. A transactional factory that encounters an unfamiliar method in an interface passed to it should throw an exception.

Transactions are managed by the `dstm2.Thread` package, which provides the following essential service: A transactional factory (or any other package) can *register* a method to be called when a transaction (1) validates, checking whether it can commit, (2) aborts, discovering that it cannot commit, or (3) commits, discovering that it can. This service allows any package to veto transaction commitment, or to clean up in an application-specific way on commit or abort.

As mentioned, every "field" of an atomic object must be scalar or atomic. This restriction means that atomic object fields cannot be arrays. Instead, DSTM2 provides an `AtomicArray<T>` class that can be used whenever an array of `T` is needed. This class provides its own class-specific synchronization and recovery. Eventually, we plan to provide a library of efficient basic types, analogous to the Java `Collections` package, to facilitate transactional applications.

3.1 The Base Factory

The transactional factories provided by DSTM2 are subclasses of a `BaseFactory` class that handles many tasks common to all transactional factories. Programmers who want to implement their own factories are advised, but not required, to inherit from `BaseFactory<T>`.

As usual for Java subclasses, each factory constructor first calls the `BaseFactory<T>` constructor, which takes a single argument, the class descriptor for the interface being implemented. The base class constructor uses reflection to examine the interface, splitting the method declarations into property definitions and other methods. It parses the property methods (checking, for example, that method names agree and that all types are `@atomic` or scalar). The property methods are stored in one symbol table, and the remaining uninterpreted methods are stored in another symbol table. When the base class constructor returns, these tables are available to the actual factory constructor.

In each of the DSTM2 factories described in the next sections, the first time a transaction accesses an object, it *opens* the object. It checks whether the object is currently in use by a conflicting transaction. If so, it consults a *contention manager* module [5, 11, 13] to decide whether to abort the conflicting transaction, to wait for a while for the conflicting transaction to commit, or to abort and restart itself. User-defined factories need not be organized in this way. For example, one could provide a factory that checks for conflicts only when a transaction is about to commit, in the style of the OSTM package of Harris and Fraser [7].

To illustrate how DSTM2 supports heterogeneous atomic-object factories, we now describe the implementations of two different factories that represent very different approaches to implementing atomic objects. We emphasize that these examples, different as they are, barely scratch the surface of possible factory implementations. These two factories represent thoroughly different approaches to transactional synchronization, so it is instructive to observe how they can both be accommodated in a single library.

3.2 Obstruction-Free Factory

A concurrent object is *obstruction-free* [10, 11] if any thread that runs by itself for long enough makes progress (pragmatically, this condition means that any thread will make 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.

The *obstruction-free* factory is based on the obstruction-free algorithm introduced in an earlier version of DSTM [11]. Objects created by the obstruction-free factory are represented in three levels: a `start` cell holds a reference to a *locator*, which has three fields: the *old version* of the object, the *new version*, and a reference to the status of the last transaction to open the object for writing (see Figure 4). The "logical value" of the object is the old version if that transaction is still active or has aborted, and is the new version if it has committed.

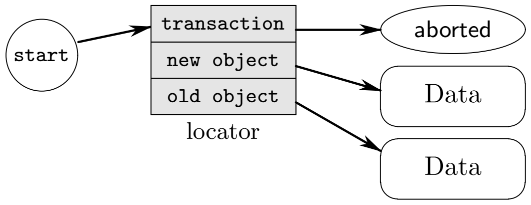

Figure 4: Structure of objects created by obstruction-free factory

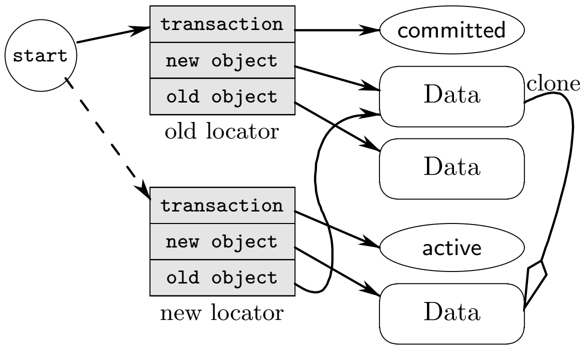

Figure 5: Opening an object created by the obstruction-free factory after recent commit

The factory synthesizes a `clone()` method that creates a "shallow copy" of an object. (In the earlier DSTM, the application programmer was required to provide such a method.)

