

# DracoSTM: A Practical C++ Approach to Software Transactional Memory

Justin E. Gottschlich    Daniel A. Connors

Department of Electrical and Computer Engineering  
University of Colorado at Boulder  
{gottschl, dconnors}@colorado.edu

## Abstract

Transactional memory (TM) is a recent parallel programming concept which reduces challenges found in parallel programming. TM offers numerous advantages over other synchronization mechanisms, yet many current TM systems require complex hardware, programming language extensions, specific compiler support or enforce impractical software design, making these models unrealistic as an immediate TM solution for early adopters. Our new software transactional memory (STM) system, DracoSTM, is a high performance lock-based C++ STM research library. DracoSTM uses only native object-oriented language semantics, increasing its intuitiveness for developers while maintaining high programmability via automatic handling of composition, locks and transaction termination. DracoSTM is the STM first solution to (1) implement both direct and deferred updating and (2) enable run-time alternation between these updating policies. DracoSTM requires no language extension, specific development environment or platform, widening its usability and increasing the novelty of its design. This paper describes DracoSTM from an architectural infrastructure viewpoint. TM-specific and library-specific aspects are discussed, as are their cross-cutting design concerns. Finally, performance benchmarks are presented, showing DracoSTM outperform another high performing C++ STM library, by upwards of two orders of magnitude.

## 1. Introduction

To continue exploiting the growth in per-chip transistor count for increased performance, hardware designers have turned to building chip-multiprocessors (CMPs) [6, 12] by replicating uniprocessor cores many times on a single die. While CMP designs provide more resources for computation, they do not offer the universal performance benefits generally found in uniprocessor performance scaling. As such, parallel programming solutions must be found to meet the computational challenge and transformation introduced by CMPs [25].

Ideal parallel programming solutions are ones which function efficiently on CMP systems and are easy to implement in existing and future software systems. While conventional synchronization mechanisms, such as mutex locks, monitors and semaphores, can

efficiently utilize resources of CMP systems, their software development costs are high as they tend to introduce cumbersome and error-prone code [2, 6, 12, 15, 16, 18]. Transactional memory (TM) contends to overcome these pitfalls by improving parallel programmability and hiding the details of thread coordination behind a synchronizing system.

Conventional lock-based synchronization primitives usually restrict execution within the atomically coordinated sequence of instructions to a single thread, referred to as a critical section [27]. Within TM systems, multiple threads can simultaneously operate on the same atomic sequence of instructions. Instead of the operations being isolated in the transaction, the memory is, leading to the term transactional memory. Fundamentally, this is what makes conventional synchronization mechanisms different than TM; classical synchronization mechanisms generally isolate operations, transactional memory isolates memory.

Yet, of the available STM systems today, few are built with library design concerns as a primary factor. Much of this is due to the current TM focus being that of problem space exploration in either performance or easily achieved atomicity [2, 14, 17–19, 22, 28]. While DracoSTM was built with the prior issues in mind, it was also built to address large scale software concerns, such as: native language compatibility, type safety, memory management, legacy system integration, platform independence and simplified maintenance. As detailed in the following sections, DracoSTM achieves these goals.

In addition to programmability, many of the rich features that make TM an ideal parallel programming solution are integrated into the DracoSTM system: direct and deferred updating [26], early and late conflict detection, composition [17], extensible polymorphic contention management [13, 19], garbage collection and a multi-threaded safe free store for memory management. All of these features are implemented using widely recognized object-oriented (OO) design patterns [11] without the use of mechanisms that could reduce the safety (void pointers, preprocessor manipulation) or flexibility of the system (compiler-specific solutions, restricted development constraints). The combination of these factors improves the overall usability of the system and results in a novel design.

**Taxonomy.** When discussing TM-specific aspects, we use the taxonomy given by Larus and Rajwar’s Transactional Memory synthesized lectures [21]. When discussing C++, object-oriented (OO), or library-specific aspects, we use taxonomy given by Matthew Austern [4], Nicolai Josuttis [20], Bjarne Stroustrup [30] and Herb Sutter [31]. We use the above common taxonomy in order to improve readability for other experts in the respective TM and programming language fields.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

LCS D’07    October 21, 2007, Montréal, Canada  
Copyright © 2007 ACM ISBN 978-1-60558-088-3...\$5.00

**Approach.** The approach we take for this paper is as follows. First, a high-level overview of the DracoSTM system is given, covering both TM-specific and C++-specific components. Second, practical examples using DracoSTM are given. These examples begin with a simple transaction, then move into a more complex, composed transaction and finally into concrete priority inversion handling using client-side implemented contention management for dynamic priority assignment. An overview of the DracoSTM API is given next. Finally, we briefly discuss performance metrics of DracoSTM and RSTM, University of Rochester’s non-blocking C++ STM system in which DracoSTM outperforms RSTM by upwards of two orders of magnitude.

## 2. Background

Transactional memory was founded on the database ACID principle (atomic, consistent, isolated and durable), except without the D [24]. Transactions are atomic; the operations all commit or none of them do. Transactions are consistent; transactions must begin and end in legal memory states. Transactions are isolated; memory changes made within a transaction are invisible until committed.

The below example gives a basic introduction into DracoSTM’s transactional framework and demonstrates DracoSTM’s ACI conformance. A `native_trans` class is defined which derives from DracoSTM’s `transaction_object`, needed for interaction with DracoSTM’s `read()` and `write()` operations.

```
template <typename T> class native_trans :
public transaction_object< native_trans<T> >
{
public:
    native_trans() : value_(T()) {}
    T& value() { return value_; }
private:
    T value_;
};

native_trans<int> global_int;

int increment_global()
{
    transaction t;
    transaction_state state = e_no_state;
    int val = 0;

    do {
        try
        {
            t.write(global_int).value()++;
            val = t.read(global_int).value();
            state = t.end_transaction();
        }
        catch (aborted_transaction_exception&)
        { t.restart_transaction(); }
    } while (state != e_committed);

    return val;
}
```

In the above example, (A) both the `t.write()` and `t.read()` operations function atomically or neither operations are performed. In addition, (C) the transaction begins and ends in legal memory states, meaning `global_int` is guaranteed to be read correctly, preventing thread data races from causing inconsistent results. Lastly, (I) the intermediate state of the incremented `global_int` is isolated until the transaction commits. These three attributes fulfill Dra-

coSTM’s conformance to the ACI principles. The above example also gives a basic introduction into DracoSTM’s transactional framework.

### 2.1 TM-Specific Concepts

This section briefly discusses some of the TM-specific concepts and how DracoSTM implements each particular concept.

**STM Synchronization Types.** There are two ways STM systems synchronize memory: (1) using non-blocking mechanisms or (2) using lock-based (or blocking) mechanisms. Non-blocking STM systems use atomic primitives, such as, compare-and-swap (CAS) or load-linked and store-conditional (LL-SC), that do not lock the STM system to perform their transactional operations. Lock-based STM systems use locks, such as mutual exclusion locks, which lock the STM system to perform some portion of their transactional operations.

DracoSTM is a lock-based STM system. At its core, DracoSTM uses one lock per thread to implement transactional reads and writes. This allows multiple transactions to simultaneously read and write without blocking other transactions’ progress. When a transaction is committing, a global locking strategy is used to temporarily block forward progress on all transactions except the committing one. Once the committing transaction completes, other transactions are allowed to resume their work. DracoSTM’s lock-based strategy allows it to gain the performance benefits of a non-blocking system, such that when transactions are not committing, the transactions do not block each other and are guaranteed to make forward progress. Yet DracoSTM maintains the benefits of a lock-based system, enabling it to perform commit-time invalidation, its primary consistency model mechanism.

Recent research shows lock-based STM systems outperform non-blocking systems [9, 10]. Our own research shows that through DracoSTM’s design, scaling concerns and other lock-based specific problems, such as deadlocking and priority inversion, can be overcome with specific contention management and conflict detection policies. Some of these lock-based details are explained in detail in later sections of this work.

**Updating.** In any TM system, an updating protocol must be used to perform transactional commits for writes. Updating policies determine how a transaction commits its memory updates to global memory. Two general ways exist to perform updating: 1) direct updating, which copies the original global memory state off to the side and then writes directly to global memory, and 2) deferred updating, which copies the original global memory off to the side and the writes to the local copy. When a transaction of a direct updating system commits its changes, no changes to global memory are made as the STM system has written directly to global memory. When a transaction of a deferred updating system commits its changes, it writes the local changes to global memory. When a direct updating system aborts, it uses the original copy of memory to update global memory, restoring it to its original state. When a deferred updating system aborts, no changes to global memory are made as the STM system has not written anything to global memory. One of DracoSTM’s novel features is its implementation of both direct and deferred updating.

**Conflict Detection.** Conflict detection is the process of identifying when two or more transactions conflict. Conflicts can exist when a transaction writes to memory that another transaction then reads or writes (write after write, write after read), or when a transaction reads memory that is then used in another transaction’s write (read after write). Unlimited readers, on the other hand, can read the same piece of memory without any conflict (read after read). Before determining how to handle a conflict, STM systems must determine when they will detect conflicts. There are two primary ways

to detect conflicts: early or late. Early conflict detection attempts to identify conflicts as soon as a transaction reads or writes to memory. Late conflict detection attempts to identify conflicts some time after the initial read or write.

