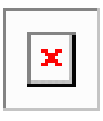
Toward.Boost.STM

Justin E. Gottchlich

Vicente J. Botet Escriba

Table of Contents



Warning

STM is not a part of the Boost libraries.

Overview

Description

Toward An Industrial Strength C++ Software Transactional Memory Library.

Transactional memory (TM) is a new parallel programming mechanism that reduces the complexity of parallel programming. TM reduces parallel programming complexity by abstracting away the necessary synchronization mechanisms from the parallel code, allowing the programmer to write parallel applications without worry of deadlocks, livelocks or race conditions.

Transactional memory is an active research interest for many academic and industry institutions with many open questions about its behavior.

TBoost.STM is a C++ lock-based software transactional memory (STM) library. Our approach to STM is to use only native language semantics while implementing the least intrusive, most type-safe object oriented solution possible.

TBoost.STM provides:

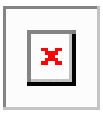
- Optimistic concurrency
- ACI transactions
  - Atomic: all operations execute or none do
  - Consistent: only legal memory states
  - Isolated: other txes cannot see until committed
- Language-like atomic transaction macro blocks
- Closed, flattened composable transactions
- Direct and deferred updating run-time policies
- Validation and invalidation conflict detection policies
- Lock-aware transactions
- Programmable contention management, enabling programmers to specify forward progress mechanisms
- Isolated and irrevocable transactions for transactions that must commit (i.e., I/O transactions)

How to Use This Documentation

This documentation makes use of the following naming and formatting conventions.

- Code is in `fixed width font` and is syntax-highlighted.
- Replaceable text that you will need to supply is in *`italics`* .
- If a name refers to a free function, it is specified like this: `free_function()`; that is, it is in code font and its name is followed by `()` to indicate that it is a free function.
- If a name refers to a class template, it is specified like this: `class_template<>`; that is, it is in code font and its name is followed by `<>` to indicate that it is a class template.

- If a name refers to a function-like macro, it is specified like this: `MACRO()`; that is, it is uppercase in code font and its name is followed by `()` to indicate that it is a function-like macro. Object-like macros appear without the trailing `()`.
- Names that refer to *concepts* in the generic programming sense are specified in CamelCase.



Note

In addition, notes such as this one specify non-essential information that provides additional background or rationale.

Finally, you can mentally add the following to any code fragments in this document:

```
// Include all of InterThreads
#include <boost/stm.hpp>

// Create a namespace aliases
namespace bstm = boost::stm;
```

Introduction

Transactional Memory

Transactional memory is a modern type of concurrency control that uses transactions as its synchronization mechanism. A transaction is a finite sequence of operations that are executed in an atomic, isolated and consistent manner. The atomicity, isolation and consistency (ACI) of transaction's are derived from the ACID principle in the database community. TM does not exhibit the D (durability) of ACID because unlike database transactions, TM transactions are not saved to permanent storage (e.g., hard drives).

Transactions are executed speculatively (optimistically) and are checked for consistency at various points in the transaction's lifetime. Programmers specify the starting and ending points of a transaction. All of the operations between those points make up the transaction's execution body. Transactions are commonly represented using the atomic structure shown in the figure below.

```
1   atomic
2   {
3       ++x;
4       --y;
5   }

Simple Transaction Using the atomic Keyword
```

Once a transaction has started it either commits or aborts. A transaction's operations are only seen once the transaction has committed, providing the illusion that all of the operations occurred at a single instance in time. The instructions of a committed transaction are viewed as if they occurred as an indivisible event, not as a set of operations executed serially. The operations of an aborted transaction are never seen by other threads, even if such operations were executed within a transaction and then rolled back.

In the case of the above example, if the transaction was committed both operations `++x` and `--y` would be made visible to other threads at the same instance in time. If the transaction the above example was aborted, neither operation (`++x` and `--y`) would appear to have occurred even if the local transaction executed one or both operations.

TM offers three distinct advantages over other parallel programming abstractions.

1. TM is simple; transactions, the synchronization mechanism of TM, are easier to program than other synchronization mechanisms because they move shared memory management into the underlying TM subsystem, removing its complexity from the programmer's

view. Moreover, TM exposes a simple programmer interface, reducing (or in some cases, removing) the potential for deadlock, livelock and priority inversion.