The first time a transaction invokes a `set` method for a field of a particular object, it opens the object for writing. To do so, it first checks whether the previous writer committed or aborted. (As mentioned, if the previous writer is still active, then the transaction consults a contention manager to decide whether to pause, allowing the other transaction a chance to complete, or whether to abort the other, allowing the transaction to proceed immediately.) If the previous writer committed, the transaction creates a new locator whose old version is the prior new version, and whose new version is a cloned copy of the old version, created using the factory's synthesized `clone` method (see Figure 5). If the prior transaction aborted, the transaction behaves similarly, except that the prior old version is used instead of the new one (see Figure 6). It then installs the new locator in the `start` cell, using the atomic `compareAndSet` method to ensure consistency in the face of conflicts with competing transactions. This protocol ensures that the logical value of the object does not change upon opening, but that the logical values of all objects so opened by the transaction become the new versions when and if the transaction successfully commits. Henceforth, this transaction's invocations of the object's `set` methods will update the new version directly.

There are two variants of the obstruction-free factory that differ in the way they deal with read sharing. In the *visible-read* version of the factory, each object maintains a list of readers' transaction descriptors, and a transaction intending to modify the object must first abort them. In the *invisible-read* version of the factory, each transaction keeps a private list of the values it has read. (As a rule, factories keep transaction-local information as thread-local data, which is

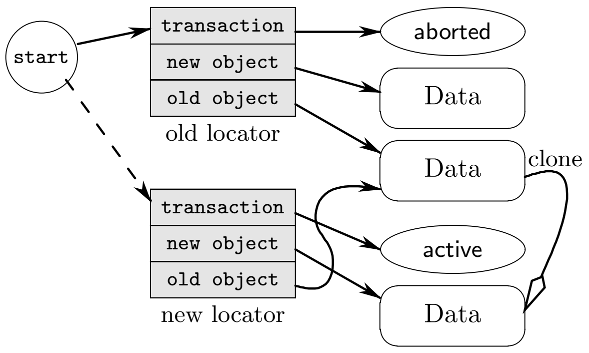

Figure 6: Opening an object created by the obstruction-free factory after recent abort

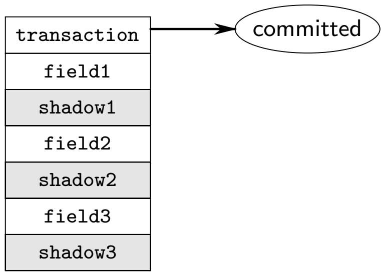

Figure 7: Structure of objects created by shadow factory

updated at important transitions such as transaction start, validation, commit, and abort.) When a transaction tries to commit, it must *validate* itself by checking that the values it read are still current. The invisible-read factory implements this functionality by registering methods to be called at transaction validation (to check currency of the values read), commit, and abort (to discard the list of values read).

3.3 Shadow Factory

The `shadow` factory uses short critical sections to avoid the indirection and allocation costs of the obstruction-free factory. This approach substantially lowers the overhead associated with opening an object, but it may not be as well suited to multiprogrammed environments (where multiple transactions share a single processor). In future architectures, some of these critical sections could be replaced by small hardware-supported transactions.

For each property defined in the interface, the shadow factory generates both a field and a shadow field (Figure 7). It synthesizes a `backup()` method that copies each regular field to its shadow, and a `restore()` method that copies the values in the other direction. As usual, when a transaction opens an object, it checks whether the last transaction to write the object committed or aborted. If it committed, then the object's fields hold the current state, so the transaction calls `backup()` to copy the fields to the shadow fields (Figure 8). If, instead, the most recent writer aborted, then the shadow fields hold the object's current state, and the transaction calls `restore()` to copy the shadow field values back to the regular fields (Figure 9). Either way, the factory has established the ability to restore the object's state if the current transaction aborts, and subsequent property

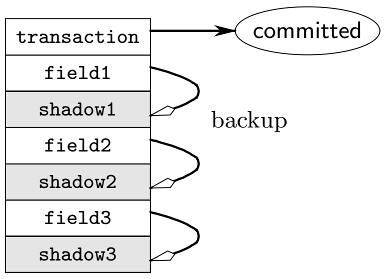

Figure 8: Opening an object created by shadow factory after recent commit

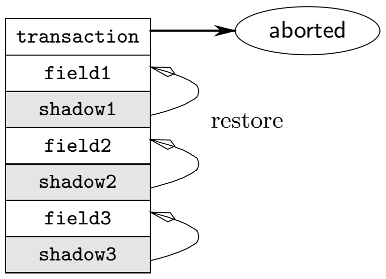

Figure 9: Opening an object created by shadow factory after recent abort

calls directly read or write the object's fields.

