Toward Simplified Parallel Support in C++

Justin E. Gottschlich† Jeremy G. Siek† Paul J. Rogers‡ Manish Vachharajani†

†Department of Electrical and Computer Engineering, University of Colorado at Boulder
‡ Raytheon Company, Aurora, Colorado
†{gottschl, jeremy.siek, manishv}@colorado.edu, ‡pjrogers@raytheon.com

Abstract

Parallel programming is an increasingly important topic in software engineering due to the recent rise of multicore computer architectures. Although substantial research efforts have been made to simplify parallel programming models, few approaches concentrate on both simplicity and practicality.

In this paper, we present a simplified programming model for practical parallel support in C++. We extend the DracoSTM C++ library with language-like constructs for both transactional memory (TM) and mutual exclusion locks. Our library-based approach promotes a simplified programming model that increases correctness by using common programming structures. We compare our design to two other well-known library-based approaches. Finally, we provide an overview of our implementation, its legal execution model and a number of examples demonstrating its use.

1. Introduction

Research in parallel programming has recently seen a flurry of attention. Among the active research is a push for high-level languages to offer native support for parallel programming primitives [18,24]. The next version of C++ will incorporate library support for threads, while numerous researchers are exploring ways to extend Java to support transactional memory (TM) [9,24]. Yet, the distinctly different goals of C++ and Java conflict with their respective approaches to parallel programming.

Java was built with parallel programming abstractions as a natural part of the language [14]. These abstractions include reserved words for constructing operations within pessimistic critical sections as well as serializing access to methods with synchronized monitors [19]. A number of proposals have also been made to support language-based optimistic critical sections (e.g., transactions) within Java, such as Ziarke et al.'s uniform approach [24]. These language-based parallel abstractions simplify parallel programming, but require the programmer to relinquish some control on the underlying behavior of locks and transactions. Moreover, a number of open problems exist in the TM domain that a language-based solution may incorrectly solve, causing irreversible damage to the language [1, 6, 15]. Since Java does not support stack-based automatic objects (aside from primitives) and has a high level of exception infrastructure within its libraries, language-based parallel abstractions seem inevitable. These language-based abstractions seem inevitable in Java, because it lacks other means to prevent locks from deadlocking or transactions from running indefinitely in the event of thrown exceptions.

A strength of C++ is its support for automatic objects [12, 22]. Rather than requiring that parallel primitives be added directly to the language, as is done in Java, automatic objects in C++ can be used to implement much of their necessary infrastructure [2, 5]. The automatic object approach is natural from a language perspective, provides full algorithmic control to the end programmer, and demonstrates C++'s linguistic elegance. The disadvantage of this approach is its added programmatic overhead. Using only automatic objects, certain programming errors, such as accidental scope removal and incorrectly programmed transactional retry behavior, can arise. Within Java, these potential errors are avoided automatically by using language-based abstractions.

In light of this, there are unique trade-offs between language-based and library-based parallel primitives. Language-based solutions minimize syntactic clutter which reduce programmer related errors, but are seemingly irreversible and, if incorrect, can have crippling effects upon the language. Library-based solutions increase programmer control and flexibility, but place substantial pressure on the programmer to avoid minute programming errors. A good compromise is a solution that behaves like a language extension, but is implemented within a library. By implementing parallel primitives within a library that uses language-like interfaces, programmer pressure is reduced, implementation updates are seamless, and full programmer control is achieved through library extensibility.

In this paper, we present such a language-like solution for C++ using the DracoSTM library coupled with a deliberate use of the preprocessor. The culmination of these components facilitate a simple, yet powerful, parallel programming interface in C++. We make the following technical contributions:

1. We present language-like extensions to the DracoSTM C++ library for mutual exclusion locks, timed locks and transactions. These extensions are easily upgradeable, thereby supporting easy integration as advances are made in TM and lock-aware TM research.
2. Every possible language-like programming model combination of transactions and locks are supported within the execution model. This is a notable achievement for a software library.

2. Synchronization Primitives

The synchronization primitives supported in DracoSTM's programming model are mutual exclusion locks, timed locks and transactions. Mutual exclusion locks have architectural support in

BoostCon'09 May 3-8, 2009, Aspen, Colorado.

Table 1. DracoSTM Mutual Exclusion Locking Parallel Constructs.

Keyword	Behavior
`use_lock(L)`	Acquires lock `L` and executes critical section. Supports single operations without `{}` scope construction such as, `use_lock(L) foo();`.
`use_timed_lock(MS, L)`	Acquires timed lock `L` and executes a critical section based on `MS` millisecond timer. If the timer expires without obtaining lock `L`, an exception is thrown. This extension allows the exception to escape the parallel construct scope. Supports single operations without `{}` scope construction such as, `use_timed_lock(1, L) foo();`.
`try_timed_lock(MS, L)`	Same as `use_timed_lock(L)`, but exceptions are caught locally. Requires `catch_lock_timeout(L)` or `lock_timeout`.
`catch_lock_timeout(E)`	Catches timed lock exception `E` thrown from `try_timed_lock(L)`. Required to follow all `try_timed_lock(L)`, unless followed by `lock_timeout`.
`lock_timeout`	Catches and discards failed timer exception thrown from `try_timed_lock(L)`. Required to follow all `try_timed_lock(L)`, unless followed by `catch_lock_timeout(E)`.

all modern instruction set architectures and are widely considered the most common synchronization primitive [11]. Transactions, a parallel programming abstraction by Herlihy and Moss [10], are currently implemented in software with potential hardware support in the near future [4].