For direct updating, DracoSTM implements a run-time configurable early and late conflict detection mechanism. For deferred updating, DracoSTM only implements late conflict detection. The decision to have DracoSTM only support late conflict detection for deferred updating was made after identifying numerous lost optimizations using early conflict detection with deferred updating. Future work may lead to the implementation of early conflict detection for deferred updating simply for symmetry.

**Consistency Checking.** An STM system can identify a conflict in two principal ways: through validation or invalidation. Validation is the process a transaction performs on its own read and write set to check itself for consistency. Invalidation is the process a transaction performs on other transaction's read and write sets to check them for consistency. Validation strategies usually have the transaction abort itself if an inconsistency is found. Invalidation strategies usually do just the opposite, aborting the other transactions if an inconsistency is found. In addition, STM systems can use contention managers to determine how best to behave when inconsistent transactions are identified.

DracoSTM currently implements consistency checking only through invalidation. One of the next goals of DracoSTM is to build run-time configuration of consistency checking for both invalidation and validation, as it is believed that both may be necessary for varying problems. This aside, DracoSTM is unique in that it is the first STM system to implement commit-time invalidation. While other systems, such as RSTM [23], have implemented invalidation, no other system implements commit-time invalidation. We believe DracoSTM is the first commit-time invalidating system due to commit-time invalidation being seemingly only possible in lock-based STM systems and as lock-based STM systems are relatively new, other lock-based systems not being far enough along to implement it yet. The two key differences we focus on in this work between invalidation and validation are; (1) invalidation can save many wasted operations by early notification of doomed transactions, whereas validation cannot and (2) invalidation can detect true priority inversion, whereas validation cannot (other significant differences exist, but are not discussed here).

(1) Fully validating systems must iterate through all transactional operations and determine consistency only at commit-time. Thus, each transaction must fully execute its transactional operations. A substantial amount of work can be saved by an invalidating system which can flag doomed transactions early, as shown in table 1. Table 1 details 4, 8 and 12 threaded runs for red-black trees, linked lists and hash tables in DracoSTM. While the percentage of operational savings decreases for each benchmark as the structure size increases, the actual operational savings improves. For example, if a linked list is inserting at the end of a 1600 node list and receives an early termination notification saving 50% of its operations, the savings gained is an 800 node iteration and insert. Likewise, performing a 90% operations savings in a linked list insert operation of size 100, saves only a 90 node iteration and insert. Furthermore, not shown in the tables here, due to space limitations, is that abort percentages grow for each benchmark as the data structure size increases. Thus, the number of aborts increases, resulting in an even high amount of abort savings per benchmark. The increasing number of aborts as the data structure grows is quite intuitive as longer running transactions are more likely to incur collisions, especially while operating on the same data structure.

(2) Priority inversion occurs in TM when a lower priority transaction causes a higher priority transaction to abort. Furthermore, priority inversion can be guaranteed to only abort true priority in-

verted transactions in an invalidating system. However, validating systems can also build priority inversion schemes, they simply must suffer penalties of potentially aborting transactions unnecessarily. The following section gives concrete examples of handling priority inversion in both validating and invalidating models.

**Updating and Consistency Configuration.** While DracoSTM benchmarks show that, deferred updating in our system usually outperforms direct updating, this is not always the case. In particular, direct updating eventually outperforms deferred updating in DracoSTM as the data structure size grows. With this in mind, we believe that direct updating is useful for specific algorithms with highly innate parallelism (such as hash tables). Likewise, we believe validation may outperform invalidation for high thread counted uses. From these conclusions, we believe final STM systems may be required to implement (1) direct updating, (2) deferred updating, (3) validation and (4) invalidation, all of which should be configurable at run-time. By doing this, each problem which demands a different four-way configuration can be handled appropriately. Rather than attempting to build a single implementation which solves all problems universally, the end resulting STM system will handle each specific problem with the most appropriate configuration.

**Memory Granularity.** STM systems must use a memory granularity size of either word or object for transactions. Word memory granularity allows transactions to read and write at the machine's architectural word size. Object memory granularity allows transactions to read and write at the object level, usually controlled by implementation of a transactional object base class using subtype polymorphism. DracoSTM implements the latter, performing reads and writes at the object level.

**Memory Rollback Capability.** STM systems must implement memory rollback capabilities for aborted transactions. Memory rollback restores the original state of memory in the event a transaction aborts. There are three rollbacking aspects any STM system must handle when implemented in an unmanaged language; 1) updates to global, 2) allocated memory and 3) deallocated memory.

DracoSTM handles rollbacking to global memory internally for both direct and deferred updating, requiring no programmer-based code. However, allocated and deallocated memory rollbacking require programmer-specific interfaces to be used. These interfaces handle C++ memory operations in their native capacity – new and delete – as well as their transactional memory capacity, ensuring no memory is leaked nor deleted prematurely. Examples of this are presented in the following section.

**Composition.** Composition is the process of taking separate transactions and adding them together to compose a larger single transaction [17]. Composition is a very important aspect of transactions as, unlike locks, transactions can compose. Without a composable TM system, nested transactions each act independently committing their state as they complete. This is highly problematic if an outer transaction then aborts, as there may be no way to rollback the state of a nested (and already committed) transaction. Therefore, implementation of composable transactions is paramount to any TM system which hopes to build large transactions.

DracoSTM implements composition via subsumption and is a flattened nested system [21]. Composition via subsumption merges all nested transactional memory of a single thread into the outer most active transaction of that same thread. The outer transaction subsumes all the inner transactions' changes. Once the outer transaction completes, all the transactional memory from the nested transactions and their parent either commit or abort. DracoSTM's

**Table 1.** Invalidation Savings by Early Notification: % of Ops Completed By Doomed Transactions Prior To Abort

nodes (rb/ll/ht)	<i>Red Black Tree</i>			<i>Linked List</i>			<i>Hash Table</i>			<i>Averages</i>	
	4	8	12	4	8	12	4	8	12	<i>executed</i>	<i>saved</i>
1k/100/1k	0.679	0.313	0.302	0.119	0.069	0.113	0.394	0.163	0.128	0.253	<b>0.747</b>
2k/200/2k	0.488	0.385	0.283	0.112	0.279	0.274	0.145	0.280	0.265	0.279	<b>0.721</b>
4k/400/4k	0.399	0.329	0.415	0.208	0.350	0.456	0.266	0.397	0.388	0.356	<b>0.644</b>
8k/800/8k	0.373	0.508	0.350	0.113	0.588	0.532	0.267	0.515	0.515	0.418	<b>0.582</b>
16k/1600/16k	0.415	0.315	0.469	0.548	0.578	0.656	0.457	0.631	0.541	0.512	<b>0.488</b>

flattened nesting system enables each nested transaction visibility into its parent’s transactional memory and vice versa, but does not allow other transactions to see this intermediate state.

## 2.2 C++ and Library-Specific Concepts

This section briefly discusses some of the C++ and library-specific concepts of DracoSTM.

**Native Language Compatibility.** Some existing STM systems require specific language extensions or compiler support for their proposed system to work. Other systems instead violate native language pragmatics by reducing or removing type-safety altogether. Yet other system’s transactional functionality is significantly reliant on the preprocessor [23].

DracoSTM is built with native language compatibility as a top priority – as such, it does not require language extensions or violate natural language semantics. Native language compatibility is a foremost concern due to C++0x set to specifically resist language extensions [29]. While in other languages, such as Java, STM systems which require language extensions may be practical, within C++ this same approach seems unrealistic. Thus, there is a very practical need for a native language ready STM solution for C++.

**Memory Management.** For unmanaged languages like C++, STM designers can build memory managers to control heap-based memory allocation and deallocation. While building a memory manager is not necessary for STM systems, performance optimizations can be achieved through such implementations. In particular, a key memory observation for STM systems is that numerous allocations and deallocations happen within transactions, irrespective of the memory design decisions. As such, DracoSTM uses a built-in templated user-configurable memory manager which generally yields 20% performance improvement over direct calls to C++’s new and delete.

As understood by most C++ experts, native new and delete operators in C++ are multi-threaded safe, using mutex locks to guarantee memory is retrieved and released in a safe manner for multiple contending threads. Improving the performance of direct calls to C++’s new and delete in a single-threaded application is relatively easy as a buffered free store can be created which requires no locking mechanism, thus naturally increasing performance. This same task is not quite as easy in a multi-threaded environment. DracoSTM improves the native performance of C++’s operator new and delete by first implementing buffered allocations which naturally perform faster than single allocations. Secondly, performance gains are made by not relinquishing ownership of deallocated memory, making second-time memory allocations faster than first-time allocations. These two aspects enable DracoSTM’s memory manager to perform faster than C++’s native new and delete operations.

DracoSTM also must lock around memory allocations and deallocations, just as native C++ new and delete must, however, it can build a more problem-specific implementation that would hinder a

generalized C++ new and delete if implemented on a global scale. The techniques used in DracoSTM are similar to those discussed in Bulka and Mayhew’s, *Efficient C++*, Chapter 7, Multi-threaded Memory Pooling [5]. In C++ semantics, the performance gains within DracoSTM’s memory manager can be thought of as the differences between using an `std::vector`’s `push_back()` iteratively compared to using an `std::vector`’s `push_back()` iteratively after calling `reserve()`, and then continuing to reuse the allocated space to avoid performance penalties of reallocations.

