Capabilities and Limitations of Library-Based Software Transactional Memory in C++ *

Luke Dalessandro Virendra J. Marathe Michael F. Spear Michael L. Scott
Computer Science Dept., University of Rochester
{luked,vmarathe,spear,scott}@cs.rochester.edu

Abstract

Like many past extensions to user programming models, transactions can be added to the programming language or implemented in a library using existing language features. We describe a library-based transactional memory API for C++. Designed to address the limitations of an earlier API with similar functionality, the new interface leverages macros, exceptions, multiple inheritance, generics (templates), and overloading of operators (including pointer dereference) in an attempt to minimize syntactic clutter, admit a wide variety of back-end implementations, avoid arbitrary restrictions on otherwise valid language constructs, enable privatization, catch as many programmer errors as possible, and provide semantics that "seem natural" to C++ programmers.

Having used our API to construct several small and one large application, we conclude that while the interface is a significant improvement on earlier efforts, and makes it practical for systems researchers to build nontrivial applications, it fails to realize the programming simplicity that was supposed to be the motivation for transactions in the first place.

Several groups have proposed compiler support as a way to improve the performance of transactions. We conjecture that compiler—and language—support will be even more important as a way to improve the programming model.

1. Introduction

Transactional Memory (TM) [17] offers an attractive alternative to locks in highly concurrent programs: the programmer indicates that certain code regions ought to run as isolated, atomic units, and the underlying system attempts to run those regions in parallel whenever possible. Implementations are typically speculative: transactions are pursued optimistically in parallel, but allowed to commit only in the absence of conflicts with other transactions.

A hardware TM system can check for conflicts on naturally occurring load and store instructions in a program's instruction stream. Software or hybrid TM systems, however, require some sort of "hook" at which to perform their checks. These hooks may be provided by the compiler or, with an appropriate API, by an STM (software TM) library. Both approaches are common in prior work. Compiler-based systems [1, 11–13, 22, 30] typically extend modern managed languages, generally provide a clean programming model, and make use of static analysis to perform important optimizations. Library-based systems [5, 7–9, 14, 15, 20, 21, 23, 27] are substantially more portable, and might, at least in principle, be easier to deploy as an incremental extension to large existing systems.

Most of our own prior work in TM has adopted the library-based approach, initially using a Java API [20] based on that of DSTM [14], and later a similar API for C++ [21]. Unfortunately, as we discuss in Sections 2.2 and 3.4, this API turns out to be both cumbersome and prone to programmer error. Motivated by experience with small-scale microbenchmarks in DSTM, Herlihy chose to leverage runtime code generation in C# to provide a simpler programming model in his SXM system [9, 15]. Subsequently, he employed byte code engineering to similar effect in Sun's Java-based DSTM2 [16].

Drawing inspiration from SXM and DSTM2, but using different techniques, we have developed a transactional API for C++ (referred to in this paper as RSTM2) that leverages macros, multiple inheritance, generics (templates), and operator overloading (including pointer dereferences) to achieve a similar reduction in syntactic clutter. As in the C# and Java systems, we require the use of accessor functions to read and write fields of transactional objects. Unlike these systems, we invoke accessor functions through *smart pointers* [2] and use initialization of the pointers as an extra bookkeeping "hook", allowing us to amortize overhead across multiple accesses to the same object.

The RSTM2 API admits a wide variety of back-end implementations. We currently use it with the original RSTM back end [21], a C++ version of ASTM [20], a blocking, zero-indirection system based on redo logs [28] (similar to the TL2 system of Dice et al. [5]), a dummy back end that implements coarse-grain locks, and several hard-

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

ware/software hybrids [26, 28]. The API enables the compiler and runtime to catch programmer errors that would be missed with the earlier interface. It also supports explicit *privatization* [29], allowing a thread to elide bookkeeping overheads when program logic guarantees that a given object will not be accessed by other threads concurrently.

In Section 2 we briefly describe the behavior of a transaction at an abstract level. We also describe our original, DSTM-like API. In Section 3 we present a more detailed description of the new API, and show how it addresses limitations of the original. We have used this API to build a half-dozen microbenchmarks and one nontrivial application. We describe the latter—an implementation of Delaunay triangulation—in Section 4. Among other things, this application makes extensive use of privatization based on geometric partitioning of the input data set.

Our experience with the mesh application confirms that RSTM2 suffices (where its predecessor did not) for large-scale experimentation. Unfortunately, we have also discovered several significant limitations and pitfalls in the API, each of which could be addressed, in whole or in part, by language and compiler support. Our conclusions are complementary to, though mostly orthogonal from, Boehm's findings with regard to library-level threads [3]. We conjecture that experience similar to ours would emerge from large-scale use of any library-based TM system, including those for languages with built-in threads.