2.1 Library-based Lock Implementations

Mutual exclusion locks, or just locks, ensure a programmer-specified set of operations are limited to one thread of execution [7, 11]. Lock-guarded operations, also known as pessimistic critical sections, require that all threads obtain a single-ownership flag before executing the guarded operations. If N threads simultaneously attempt to execute the same pessimistic critical section, one thread is allowed forward progress while the remaining $N-1$ threads are stalled. An example of three distinct types of locking is shown in Figure 1.

```
// mutual exclusion with lock()/unlock().
pthread_mutex_lock(l);
int ret = foo();
pthread_mutex_unlock(l);
return ret;
```

```
// mutual exclusion with automatic objects.
{
  scoped_lock<pthread_mutex_t> lock(l);
  return foo(); // lock released
}
```

```
// language-like mutual exclusion.
use_lock(l) return foo(); // lock released
```

Figure 1. Library-based Lock Implementations.

In Figure 1, the automatic object and language-like mutual exclusion interfaces are safer than the `pthread_mutex_lock()` and `pthread_mutex_unlock()` interfaces. If an exception is thrown from `foo()` after `pthread_mutex_lock()` is called, its corresponding `pthread_mutex_unlock()` will not be called causing the application to deadlock. Both the automatic object and language-like implementations avoid deadlock in the event of an exception thrown from `foo()`. However, these implementations have notable differences in programmatic scoping and programmer error.

2.1.1 Pitfalls in Scoping of Automatic Object Locks

A closer examination of the automatic object and language-like approaches to locking reveal differences in potential programmer-induced error. Figure 2 demonstrates a sequential locking order example, a common solution to complex fine-grained locking implementations, in which locks must be obtained in a specific sequential order to avoid deadlocking. If the scope (`{}`) operators are removed from Figure 2 the behavior of both the `scoped_lock`s and the `use_lock`s are changed. In the case of the `scoped_lock`s, the automatic objects will no longer be terminated after `operation_C()` and they will thereby retain any locks acquired upon construction until the next immediate scope is terminated. In the case of the `use_lock`s, the locks remain locked *only* for `operation_A()` and are then released. Both locking structures are changed to behave incorrectly when the scope operators (`{}`) are removed, but we believe programmers are less likely to accidentally remove the scope from the `use_lock` idiom than the `scoped_lock`.

```
// mutual exclusion with automatic objects.
{
  scoped_lock<pthread_mutex_t> lock1(L1);
  scoped_lock<pthread_mutex_t> lock2(L2);
  scoped_lock<pthread_mutex_t> lock3(L3);
  operation_A();
  operation_B();
  operation_C();
}
```

```
// language-like mutual exclusion.
use_lock(L1) use_lock(L2) use_lock(L3)
{
  operation_A();
  operation_B();
  operation_C();
}
```

Figure 2. A Required Sequential Locking Order.

The reason why programmers are less likely to accidentally remove the scope from `use_lock` as opposed to `scoped_lock` is

because the programming structure of `use_lock` is native to C++ in many ways. For example, the code structure shown below is found throughout C, C++ and Java:

```
grammatical-phrase
{
  operations ...
}
```

A number of control structures, specifically in C++, follow this format, such as, `if` statements, `for` loops, `switch`es, `class`es, `struct`s and so forth. While the `use_lock` structure follows this common practice, scoped automatic objects (e.g., `scoped_lock`s) do not. Because `scoped_lock`s unavoidably deviate from the above common code structure, due to their automatic object basis, their scope is more likely to be accidentally removed by novice programmers as unnecessary.

The side-effect of accidental scope removal for automatic objects is that resources obtained by these objects are not released until the following scope closure is reached. In some cases, such as file closing, it may be acceptable for automatic objects to delay the release of resources. However, in other cases, such as the case with locks, it is unacceptable for automatic objects to delay the release of resources. In particular, if locks are not released when the programmer intended, the resulting behavior can temporarily or permanently stall all other threads from making forward progress. If threads are temporarily suspended from forward progress, program performance is degraded, yet if threads are permanently suspended from making forward progress, the program results in a deadlock. Both cases are unacceptable for locks because locks are used as a tool to optimize program performance and in both cases the delayed release of locks results in unoptimized performance.

2.2 Library-based Transaction Implementations

Transactions allow an unlimited number of threads to execute their optimistic critical sections. Transactions can perform their writes off to the side, ensuring global memory is preserved until the transaction's atomic operations are complete. To ensure conflicting transactions are identified and prevented, transactions perform correctness verification immediately before committing. The consistency checking performed by transactions ensures that transactions that write to or read from the same memory are restricted in their concurrent execution [13, 21]. Figure 3 demonstrates three different implementations for transactions from a library-based approach where `x` is shared memory that must be synchronized.

```
// transaction with begin()/end().
begin_transaction(t);
tx_write(t, x) = val;
end_transaction(t);
```

```
{ // transaction with automatic object.
  transaction t;
  t.write(x) = val;
  t.end();
}
```

```
// language-like transaction.
atomic(t) { t.write(x) = val; } end_atom
```

Figure 3. Library-based Transaction Implementations.

Inspection of Figure 3 reveals that `begin_transaction()` and `end_transaction()` are susceptible to the same problem as found in Figure 1 where a thrown exception can interfere with correct interface calls. As such, the `begin_transaction()` and `end_transaction()` approach can be immediately discarded from further consideration. The two remaining approaches, similar to the prior locking implementations, use automatic objects and a language-like approach. An initial observable difference between the two approaches is that the language-like approach has a smaller programmatic footprint than the automatic object approach. Furthermore, the automatic object approach introduces more programmatic complexity for transactional retry mechanics and composed transactional behaviors [8].

2.2.1 Pitfalls in Transactional Execution of Automatic Objects

Transactions use optimistic critical sections which generally require transactions be retried if they do not commit. As such, transactions are usually implemented as loops which re-execute until they commit. Figure 4 illustrates the client code necessary to implement a basic retry behavior for automatic objects and language-like transactions.

```
// automatic object transaction with retry.
for (transaction t; !t.committed();
     t.restart()) {
  try {
    t.write(x) = val;
    t.end();
  } catch (...) {}
}
```

```
// language-like transaction with retry.
atomic(t) { t.write(x) = val; } end_atom
```

Figure 4. Retry Mechanics of Transactions.