**RAII.** An STM system needs a transaction interface to identify where transactions begin, end and which operations are performed within the transaction. DracoSTM achieves this by implementing transactions as objects using the Resource Acquisition Is Initialization (RAII) principle [30, 31].

RAII is a common concept in C++ when dealing with resources that need to be both obtained and released, like opening and closing a file. RAII uses the concept that if a resource is obtained it must be released even if the programmer fails to do so. RAII’s behavior is implemented per class, usually requiring the destructor of the class to guarantee any resources gathered in the lifetime of the object be released. A primary benefit of RAII is its natively correct behavior in the event of exceptions. If an exception occurs causing an RAII class instance to destruct, due to stack unwinding, the deterministic destruction of the object is invoked. The destructor then releases any resources previously collected. This guarantees any object implementing RAII semantics will always release resources it controls, irrespective of program flow (normal or abnormal).

The DracoSTM’s transaction class is based on the RAII concept for two primary reasons. First, C++ programmers implicitly understand stack based (automatic) objects and their native RAII semantics. In fact, all of C++’s Standard Template Library (STL) containers are implemented using the RAII philosophy. Second, exceptions in C++ are not required to be handled by the programmer as they are in other languages, like Java. Using RAII for transactions ensures proper and guaranteed termination of transactions regardless of program flow, a very important attribute for correct transactional behavior.

**Exception Safety.** DracoSTM fulfills Abrahams’ basic exception safety guarantee for deferred updating, but cannot supply any exception safety guarantee for direct updating. The basic guarantee for exception safety states that if an exception is thrown, the operation may have side-effects but is in a consistent state [1, 30]. The basic guarantee is less strict than the strong guarantee which specifies if an exception is thrown there are no side-effects. The highest level of exception safety, above the strong guarantee, is the nothrow guarantee which disallows exceptions from being thrown entirely.

Within deferred updating, DracoSTM can only afford to implement the basic guarantee because if memory is partially committed and then a user exception is thrown, no original state exists for the already committed memory. Therefore, already committed mem-

ory in a deferred updating system, must stay committed since no reverted original state can be used to revert the changes. To implement such a system would result in a substantial performance degradation to the overall system, effectively doubling memory size and copy operations. Due to these costs, a double-copy implementation is not performed and the basic guarantee for deferred updating is deemed acceptable.

Within direct updating, memory updates are done immediately on global memory, so transactions naturally achieve strong exception safety guarantees for commits. Aborts within direct updating, however, invoke copy constructors for restoration of the original global memory state. These copy constructors can throw exceptions which then can lead to a partially restored global state for aborted exceptions that are short-circuited by user-defined copy constructor exceptions. As such, no exception safety guarantee can be made for direct updating when used in C++, a downfall of the updating policy.

**Parametric Polymorphism and Subtype Polymorphism.** Type abstraction in C++ to create general purpose code can be achieved in numerous ways. Some of these ways, such as the use of C++ template classes and template functions (parametric polymorphism), as well as inheritance (subtype polymorphism), are considered practical and robust ways to build general purpose functionality while still ensuring a certain degree of type-safety is maintained. C++ templates, also known as parametric polymorphism, exhibit the same type-safety as if the general purpose code was written specifically for the templated instantiated type. Inheritance, on the other hand, reduces type-safety to some degree, but gains run-time flexibility unachievable with C++ templates alone. Other mechanisms also exist to create general purpose code, such as void pointers or preprocessor macros, but are considered unsafe and error-prone [8] and thusly, not used in DracoSTM.

DracoSTM uses both parametric and subtype polymorphism throughout its internal implementation and exposed interfaces. In cases where strict type-safety can be achieved, C++ templates are used. In other cases where exact type-safety cannot be achieved without reducing DracoSTM's functionality, inheritance is used. All of these factors considered, DracoSTM is a research library that requires type-safety to be a foremost concern, as its usage would be hampered if type-safety was relaxed in areas where it could have been retained. As such, C++ templates are used due to their retention of full type information, in cases where inheritance would have also sufficed with a slight loss of type-safety.

### 3. DracoSTM in Practice

A number of example transactions are presented in this section using the DracoSTM library. The first example illustrates how to write a transactional linked list insert operation. The second example demonstrates composition, combining a transactional insert operation with a transactional remove operation which compose into a larger, single move transaction. Next, a minor but important detail regarding memory addresses within the transactional workspace is given. Finally, an example of how to handle priority inversion for validating and invalidating consistency schemes using DracoSTM's extensible contention manager and compositional framework is provided. The final example demonstrates a number of important aspects of DracoSTM's implementation, such as, differing priority inversion mechanics for different consistency models, transactional attribute enrichment via composition and threaded memory sharing amongst transactions.

The DracoSTM interfaces used in the below examples are defined in the following DracoSTM API section. While most of the examples are intuitive and a complete understanding of the DracoSTM API is not needed for a high-level understanding of its

functionality, a complete description of all interfaces used below can be referenced in the following API section.

#### 3.1 A Simple Transaction

In this example, we build a linked list insert transactional operation using DracoSTM. The example is shown in three segments: (1) the client code which inserts 100 items into the list, (2) the insert operation which client code calls, (3) the internal insert operation which the exposed insert operation calls.

##### 1. Client Invoked Inserts.

```
linked_list<int> llist;
list_node<int> node;

for (int i = 0; i < 100; ++i)
{
    node.value() = i;
    llist.insert(node);
}
```

After inspecting the above client invoked insert code it is apparent that the code itself shows no signs of being transactional. This is our desired behavior. As far as the client side programmer is concerned, there is no additional code needed to perform a transactional linked list insert over a non-transactional linked list insert. Obviously, this simplistic behavior eases the introduction of TM solutions into algorithms of new and legacy systems.

##### 2. Exposed Insert.

```
bool insert(list_node<T> const &node)
{
    bool success = true;
    transaction_state state = e_no_state;

    do {
        try { state = internal_insert(node, success); }
        catch (aborted_transaction_exception&) {}
        if (!success) return false;
    } while (state != e_committed);

    return true;
}
```

The exposed insert code performs two key operations: (1) it retries the transaction until it succeeds (commits) and (2) it catches aborted transaction exceptions. The retry code is perhaps the largest visible section of code overhead for the transactional linked list insert operation. While there are other C++ mechanisms to retry transactions, like `gotos` or macro-based approaches, we believe a simple loop is currently the best solution for TM retry behavior in C++. Others before us have implemented differing solutions that have smaller code footprints, but violate large-scale design concerns, break compositionality potential and hide or impose large language penalties. As such, we currently accept the loop overhead as a small inconvenience and avoid breaking language semantics.

The `aborted_transaction_exception` allows DracoSTM to be exception neutral while also gaining performance benefits of early notification of doomed transactions. The above example demonstrates this behavior in practice with its absorption of aborted transactions and only aborted transactions.

### 3. Internal Insert.

```
transaction_state internal_insert
(list_node<T> const &rhs, bool &success)
{
    transaction t;
    list_node<T> *prev = &t.read(head_),
    *cur = t.read_ptr(head_.next());
    list_node<T> *node = t.new_memory_copy(rhs);

    if (!cur->next()) t.write_ptr(cur->next(node, t);
    else {
        node->next_for_new_mem(cur);
        t.write_ptr(prev->next(node, t);
    }
    return t.end_transaction();
}
```

The above example illustrates the simplicity of DracoSTM transactions and their interfaces. The transactional `internal_insert()` is nearly identical to a non-transactional implementation with the exception of the `transaction`, some annotations and a call to `end_transaction()`. The templated functions within the `transaction` class ensure type-safety is maintained without any necessary type-casts. Due to exact type correctness, as demonstrated in the calls to `write_ptr()`, daisy-chained method invocation can be performed allowing streamlined usage. These aspects help make DracoSTM transactions small and easy to understand.

One minor, but vital, detail is in the difference between `next()` and `next_for_new_mem()`. Rather than hide this difference, it is intentionally exposed here to draw out the memory access differences required for writes to new and existing memory. We explain this difference in detail later in this section.

### 3.2 A Composable Transaction

The below example builds upon the previous example by adding a remove operation. We combine the insert and remove operations and build a transactional move operation that compose into a single transaction. Composition is a key aspect for TM systems. DracoSTM's ability to compose transactions from pre-existing transactions is fundamental to its design.

In the following example, the first section shows client code invoking the move operation. Next, the internal remove operation is shown, demonstrating its transactional independence. Last, the external move operation is explained, combining the internal insert and remove linked list operations resulting in a composed, single transaction.

#### 1. Client Invoked Inserts / Moves.

```
linked_list<int> llist;
list_node<int> node1, node2;

for (int i = 0; i < 100; ++i)
{
    node1.value() = i;
    llist.insert(node1);
}
// 0 -> 0, 1 -> -1, 2 -> -2, ...
for (int j = 0; j < 100; ++j)
{
    node1.value() = j;
    node2.value() = -j;
    llist.move(node1, node2);
}
```

The client invoked inserts and moves are fairly straight forward. The insert operations are performed first then the original items are moved to a new location by inverting their value. Again, from a client programming perspective, there is no hint that this code is transactional, which is our intended goal.