Our conclusion, to which we return in Section 5, is that software transactional memory will not succeed at making synchronization easy unless it becomes an integral part of programming languages and compilers. While this conclusion may not be surprising, it was nonetheless not obvious to us until we tried to pursue the library-based approach as far as possible. We hope our experience will encourage other groups to experiment with language features, and in particular to consider the relationships among transactional, privatized, and nonshared data and references.

2. Library-Based Software Transactional Memory

A *transaction* is a program fragment that appears to execute atomically and in isolation from concurrent transactions. In typical implementations a transaction speculatively accesses a set of shared memory locations and then attempts to commit its speculative state, succeeding only if no conflict was detected with other committed transactions.

STM runtimes rely on the notion of *ownership* of locations to enforce atomicity and isolation. Object-based STMs manage ownership by embedding *ownership metadata* in each transactional object. Word (or block) based STMs store ownership metadata in a global hash table of *ownership records*. Transactions must acquire exclusive ownership of a location to write to it, while shared ownership is sufficient for read-only access. Alternatively, a transaction may snapshot the metadata of locations it reads, verifying consistency at commit time. Policies for resolving conflicts (contention managers) are described in other papers [24].

2.1 STM API for Unmanaged Code

It is possible to use word-based STMs in strongly-typed languages, however these systems are a more natural fit in languages like C. The basic API for such systems has only four operations: `begin_txn` initializes thread-specific transaction data; `stm_read(&l)` safely reads from location `l`; `stm_write(&l, v)` speculatively writes value `v` to location `l`; and `end_txn` tries to commit all speculative writes.

With this API, programmers bear the entire burden of inserting calls to read and write shared data. There are no mechanisms to detect access to shared locations outside the API, or to elide redundant calls. Features like inheritance, virtual methods, and operator overloading exacerbate the difficulty of writing correct transactional code when this API is used for C++.

2.2 Basic API for Object-Oriented Code

The original DSTM library [14] established a number of practices that influence most object-oriented STM libraries. In particular, DSTM hid each transactional object behind an opaque `TMObject` header. The header can be queried to obtain an `Object` reference, which must then be cast to the appropriate type. The DSTM API has three main methods:

`void beginTransaction()`: initializes transaction data structures and begins execution of a transaction.

`Object open(mode)`: method called on an opaque object header; returns an object that can be safely read or written (depending on the value of `mode`).

`bool commitTransaction()`: attempts to commit the transaction; fails if the transaction is not guaranteed to be atomic and isolated.

The object returned by `open` can be accessed without further API calls. This provides a caching benefit relative to word-based STM, where calls to `stm_read` must repeatedly inspect object metadata.

There are a number of programming pitfalls in the original DSTM API. The programmer is responsible for creating wrappers for transactional objects, and can write code that creates 0 (or > 1) headers for a single transactional object. Likewise, the programmer must take care to ensure that references to objects are replaced with references to transactional headers in class member declarations. Since exceptions are used to interrupt transactions that abort, the programmer must wrap each transaction in a `try/catch` block. Code can open an object with `mode==READ` and then modify the object (or continue to access it after the transaction completes). If a single object is opened for reading and then writing, the readable version remains usable, but logically invalid (we refer to this as an alias error). Lastly, anecdotal

evidence suggests the API is unnatural to the programmer, and adds significant syntactic clutter.

2.3 Reflection-Based API

The SXM [9, 15] and DSTM2 [16] libraries extend and improve upon the DSTM API by leveraging the robust runtime support of C# and Java, respectively.

In both SXM and DSTM2 a transactional block is generated as a closure and is passed to the STM runtime, which executes it inside a `try/catch` block and, if desired, nests it in a loop that retries it until it commits. Both systems also make heavy use of reflection and dynamic code generation (C# `Reflection.Emit`; Java byte-code engineering) to create objects that contain both transactional metadata and *accessor* methods; the latter make hidden `open` calls, detect alias errors, and perform other algorithm-specific operations. Finally, both exploit automatic garbage collection for transactionally safe memory management. Though available only in managed code, we find the accessor-based API to be a significant improvement over explicit `open` calls. It eliminates the performance-enhancing caching of DSTM, but makes programming simpler and more natural.

2.4 Object-Based API for Unmanaged Code

The original RSTM library (referred to here as RSTM1 [21]) used a C++ version of the DSTM API, with four extensions. First, RSTM1 delineates transactions with `BEGIN_TRANSACTION` and `END_TRANSACTION` macros, hiding the nested `while/try` blocks of DSTM. Second, it distinguishes more explicitly between opening for read and opening for write, with `open_RO` returning a `const` pointer and `open_RW` returning a bare pointer. Third, it uses multiple inheritance to allow a back end to inject metadata into each shared object—in particular a back-pointer to the object header, used for run-time checks. Fourth, it uses generics (templates) to replace the general-purpose `TMObject` header type with specific `Shared<foo>` types, thereby avoiding casts on the returns from `open_RO` and `open_RW`.