To complicate matters, some transactions must not implement a retry. Failed subtransactions often require the entire transaction be re-executed from the beginning [16]. While the methods used to perform transactional retries vary between TM implementations, DracoSTM uses an exception-based approach for all transactional interfaces. These DracoSTM interfaces throw exceptions if transactions are found to be inconsistent [5]. Therefore, parent transactions should use retry mechanics while their child transactions should not. Figure 5 shows the differences between an automatic object and language-like implementation for parent and child transactions.

In Figure 5, the retry mechanics' syntactic overhead for automatic objects is nearly double that of the language-like semantics. The complexity of the additional retry code is significant and exhibits a number of locations where programmer-induced errors could be made. The key benefit of the language-like `atomic` syntax is that its structure is identical for parent and nested transactions and it behaves correctly when *any* transaction is used as a parent or child (details to follow).

While the automatic object syntax could also be created to be identical for parent and nested transactions, the impact of creating such identical behavior would result in an increase in the child transaction's code size by $\approx 266\%$ for single instruction transactions. The resulting increased code size and complexity would increase the likelihood for programmer-induced errors. For these reasons, a number of TM researchers have been in favor of direct language integration of TM instead of API-only approaches [3, 18, 24].

2.2.2 Disadvantages of Language-Based Transactional Integration

Unfortunately, there are a number of disadvantages to direct language-based support for transactions. To begin, transactional memory is still in the early stages of research investigation. A number of open TM questions should be answered before transactions

Table 2. DracoSTM Transactional Memory Parallel Constructs.

Keyword	Behavior
`atomic(T)`	Uses atomic transaction `T` for optimistic critical section. Supports single-depth and multiple-depth (nested) transactions. Performs automatic retry when `T` is a parent transaction, throws exception when `T` is a child transaction. This automatic switch enables correctly behaved nested and non-nested transactions.
`catch_before_retry(E)`	Catches exception when transaction is aborted. This, `before_retry`, or `end_atom` must follow an `atomic` block. Once all operations within the `catch_before_retry` block are executed, control is returned to the local `atomic` where the transaction is retried (if it is the parent) or the `atomic` block is exited by exception (if it is a child).
`before_retry`	Same as `catch_before_retry(E)` except the exception is discarded.
`end_atom`	Same as `before_retry` except the exception is discarded and {} are automated.

```
// parent tx with automatic object.
for (transaction t; !t.committed();
     t.restart()) {
  try {
    t.write(x) -= val;
    foo();
    t.end();
  } catch (...) {}
}

// child tx with automatic object.
void foo(int val)
{
  transaction t;
  t.write(y) += val;
  t.end();
}
```

```
// parent tx with language-like transaction.
atomic(t) {
  t.write(x) -= val;
  foo();
} end_atom

// child tx with language-like transaction.
void foo(int val)
{
  atomic(t) { t.write(y) += val; } end_atom
}
```

Figure 5. Parent and Nested Transactions.

are integrated directly into high-level languages. Some of the open questions for transactions are regarding validation and invalidation consistency checking, fairness and priority-based transactions, open and closed nesting, exception behavior within transactions, lock-based and non-blocking solutions, and hardware-software transactional communication. Furthermore, some TM problems, such as contention management strategy selection, seem more naturally placed within libraries than languages due to their continually evolving and workload-specific nature [6, 20].

In light of this, direct integration of TM into a programming language today may lead to errors that are irreversible. These errors may have long-term consequences for the language. Language-based integrations are also slow to emerge, even in languages that are quick to evolve, such as Java. A language-based approach to TM may take several years before it is available. Yet, the emergence of multi-core hardware is rushing programmers to develop multi-threaded applications today. Without wide TM availability, the primary parallel programming construct used today is locks [11]. Parallel programming research experts unanimously agree that fine-grained locking alone leads to notoriously complex software to implement and maintain.

The culmination of the above points illustrate the need for an extensible, simplified, parallel programming model today. Our language-like approach provides such a solution for C++ that neither library-based automatic objects nor language-based parallel abstractions alone can provide.

3. Implementation Details

Our language-like lock and transaction parallel constructs are implemented in DracoSTM as automatic objects wrapped inside of preprocessor macros. Automatic objects are common in C++ and are helpful in ensuring correct lock and transaction behavior as they create deterministic termination points [5, 22]. These deterministic termination points are invoked when the scope containing the automatic object is exited, guaranteeing locks are released and transactions are terminated even in the event of uncaught exceptions. The preprocessor macros used for the locking and transaction constructs are shown in Figures 6 and 8, respectively.

3.1 Locking Macros

The lock macros use an unoptimized `if` statement to ensure the variables inside their local `for` loop are terminated for non-standard conforming C++ compilers (more details to follow). Once the `for` loop is entered an `auto_lock` is constructed. Upon construction of the `auto_lock`, acquisition of the supplied lock is attempted if it has not already been acquired by the locking thread. For the `use_lock` macro, a blocking lock acquisition call is made which blocks forward progress until the lock is obtained. For the `use_timed_lock` and `try_timed_lock` calls, a non-blocking lock acquisition call is made which returns control to the calling thread via exception after `MS` milliseconds if the lock was not successfully obtained.

The `post_step()` call within the lock macros releases any locks the `auto_lock` had acquired. The `for` loop conditional, `done_post_step()`, returns `false` until `post_step()` has been executed. This ensures the lock macro `for` loops are executed once and only once.

```
#define use_lock(L) if (unoptTrue()) \
  for (core::auto_lock __var(L); \
  !__var.done_post_step(); \
  __var.post_step())

#define use_timed_lock(T, L) if (unoptTrue()) \
  for (core::auto_lock __var(T, L); \
  !__var.done_post_step(); \
  __var.post_step())

#define try_timed_lock(T, L) \
  try { for (core::auto_lock __var(T, L); \
  !__var.done_post_step(); \
  __var.post_step())

#define catch_lock_timeout(E) } \
  catch (core::timed_lock_exception &E)

#define lock_timeout } \
  catch (core::timed_lock_exception &E)
```

Figure 6. Preprocessor Macros for Language-like Locks.