#### 2. Internal Remove.

```
transaction_state internal_remove
(list_node<T> const &rhs, bool &success)
{
    transaction t;
    list_node<T> *prev = &t.read(head_);

    for (list_node<T> *cur = prev; cur != NULL;
         prev = cur, cur = t.read(*cur).next())
    {
        if (cur->value() == rhs.value())
        {
            t.write(*prev).next
            (t.read_ptr(cur)->next(), t);
            t.delete_memory(*cur);
            return t.end_transaction();
        }
    }

    success = false;
    return e_aborted;
}
```

As was the case with the `internal_insert()`, the above `internal_remove()` method is almost identical to how a normal linked list remove operation would be implemented, with the exception of the `transaction` object and a few DracoSTM API calls. Again, this is ideal, as it leads to intuitive transactional programming, requiring only a minor learning curve for the algorithms developer.

#### 3. Composed External Move.

```
bool move(list_node<T> const &node1,
          list_node<T> const &node2)
{
    bool success1 = true, success2 = true;
    transaction_state state = e_no_state;

    do {
        try {
            transaction t;
            internal_remove(node1, succeeded1);
            internal_insert(node2, succeeded2);
            state = t.end_transaction();
        }
        catch (aborted_transaction_exception&) {}

        if (!success1 || !success2) return false;
    } while (e_committed != state);

    return true;
}
```

The external `move()` interface requires the most code overhead not required in non-transactional implementations, yet the `move()` operation is fairly simple. In this example, there are two booleans which determine the success of the operations, one for remove and one for insert. In addition, the `transaction_state` stores the result of the `end_transaction()`. Yet, this time the `transaction_state` is obtained from a local transaction instance call to `end_transaction()`. The remainder of the code

is functionally the same as the prior example found in the external `insert()` interface.

The above move example demonstrates how two independent transactions can be composed together forming a single, larger transaction. All of the composition is achieved in a highly intuitive manner that does not require any changes to the original independent transactions. This is a primary goal of DracoSTM as it enables legacy transactional code to be extended seamlessly.

**Decomposing the Composition.** Understanding the high-level details of the composed `move()` transaction can be done in a few steps. First, the `move()` operation constructs a parent transaction that encapsulates the transactions of `internal_remove()` and `internal_insert()`. Second, the internal transactions are called between the `move()`'s transaction construction and its call to `end_transaction()`. Finally, a call to `end_transaction()` is made on the parent transaction. This signals the completion of the composed transaction, thus any intermediate state constructed from the nested transactions within the lifetime of the parent's transaction is committed to global memory and the new state is made visible to other threads.

When the children transactions of `internal_remove()` and `internal_insert()` are constructed they communicate with the parent transaction and share its read and write sets. When the first child calls `end_transaction()`, its read and write sets are simply handed off to the parent transaction, which exists for the duration of both transactions. Then, when the second child transaction is constructed, from the `internal_insert()` call, it shares the read and write set of the parent transaction. This contains the intermediate state of memory changed by the `internal_remove()` operation. Once `end_transaction()` is called from `internal_insert()`, the read and write sets are again handed off to the parent transaction of the `move()` method. Finally, when the parent transaction calls `end_transaction()`, the transaction commits or aborts. As no further transactions exist on the stack for the specific thread, the cumulation of state gathered from the three transactions are then committed to global memory or aborted in their entirety.

From the C++ programmer perspective, in order to compose multiple transactions into a single transaction all he or she needs to do is have one transaction wrap the other transactions. Once the transactional operations are complete, the outer transaction need simply make a call to `end_transaction()` which signals the composed transaction is complete.

DracoSTM implements flattened nesting, which requires all nested transactions to abort if a single transaction within the nested transactions aborts. This keeps the linked list in a consistent state. Even if the transaction within `internal_insert()` aborts after `internal_remove()` succeeds, the changes from `internal_remove()` are not committed. Furthermore, due to the flattened-nesting of the transaction, no other threads or transactions outside of the current transaction see any intermediate state leading up to the fully committed removed and re-inserted state. This further reinforces DracoSTM's ACI (atomic, consistent and isolated) conformance.

**A Minor, but Important Detail.** The first example in this section noted a subtle parameter difference between `next()` and `next_for_new_mem()` for the linked list insert operation. The overall reason for this is based in the copy semantics employed by deferred updating. Recall that deferred updating makes a copy of memory and returns the new copy when a write attempt is made on an object. The local copy is what is used for most operations. When a sequence of operations are performed requiring sequential transactional writes these local copies must be used to maintain a consistent state. For example, consider the below code:

```
native_trans<int> x = 0, y = 0, z = 0;
transaction t;
t.write(x).value() = 5;
t.write(y).value() = t.read(x).value() + 1;
t.write(z).value() = t.read(y).value() + 1;
t.end_transaction();
```

The above code demonstrates why local modifications to memory must be used within a transaction for the data to be consistent. Before the transaction begins, x, y and z are 0. The first transactional operation sets x to 5. The following operation sets  $y = x + 1$ , then  $z = y + 1$ . Intuitively, we can see y should be set to 6 and z should be set to 7. However, if the `read()` operation for the transaction does not return the local copy of x, but instead returns the global value of x, y would be set to 1. The same would result for z because of y's global value of 0.

Clearly this is incorrect behavior. As such, in nearly all deferred updating cases, the local copy of memory from a transactional write is the value needed for read operations within that transaction. However, there are some very subtle cases where this behavior is incorrect.

Consider the prior linked list move operation. In particular, consider the case where node 1 is being removed and node -1 is being inserted. The sequence of operations is as follows:

```
global list: h -> 0 -> 1 -> 2 -> ...
expected list: h -> -1 -> 0 -> 2 -> ...

1. remove node val 1
   a. point node val 0 to node val 2
      (creates copy of node val 0, which
       now is local ref for node 0)
   b. delete node val 1

   local list: h -> L0 -> 2 -> ...
   global list: h -> G0 -> 1 -> ...

2. insert node val -1
   a. new transaction mem for node val -1
   b. point head node to node val -1
      (creates copy of head node, which now
       is local ref for head node)
   c. point node val -1 to node val 0
      (node val -1 points to local node val 0)

   local list: Lh -> -1 -> L0 -> 2 -> ...
   global list: Gh -> G0 -> 1 -> ...

3. commit transaction
   a. copy local node 0 to global node 0
      global list: h -> 0 -> 2 -> ...
   b. delete local node 0

      ERROR: node val -1 was pointing to local
             node val 0

   c. copy local node h to global node h

global list: h -> -1 -> deleted memory
```

As can be seen from the above illustration, step 3.b deletes memory which was being referenced by node val -1. At first observation, it may seem as though the problem lies here. Further investigation, however, reveals that node val -1 should have never pointed to a locally copied node 0, as local copies are deleted in deferred updating once the transaction commits. Instead, node val

-1 should have instead pointed to the true address (global address) of node 0.

This minor but critical observation is why the `next()` method for the `list_node` is slightly more complex than we might otherwise suspect. Additionally, it also explains why there are two methods; one for pre-existing transactional memory and one for newly allocated transactional memory. Newly allocated transactional memory always references true global memory, thus when a pointer is being set to it, it can do so directly. However, pre-existing transactional memory may have been modified within the transaction and thus may have a local copy. Pointing to the address of this local copy is incorrect and must be programmatically avoided.

However, as shown in the `x, y, z` example, normal behavior requires local copies to be used. What is seen here is that only when transactions reference the address of a transactional piece of memory, should they refer to the global piece of memory. In all other cases that we have observed, the local copy should be referenced. Below gives the actual implementation used for the `list_node` `next()` and `next_for_new_mem()` used in the previous examples.

```
void next(list_node *rhs, transaction &t)
{
    if (NULL == rhs) next_ = NULL;
    else next_ = &t.find_original(*rhs);
}
void next_for_new_mem(list_node *rhs)
{
    next_ = rhs;
}
```

As demonstrated above, an important interface needed for transactions is `find_original()`, which is described in further detail in the DracoSTM API section. New memory can also be passed to `find_original()` which simply returns the memory passed in. Therefore, it is possible to build a single `next()` method for both new memory and existing memory, simplifying the `list_node` implementation. However, such usage incurs an unnecessary functional call penalty for new memory. While this penalty may be insignificant, it can be side-stepped entirely when performing new memory address references. The importance of this example is to explain the underlying complexity of referencing global addresses of transactional memory and clarify that new and existing memory behave in functionally different ways within transactions. With this aside, it is recommended that only one `next()` method be implemented in normal client code to reduce code complexity, even though `find_original()` is unnecessary for new memory.

### 3.3 A Dynamically Prioritized, Composed Transaction

The following subsection discusses how priority inversion is handled within DracoSTM using dynamic priority assignment. Two solutions are presented for the different consistency models, one for validation and one for invalidation. Following the two examples which detail how to override contention management interfaces, a dynamically prioritized transaction is presented, demonstrating how transactions interact with the prior implementations.

Priority inversion in transactional memory occurs when a lower priority transaction causes a higher priority transaction to abort. With STM lock-based (and non-blocking) systems, priority inversion does not happen on the same scale as that of direct lock-based solutions. The different cases of priority inversion between direct locking solutions and TM solutions are due to TM's natural avoidance of critical sections. However, priority inversion in TM can easily occur if, for example, a long running transaction is continually preempted by shorter running transactions which always commit before the longer transaction.