3. Smart Pointer API

Our new RSTM2 API uses the C++ *smart pointer* pattern [2] to hide the internal metadata organization of the back-end implementation and to provide the necessary STM system hooks. This allows application code to closely resemble non-transactional code, while remaining independent of any particular back end. At the same time programming errors that are unique to library STM implementations, such as alias errors caused by out of place modification [16], can be handled correctly and transparently.

Smart pointers rely heavily on template metaprogramming and operator overloading support in C++. The first three subsections below describe transactional objects, smart pointer types, and other library calls. The fourth presents brief examples that illustrate the RSTM2 API and its advantages over RSTM1.

3.1 Making Objects Transactional

As in RSTM1 we use multiple inheritance to inject metadata into objects by providing a class from which all transactional objects must inherit. This metadata is implementation specific, thus the RSTM2 API does not directly provide such a class, but rather assumes that it is available in the back end and exported as `stm::Object`. Implementations of this class must provide transaction-aware versions of `operator new` and `operator delete`. These must be compatible with any allocator, and must defer reclamation until no concurrent transaction can access a deleted object [8, 18].

Implementations of `stm::Object` also define abstract methods to clone an object, deactivate a clone, and (for some back ends) copy a clone back to the original. These must be implemented by the user program. For objects that have neither internal pointers nor heap-allocated components, the cloning and copy-back routines can simply call the copy constructor; the deactivation routine can be empty.

Since the API is independent of the back end, but must support a variety of metadata organizations, we require that any user-level reference to a transactional object, whether stand-alone or embedded in an object itself, be realized as a smart pointer. Thus in a linked list of `node` objects, the `next` pointer must be of type `sh_ptr<node>` rather than `node*`.

Additionally, in order to support per-access validation in systems that require it (these include the redo-log and hybrid systems mentioned in Section 1), and to provide a natural hook for word-based STMs, RSTM2 requires the use of accessor-declaring macros. Each use of the `DECLARE_FIELD(t, n)` macro creates a `protected` field of type `t` with name `n`, and generates public getter and setter methods to access that field. The getter takes a parameter used for post-access validation. Individual STM implementations define the type of this parameter, so that it can be a no-op when appropriate.

3.2 Smart Pointer Types

The following four generic smart pointers hide all metadata manipulation behind operators for construction, assignment (`operator =`), and dereference (`operator ->` and the rarely-used `operator *`).

`sh_ptr<T>:` opaque reference to a transactional object of type `T`. An `sh_ptr` cannot be dereferenced, but it can be copied and compared to other `sh_ptr`s, or used to initialize other kinds of smart pointers when dereference is required. Reference fields in linked structures should always be `sh_ptr`s.

`rd_ptr<T>:` similar to a `const` pointer; permits reading of a transactional object, but does not allow writing.

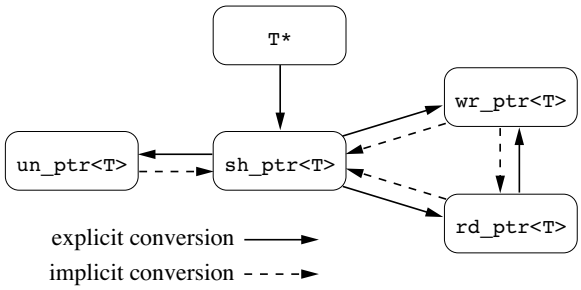

Figure 1. Permitted conversions among smart pointer types. The `T*` to `sh_ptr<T>` case is for initialization only (`operator new` returns an ordinary pointer).

`wr_ptr<T>`**:** permits reading and writing of a transactional object.

`un_ptr<T>`**:** similar to `wr_ptr`, but for privatized objects, outside transactions only.

Figure 1 shows the implicit and explicit conversions permitted among the different smart pointer types. Implicit conversions can be performed using implicit constructor (`rd_ptr<foo> p = q`) or `operator =` syntax. Explicit conversions are performed using explicit constructors (`rd_ptr<foo> p(q)`), casts, or, when transitioning from an `sh_ptr`, `operator =`. Within a transaction, these conventions make it easy to move back and forth between `sh_ptrs` and either `rd_ptrs` or `wr_ptrs` (similarly, outside a transaction, between `sh_ptrs` and `un_ptrs`). Explicit syntax is required to upgrade a `rd_ptr` to a `wr_ptr`. Both `rd_ptrs` and `wr_ptrs` can be used only within a transaction; `un_ptrs` can be used only outside. Only `sh_ptrs` can safely refer to (nonprivatized) transactional objects outside a transaction.