2. TM is scalable; it achieves increased computational throughput when compared to other parallel programming abstractions by allowing multiple threads to speculatively execute the same critical section. When concurrently executing threads do not exhibit shared data conflicts, they are guaranteed to make forward progress.

3. TM is modular; transactions can be nested to any depth and function as a single unit. This behavior allows application programmers to extend atomic library algorithms into atomic domain-specific algorithms without requiring the application programmers to understand the implementation details of the library algorithm.

For these reasons, transactions are considered an important synchronization mechanism and TM is viewed as an important type of concurrency control. The remainder of this section presents TM from a viewpoint of (1) simplicity, (2) scalability and (3) modularity.

Simplicity

Synchronization problems, such as deadlocks, livelocks and priority inversion are common in software systems using mutual exclusion. TM avoids many of these problems by providing a synchronization mechanism that does not expose any of its implementation details to the programmer. The only interfaces the programmer needs to use for TM is as follows:

- begin_tx() - the signaled start of the transaction.
- read(loc) - reads the specified memory location, storing its location in the transaction's read set and returning its current value.
- write(loc, val) - writes the specified memory location to the supplied val, storing its location in the transaction's write set.
- end_tx() - the signaled end of the transaction. end_tx() returns true if the transaction commits, otherwise it returns false.

The above interfaces allow the programmer to create a transaction (using begin_tx()), specify its memory operations (using read() and write()) and terminate (using end_tx()()). Moreover, none of the interfaces specify details of the TM subsystem's implementation. This leaves the TM system's implementation disjoint from the interfaces it supplies, a key characteristic for TM's simplicity.

All TM implementations use some combination of the above interfaces. TMs implemented within compilers tend to implicitly annotate transactional read() and write() operations, whereas those implemented within software libraries tend to require the programmer explicitly state which operations are transactional reads and writes. An example of a transaction using the above interfaces alongside an actual STM library implementation is shown in Figure 3.

```
1   // TM interfaces            // __Boost_STM__
2   do {                        atomic(t)
3       begin_tx();             {
4       write(x, read(x)+1);        ++t.write(x);
5       write(y, read(y)-1);        --t.write(y);
6   } while (end_tx());         } before_retry {}

Figure 3. Transaction Using (1) Explicit TM Interfaces and (2) TBoost.STM.
```

Figure 3 implements the same transaction as shown in Figure 2, except all transactional memory accesses, including the transaction's retry behavior (e.g., its loop), are demonstrated from a simple TM interface perspective and an actual library implementation (TBoost.STM). While most TM systems handle some portion of these interface calls implicitly, as is shown in the TBoost.STM transaction, it is important to note that even when all operations are made visible to the end programmer, transactions are still devoid of many concurrency problems, such as data races and deadlocks (explained below), that plague other types of concurrency control.

For example, as long as the programmer properly annotates the access to the shared variables x and y as shown in Figure 3, it is impossible for race conditions or deadlocks to occur. Furthermore, the programmer does not need any program-specific knowledge to use shared

data; he or she simply uses the TM interfaces supplied by the system and the resulting behavior is guaranteed to be consistent. This is explained in greater detail in Section 3.1.

Other types of concurrency control, such as mutual exclusion, cannot achieve the same interface simplicity, because part of their implementation is associated with, or exposed through, their interface. To demonstrate this, consider the fine-grained locking example of Figure 1 as shown below.

```
1   // fine-grained locking
2   lock(mutexX);
3   lock(mutexY);
4   ++x;
5   --y;
6   unlock(mutexX);
7   unlock(mutexY);
```

There is no universal interface that can be used to properly access the shared data protected by the mutual exclusion in the above fine-grained locking example. Instead, the programmer must be aware that mutexX and mutexY protect shared data x and y and, therefore, the locks must be obtained before accessing the shared data. In short, the programmer is responsible for knowing not only that mutual exclusion is used, but also how it is used (e.g., which locks protect which shared variables). In this case, mutexX must be obtained before mutexY. If another section of code implements the following, a deadlock scenario will eventually occur.

```
1   // fine-grained locking
2   lock(mutexY);
3   lock(mutexX); // deadlock here
4   --y;
5   ++x;
6   unlock(mutexY);
7   unlock(mutexX);
```