In order to prevent such priority inversion scenarios, two extensible contention manager (CM) virtual methods are provided to allow client-side implementations a way to handle different scenarios based on the consistency model currently in use. The first interface, `abort_before_commit()`, allows a user-defined contention manager mechanism to abort a transaction before it commits. Although DracoSTM does not yet implement validation, once it becomes available, client-side validating algorithms which want to avoid priority inversion will need to override `abort_before_commit()` to iterate over in-flight transactions and abort the current in-process transaction if another in-flight transaction exists of higher priority. All in-flight transactions can be accessed by a call to `in_flight_transactions()` which returns the set of active transactions. As such, one could build an overridden `abort_before_commit()` which always caused lower priority committing transactions to abort in the event a higher priority transaction is currently in-flight. One possible implementation is shown below. For code simplicity, the following code has removed some static class accessors and namespaces.

#### 1. Priority Inversion for Validating Consistency.

```
class priority_cm :
    public core::base_contention_manager
{
public:
    // method invoked prior tx commit
    bool abort_before_commit(transaction const &t)
    {
        in_flight_transaction_container::const_iterator
            i = in_flight_transactions().begin();

        for (; in_flight_transactions().end() != i; ++i)
        {
            if (t.priority() < (*i)->priority())
            {
                return true;
            }
        }
        return false;
    }
};
```

While the above approach is necessary for a validating system, it is largely a poor way to perform priority inversion checking. Firstly, it has the side-effect of causing unnecessary aborts for transactions which do not necessarily conflict, but simply have differing priority levels. Secondly, it is slow in that all transactions must be walked through each time a transaction commits. However, as an ad hoc solution for a validating system, the above priority inversion mechanism may be as close to correct as is possible. This solution is useful for validation, but should never be used for invalidation. Instead a second and more natural approach for invalidating systems to prevent priority inversion is to override the `permission_to_abort()` interface. The `permission_to_abort()` interface can only be used when DracoSTM is performing invalidation.

The `permission_to_abort()` interface is called from DracoSTM's `end_transaction()` method when a committing transaction has found a second transaction it needs to abort for consistency. As such, the method takes two parameters, an lhs (left-hand side), the committing transaction, and an rhs (right-hand side), the transaction requested to be aborted. If permission is granted to abort the second transaction, the method returns true and the second transaction is aborted. If permission is not granted to abort the second transaction, the method returns false and upon returning the committing transaction aborts itself. All consistency checking for

deferred updating is performed prior to any updating operation and thus memory is still in a completely legal uncommitted state until all consistency is performed. For direct updating aborts, the system simply follows its normal semantics of aborting the transaction by restoring global memory to its original state. Similar to the prior example, overriding the `permission_to_abort()` method can be done in such a manner which prevents lower priority transaction from aborting a higher priority transaction as shown below:

### 2. Priority Inversion for Invalidating Consistency.

```
class priority_cm :
    public core::base_contention_manager
{
public:
    // method invoked before lhs transaction
    // aborts rhs transaction
    bool permission_to_abort(transaction const &lhs,
        transaction const &rhs)
    {
        return lhs.priority() >= rhs.priority();
    }
};
```

With priority inversion preventable for both validating and invalidating consistency modes, transactions now need some mechanism to control their priority. DracoSTM allows for such control through the `raise_priority()` interface. By iteratively calling `raise_priority()`, preempted transactions can raise their priority at each preemption ensuring their eventual commit. The `raise_priority()` interface is implemented using a `size_t` type. Additionally, DracoSTM supplies a `set_priority()` interface taking a `size_t` parameter allowing client code to set the priority directly.

In order for `raise_priority()` to function correctly, the affected transaction must not be destroyed upon transactional abort. If the prioritized transaction is destroyed at each transactional abort, `raise_priority()` will only raise the transaction's priority by one each time. In order to demonstrate how `raise_priority()` can be used in practice, we use a wrapper transaction around the `internal_insert()`'s transaction. However, in this case the wrapper transaction is not destroyed upon successive iterations.

The `restart_transaction()` interface must be called for transactions that are not destroyed after being aborted. This necessary step clears the state from the previously failed transactional run. As shown in the below code, the `restart_transaction()` is only called when an aborted exception is caught. This is because `end_transaction()` throws an exception when the transaction is aborted. Following this implementation paradigm, handling aborted transactions is relatively straightforward as all aborted transactions follow the same exception-based path.

The below example combines all of these aspects together into a dynamically prioritized composed transaction. The composition is slightly different than what has been shown previously - instead of using composition for wrapping two methods into a larger transaction, we use composition to override the internal transaction's implementation to improve the richness of its behavior, a relatively novel concept for composition.

### 3. Dynamically Prioritized Composed Transaction.

```
bool insert(list_node<T> const &node)
{
    bool success = true;
    transaction_state s = e_no_state;
    transaction t;
    for (; s != e_committed; t.raise_priority())
    {
```

```
        try
        {
            internal_insert(node, success);
            s = t.end_transaction();
        }
        catch (aborted_transaction_exception&)
        { t.restart_transaction(); }

        if (!success) return false; // on list
    }
    return true;
}
```

The above example demonstrates a number of important concepts with DracoSTM's extensible contention manager and its implementation of composition. First, it shows how to avoid priority inversion, using dynamically prioritize transactions, in conjunction with a prioritized overridden contention manager for both validation and invalidation consistency schemes. Second, it demonstrates how transactions which are aborted but not destroyed can be restarted with the aborted transaction catch clause. Third, the example explains how ordinary transactions can be enriched by layering composed transactions on top of them without changing the underlying original code. Lastly, it reveals some of DracoSTM's internal priority processing which requires an additional amount of explanation, as follows.

**4. DracoSTM's Internal Write-Write Abort Process.** As the above priority assigned transaction demonstrates, the outer transaction has increasing priority while the inner transaction, the one within `internal_insert()`, does not. Yet, the inner transaction is not aborted due to the outer transaction's priority. This is handled internally via DracoSTM's abort process by two fundamental ideas. (1) As previously explained, all transactions of the same thread share transactional memory, this allows the outer transaction to be seen as using the same memory as the inner transaction. Thus, when the inner transaction is flagged to be aborted, the outer transaction must also be flagged to be aborted as well, since it would have the same memory conflicts. However, when checking the outer transaction's priority, the contention manager's priority method would see the outer transaction as having higher priority than the committing transaction if it had already been aborted once and the committing transaction had not. The priority analysis of the outer transaction compared to the committing transaction would thereby force the committing transaction to abort instead of the outer transaction. (2) DracoSTM's abort mechanism does not abort any transactions until it has walked all transactions, passing all the `permission_to_abort()` checks. Therefore, even if the inner transaction is flagged to be aborted, since all transactions must be successfully walked in order to abort any transaction, the outer transaction's priority will cause the committing transaction to abort, thereby saving the inner transaction from being affected. An example of this, taken directly from DracoSTM's implementation, is shown below (some code has been removed or shorted to simplify the example):

```
void abort_conflicting_writes_on_write_set()
{
    trans_list aborted;
    // iterate through all tx's written memory
    for (write_set::iterator i = writes().begin();
        writes().end() != i; ++i)
    {
        // iterate through inflight transactions
        for (trans::iterator j = inflight_.begin();
            inflight_.end() != j; ++j)
        {
```

```

transaction *t = (transaction*)j;

// if writing to this write_set, store it
if (t->writes().end() !=
    t->writes().find(i->first))
{
    if (cm->permission_to_abort(*this, *t))
        aborted.push_front(t);
    else
        throw aborted_transaction_exception("");
}
}
}
// ok, forced to aborts are allowed, do them
for (trans_list::iterator k = aborted.begin();
    aborted.end() != k; ++k)
{
    (*k)->forced_to_abort() = true;
}
}

```

**Priority Inversion Allowed.** From the above code examples, one may question why the default behavior implemented within DracoSTM does not automatically integrate priority into transactions, as it could be integrated within `restart_transaction()`. First, each problem is different and integrating priority only into `restart_transaction()` would not cover all cases (e.g., when the outer transaction was terminated). Second, building an automatic priority inversion handling scheme would eliminate some of the natural optimizations granted from different updating policies. For example, deferred updating allows multiple writers of the same memory to execute simultaneously. This behavior enables deferred updating the ability to process the fastest completing transactions first. If a priority system was integrated directly into DracoSTM, this optimization would be lost. In addition, direct updating optimizes writes by writing directly to global memory. As such, direct updating suffers greater penalties for aborted transactions due to required restoration of global memory. In this case, more transactional aborts would occur if DracoSTM built-in a default priority inversion handler. Considering these factors, as well as many others, DracoSTM does not build transactional priority into its system. Instead, we leave this implementation up to client-side implementors, as they will have a better understanding of their problem domain and be able to more correctly implement the right contention manager for their specific needs.

**The Future of Parallel Programming.** An important distinction regarding priority within transactions versus priority within more classical synchronization mechanisms, like locks, is that same functional units can be executed simultaneously by different threads yielding different priorities. For example, two threads can be executing the above insert transaction, one thread which has just begun its first run will have a priority of 0, while a second transaction which has attempted to run the insert operation 99 times previously, would have a priority of 99. The important distinction here is that classical critical section synchronization mechanisms can have only a single priority per functional unit (e.g., insert, remove, lookup operation) due to the innate limitations of single thread critical section execution. With transactions, this limitation is removed and new concepts of priority begin to emerge. Priority inversion can then extend beyond its traditional meaning and extend into a new category which incorporates differing priority within the same functional unit. These new concepts may reshape the way classical parallel problems are thought of in the future, especially in relation to transactional memory.