Enforcing these conventions results in one peculiarity: a `wr_ptr` can be initialized from a `rd_ptr` using explicit constructor syntax (e.g., `wr_ptr<T> w(r);`), but not using "assignment" syntax (`wr_ptr<T> w = r;`). Explicit-constructor examples can be seen on lines 3, 4, and 12 of Figure 3. To enable the more intuitive assignment-style declaration we would have to provide an implicit constructor, which C++ could then use when passing parameters, allowing a `rd_ptr` to be passed to a function that expects a `wr_ptr`, thereby violating `const` semantics.

3.3 Other Library Calls

Each thread, when created, must call `stm::init` to gain access to the TM system. Before completing, it must call `stm::shutdown`. These calls ensure that the thread has a transaction descriptor, and that the necessary memory management infrastructure is in place for correct creation and deletion of transactional objects. In some C++ implementations it may not be safe to call destructors from within a transaction. The `tx_delete` function schedules destruction for the end of the current transaction.

As in RSTM1, transactions are delimited by `BEGIN_TRANSACTION` and `END_TRANSACTION` macros. These contain loop control and `try` block fragments that compile without error only when lexically matched.

Lastly, we provide two special-purpose functions. The first, `tx_release`, can be used for *early release* [14], in which an object is removed from a transaction's read set. The second, `stm::fence`, is used outside of a transaction to ensure that all speculative state has coalesced before nontransactional code uses `un_ptrs`. Early release is an application-specific optimization, and must be used with care to ensure consistency. The transactional fence serves as a hook to *privatize* shared objects [29]. Its use is not required if (as in the mesh application of Section 4) privatization is achieved by global consensus (e.g., barriers).

3.4 Examples

The two chief benefits of the RSTM2 API are its approximation of the look of nontransactional code and its ability to catch programmer errors, either statically (via the type system) or dynamically (via hooks into the runtime).

Figures 2 and 3 show code to insert an object in a sorted linked list using the RSTM1 and RSTM2 APIs, respectively. Where the old API requires frequent calls to `open_XX` (e.g., to traverse the list), the new relies on smart pointers whose presence is apparent only in declarations and casts. Note the explicit constructors mentioned in Section 3.2: the declaration at line 12, for example, initializes `insert_point` to be a copy of `previous`. In addition to simplifying the code, the lack of `open_XX` calls allows us to create generic functions that can be called in both transactional and privatized contexts; we discuss this further in Section 4.

For back ends that require per-access hooks, the programmer must use the accessors, validators, and `DECLARE_FIELD` macros mentioned in Section 3.1. In the loop of Figure 3, for example, the last line would be written `current = previous->get_next(previous.v());`. The need for accessors and validators is independent of the choice between smart pointers and `open_XX`.

As an example of error checking, consider Figure 4, which depicts a bug caused by updates to clones. With the RSTM back end, opening an object for writing creates a new version of the object, and turns existing read-only references into dangling pointers. In this example, `r1` is initialized to a clean object. When `w1` is initialized, it clones that object and modifies the clone. The initialization of `r2` uses the clone as well, but `r1` now holds a reference to a stale immutable copy. The assertion at the end of the transaction may fail when logically it should succeed.

The assertion need not fail, however, if we add a check in `rd_ptr::operator ->`. The check can detect that the object `r1` points to has been acquired by the current transaction, and then silently change the reference in `r1` to the clone. Alternatively, if we want to minimize overhead, we

```
 1  void list::insert(int val) {
 2    BEGIN_TRANSACTION;
 3      const node* previous = head->open_RO();
 4      const node* current =
 5          previous->next->open_RO();
 6
 7      while (current != NULL) {
 8        if (current->val >= val) break;
 9        previous = current;
10        current = current->next->open_RO();
11      }
12      if (!current || current->val > val) {
13        node* n =
14            new node(val, current->shared());
15        previous->open_RW()->next =
16            new Shared<node>(n);
17      }
18    END_TRANSACTION;
19  }
```

Figure 2. Insertion in a sorted linked list using RSTM1.

```
BEGIN_TRANSACTION;
  rd_ptr<LLNode> r1(head);
  wr_ptr<LLNode> w1(head);

  w1->val++;

  rd_ptr<LLNode> r2(head);

  assert(r1->val == r2->val);
END_TRANSACTION;
```

Figure 4. An alias error, which RSTM2 can detect.

can arrange for the check to print an error message when run in debug mode, and then elide it in production code. Similar optional checks ensure that `rd_ptr`s and `wr_ptr`s are used only within transactions, and that `un_ptr`s are used only outside transactions. RSTM1 and other traditional library-based APIs provide no way to catch such errors.

4. Experience with the API

We have used our API to construct a parallel implementation of Delaunay triangulation [4]. Given a set of points $\mathcal{P}$ in the plane, a *triangulation* partitions the convex hull of $\mathcal{P}$ into a set of triangles such that (1) the vertices of the triangles, taken together, are $\mathcal{P}$, and (2) no two triangles intersect except by sharing an edge. A *Delaunay* triangulation has the added property that no point lies in the interior of any triangle's circumcircle (the unique circle determined by its vertices). If not all points are colinear, a triangulation must exist. If no four points are cocircular, the Delaunay triangulation is unique.