3.1.1 Generating Fairly Unique Variable Names

The use of `__var` within the preprocessor lock macros is used to prevent compiler warnings of duplicate variable name generation (e.g., nested variables of the same name prevent access to the outer variable). The `__var` macro invokes a sequence of macros to generate a unique variable name. This unique variable name is important in overcoming warnings that would otherwise be emitted from standard conforming compilers when embedding locks within locks or transactions within transactions. The details of the `__var` implementation are shown in Figure 7.

```
#define __concat(a,b) a##b
#define _concat(a,b) __concat(a,b)
#define __var _concat(_name_, __LINE__)
```

Figure 7. Preprocessor Macro for Unique Variable Name Construction.

3.2 Transaction Macros

The preprocessor `atomic` macro for transactions is slightly more complex than the preprocessor locking macros. The additional complexity behind the `atomic` macro is needed to guarantee two fundamental goals. First, transactions must start and end correctly. Second, transactions must change their retry behavior based on whether they are a child or parent transaction to ensure proper closed nesting, flattened transaction behavior is performed.

The `atomic` preprocessor behaves as follows. Like the lock macros, the `atomic` macro begins with an unoptimized `if` statement (details to follow). When the transactional `for` loop is entered, a `transaction` automatic object is constructed which initializes the transaction and puts it in-flight. The `for` loop conditional ensures the following conditions: (1) the transaction is uncommitted, (2) the transaction has the opportunity to throw an exception if necessary, and (3) the transaction is in-flight. Once a transaction commits, the check on (1) `!T.committed()` ensures the transaction is not executed again. If the transaction has been aborted but is a child transaction, the transaction must be restarted at the parent level. The call to (2) `T.check_throw_before_restart()` allows an aborted child transaction to throw an exception upward (before it is restarted) so the entire transaction can be restarted from the parent. The `check_throw_before_restart()` API checks the current run-time state of the thread to determine if another transaction is active above it. This behavior allows transactions to be used at any nesting level while dynamically ensuring the correct retry behavior. Finally, the call to `restart_if_not_inflight()` ensures the transaction is correctly restarted after each subsequent abort.

Once all of the transactional operations within the `for` loop are executed, a call to `no_throw_end()` is made which ends the transaction. The `no_throw_end()` terminates the transaction by either committing or aborting it. Note, however, that `no_throw_end()` does not throw an exception if the transaction is aborted, whereas the prior DracoSTM API `end()` does. This non-throwing behavior deviates from the prior DracoSTM implementation of automatic objects when `end()` was invoked within the `try` / `catch` body. Furthermore, due to `no_throw_end()` not throwing an exception if the transaction is aborted, some cases may arise where `catch_before_retry` or `before_retry` operations are *not* invoked when a transaction is aborted. This is a current limitation of the system and is overcome by inserting a manual `end()` operation as the last operation in the `atomic` block. The explicit `end()` (Figure 14) ensures any operations within the `before_retry` block are executed if the transaction is aborted.

```
#define atomic(T) if (unoptTrue()) \
  for (core::transaction T; !T.committed() && \
  T.check_throw_before_restart() && \
  T.restart_if_not_inflight(); \
  T.no_throw_end()) try

#define catch_before_retry(E) \
  catch (core::aborted_tx &E)

#define before_retry catch \
  (core::aborted_tx &)

#define end_atom catch \
  (core::aborted_tx &) {}
```

Figure 8. Preprocessor Macros for Language-like Transactions.

3.3 Correcting Non-Compliant Compilers

The `if (unoptTrue())` call in the preprocessor macros is used to prevent `for` loop errors in non-standard conforming C++ compilers. In these non-compliant compilers, automatic objects constructed as index variables for the `for` loop are leaked out beyond the scope of the `for` loop, incorrectly extending the liveness of these variables. In order to correct this behavior, the `for` loops that are not encapsulated within `try`s are wrapped within `if` statements. The `if` statements and `for` loops naturally nest without delineated scope (e.g., {, }) allowing programmers to execute single or multiple operations based on their preference.

The `unoptTrue()` function always returns `true` and cannot be optimized away by an optimizing compiler. By using an always `true` non-optimizable function inside an if statement, a variable scope is generated that guarantees automatic objects which are placed within these scopes are properly destroyed once the scope is exited. These scopes properly terminate variables which would otherwise be leaked in non-compliant `for` loop C++ compilers. The proper termination of automatic `auto_lock`s and `transaction`s is necessary to release acquired locks, terminate transactions and release transactional memory.

4. Parallel Constructs for Mutually Exclusive Locks

In this section we describe the behavior, and give examples, of the DracoSTM language-like parallel constructs for mutually exclusive locks. The DracoSTM mutual exclusion locking constructs are shown in Table 1. Each of these constructs allow any type of nesting, ordering and intermixing. Examples of the DracoSTM locking constructs are shown in Figure 9.

```
// single-line sequential locking
use_lock(L1) use_lock(L2) use_lock(L3) foo();

// multi-line sequential locking
use_lock(L1) use_lock(L2) use_lock(L3) {
  foo();
}

// multi-line sequential single 100 ms timed lock
while (polling)
  try_timed_lock(100, L1) {
    use_lock(L2) use_lock(L3)
    {
      foo();
      polling = false;
    }
  } lock_timeout { execute_on_failed(); }
```

Figure 9. Mutual Exclusion Locks with DracoSTM.

As demonstrated in Figure 9, a number of locking models can be implemented using a mixture of the `use_lock(L)`, `use_timed_lock(MS, L)` and `try_timed_lock(MS, L)` parallel constructs. Each locking interface addresses a unique requirement in client code.