## 4. DracoSTM API Overview

The following section presents the major design components identified in the background section and discusses how they are implemented within the DracoSTM library.

**Direct and Deferred Updating.** One of DracoSTM's novel aspects is its capability of run-time switching between direct and deferred updating. The only restriction to this is that no transactions can be in-flight when the updating model is switched. To switch between updating models and identify which updating model is active, four static interfaces are supplied from the transaction class:

- `bool do_direct_updating()`  
Attempts to switch to direct updating. Returns `false` if in-flight transactions are found and DracoSTM is unable to switch updating models. Otherwise, returns `true` and enables direct updating.
- `bool do_deferred_updating()`  
Attempts to switch to deferred updating. Returns `false` if in-flight transactions are found and DracoSTM is unable to switch updating models. Otherwise, returns `true` and enables deferred updating.
- `bool direct_updating()`  
Returns `true` if direct updating is active, otherwise returns `false`.
- `bool deferred_updating()`  
Returns `true` if deferred updating is active, otherwise returns `false`.

**Early or Late Conflict Detection.** Four additional static interfaces within the transaction class are available to control early and late conflict detection at run-time. Recall that DracoSTM does not allow for early conflict detection for deferred updating and thus the API prevents this behavior. Additionally, conflict detection changes are not allowed while transactions are in-flight. A summary of the available interfaces is as follows:

- `bool do_early_conflict_detection()`  
Attempts to switch to early conflict detection. Returns `false` if in-flight transactions are found or if deferred updating is active. Otherwise returns `true` and enables early conflict detection.
- `bool do_late_conflict_detection()`  
Attempts to switch to late conflict detection. Returns `false` if in-flight transactions are found, otherwise returns `true` and enables late conflict detection.
- `bool early_conflict_detection()`  
Returns `true` if direct updating is active and early conflict detection, otherwise returns `false`.
- `bool late_conflict_detection()`  
Returns `true` if deferred updating is active or if direct updating and late conflict detection are active, otherwise returns `false`.

Currently, late conflict detection within DracoSTM is only performed at commit-time. While it is possible to perform late conflict detection at times other than commit-time, we have found that commit-time conflict detection yields a number of optimizations that do not exist elsewhere. In the future, we may extend run-time configuration of late conflict detection to locales other than commit-time.

**Invalidation Consistency Checking.** A key class of DracoSTM's commit-time invalidation is the `aborted_transaction_exception`. Its skeletal structure is as follows:

```

class aborted_transaction_exception
: public std::exception
{
public:
    aborted_transaction_exception
        (char const * const what);

    virtual char const * what() const;
};

```

When a committing transaction flags other transactions as being inconsistent, it does so by setting their `forced_to_abort` flag to `true`. Each time a transaction performs a transactional read, write or attempts to commit, it first checks its abort flag. When a transaction sees it has been forced to abort, it immediately calls the contention manager for directions on how to proceed.

The default behavior for an aborted transaction is to sleep for a brief period of time, to avoid creating further contention (memory-based and lock-based), and then throw the aborted transaction exception. Throwing an exception helps the transaction's overall performance as it enables early aborts for doomed transactions as shown in table 1, which would otherwise perform useless operations, as the transaction must eventually abort.

The DracoSTM `aborted_transaction_exception` type is important to the exception neutrality of DracoSTM. It signals a specific type of exception which can be caught directly by transaction code that is guaranteed to not interfere with other exception types thrown from within the STM system that are caused by client-side code (i.e. copy constructors, `copy_state()` calls, etc.).

**Object Granularity: Part I.** DracoSTM implements memory reads and writes within transactions at the object level. The `base_transaction_object` is the base class used for this behavior. A single virtual function must be overridden by the client code which inherits from it, in order for user-defined classes to be used within transactions. A high-level analysis of the `base_transaction_object` is shown below:

```

class base_transaction_object
{
public:
    base_transaction_object();
    virtual ~base_transaction_object();

    virtual void copy_state
        (base_transaction_object const * const rhs) = 0;

    static void* retrieve_mem(size_t size);
    static void return_mem(void *mem, size_t size);
    static void alloc_size(size_t size);
};

```

- `base_transaction_object()`  
Default constructor (ctor) with no parameters allows derived `base_transaction_objects` to implicitly construct the `base_transaction_object` base class for easier integration.
- `virtual ~base_transaction_object()`  
Virtual destructor (dtor) ensures correct destructors are called in the inheritance hierarchy for delete operations invoked on `base_transaction_object` pointers, which occur in numerous places throughout the internal transaction code.
- `virtual void copy_state(base_transaction_object const * const)`  
Pure virtual method which is the only required user-defined function for derived transaction objects. Derived classes usually

simply override this method and perform an `operator=()` function call for the specific derived type. The `copy_state()` method is called each time global memory is updated, for either direct or deferred updating policies. With this in mind, it is vital that the `this` object be set to the exact state of the input parameter.

- `void* retrieve_mem(size_t)`

Static interface into DracoSTM's memory management system for retrieving memory. The supplied parameter is the requested block size, the return parameter is a `void*` to the allocated block. Usually access to this interface is done by overloading operator `new` in the derived `base_transaction_object` class. Void pointers are the natural and preferred manner to handle memory allocations and deallocations and are therefore safe in this context.

- `void* return_mem(void*, size_t)`

Static interface into DracoSTM's memory management system for returning memory. The first parameter points to the memory block being returned, the second parameter specifies its size. Usually access to this interface is done by overloading operator `delete` in the derived transaction object class. Void pointers are the natural and preferred manner to handle memory allocations and deallocations and are therefore safe in this context.

- `void alloc_size(size_t)`

Static interface into DracoSTM's memory management system which allows the user to specify the allocation chunk size for the memory manager. The input parameter specifies the number of transactional objects that should be allocated at startup and with each subsequent buffer increase. If no size is specified, the allocation chunk size uses DracoSTM's default value, currently 8192. The `alloc_size()` interface can be reconfigured at runtime and is used upon the next buffer increase.

To further simplify the usage of DracoSTM, an intermediate template class was built which is meant to sit between the `base_transaction_object` and the user-defined transaction objects. The purpose of this intermediate class is to reduce the code overhead needed for user-defined transaction objects. To do this, the curiously recurring template pattern developed by James Coplien was used [7]. Its implementation is shown below:

```

template <class Derived> class transaction_object :
public base_transaction_object
{
public:
    virtual void copy_state
        (base_transaction_object const * const rhs)
    {
        static_cast<Derived *>(this)->operator=
            (*(static_cast<Derived const * const>(rhs)));
    }

    void* operator new(size_t size) throw ()
    { return retrieve_mem(size); }

    void operator delete(void* mem)
    {
        static Derived elem;
        static size_t e_size = sizeof(elem);
        return_mem(mem, e_size);
    }
};

```

With the templated `transaction_object`, client-side transaction objects need only to derive from it and pass their class type as its template parameter. At compile-time the `transaction_object` generates the necessary code to override `copy_state()` and implement operator `new` and operator `delete` using DracoSTM's memory manager for all user-defined types derived from it.

Below is an example of a user-defined class deriving from `transaction_object`:

```
template <typename T> class list_node :
public transaction_object < list_node<T> >
{
    // copy_state, operator new & delete are
    // defined using the curiously recurring
    // template pattern in transaction_object
private:
    T value_;
    list_node *next_;
};
```

The `list_node` class does not define an `operator=()` (which `transaction_object` uses), yet the above code is fully functional. This is due to the C++ standard requiring standard conforming compilers to synthesize certain class operators, such as the default constructor, copy constructor, destructor and `operator=()` [3]. The above `list_node` invokes the compiler synthesized `operator=()`, gaining access to an optimized object copy while simultaneously reducing the maintenance required by implementing `operator=()` directly.

To the best of our knowledge, using the curiously recurring template pattern for our `transaction_object` makes DracoSTM the least intrusive STM system to date, requiring the smallest amount of client-side code needed for making objects transactional.

**Object Granularity: Part II.** While in most cases user-defined transaction objects should derive from the `transaction_object`, as the client-side code is significantly simplified, some cases may exist where doing so would break the existing system's framework. Embedded software systems usually require memory allocations to be done prior to normal program execution due to the overhead of global locking needed for memory allocation and deallocation. As such, these systems usually require the use of only their own free store. Other large scale systems may hide (make private) copy constructors or `operator=()` for certain classes due to their singleton-like behavior [11]. As such, designers of these system may be reluctant to make an `operator=()` visible due to their potential misuse from other software engineers.

Taking heed to the above circumstances, DracoSTM allows client code to derive directly from the `base_transaction_object` and override `copy_state()` in an explicit user-defined manner. Derivation of the `base_transaction_object` also allows the client-side code to control how `new` and `delete` function for the specific user-defined transaction object. An example of derivation from the `base_transaction_object` is as follows:

```
template <typename T> class list_node :
public base_transaction_object
{
public:
    void copy_state(base_transaction_object
        const * const rhs);

    // operator new and delete do not need to
    // be defined, but can be user controlled

    void* operator new(size_t size) throw ();
    void operator delete(void* mem);
```

```
private:
    T value_;
    list_node *next_;
};
```

While the above code does require the overriding of the `copy_state()` method it enables legacy systems to retain complete control of their derived transactions objects. In the cases explained above, this may be critical to system correctness.