Delaunay triangulation is widely used in finite element analysis, where it promotes numerical stability, and in graphical rendering, where it promotes aesthetically pleasing shading of complex surfaces. In practice, Delaunay meshes are typically *refined* by introducing additional points where needed to eliminate narrow triangles.

```
 1  void list::insert(int val) {
 2    BEGIN_TRANSACTION;
 3      rd_ptr<LLNode> previous(head);
 4      rd_ptr<LLNode> current(previous->next);
 5
 6      while (current != NULL) {
 7        if (current->val >= val) break;
 8        previous = current;
 9        current = previous->next;
10      }
11      if (!current || current->val > val) {
12        wr_ptr<LLNode> insert_point(previous);
13        insert_point->next =
14            sh_ptr<LLNode>(new LLNode(val, current));
15      }
16    END_TRANSACTION;
17  }
```

Figure 3. Insertion in a sorted linked list using RSTM2 (with smart pointers, but without accessors and validators).

4.1 Transactional Implementation

At the 2006 Workshop on Transactional Workloads, Kulkarni et al. proposed refinement of a (preexisting) Delaunay triangulation as an ideal TM application [19]. Our code addresses the complementary problem of constructing the initial triangulation; we do not yet consider refinement.

We begin by sorting points into geometric regions, one for each available processor. We then employ Dwyer's sequential solver [6] to find, in parallel, the Delaunay triangulations of the points in each region. For uniformly distributed points (which we assume), Dwyer's algorithm runs in $O(n \log \log n)$ time, where n is the number of points. Given triangulations of each region, we employ a mix of transactions and thread-local computation to "stitch" the regions together, updating previously chosen edges when necessary to maintain the Delaunay circumcircle property.

All told, our application comprises approximately 3200 lines of heavily-commented C++, spread across 24 source files. There are three transactional types (subclasses of `stm::Object`) and three static occurrences of transactions.

The first transactional type represents an edge between two points. As suggested by Guibas and Stolfi [10], an `edge` object contains pointers to its two endpoints and to the four neighboring edges found by rotating clockwise and counterclockwise about those endpoints. The second transactional type contains, for a given point, a reference to some adjacent edge, from which others can be found by following neighbor links. The third transactional type is used to create links in the chains of a global hash set, used to hold created edges.

The first static transaction protects a call to the `edge` constructor. This in turn calls 11 additional (non-accessor, non-library) routines, directly or indirectly, for a total (not including headers) of 72 lines of code. The second static transaction protects the body of a subroutine used (twice) when stitching regions together. Together with the bodies of 17 called routines, it comprises 155 lines of code. The third

static transaction is used to "reconsider" edges that may not satisfy the circumcircle property in light of subsequent region stitching. Together with the bodies of 16 called routines, it comprises 214 lines of code.

Performance results are reasonable (details appear in a companion paper [25]). Linked to our redo-log (TL2-like) back end and running on a 16-processor SunFire 6800, triangulation of 100,000 points takes 0.26 seconds, a speedup of 7.7 over an optimized, single-threaded implementation of Dwyer's sequential algorithm. On an 8-core Sun T1000, the maximum speedup is 3.6. Performance is limited by memory bandwidth on both machines; speedup decreases with larger numbers of points.

Transaction overhead is a relatively insignificant contributor to run time: initial triangulation of thread-private regions consumes more than 90% of total run time (usually more than 95%). Run times for redo-log, coarse-grain locks, and fine-grain locks (implemented separately with `#ifdef`s) are within a few percent of one another. This means, of course, that transactional memory offers no advantage over coarse-grain locks in this particular program—neither performance nor convenience—except perhaps for the comfort of knowing from the outset that lock-induced bottlenecks would never require a major re-write of the code.

Interestingly, with so much geometrically-partitioned computation, performance is critically dependent on avoiding overhead outside transactions. Because it imposes an extra level of indirection on every inter-object reference (leading to a two-fold increase in cache misses in memory-limited private code), the RSTM back end achieves only half the performance of redo-log. We consider this a powerful argument for zero-indirection systems.

4.2 Capabilities

Starting with an implementation of Dwyer's sequential solver he had previously written in Java, one of the authors (Scott) was able to build and tune the application in his spare time over the course of a three-month period. The result is, in our opinion, reasonably clear and readable. We believe it would have been much more difficult to build using RSTM1. Moreover automatic checks on smart pointers caught several usage errors that would simply have resulted in erroneous behavior with the earlier API. Approximately half the development time was devoted to (1) finding the constructor bug discussed in Section 4.3.1 below, and (2) developing a template-based means of sharing subroutines between transactional and privatized computations.