4.1 Exception-based Timed Locks

A notable feature of our DracoSTM language-like extensions is the use of exceptions in conjunction with timed locks. It is our belief that timed locks are used primarily for polling models and generally require two distinct paths of execution. One path of execution is taken when the lock is obtained before the timeout. The other path of execution is taken when the lock is not obtained before the timeout. In light of this, exceptions thrown when timed locks are not acquired can facilitate a distinction between successful lock acquisition and failed lock acquisition. Client code can become excessively unoptimized or complex if timed sequential locking is necessary and non-throwing automatic objects are used. An example of such unoptimized timed locking is demonstrated in Figure 10.

```
// unoptimized timed locking
while (polling) {
  timed_lock tl1(100, L1);
  timed_lock tl2(100, L2);
  timed_lock tl3(100, L3);

  if (tl1.has_lock() && tl2.has_lock() &&
      tl3.has_lock()) {
    foo();
    polling = false;
  }
  else execute_on_failed();
}
```

Figure 10. Unoptimized Timed Locking with Automatic Objects.

Figure 10 illustrates how non-throwing automatic objects can hamper performance if incorrectly implemented and demonstrates an unoptimized implementation of timed locks. Delaying lock acquisition checks until after all timed locks are constructed can result in an a substantial performance degradation if neither `L1` nor `L2` are acquired. This performance degradation is caused by performing additional unnecessary work after failing to acquire lock `L1` and `L2`.

```
// optimized timed locking
while (polling) {
  timed_lock tl1(100, L1);
  if (tl1.has_lock()) {
    timed_lock tl2(100, L2);
    if (tl2.has_lock()) {
      timed_lock tl3(100, L3);
      if (tl3.has_lock()) {
        foo();
        polling = false;
      }
      else execute_on_failed();
    }
    else execute_on_failed();
  }
  else execute_on_failed();
}
```

Figure 11. Optimized Timed Locking with Automatic Objects.

In order to optimize Figure 10's timed locks, a complex level of `if` nesting is needed as shown in Figure 11. Figure 11 demonstrates an optimized level of timed locking, but drastically increases the complexity of the software structure. Figure 11 introduces a high level of software complexity as a trade-off for performance.

Intermixing the DracoSTM `try_timed_lock(MS, L)` and `use_timed_lock(MS, L)` constructs builds a software structure that is simpler than either Figure 10 or 11 while achieving the same level of performance found in Figure 11. An example of `try_timed_lock(MS, L)` and `use_timed_lock(MS, L)` is shown in Figure 12.

```
// multi-line sequential all 100 ms timed lock
while (polling)
  try_timed_lock(100, L1) {
    use_timed_lock(100, L2)
    use_timed_lock(100, L3)
    {
      foo();
      polling = false;
    }
  } lock_timeout { execute_on_failed(); }
```

Figure 12. Optimized Timed Locking with DracoSTM.

5. Parallel Constructs for Transactional Memory

In this section we describe the behavior, and give examples, of the DracoSTM language-like parallel constructs for transactional memory. The DracoSTM transaction-based parallel constructs are shown in Table 2. These transactional constructs enable open-ended transactional nesting behaviors that are important to composition. Examples of the DracoSTM transactions are shown in Figure 13.

Figure 13 demonstrates how transactions are used in various ways. In some cases transactions are single operations, such as in the `inc_x()` case. At other times transactions consist of

```
1  void inc_x(int val)
2  {
3    atomic(t) { t.write(x) += val; } end_atom
4  }
```

```
1  void inc_y_and_z(int val)
2  {
3    atomic(t) {
4      t.write(y) += val;
5      t.write(z) += val;
6    } end_atom
7  }
```

```
1  void transfer_x_to_y_z(int val)
2  {
3    atomic(t) {
4      inc_x(-val);
5      inc_y_and_z(val / 2);
6    } end_atom
7  }
8
```

Figure 13. Transactions with DracoSTM.

multiple operations, such as `inc_y_and_z()`. More cases exist when transactions are made up of other transactions, such as `transfer_x_to_y_z()`. The last case, often referred to as *composition*, is an important characteristic of transactions that most (if not all) other synchronization primitives do not exhibit (more on this later).

In the simple case, when no transactional nesting occurs, the transactional retry behavior happens locally. For example, if two threads try to concurrently execute the `inc_x()` function of Figure 13, only one thread will succeed since both threads cannot simultaneously access and change the same shared memory location. The failing thread will have to retry its transaction. Since `inc_x()` is a single-level transaction, the failing thread can retry the transaction locally (i.e., restarting at the top of the `atomic` structure inside `inc_x()`).

Non-trivial cases exist when transactions may be nested and retrying a transaction locally will cause an infinite loop. These cases use composed transactions and (generally) require that a failed transaction be retried from the parent transaction [1]. An example of a composed transaction that, if aborted, would require the transaction to be retried from the parent is `transfer_x_to_y_z()`. This transaction must be retried from the parent, rather than by either of its child transactions individually, (`int_x()` and `inc_y_and_z()`) because the values within the shared memory `x`, `y`, and `z` could have been altered by another interleaved transaction.

DracoSTM handles the shifting of transactional retry mechanics from child to parent dynamically and automatically. As demonstrated in Section 3.2, the `atomic` macro makes numerous calls into the currently active transaction to determine whether its failed commit should result in a local retry or a thrown `aborted_transaction` exception. The correct path of execution for the transactional retry is based on the run-time state. For example, if `int_x()` has failed, but it is the only transaction currently being executed by a thread, then it should be retried locally. However, if `int_x()` has failed, but it is being executed from within `transfer_x_to_y_z()` then `transfer_x_to_y_z()` should be re-executed. The re-execution of a parent transaction is enabled by a child transaction identifying the correct behavior and then throwing an exception upward.

[1] This assumes the TM system uses closed nesting with flattened transactions. The use of open nesting does allow failed transactions to be retried locally [15, 17].

5.1 Transaction Nesting

While the retry behavior of nested transactions is important, the composed behavior of independently designed transactions is even more important. Transactional composition [8, 15] is the cornerstone of transactional memory. In short, TM allows independently designed transactions to be placed together under the umbrella of a single transaction. This single transaction then behaves in an atomic, consistent and isolated fashion. Furthermore, the single transaction that is composed of many individual transactions is seen as a single indivisible operation to other transactions.