**Transaction Interface.** DracoSTM defines transactions as stack-based objects using RAII. In addition to the static interfaces for direct and deferred updating described earlier in this section, the following interfaces are necessary for performing transactions.

- `static void initialize()`  
This method must called before any transaction objects are constructed. The initialize method initializes the overall transaction locking framework.
- `static void initialize_thread()`  
This method must be called for each thread before any transactions are constructed. This method initializes the thread's read and write sets, new and deleted memory sets, mutex locks and thread-based flags.
- `static void terminate_thread()`  
This method should be called before destroying a thread. While it is not needed, it will keep transaction operations functioning optimally by reducing static overhead within the transaction class that is no longer needed once a thread's lifetime has ended.
- `transaction()`  
Default ctor for a transaction immediately puts the transaction in-flight. In addition, the ctor points referenced members to a number of thread-specific sets which it does by referencing the thread id.
- `~transaction()`  
The transaction dtor releases the thread-specific transaction lock if it is obtained. The dtor then immediately returns if the transaction was not in-flight, or if the transaction was in-flight, it forces the transaction to abort which performs a number of clean-up operations.
- `void restart_transaction()`  
This method is similar to the transaction ctor as it signals the start of a transaction and attempts to put it in-flight. A fundamental difference between `restart_transaction()` and the transaction ctor is that if the transaction is already in-flight, `begin_transaction()` aborts the transaction and restarts it. This behavior is important for composed transactions where the outer transaction is never destructed, due to continually excepting inner transactions which also prevent the outer transaction from reaching its call to `end_transaction()`. This is shown concretely in the composable transaction example.
- `transaction_state end_transaction()`  
This method signals that the transaction should try to commit. The return value stored in a `transaction_state` enumeration is either 1) `e_hand_off`, meaning the transaction was nested and has handed off its state to the parent transaction or 2) `e_committed`, meaning the transaction was committed and global memory has been updated. If a transaction is aborted, an `aborted_transaction_exception` is thrown. A call to `end_transaction()` will never return an enumerated state which signals the transaction was aborted. Instead, if a trans-

action is found to be aborted at a call to `end_transaction()`, an aborted transaction exception is thrown. This behavior enables all aborted transactions to be handled in a similar fashion, resulting in an aborted transaction exception.

- `template <typename T> T& read(T&)`

The read method is used when a transaction is attempting to read a piece of memory that is shared between threads (i.e. global heap space). The input argument is a reference to a derived `base_transaction_object` instance, the return value reference is the correct `base_transaction_object` to read based on the current state of the transaction and the currently employed updating policy.

- `template <typename T> T& write(T&)`

The templated write method is used when a transaction is attempting to write to a piece of memory that is shared between threads (i.e. global heap space). The input argument is a reference to a derived `base_transaction_object` instance, the return value reference is the correct `base_transaction_object` to write to based on the current state of the transaction and the currently employed updating policy.

- `template <typename T> T* read_ptr(T*)`

Identical to the `read()` interface, except the input parameter is a `base_transaction_object` pointer as is the return parameter.

- `template <typename T> T* write_ptr(T*)`

Identical to the `write()` interface, except the input is a `base_transaction_object` pointer as is the return parameter.

- `template <typename T> T& find_original(T&)`

This method searches the transaction's write list for the original piece of memory referred to by the transactional memory passed in. If the memory passed in is the original global memory it is simply returned. Otherwise, the copied memory passed in, is used to find the original global memory stored in the transaction's write set and then the original memory is returned. While new memory constructed within a transaction can be passed into `find_original()`, it is not necessary as it always will return a reference to itself. However, to reduce the complexity of client-side code, it may be preferred to build a single method for both new and existing memory address lookup.

- `template <typename T> T* new_memory(T const &)`

This method constructs new memory of a derived `base_transaction_object` type and returns a pointer to the newly allocated memory. While the input template parameter is not used, C++ does not allow functions to differ only by return type. Since different template function instantiations would be constructed here, one per derived `base_transaction_object` type used within the transaction, a compiler error would be generated for differing return types if an input parameter was not supplied. To overcome this, a `void*` could be used as the return value, but that would incur client-side casting on the return value, a costly side-effect. To overcome the limitations of C++, while still ensuring strong type-safety, an input parameter which is never referenced is required.

- `template <typename T> T* new_memory_copy(T const &)`

This method behaves similarly to the `new_memory()` interface except that it makes an exact replica of the input parameter. The return value is a pointer to the newly allocated, replicated object.

- `template <typename T> void delete_memory(T &)`

This method places the input parameter into a garbage set which is emptied (deleted) once the transaction commits.

**Memory Rollbacking.** STM systems must restore allocated and deallocated transactional memory to its original memory state if a transaction aborts. These operations are handled by the `new_memory()`, `new_memory_copy()` and `delete_memory()` interfaces. New transactional memory is stored in a transactional list that is emptied if the transaction commits, or if the transaction aborts, each item is deleted and the list emptied. Deleted transactional memory is also stored in transactional list where each item is deleted and the list emptied if the transaction commits, or if the transaction aborts, the list is simply emptied. This guarantees allocated or deallocated memory is consistent based on the transaction's final state.

**Contention Manager Interface.** DracoSTM implements an extensible contention manager (CM) system. As such, any client can construct a new CM with a user-specified implementation. Two static interfaces within the transaction class allow client-side CMs to be plugged in.

- `void contention_manager  
(base_contention_manager *rhs)`

Sets DracoSTM's contention manager to point to the passed in contention manager. When a new CM is passed in to DracoSTM, it deletes the previously pointed to CM and points to the new one. Client code is responsible for constructing new CMs that are passed to this method, but DracoSTM then takes ownership of these CMs.

- `base_contention_manager* get_contention_manager()`

Returns a pointer to the current CM used by DracoSTM. This method is mostly used for validation purposes after setting a CM to a user-specified implementation.

The skeletal structure and API examination for DracoSTM's contention manager is given below. The following interfaces are the overridable methods for the `base_contention_manager`, as briefly explained in the previous section's priority inversion examples. All of the interfaces within the `base_contention_manager` are pure virtual and must be overridden if a client-side CM is implemented.

```
class base_contention_manager
{
public:
    virtual void abort_on_new(transaction const &) = 0;
    virtual void abort_on_delete(transaction const &,
        base_transaction_object const &) = 0;
    virtual void abort_on_read(transaction const &,
        base_transaction_object const &) = 0;
    virtual void abort_on_write(transaction &,
        base_transaction_object const &) = 0;
    virtual bool abort_before_commit
        (transaction const &t) = 0;
    virtual bool permission_to_abort
        (transaction const &, transaction const &) = 0;
    virtual ~base_contention_manager() {};
};
```

- `virtual void abort_on_new(transaction const &t)`

Supplies the behavior for transactional aborts when identified as doomed from within a `new_memory()` or `new_memory_copy()` operation. The input parameter is the doomed transaction. Throwing `aborted_transaction_exceptions` are the usual mechanism for aborting.

- virtual void abort\_on\_delete(transaction const &t, base\_transaction\_object const &in)

Supplies the behavior for transactional aborts when identified as doomed from within a `delete_memory()` operation. The input parameters are the doomed transaction and the object being deleted. Throwing `aborted_transaction_exceptions` are the usual mechanism for aborting

- virtual void abort\_on\_read(transaction const &t, base\_transaction\_object const &in)

Supplies the behavior for transactional aborts when identified as doomed from within a `read()` operation. The input parameters are the doomed transaction and the object being read. Throwing `aborted_transaction_exceptions` are the usual mechanism for aborting

- virtual void abort\_on\_write(transaction const &t, base\_transaction\_object const &in)

Supplies the behavior for transactional aborts when identified as doomed from within a `write()` operation. The input parameters are the doomed transaction and the object being written. Throwing `aborted_transaction_exceptions` are the usual mechanism for aborting

- virtual bool abort\_before\_commit(transaction const &t)

This method is called prior to a transaction performing its commit operation. The parameter passed in is the transaction preparing to commit. The client code should return `true` if the transaction should abort before committing. Otherwise, client implementation should return `false`.

- virtual bool permission\_to\_abort(transaction const &lhs, transaction const &rhs)

This method is invoked when a transaction is requesting permission to abort another transaction due to a memory inconsistency. Client code should return `true` if the transaction should abort before committing. Otherwise, client implementation should return `false`. The lhs input parameter is the transaction requesting to abort rhs, the rhs input parameter is the transaction which will be aborted if return `true`. Otherwise, if the method returns `false`, lhs will be aborted.

## 5. Experimental Results

While the focus of this work is on DracoSTM’s architecture, we also report on experimental benchmarks that demonstrate our architecture goals do not come at a cost to performance.

Figure 1 and 2 illustrate DracoSTM’s performance for four threaded linked list and red-black trees. These performance operations compare RSTM, University of Rochester’s non-blocking C++ STM library, to DracoSTM using deferred and direct updating. Both Figure 1 and 2 double the number of nodes inserted and looked up with each iteration. The x-axis shows the number of nodes inserted (and looked up) in the container and the y-axis shows the number of transactions per second. The higher the transactions per second, the faster the STM system is performing. These tests were run on a 3.2 GHz 4-processor Intel Xeon with 16GB RAM.