Our code contains 11 functions that operate on transactional data and that are called both within a transaction (on data that are actively shared) and in the course of geometrically-partitioned computation (on data that are temporarily private to one thread). Examples include the `edge` constructor and destructor methods, the hash set insert and remove methods, several utility routines, and the key routine that reconsiders a potentially non-Delaunay edge.

```
template<class edgeRp, class edgeWp>
bool reconsider(edgeWp self, const int my_seam, ... ) {
  ...
  point* x = ...
  point* y = ...
    // outlying corners of bounding quadrilateral
  if (!txnal<edgeRp>() &&
        (closest_seam(x) != my_seam
     ||  closest_seam(y) != my_seam)) {
           return false; // defer to synchronized phase
  }
  ...
  return true;
}
```

Figure 5. Skeleton of code to reconsider a potentially non-Delaunay edge. Reconsideration must be deferred if we are running (barrier-protected) private code and the edge's bounding quadrilateral does not lie entirely within the current thread's geometric region.

A simplified snippet of the latter routine appears in Figure 5. It is instantiated in private code with template parameters `<un_ptr<edge>, un_ptr<edge> >`, and in transactional code with `<rd_ptr<edge>, wr_ptr<edge> >`. Internally, it uses a `txnal<`*pointer_type*`>()` generic predicate to choose between alternative code paths.

In a few other places in the code, we use generic type constructors to declare pointer types analogous to (i.e., transactional or nontransactional, as appropriate) some other pointer type. For example, if `edgeRp` is a template parameter that may be either `rd_ptr<edge>` or `un_ptr<edge>`, then `edgeRp::w_analogue<foo>::type` will be `wr_ptr<foo>` or `un_ptr<foo>`, respectively.

4.3 Limitations

The most obvious problem with the API is simple awkwardness. Accessors are not a natural way to access fields of an object in C++, though they could easily be made so with compiler support, as in C#. Validator arguments to getters, required for post-access consistency checks in zero-indirection systems, are pure syntactic clutter. While they could be eliminated with trivial compiler support, we see no way to hide them in a purely library-based system.

Several of our back ends, including RSTM and redo-log, create new copies of objects. Any class for which bitwise copying does not suffice must provide `clone` and `deactivate` or `redo` methods. These expose implementation details that should ideally be hidden from the programmer. (None of our current applications requires nontrivial copies.) In a similar vein, the use of templates to support transactional/private code sharing is too cumbersome to ask of naive programmers. With compiler support (as in McRT and Bartok), a similar effect could be achieved transparently via automatic function cloning.

More problematically, our four different flavors of smart pointers, while useful for catching errors, introduce a level

of complexity that seems out of place in a programming model intended to simplify concurrency. In future work, we hope to develop language mechanisms that provide similar functionality in a more implicit fashion.

4.3.1 Programming Restrictions

Beyond simple awkwardness, our API imposes several important restrictions on transactions and transactional objects. One cannot safely escape a transaction, for example, in any way other than falling off the end—no `goto`s, no `break`s, no `return`s. This restriction would disappear if C++ offered a Java-like `try...finally` construct, but in the current language we can neither support nonlocal exits nor catch them statically. One early version of the mesh application had an errant `return` in a transaction; the behavior that resulted would not have made any sense to a programmer unfamiliar with STM system internals.

A more significant restriction imposed by the API is a prohibition against most nonstatic methods in transactional classes: because we insist that fields be accessed through smart pointers, we cannot safely use the `this` parameter of a C++ member function. The replacement idiom—a static method with explicit `sh_ptr` parameter—is unnatural to a C++ programmer. With compiler support, `this` could *be* an `sh_ptr`.

Destructors are also problematic, for reasons similar to those that led to their omission from garbage collected languages: we want to allow a TM-specific memory manager to delay space reclamation until any concurrent transaction that may be using the object has aborted. This means we can't control when the destructor will execute. A similar problem occurs with constructors in our RSTM and redo-log systems: if a constructor does anything that conflicts with another transaction, and leads to an abort, normal C++ exception propagation causes a call to the destructor for the base class, followed by the (TM-specific) `operator delete`. Unfortunately, our TM-specific `operator new` has already, at this point, placed the object on a list to be deleted on abort, at which point it detects a dangling reference. This unexpected interaction was the source of the most time-consuming bug in the development of the mesh. While it could be addressed by a change to the memory management routines, it illustrates a more general problem: a library that changes such fundamental notions as the timing of storage reclamation can interact with user code in unexpected ways—ways that may be extraordinarily difficult for the programmer to diagnose.