4. PERFORMANCE

We ran a number of simple benchmarks using the obstruction-free factory, the obstruction-free factory with invisible reads, and the shadow factory. We present results for two of these benchmarks, to demonstrate how DSTM2 can be used experimentally to evaluate the relative performance of different factories. (A thorough analysis of factory performance would require many more experiments which would distract from the point of the paper.)

All the results presented here are from runs on a Sun Fire^{TM} T2000 server. This server has a single UltraSPARC® T1 processor containing eight computing cores, each with four hardware strands, clocked at 1200 MHz. Each four-strand core has a single 8 KByte level-1 data cache and a single 16 KByte instruction cache. All eight cores share a single 3 MByte level-2 unified (I and D) cache, and a four-way interleaved 32GB main memory. Data access latency ratios are approximately 1:8:50 for L1:L2:Memory accesses.

4.1 List Benchmark

We first considered a simple linked list using the same list node interface shown in Figure 1. The list is sorted, and threads randomly insert, remove, or search for numbers. Figures 10–12 show the transactions per second executed in a 20-second period, where the number of updates executed (insert and remove calls) varies is 100%, 50%, or none. The shadow factory has substantially higher throughput than the obstruction-free factories, with the advantage becoming slightly more pronounced as the percentage of updates decreases. The two obstruction-free factories are roughly the same across the board.

4.2 Skip-List Benchmark

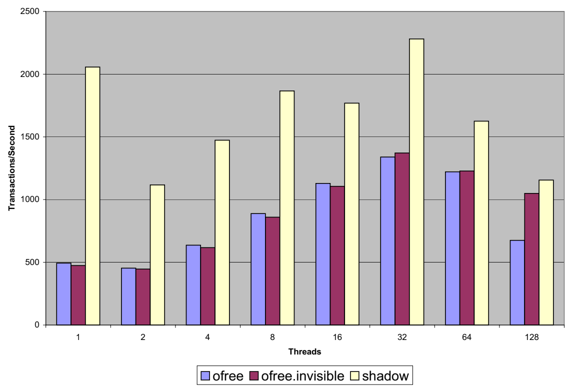

Figure 10: List Benchmark with 100% updates

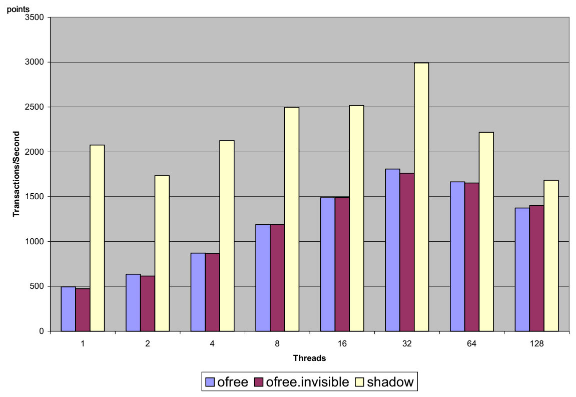

Figure 11: List Benchmark with 50% updates

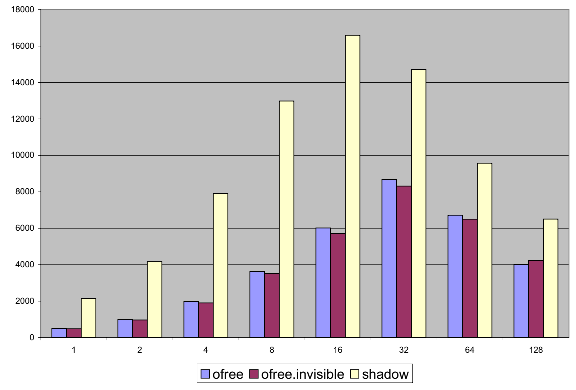

Figure 12: List Benchmark with no updates

```
@atomic public interface Node {

  /**
   * Get array of nodes further along in the skip list .
   **/
  public AtomicArray<Node> getForward();

  /**
   * Set array of nodes further along in the skip list .
   **/
  public void setForward(AtomicArray<Node> value);

  /**
   * Get node value.
   **/
  public int getKey();

  /**
   * Set node value.
   **/
  public void setKey(int value);

}
```

Figure 13: Atomic Skip-List Node

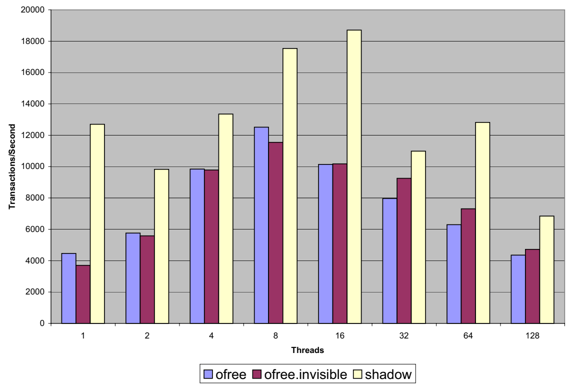

Figure 14: Skip-List Benchmark with 100% updates

In the next benchmark, we replace the list with a skip list [22]. The skip list node declaration, shown in Figure 13, illustrates a use of the built-in `AtomicArray` class. As shown in Figures 14–16, the shadow factory dominates the obstruction-free factory when the percentage of updates is small, but not when it is large.