As such, an important aspect of transaction nesting is the construction of a transaction without the need to specify it is nested. A similar line of thinking is the construction of a function which is then used inside of another function and so forth. In C++ these nested functions are not created in any special way that describes their use in a given nested structure. Transactions, therefore, should be programmatically constructed in a similar way.

To the best of our knowledge, DracoSTM is the first C++ STM library to implement transactions under a single keyword (`atomic`) which behaves correctly for both nested and non-nested transactions. While other correct implementations exist at the language-level [24] and compiler-level [18], no other library-based solution allows the programmer the practical freedom to implement transactions under a single keyword for transactions at any nesting level.

A more realistic example of the importance of nested and non-nested transactions and a universal keyword for both, is shown in Figure 14. Here we implement some basic list operations that are useful when used independently and when used in an cooperative, nested fashion.

Figure 14 demonstrates how transactions are composed using DracoSTM. The goal of the code shown in Figure 14 is to transfer all of the elements from list `l` to list `j` that do not already exist in list `j`. The entire operation is done atomically using four core transactional operations, `insert`, `remove`, `transfer`, and `complete_transfer`. The additional transactional operation, `priority_transfer`, is added to demonstrate how client code can wrap a composed transaction and alter its underlying performance in relation to other transactions by modifying the transaction's priority. Wrapping transactions with priority is useful in cases where the original transaction was written in a priority-neutral manner, but now must be used in an explicitly prioritized system. More details of prioritized transactions can be found in [6, 20].

6. Unifying Parallel Constructs

The DracoSTM library supports the concurrent execution of locks and transactions. To achieve this unified behavior, locks and transactions must coordinate their execution [7]. The automatic locking objects implemented behind DracoSTM's `use_lock(L)`, `use_timed_lock(L)`, and `try_timed_lock(L)` notify the TM subsystem when locks are acquired and released. DracoSTM then coordinates when transactions execute around lock acquisition and release using one of three different *lock-aware transactional memory* (LATM) policies.

The most basic LATM policy, called full lock protection, requires no programmer intervention. Full lock protection works by disallowing transactions and locks from running concurrently. While full lock protection is trivial to prove correct, its performance is suboptimal.

The intermediate LATM policy, transactional memory lock protection (TM-lock protection), requires the programmer to specify, at the software system level, which locks can conflict with any of the transactions in the software. The system then ensures that be-

```
1  bool insert(ilist &L, int V)
2  {
3    bool r = false;
4    atomic(t) { r = t.write(L).insert(V); } end_atom
5    return r;
6  }
7
8  bool remove(ilist &L, int V)
9  {
10   bool r = false;
11   atomic(t) { r = t.write(L).remove(V); } end_atom
12   return r;
13 }
14
15 void transfer(ilist &L1, ilist &L2, int V)
16 {
17   atomic(t) {
18     if (insert(L2, V)) remove(L1, V);
19   } end_atom
20 }
21
22 void complete_transfer(ilist &l, ilist &j)
23 {
24   atomic(t) {
25     while (l.size() > 0)
26       transfer(l, j, l.front());
27   } end_atom
28 }
29
30 void priority_transfer()
31 {
32   int p = 0;
33
34   atomic(t) {
35     t.set_priority(p);
36     complete_transfer(L1, L2);
37
38     // explicit call to t.end() to ensure
39     // before_retry is called if tx fails
40     t.end();
41   } before_retry { ++p; }
42 }
```

Figure 14. Example of Composable Transactions in DracoSTM.

fore such conflicting locks are acquired all in-flight transactions are either committed or aborted.

The advanced LATM policy, transaction lock protection (TX-lock protection), requires that the programmer specify, per transaction, which locks conflict with it. The system then ensures that, before such conflicting locks are acquired, any in-flight transactions flagged as conflicting with such locks either commit or abort. While TX-lock protection requires the most programmer involvement, it also yields the greatest performance throughput.

6.1 The Importance of a Unified Library Approach

In order for a locking system to run in conjunction with a TM system, the systems must be aware of each other. All existing LATM systems are implemented in such a manner [7, 23, 24]. As such, a locking library and a TM library written with no knowledge of each other will not operate correctly. Figure 15 demonstrates the incorrect behavior of locks and transactions in a non-unified model.

In Figure 15, let `x = y = z = 0`. Thread 1 executes lines 3-4, obtaining lock `L` and setting `x = 1`. Thread 2 executes lines 5-6, beginning a transaction and setting `z = 1`. Thread 1 executes

```
1  Thread 1                    Thread 2
2
3  lock(L);
4  ++x;
5                              atomic {
6                                z = x + y;
7  ++y;                        }
8  unlock(L);
```

Figure 15. Lock and transaction violation (code).

line 7, setting `y = 1`. Both threads 1 and 2 then complete their operations. The resulting value of `z = 1` is incorrect. `L`'s mutually exclusive critical section is not respected by thread 2's transaction and therefore half of `L`'s critical section operations are exposed to thread 1. In a unified model, transactions respect the critical sections created by locks and therefore transactions do not see incomplete intermediate states of locks. Executing Figure 15's code within a unified model results in `z = 2`.

A critical observation is that a library written solely to operate on locks may require updating if a TM library is used in the future. A lock-aware TM library, alone, may not be sufficient to coordinate locks and transactions [7, 23, 24]. As is implemented in all existing unified models to date, DracoSTM's software library, Ziarek et al.'s P-SLE in the Java programming language, and Volos et al.'s OS-level TxLocks [23], locks and transactions must both communicate with the TM subsystem in order for the locks and transactions to behave correctly. This behavior requires both the locking library and the TM library to interface with a common TM communication protocol. Unfortunately, the current specification of C++0x's standard locking library does not include such a communication protocol for a TM subsystem [2]. In light of this, if transactions are adopted into the C++ standard, a TM library and an update to the existing C++0x locking library will be required.