4.3.2 Pitfalls

Because nontransactional (nonshared) objects do not revert their values on abort, care must be taken when communicating information across transaction boundaries. In particular, a transaction can safely read or write a nontransactional variable (assuming, in the latter case, it *always* writes before committing), but not both. The second-most difficult bug in the mesh had the following general form (hidden in quite a bit of other code):

```
int x = 0;
BEGIN_TRANSACTION(X)
  ...
  x++;
  ...       // an abort here is dangerous
END_TRANSACTION(X)
// x == 1 + number of post-increment aborts
```

The correct alternative looks like this:

```
int x = 0;
int x_t;
BEGIN_TRANSACTION(X)
  x_t = x;
  ...
  x_t++;
  ...
END_TRANSACTION(X)
x = x_t;    // x == 1
```

A compiler could recognize the problem here, and generate the second version of the code, though it might in the general case require run-time alias checks. Unfortunately, nothing in the smart-pointer API prevents the programmer from writing the incorrect version.

4.3.3 Fundamental Problems

The example above is a manifestation of a deeper problem: by default, data in our API are nontransational; they retain their values on abort. Only instances of classes derived from `stm::Object` revert to earlier values. Purists would argue that in the absence of open nesting, *every* object should revert its value on abort, unless the compiler can prove that it will have no effect on program behavior. Even if one believes, however, that an STM implementation should distinguish between transactional and nontransactional data (to avoid bookkeeping overhead on the latter, or to provide a simple alternative to open nesting), it seems clear that data should be transactional by default, rather than the other way around. Unfortunately, we see no way to obtain this behavior with a library-based API.

Finally, while we are able to use templates (awkwardly) to share code between transactional and privatized uses of transactional data, there is no comparable way to share code between transactional and nonshared instances of otherwise identical abstractions. For example, if we have a preexisting (nontransactional) `queue` class, we cannot share its code with a transactional queue whose components are derived from `stm::Object`. Again, it seems easy to address this problem with language and compiler support.

5. Conclusions

We have described an application programming interface for transactional memory in C++. Our interface is shared among several back-end systems, covering nonblocking

(indirection-based) and (blocking) zero-indirection software systems, coarse-grain locks, and several software/hardware hybrids. It addresses several problems encountered with an earlier, more conventional library interface, including the awkwardness of repeated `foo*` ↔ `Shared<foo>` conversions, accidental use of `foo*` references outside their transactional context, silent invalidation of `const` references when an object is upgraded to writable, and the inability to share functions between transactional and privatized code. The key to the new API is its use of C++ *smart pointers*, which provide both per-access and amortized (per reference) hooks into the back-end system.

We have used our interface to construct a parallel implementation of Delaunay triangulation. Our experience confirms that the API can be used to build nontrivial applications. It allows us to access sharable objects in both transactional and privatized contexts and, using generics, to call a common set of functions from both. At the same time, we find that the programming model supported by the API is too complex and nonorthogonal to give to naive users. Problems include remaining awkward constructs, limitations on the programming model, pitfalls in the communication of values across transaction boundaries, and fundamental issues with default nontransactional semantics and the distinction between sharable and nonsharable data.

For most of the problems we encountered, there exist straightforward language and compiler-based solutions. Among other things, a compiler can easily support real field references (instead of accessor functions); invisible post-access consistency checks; `break` and `return` inside transactions; and nontrivial constructors and nonstatic methods on transactional objects (in effect, making `this` a smart pointer). With just a bit more work, the compiler can create `clone`, `deactivate`, and `redo` methods automatically; generate alternative versions of functions that are called on both transactional and privatized data, eliminating the template hackery of Figure 5; and identify and automatically re-write code of the form shown in Section 4.3.2 (a simple dataflow problem). Finally, a compiler can (as in all existing compiler-based TM systems) arrange to revert *all* data on transaction abort (using lightweight checkpoints or undo logs for thread-local objects).

At one time, we had hoped that library-based systems might allow important classes of programmers to move to transactional memory at low investment cost. Given that the whole point of TM is to simplify concurrent programming, however, our experience suggests that this hope was naive. Interfaces like RSTM2 can remain a useful tool for systems researchers, but application programmers are going to need language and compiler support.

In future work, we hope to integrate solutions for the problems of Section 4.3 into a compiler or source-to-source translator. Toward this end, we are exploring ways to distinguish among transactional, privatized, and nonshared (thread-local) data. Questions include: Should the notion of sharing status be associated with classes, object instances, references, or some combination of these? Should it be determined statically or dynamically? How should it propagate through parameters? While these questions should probably not be exposed to programmers in most cases, it seems likely that the compiler will need to reason about them internally. Ultimately, a successful transactional language will need to combine simple, intuitive behavior (at least in the common case) with implementations that avoid unnecessary overhead when run on legacy machines.

References

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

[2] A. Alexandrescu. Smart Pointers. Chapter 7 or *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison Wesley Professional, 2001.

[3] H.-J. Boehm. Threads Cannot Be Implemented As a Library. *SIGPLAN 2005 Conf. on Programming Language Design and Implementation*, Chicago, IL, June 2005.