5. RELATED WORK

The goal of our work has been to provide a flexible framework for allowing a wide range of researchers to experiment with different STM implementation mechanisms, while providing a reasonable API to encourage the development of a shared body of transactional programs. We believe that our framework provides the first STM implementation with a reasonable transactional programming interface that does not depend on modifications to compilers and/or runtime

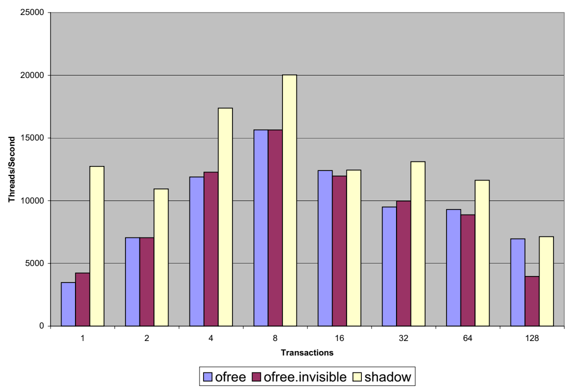

Figure 15: Skip-List Benchmark with 50% updates

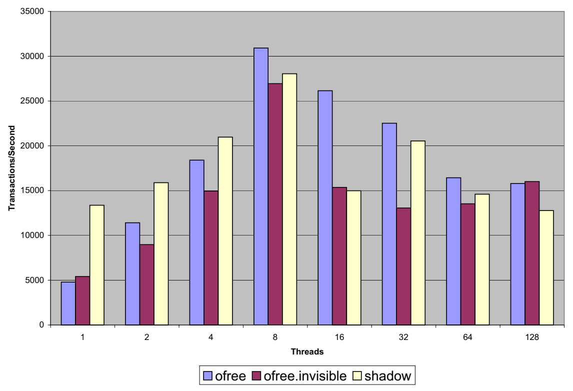

Figure 16: Skip-List Benchmark with no updates

systems.

As explained in Section 2.2, our previous DSTM [11] is implemented as a library, but does not provide an adequate programming interface to enable a significant body of transactional applications to be developed. Several other STM implementations that are based on DSTM inherit the same disadvantage, including the OSTM of Harris and Fraser [4] and the ASTM of Marathe, Scherer, and Scott [16]. The WSTM implementation of Harris and Fraser [4, 7] similarly requires the programmer to access memory through explicit library calls.

The SXM implementation of Herlihy [9] is a precursor to the work reported here, and has many of the same advantages.[2]

Most other STM implementations we are aware of are implemented via changes to compilers and/or run-time systems, and are therefore much less flexible and more difficult to distribute and modify than implementations based on our DSTM2 framework. These include:

- The conditional critical region construct of Harris and Fraser [7], which is implemented through modifications to both the source-to-bytecode and bytecode-to-native compilers for the Sun Java Virtual Machine for Research.
- The Composable Memory Transactions of Harris, Marlow, Peyton-Jones and Herlihy [8], which is implemented in the Glasgow Haskell compiler;
- The McRT STM of Saha et al. [24], which depends on specialized compiler and run-time support; and
- The Transactional Monitors of Welc, Jagannathan, and Hosking [26], which is implemented in a modified version of the Jikes RVM compiler; and
- The HybridTM of Moir [18], which is designed to exploit whatever hardware transactional support is available, includes an STM that runs on current systems but uses a modified C/C++ compiler to achieve it.

6. CONCLUSIONS

Our experience moving from the original DSTM to DSTM2 has convinced us that designing a simple, relatively pitfall-free, (mostly) implementation-agnostic STM library is harder than it may look. The DSTM2 design described here is a result of our collective experience designing and working with earlier STM packages, both in Java programming language [11] and in C# [9]. Our aspiration is that DSTM2 can be used as a common framework for experimenting with a variety of transactional algorithms and implementations, sparing researchers from the time-consuming chore of constructing their own infrastructures, and providing a common ground for comparing alternatives. As noted, no prior work

[2]A note for OOPSLA referees: a short paper describing SXM at a high level has been submitted to the Transact '06 workshop. This workshop will not publish proceedings, and as such is not a "publication venue"; it is our hope that OOPSLA will be the primary publication venue for presenting this work.

combines the same degree of simplicity, portability, and flexibility.