Understanding Concurrency Hazards

Informally, a concurrency hazard is a condition existing in a set of operations that, if executed with a specific concurrent interleaving, results in one or many unwanted sideeffects. Most errors in parallel systems, such as deadlocks and priority inversion, are the specific execution of concurrency hazards resulting in the unwanted side-effect(s) they contain. If the concurrency hazards are eliminated, the parallel system errors contained within the concurrency hazards are also eliminated. Unfortunately, detecting existing concurrency hazards is non-trivial and therefore eliminating them is also non-trivial.

Mutual exclusion exhibits more concurrency hazards than TM because its implementation details (i.e., its locks) must be exposed and used by the end programmer. While the locks used to enforce mutual exclusion by themselves are not concurrency hazards, their use can lead to a number of hazards. As such, using locks leads to concurrency hazards.

Because the mutual exclusion locking details are exposed to the programmer and because the programmer must maintain a universal and informal contract to use these locks, concurrency hazards can arise due to the number of possible misuses that can be introduced by the programmer. In particular, if the programmer accidentally deviates from the informal locking contract, he or she may inadvertently introduce a concurrency hazard that can cause the program to deadlock, invert priority or lead to inconsistent data.

In contrast, TM has no universal or informal contract between shared data that the end programmer needs to understand and follow as is required in mutual exclusion. Due to this, TM can hide its implementation details which results in reduced concurrency hazards. In particular, each transaction tracks the memory it uses in its read and write sets. When a transaction begins its commit phase, it verifies its state is consistent and commits its changes. If a transaction finds its state is inconsistent, it discards its changes and restarts. All of this can be achieved using the basic TM interfaces shown in Section 3 without exposing any implementation details. In order to use TM, the end programmer only needs to know how to correctly create a transaction. Once the transaction is executed, regardless of how it is executed, it results in a program state that is guaranteed to be consistent.

Fundamentally, TM exhibits less concurrency hazards than mutual exclusion because its implementation details are divorced from its interface and can therefore be hidden within its subsystem. Any number of implementations can be used in a TM subsystem using only the basic TM interfaces shown in Section 3. The same is not true for mutual exclusion. Mutual exclusion, regardless of how it is implemented, exposes details of its implementation to the programmer. As demonstrated in Section 5, mutual exclusion does not provide software modularity specifically because extending an existing module requires an understanding and extension of that module's implementation. When such locking implementations are hidden inside of software libraries, extending these modules can range from difficult to impossible.

Testing: Race Conditions and Interleavings

A race condition is a common concurrency hazard that exists in parallel or distributed software. As with all concurrency hazards, race conditions rely on a specific interleaving of concurrent execution to cause their unwanted side-effect. In this section we demonstrate that race conditions do not exist in TM and therefore, software testing is greatly simplified because all possible interleavings do not need to be tested to ensure correct system behavior. In order to demonstrate that race conditions are absent from TM, we must first show that they are present in other types of concurrency control.

```
1   // Thread T1 // Thread T
2   lock(L2);
3   lock(L1);
4   ...
5   unlock(L1);
6   unlock(L2);
7   lock(L1);
8   lock(L2);
9   ...
```

Figure 4. Mutual Exclusion Race Condition.

Consider the race condition present in the mutual exclusion example shown in Figure 4. The race condition present in the example results in a deadlock if thread T1 executes line 7 followed by thread T2 executing line 2. However, if the program executes the lines in order (e.g., line 1, then line 2, then line 3, etc.), the system will execute properly. The fundamental problem in Figure 4 is that it contains a concurrency hazard; in particular, it contains a race condition. To further complicate matters, the race condition can only be observed in two of many possible concurrent executions. Those two executions are: T1 executes line 7 followed by T2 executing line 2 or T2 executes line 2 followed by T1 executing line 7. All other possible concurrent interleavings of threads T1 and T2 avoid the deadlock race condition. More specifically, as long as T1 executes lines 7-8 atomically or T2 executes line 2-3 atomically, all remaining concurrent interleavings are free of the deadlock race condition.