[4] B. Delaunay. Sur la Sphère Vide. *Bulletin of the USSR Academy of Sciences, Classe des Sciences Mathématiques et Naturelles*, 7:793-800, 1934.

[5] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. *Intl. Symp. on Distributed Computing*, Stockholm, Sweden, Sept. 2006.

[6] R. A. Dwyer. A Faster Divide and Conquer Algorithm for Constructing Delaunay Triangulation. *Algorithmica*, 2:137-151, 1987.

[7] R. Ennals. Software Transactional Memory Should Not Be Lock Free. Technical Report IRC-TR-06-052, Intel Research Cambridge, 2006.

[8] K. Fraser. Practical Lock-Freedom. Ph. D. dissertation, UCAM-CL-TR-579, Computer Laboratory, University of Cambridge, Feb. 2004.

[9] R. Guerraoui, M. Herlihy, and B. Pochon. Polymorphic Contention Management in SXM. *Intl. Symp. on Distributed Computing*, Cracow, Poland, Sept. 2005.

[10] L. Guibas and J. Stolfi. Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams. *ACM Trans. on Graphics*, 4(2):74-123, Apr. 1985.

[11] T. Harris and K. Fraser. Language Support for Lightweight Transactions. *OOPSLA 2003 Conf. Proc.*, Anaheim, CA, Oct. 2003.

[12] T. Harris, S. Marlow, S. P. Jones, and M. Herlihy. Composable Memory Transactions. *ACM Symp. on Principles and Practice of Parallel Programming*, Chicago, IL, June 2005.

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

[14] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software Transactional Memory for Dynamic-sized Data Structures. *ACM Symp. on Principles of Distributed Computing*, Boston, MA, July 2003.

[15] M. Herlihy. SXM: C# Software Transactional Memory. Unpublished manuscript, Brown Univ., May 2005.

[16] M. Herlihy, V. Luchangco, and M. Moir. A Flexible Framework for Implementing Software Transactional Memory. *OOPSLA 2006 Conf. Proc.*, Portland, OR, Oct. 2006.

[17] M. Herlihy and J. E. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. *Intl. Symp. on Computer Architecture*, San Diego, CA, May 1993.

[18] R. L. Hudson, B. Saha, A.-R. Adl-Tabatabai, and B. Hertzberg. McRT-Malloc—A Scalable Transactional Memory Allocator. *2006 Intl. Symp. on Memory Management*, Ottawa, ON, Canada, June 2006.

[19] M. Kulkarni, L. P. Chew, and K. Pingali. Using Transactions in Delaunay Mesh Generation. *Workshop on Transactional Memory Workloads*, Ottawa, ON, Canada, June 2006.

[20] V. J. Marathe, W. N. Scherer III, and M. L. Scott. Adaptive Software Transactional Memory. *Intl. Symp. on Distributed Computing*, Cracow, Poland, Sept. 2005.

[21] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer III, and M. L. Scott. Lowering the Overhead of Software Transactional Memory. *ACM SIGPLAN Workshop on Transactional Computing*, Ottawa, ON, Canada, June 2006.

[22] M. F. Ringenburg and D. Grossman. AtomCaml: First-Class Atomicity via Rollback. *ACM SIGPLAN Intl. Conf. on Functional Programming*, Tallinn, Estonia, Sept. 2005.

[23] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. McRT-STM: A High Performance Software Transactional Memory System for a Multi-Core Runtime. *ACM Symp. on Principles and Practice of Parallel Programming*, New York, NY, Mar. 2006.

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

[25] M. L. Scott, M. F. Spear, L. Dalessandro, and V. J. Marathe. Delaunay Triangulation with Transactions and Barriers. *IEEE Intl. Symp. on Workload Characterization*, Boston, MA, Sept. 2007. Benchmarks track.

[26] A. Shriraman, M. F. Spear, H. Hossain, S. Dwarkadas, and M. L. Scott. An Integrated Hardware-Software Approach to Flexible Transactional Memory. *Intl. Symp. on Computer Architecture*, San Diego, CA, June 2007.

[27] M. F. Spear, V. J. Marathe, W. N. Scherer III, and M. L. Scott. Conflict Detection and Validation Strategies for Software Transactional Memory. *Intl. Symp. on Distributed Computing*, Stockholm, Sweden, Sept. 2006.

[28] M. F. Spear, A. Shriraman, L. Dalessandro, S. Dwarkadas, and M. L. Scott. Nonblocking Transactions Without Indirection Using Alert-on-Update. *ACM Symp. on Parallelism in Algorithms and Architectures*, San Diego, CA, June 2007.

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

[30] A. Welc, S. Jagannathan, and A. L. Hosking. Transactional Monitors for Concurrent Objects. *European Conf. on Object-Oriented Programming*, June 2004.