While RSTM is an exceptionally fast performing non-blocking STM library, our tests have demonstrated that DracoSTM outperforms it in most cases. However, our benchmarks have found RSTM to outperform DracoSTM when small transactions dominate the workload as can be seen in the early benchmarks of Figure 2. Figure 1 and 2 demonstrate the superlinear performance improvement DracoSTM has over RSTM as the size of the container doubles. For example, the 6400 linked list insert + lookup test took

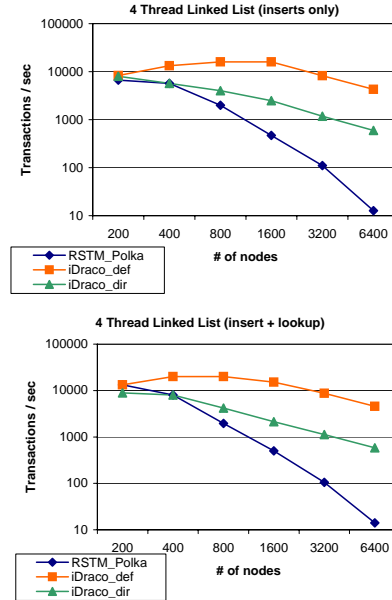


Figure 1. Four Threaded Linked List Benchmark.

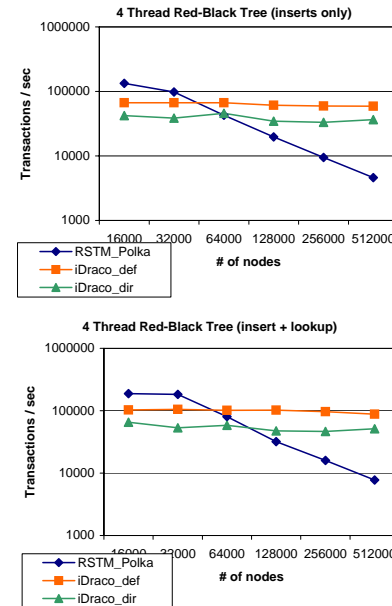


Figure 2. Four Threaded Red-Black Tree Benchmark.

RSTM 908 seconds to perform while DracoSTM performed the same workload in 2 seconds using deferred updating (454x faster) and in 22 seconds using direct updating (41x faster). The 512,000 red-black tree insert + lookup test took RSTM 132 seconds to perform, while DracoSTM performed the same operation in 10 seconds using deferred updating (13x faster) and 19 seconds using direct updating (7x faster). We believe these performance metrics are partially induced by the library optimizations found within DracoSTM, but are mostly the result of the specific TM theories implemented within the system (e.g., lock-based deferred updating using commit-time invalidation). A significant number of additional tests were run, but were not included due to space limitations.

## 6. Conclusion

This paper presented the DracoSTM C++ library from an architectural viewpoint, detailing the TM-specific and library-specific high level aspects. For each TM aspect, we explained its purpose from a theoretical viewpoint and then explained how DracoSTM implements it. We then followed the same mechanical process for C++ and library aspects. Next, we showed some working examples, starting with a simple transaction and then building a more complex, composable transaction. Discussion of a minor, but important piece of complexity, followed. We then gave a more complex priority inversion example using composition, resulting in transactional attribute enrichment. After this, DracoSTM's API was covered, touching on the most visible and static interfaces first and then drilling down into specific transaction-based interfaces. Last, we presented some brief performance results highlighting DracoSTM performance against RSTM, University of Rochester's C++ non-blocking STM library, showing DracoSTM outperforms RSTM by upwards of two orders of magnitude for large transactions.

## Acknowledgments

We would like to thank the entire University of Rochester High-Performance Synchronization Group for their generosity with their STM library (RSTM) and detailed explanation of its usage. We would like to thank Michael Spear, specifically, for his directed commentary regarding many TM concepts mentioned (and not mentioned) here. We also give our thanks to Lori Peek-Gottschlich, Sue Lewis and Dwight Winkler for their numerous edits and valuable feedback. We give special thanks to Jeremy Siek who gave insightful and critical DracoSTM design suggestions, improving the overall implementation of the system as well as the structure of this work. Last, we would like to thank all of the anonymous reviewers who pointed out several important gaps and places for improvement within this work. We have tried our best to integrate as many of these comments as possible.

## References

- [1] D. Abrahams. Exception-safety in generic components. *Online: [www.boost.org/more/generic\\_exception\\_safety.html](http://www.boost.org/more/generic_exception_safety.html)*.
- [2] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *Proceedings of the Eleventh International Symposium on High-Performance Computer Architecture*, pages 316–327. Feb 2005.
- [3] ANSI/ISO. International standard for programming language C++. Technical Report ANSI X3J16/96-0225 ISO WG21/N1043, ANSI/ISO, 1996.
- [4] M. H. Austern. *Generic Programming and the STL: using and extending the C++ Standard Template Library*. Addison-Wesley professional computing series. Addison-Wesley Longman Publ. Co., 1998.
- [5] D. Bulka and D. Mayhew. *Efficient C++: performance programming techniques*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.
- [6] B. Carlstrom, J. Chung, H. Chafi, A. McDonald, C. Minh, L. Hammond, C. Kozyrakis, and K. Olukotun. Transactional execution of java programs, 2005.
- [7] J. O. Coplien. Curiously recurring template patterns. *C++ Rep.*, 7(2):24–27, February 1995.
- [8] S. Dewhurst. *C++ Gotchas: Avoiding Common Problems in Coding and Design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
- [9] D. Dice and N. Shavit. What really makes transactions faster? *ACM SIGPLAN Workshop on Transactional Computing*, June 2006.
- [10] R. Ennals. Software transactional memory should not be obstruction-free. Technical report, Intel Research Tech Report, Jan 2006.
- [11] Gamma, Helm, Johnson, and Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley, Massachusetts, 2000.
- [12] C. Gniady, B. Falsafi, and T. N. Vijaykumar. Is SC + ILP=RC? In *ISCA*, pages 162–171, 1999.
- [13] R. Guerraoui, M. Herlihy, and B. Pochon. Toward a theory of transactional contention managers. In M. K. Aguilera and J. Aspnes, editors, *PODC*, pages 258–264. ACM, 2005.
- [14] L. Hammond, B. D. Carlstrom, V. Wong, M. K. Chen, C. Kozyrakis, and K. Olukotun. Transactional coherence and consistency: Simplifying parallel hardware and software. *IEEE Micro*, 24(6):92–103, 2004.
- [15] L. Hammond, B. D. Carlstrom, V. Wong, B. Hertzberg, M. Chen, C. Kozyrakis, and K. Olukotun. Programming with transactional coherence and consistency (TCC). *j-SIGPLAN*, 39(11):1–13, Nov. 2004.
- [16] T. Harris and K. Fraser. Language support for lightweight transactions. *j-SIGPLAN*, 38(11):388–402, Nov. 2003.
- [17] T. Harris, S. Marlow, S. L. P. Jones, and M. Herlihy. Composable memory transactions. In K. Pingali, K. A. Yelick, and A. S. Grimshaw, editors, *PPOPP*, pages 48–60. ACM, 2005.
- [18] Herlihy, Luchangco, Moir, and Scherer. Software transactional memory for dynamic-sized data structures. In *PODC*, 2003.
- [19] W. N. S. III and M. L. Scott. Advanced contention management for dynamic software transactional memory. In M. K. Aguilera and J. Aspnes, editors, *PODC*, pages 240–248. ACM, 2005.
- [20] N. M. Josuttis. *The C++ Standard Library: a tutorial and reference*. Addison-Wesley, 1999.
- [21] J. R. Larus and R. Rajwar. *Transactional Memory*. Morgan & Claypool, 2006.
- [22] Marathe, Scherer, and Scott. Adaptive software transactional memory. In *DISC: International Symposium on Distributed Computing*. LNCS, 2005.
- [23] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer, III, and M. L. Scott. Lowering the overhead of nonblocking software transactional memory. Technical Report TR893, University of Rochester, Computer Science Department, May 2006.
- [24] A. McDonald, J. Chung, B. D. Carlstrom, C. C. Minh, H. Chafi, C. Kozyrakis, and K. Olukotun. Architectural semantics for practical transactional memory. In *Proceedings of the Thirty-Third Annual International Symposium on Computer Architecture, ISCA 2006, Boston, MA, USA, June 17-21, 2006*, June 2006.
- [25] K. E. Moore. Thread-level transactional memory. In *Wisconsin Industrial Affiliates Meeting*. Oct 2004. Wisconsin Industrial Affiliates Meeting.
- [26] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. Logtm: Log-based transactional memory. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, pages 254–265. IEEE Computer Society, Feb. 2006.
- [27] R. Rajwar and J. R. Goodman. Speculative lock elision: enabling highly concurrent multithreaded execution. In *MICRO*, pages 294–305. ACM/IEEE, 2001.
- [28] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. *isca*, 00:494–505, 2005.
- [29] B. Stroustrup. A brief look at C++0x. *Online: [www.artima.com/cppsource/cpp0x.html](http://www.artima.com/cppsource/cpp0x.html)*.
- [30] B. Stroustrup. *The C++ Programming Language (3rd Edition)*. Reading, Mass., 1997.
- [31] H. Sutter. *Exceptional C++: 47 engineering puzzles, programming problems, and solutions*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.