Because it is unlikely that the deadlock race condition will occur, the programmer may never observe it, no matter how many times the program is tested. Only exhaustive testing, which tests all possible concurrent interleavings, is guaranteed to identify the presence of the deadlock. Regrettably, exhaustive testing is an unrealistic solution for most programs due to the time it would take to execute all possible concurrent interleavings of the program.

An alternative to exhaustive testing is for programmers to use types of concurrency control that are devoid of certain concurrency hazards. For example, if mutual exclusion did not emit the race condition concurrency hazard, it would be impossible for a program using it to deadlock. Therefore, exhaustive testing would not be necessary. While this scenario is hypothetical, it illustrates our larger argument: in order to avoid common parallel problems in a practical fashion, programmers may need to only use types of concurrency control that are devoid of certain concurrency hazards. By doing this, the program using the specific type of concurrency control will be guaranteed to be free of certain common parallel problems.

TMs are required to be devoid of race conditions within their implementations because they must enforce the ACI (atomic, consistent and isolated) principles. Transactions must execute as atomic and isolated and, therefore, TMs are not capable of supporting concurrent interleavings between multiple transactions as that would violate the atomic and isolated principles of ACI. Due to this, programs only using TM are guaranteed to be free of deadlocks (i.e., deadlockfreedom). Moreover, because TM implementations can guarantee freedom of race condition concurrency hazards, programmers only need to verify their transactional code is correct in a sequential (non-parallel)

manner. Once the sequential execution of the transactional code has been verified, no more testing is required as the TM system is required to behave in a consistent manner for all serial orders.

Development: Mutual Exclusion and TM

The development of fine-grained locking is notoriously difficult. Designing such software is equally as hard. The difficulty in developing and designing fine-grained locking systems is rooted in conflicting heuristics. A primary goal of software design is to identify the most simplistic software solution that exists for a particular problem. A primary goal of fine-grained locking is the most efficient concurrent implementation of a software system. The goals of software design and fine-grained locking are conflicting because the most efficient fine-grained locking solution usually requires some of the most complex software design implementations to achieve such performance.

TM achieves scalability by using optimistic concurrency that is implemented within its subsystem (see Section 4). Since the TM subsystem is the efficiency throttle for TM, which is unexposed to the programmer, the software architecture and design never needs to be complicated (nor can it be) in order to achieve increased parallelism when using transactions. As will be demonstrated in the following section, transactions run efficiently using the interfaces shown in this section and are never complicated in order to achieve improved performance, as is commonly found in fine-grained mutual exclusion implementations.

Scalability

In this section we analyze the scalability of TM compared to mutual exclusion. We measure scalability by two metrics: consistency and performance. A concurrency control type has consistent scalability if it guarantees correct behavior for an arbitrarily large number of concurrently executing processes.4 Performance scalability is measured by the maximum number of consistent processes supported by a concurrency control type while executing concurrently.

Pessimistic and Optimistic Critical Sections

Critical sections can be pessimistic or optimistic. Pessimistic critical sections limit their critical section execution to a single thread. Locks are an example of a synchronization mechanism that use pessimistic critical sections. Optimistic critical sections allow unlimited concurrent threaded execution. Transactions are an example of a synchronization mechanism that use optimistic critical sections.

Truly Optimistic Critical Sections

Truly optimistic critical sections are those critical sections which allow multiple conflicting threads to simultaneously execute the same critical section. A deferred update (or lazy acquire) TM system supports truly optimistic critical section. A direct update (or eager acquire) TM system does not support truly optimistic critical sections. More details on deferred and direct update TM systems are presented in the subsequent sections.

Truly optimistic critical sections are important because they allow simultaneous conflicting critical section execution, as opposed to disallowing such behavior. It is important to allow conflicting critical section execution because prematurely preventing concurrently executing threads pessimistically degrades performance. To demonstrate this, consider two transactions, called T1 and T2, executing the same critical section. Transaction T1 starts first and tentatively writes to memory locationM. Transaction T2 then starts and tries to write to memory locationM. In a truly optimistic TM system, T2 would be allowed to tentatively write to location M while T1 is also writing to M. This behavior then allows T2 to commit before T1 in the event T2 completes before T1.