7. Transaction-Lock Execution Model

All possible programmed combinations of transactions and locks are legal in DracoSTM. We call these programmed combinations the *transaction-lock execution model*. The three possible combinations of transactions and locks in the transaction-lock execution model represent three distinct classes of problems. These problems are discussed in detail in [7]. The first combination of locks and transactions is when one thread uses only locks and another thread only uses only transactions. Gottschlich et al. call this behavior locks outside of transactions or LoTs. The second combination of locks and transactions is when transactions are run inside of locks, called TiLs. TiLs require no special effort and are trivial to proof correct. TiLs are described in more detail by Gottschlich et al. and Volos et al. [23]. The third combination of locks and transactions is when locks are placed inside of transactions, causing the locks to gain composable characteristics due to their transactional encapsulation. Gottschlich et al. refer to this behavior as locks inside of transactions or LiTs. LiTs are the most complex combination of locks and transactions.

In their LATM paper, Gottschlich et al. deem one transaction-lock execution using LiTs as illegal [7]. The illegal behavior occurs when a lock begins its critical section outside of a transaction, but ends its critical section inside a transaction (also known as Early Release [23]). Both Gottschlich et al. and Volos et al. demonstrate the deadlocking potential of this behavior. Volos et al. attempt to break the deadlock behavior through a conflict resolution system. Gottschlich et al. state the behavior is illegal, and throw an exception when it is detected at run-time.

```
1  Locking API            Automatic Object
2
3  lock(L);               auto_lock obj(L);
4
5  atomic(t) {            atomic(t) {
6    ...                    ...
7    unlock(L);             obj.~auto_lock();
8    ...                    // or obj.unlock();
9  } end_atom             } end_atom
```

Figure 16. Locking API and Automatic Object Scope Violation.

Using our DracoSTM language constructs, Early Release scenarios, such as those seen in Figure 16, are not possible. The `use_lock()`, `use_timed_lock()` and `try_timed_lock()` do not allow a lock to be acquired in one scope and released in another. Specifically, both of DracoSTM's parallel constructs (transactions and locks) create critical sections that must begin and end in the same scope. As shown in Figure 16, both locking API and automatic object implementations create critical sections that can begin and end in different scopes, leading to deadlocks.

7.1 Essential Execution Modes

The three combinations of the transaction-lock execution model (e.g., LoTs, LiTs and TiLs) lead to different modes of execution. Some of these execution modes are more important than others. In this section we describe the importance of LoT and LiT execution modes. While TiLs are also important, we believe their use will be minimized once transactions become mainstream. TiLs will rarely be used in practice because programmers will favor transactions over locks, and therefore rarely use locks to wrap transactions.

Locks outside of transactions are important. Herlihy and Shavit note that mutual exclusion locks are arguably the most predominant form of synchronization in parallel programming [11]. In light of this, transactions must function in the presence of locks in legacy code. An example of such an execution is shown in Figure 17.

```
1  Thread T1              Thread T2
2
3  use_lock(A)
4  use_lock(B)            atomic(t) {
5                           t.lock_conflict(A);
6                           t.lock_conflict(B);
7    ++x;
8    ++y;                   // stall for T1
9  }
10                          retX = t.read(x);
11                          retY = t.read(y);
12                         } end_atom
```

Figure 17. Locks Outside of Transactions (LoT) Example.

Figure 17 illustrates how locks outside of transactions are used in a legal manner using TX-lock protection, DracoSTM's fastest LATM mode of execution. Thread T_1 executes lines 3-4 and obtains both locks `A` and `B`. Thread T_2 executes lines 4-6, starting transaction `t` and identifying `A` and `B` as conflicting locks with the transaction. These identified conflicts inform the TM subsystem that the transaction will conflict with the specified locks. Since both locks are currently acquired by thread T_1, the transaction stalls. Thread T_1 increments `x` and `y` and completes its critical section, unlocking locks `A` and `B`. The TM subsystem acknowledges the locks are available and begins executing transaction `t`. Both `A` and `B` are read and the transaction commits. As demonstrated above, LoTs are useful as they enable existing lock-based software to concurrently execute alongside transactions.

Locks inside of transactions are important. Existing lock-based legacy code may need to be wrapped inside of a transaction. Alternatively, new lock-based code may need to be written that requires some portion to be composed within a transaction. The following example demonstrates such a need.

Consider the logic of a multi-threaded web server receiving network messages and storing them locally. These messages are processed, stored in a serialized buffer by one thread, and saved to a log file on the server by another thread. Since these operations do not emit failure atomicity (e.g., they cannot be unwound once partially executed) they are ideal candidates for mutual exclusion locks. Two methods are developed: `lock_recv()` and `lock_write()`. These methods respectively implement the lock-based socket receive and lock-based file write operations. The `lock_recv()` socket implementation uses lock `S` and the `lock_write()` file implementation uses lock `F`. Both methods can then be executed concurrently and use lock `Q` to push and pull messages off of the synchronized message queue, called `q`. This implementation allows socket operations and file operations to be performed concurrently, limiting thread coordination to the shared message queue. A basic implementation of this framework is given in Figure 18.

```
1  void lock_recv(Q, q)
2  {
3    use_lock(S) buff = read_sock();
4    use_lock(Q) q.push_front(buff);
5  }
6
7  void lock_write(Q, q)
8  {
9    use_lock(Q) buff = q.pop_back();
10   use_lock(F) write_file(buff);
11 }
```

Figure 18. Pseudocode for Lock-based I/O Operations.

Now consider a new operation that combines the socket receive and file write into a single operation after a transactional operation is successfully performed. Since both `lock_recv()` and `lock_write()` are mature and thoroughly tested, they can be used directly inside the transaction. An example of such functionality is shown in Figure 19.

```
1  atomic(t) {
2    bool shutDown = true;
3    for (int i = 0; i < server.size(); ++i) {
4      shutDown &= t.read(server[i]).state();
5    }
6
7    // if all systems shutdown, dump sockets
8    if (shutDown) {
9      t.lock_conflict(S, Q, F);
10
11     while (socket_alive()) {
12       lock_recv(Q, q);
13       lock_write(Q, q);
14     }
15   }
16 } end_atom
```

Figure 19. Pseudocode for Locks Inside of Transaction (LiT) I/O Operations.