DSTM2 is *object-based* in the sense that transactional synchronization works by intercepting and synchronizing calls to object methods. An alternative approach is *word-based* synchronization, which works by intercepting and synchronizing field (or memory) accesses. Word-based synchronization is intended for environments where one cannot rely on a high-level language to mediate memory accesses, such as C and C++ programs that allow pointer arithmetic. The engineering issues that arise in word-based STM systems are quite different from those that arise in the object-based alternatives considered here. Most importantly, the lack of facilities for reflection and run-time code generation in the environments for which word-based STMs are typically designed mean that it is difficult or impossible to implement a word-based STM that provides a clean API without modifying the compiler.

We think that DSTM2 provides a unique combination of a simple API and substantial flexibility to experiment with run-time mechanisms. This flexibility follows from two innovations: atomic object factories allow user-defined synchronization algorithms to intercept method calls, combined with the ability to register user-defined methods to veto transaction commitment and to clean up when transactions finish.

7. REFERENCES

[1] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded transactional memory. In *Proc. 11th International Symposium on High-Performance Computer Architecture*, pages 316–327, February 2005.

[2] C. Cole and M.P. Herlihy. Snapshots and software transactional memory. *Science of Computer Programming*, 58(3):310–324, December 2005.

[3] Apache Software Foundation. Byte-code engineering library. `http://jakarta.apache.org/bcel/manual.html`.

[4] Keir Fraser and Tim Harris. Concurrent programming without locks. Submitted for publication.

[5] Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Polymorphic contention management in sxm. In *Proceedings of the 19th International Symposium on Distributed Computing*, September 2005.

[6] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. In *Proc. 31st Annual International Symposium on Computer Architecture*, June 2004.

[7] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 388–402. ACM Press, 2003.

[8] Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable memory transactions. In *Principles and Practice of Parallel Programming*, 2005.

[9] Maurice Herlihy. http://www.cs.brown.edu/m̃ph/sxm.htm.

[10] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDS)*, pages 522–529, May 2003.

[11] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101. ACM Press, 2003.

[12] Maurice Herlihy and J. Eliot 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.

[13] W. N. Scherer III and M. L. Scott. Contention management in dynamic software transactional memory. In *PODC Workshop on Concurrency and Synchronization in Java Programs*, July 2004.

[14] Amos Israeli and Lihu Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing*, pages 151–160. ACM Press, 1994.

[15] V. J. Marathe, W. N. Scherer III, and M. L. Scott. Design tradeoffs in modern software transactional memory systems. In *7th Workshop on Languages, Compilers, and Run-time Support for Scalable Systems*, October 2004.

[16] Virendra Marathe, William Scherer, and Michael Scott. Adaptive software transactional memory. In *Proceedings of the 19th International Symposium on Distributed Computing*, September 2005.

[17] Mark Moir. Practical implementations of non-blocking synchronization primitives. In *Proceedings of the sixteenth annual ACM symposium on Principles of distributed computing*, pages 219–228. ACM Press, 1997.

[18] Mark Moir. Hybrid transactional memory, Jul 2005. Unpublished manuscript.

[19] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. LogTM: Log-based transactional memory. In *Proc. 12th Annual International Symposium on High Performance Computer Architecture*, 2006.

[20] Kevin E. Moore, Mark D. Hill, and David A. Wood. Thread-level transactional memory. Technical Report CS-TR-2005-1524, Dept. of Computer Sciences, University of Wisconsin, March 2005.

[21] J. Eliot B. Moss and Antony L. Hosking. Nested transactional memory: Model and preliminary architecture sketches. In *SCOOL*, 2005.

[22] W. Pugh. Skip lists: a probabilistic alternative to balanced trees, 1990.

[23] Bratin Saha, Ali-Reza Adl-Tabatabai, Rick Hudson, Chi Cao Minh, and Benjamin Hertzberg. Mcrt-stm. In *PPoPP*, 2006.

[24] Bratin Saha, Ali-Reza Adl-Tabatabai, Rick Hudson, Chi Cao Minh, and Benjamin Hertzberg. Mcrt-stm: A high performance software transactional memory system for a multi-core runtime. In *Symposium on Principles and Practice of Parallel Programs (PPoPP)*, 2006.

[25] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213. ACM Press, 1995.

[26] Adam Welc, Suresh Jagannathan, and Antony L. Hosking. Transactional monitors for concurrent objects. In *ECOOP*, pages 519–542, 2004.