While it is possible to implement the `shutDown` logic in Figure 19 outside of the transaction, its natural location is within the transaction. Furthermore, the locking behavior within `lock_recv()` and `lock_write()` naturally compose within the transaction [7]. The composition of `lock_recv()` and `lock_write()` is critical to ensuring the atomicity of the shut down behavior. If the atomicity of the shut down operation is not respected, other threads may continue processing socket or file data, breaking the correctness of the system. In particular, in a non-atomic shut down operation, other threads may be able to read and process the log file. The incurred overhead of reading and processing the log file will cause a delay in the shut down procedure. The delayed shut down may then trigger a forced shut down operation which kills the execution of the software, causing an unrecoverable loss of data in the log file.

8. Conclusion

We presented a simplified parallel programming model for C++. We briefly compared our simplified approach to a Java language extension approach and the standard C++ threading library approach. While the upcoming C++ threading library is important as a general solution, it may have delays before becoming available and place additional pressure on novice and expert parallel programmers. Our solution provides a parallel programming model that is simple and extensible, yet powerful in that it unifies locks and transactions under a single model. Our programming model supports locks, timed locks and transactions. The implementation of the programming model is achieved entirely through a software library, using automatic objects and preprocessor macros, making it easy to adopt while imposing negligible development environment costs compared to other LATM approaches.

We also explored DracoSTM's transaction-lock execution model. The paper provided a basic overview of DracoSTM's lock-aware transactional memory policies, as well as a brief explanation of why transaction-lock interaction requires management to avoid transaction violation. Next, we presented the essential modes of execution. These modes are useful for integrating lock-based legacy code with transactions and integrating lock-based idioms, such as irreversible operations, within transactions. We closed with an example expressing the benefit of lock-based implementations performing I/O operations coupled with transactional composition for correctness.

9. Acknowledgements

We would like to thank Vicente Botet and Joaquin Lopez for their feedback on this work and our underlying implementation. We were unable to augment the language-like constructs in a timely fashion for this paper, but we will incorporate their insightful recommendations into the next releasable version of DracoSTM.

References

[1] A.-R. Adl-Tabatabai, D. Dice, M. Herlihy, N. Shavit, C. Kozyrakis, C. von Praun, and M. Scott. Potential show-stoppers for transactional synchronization. In *PPOPP*, page 55, 2007.

[2] P. Becker. A multi-threading library for standard C++, revision 1. Technical Report N1907=05-0167, Oct 2005.

[3] L. Dalessandro, V. J. Marathe, M. F. Spear, and M. L. Scott. Capabilities and limitations of library-based software transactional memory in c++. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Transactional Computing*. Portland, OR, Aug 2007.

[4] D. Dice, M. Moir, and N. Shavit. Sun Microsystems: Transactional memory.

[5] J. E. Gottschlich and D. A. Connors. DracoSTM: A practical C++ approach to software transactional memory. In *ACM SIGPLAN Library-Centric Software Design (LCSD)*, Oct. 2007.

[6] J. E. Gottschlich and D. A. Connors. Extending contention managers for user-defined priority-based transactions. In *Proceedings of the 2008 Workshop on Exploiting Parallelism with Transactional Memory and other Hardware Assisted Methods*. Apr 2008.

[7] J. E. Gottschlich, D. A. Connors, D. Y. Winkler, J. G. Siek, and M. Vachharajani. An intentional library approach to lock-aware transactional memory. Technical Report CU-CS 1048-08, University of Colorado at Boulder, Oct 2008.

[8] 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.

[9] M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 253–262, New York, NY, USA, 2006. ACM.

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

[11] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Elsevier, Inc., 2008.

[12] N. M. Josuttis. *The C++ Standard Library: a tutorial and reference*. Addison-Wesley, 1999.

[13] J. R. Larus and R. Rajwar. *Transactional Memory*. Morgan & Claypool, 2006.

[14] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[15] E. Moss. Nesting transactions: Why and what do we need? *TRANSACT*, Jun 2006.

[16] J. E. B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. PhD thesis, 1981.

[17] Y. Ni, V. S. Menon, A.-R. Adl-Tabatabai, A. L. Hosking, R. L. Hudson, J. E. B. Moss, B. Saha, and T. Shpeisman. Open nesting in software transactional memory. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, 2007.

[18] Y. Ni, A. Welc, A.-R. Adl-Tabatabai, M. Bach, S. Berkowits, J. Cownie, R. Geva, S. Kozhukow, R. Narayanaswamy, J. Olivier, S. Preis, B. Saha, A. Tal, and X. Tian. Design and implementation of transactional constructs for C/C++. In *Proceedings of the 23rd ACM SIGPLAN conference on Object oriented programming systems languages and applications*. ACM, 2008.

[19] T. Peierls, B. Goetz, J. Bloch, J. Bowbeer, D. Lea, and D. Holmes. *Java Concurrency in Practice*. Addison-Wesley Professional, 2005.

[20] M. F. Spear, L. Dalessandro, V. Marathe, and M. L. Scott. A comprehensive strategy for contention management in software transactional memory. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming 2009*, Feb. 2009.

[21] M. F. Spear, V. J. Marathe, W. N. Scherer III, and M. L. Scott. Conflict detection and validation strategies for software transactional memory. In *Proceedings of the 20th International Symposium on Distributed Computing*, Sep 2006.

[22] B. Stroustrup. *The C++ Programming Language (3rd Edition)*. Reading, Mass., 1997.

[23] H. Volos, N. Goyal, and M. M. Swift. Pathological interaction of locks with transactional memory. February 2008.

[24] L. Ziarek, A. Welc, A.-R. Adl-Tabatabai, V. Menon, T. Shpeisman, and S. Jagannathan. A uniform transactional execution environment for Java. In *ECOOP*, pages 129–154, 2008.