In comparison, if the TM system is not truly optimistic, once T1 writes to M, T2 must stall until T1 completes. This pessimistically degrades the performance of the system by prematurely deciding that T1's transactional execution should have higher priority than T2's.

```
1   // global variables
2   int g1 = 0; int g2 = 0;
3
4   void set1(int val) { atomic { g1 = val; } }
5   void set2(int val) { atomic { g2 = val; } }
6   int get1() { atomic { return g1; } }
7   int get2() { atomic { return g2; } }
```

Figure 5. Truly Optimistic Concurrency Diagram.

Furthermore, and perhaps more importantly, truly optimistic critical sections allow readers and writers of the same memory location to execute concurrently. This behavior is important because in many cases, both the readers and writers of the same memory can commit with consistent views of memory.

An example of this is shown in Figure 5. As demonstrated in Figure 5 thread 1 and thread 2, which we'll refer to as T1 and T2 respectively, operate on the same memory locations (g1 and g2). Because the TM system supports optimistic concurrency, T2 is allowed to execute concurrently alongside T1 even though their memory accesses conflict. However, in this scenario, because T2 completes its workload before T1, both transactions are allowed to commit. T2 captures the state of g1=0,g2=0 while T1 sets the state of g1=1,g2=1. As the example addresses, both g1=0,g2=0 and g1=1,g2=1 are legal states.

Direct and Deferred Update

Updating is the process of committing transactional writes to global memory and is performed in either a direct or deferred manner. Figure 6 presents a step-by-step analysis of direct and deferred updating.

Deferred update creates a local copy of global memory, performs modifications to the local copy, and then writes those changes to global memory if the transaction commits. If the transaction aborts, no additional work is done. Direct update makes an original backup copy of global memory and then writes directly to global memory. If the transaction commits, the transaction does nothing. If the transaction aborts, the transaction restores global memory with its backup copy. Some TM systems favor direct update due to its natural optimization of commits (BSTM, McRTSTM and LogTM). However, other TM systems favor deferred update due to its support for truly optimistic critical sections (TBoost.STM and RingSTM).

Direct update enables greater TM throughput when aborts are relatively low because it optimizes the common commit case. Deferred update enables greater TM throughput when

1. aborts are relatively high or

2. short running transactions (e.g., those that complete quickly) are executed alongside long running transactions (e.g., those that do not complete quickly) because long running transactions do not stall shorter running ones as they would in direct update systems, and therefore the fastest transactions can commit first.

It is important to note that deferred update supports truly optimistic critical sections without special effort, while direct update does not. Truly optimistic critical sections enable the speculative execution of transactions that arrive after a memory location has already been tentatively written to by another transaction. This allows the first transaction, of potentially many competing transactions, to complete its commit, whether it be the later arriving transaction or the earlier arriving writer. This scenario is not possible with direct update without special effort.

Scalability: Mutual Exclusion and Transactional Memory

The scalability of mutual exclusion is limited to pessimistic concurrency. By definition, mutual exclusion's critical sections must be pessimistic, otherwise they would not be isolated to a single thread (i.e., they would not be mutually exclusive). TM, however, is generally implemented using optimistic concurrency, but it can enforce pessimistic concurrency amongst transactions if that behavior is required for certain conditions. In certain cases, TMs becomemore strict and execute pessimistically to enable inevitable or irrevocable transactions. Such transactions have significant importance for handling operations that, once started, must complete (e.g., I/O operations).

Since TM can execute optimistically and pessimistically, it is clear that whatever benefits pessimistic concurrency has can be acquired by TM. However, since mutual exclusion can only execute pessimistically, the advantages found in optimistic concurrency can never be obtained by mutual exclusion.

When one first analyzes pessimistic and optimistic concurrency, it may seem that the only benefit optimistic concurrency has over pessimistic concurrency is that multiple critical sections, which conflict on the memory they access, can execute concurrently. The simultaneous execution of such conflicting critical sections allows the execution speed of such critical sections to guide the system in deciding which execution should be allowed to commit and which should be aborted. In particular, the first process to complete the critical section can be allowed to abort the other process of the system. The same scenario cannot be achieved by pessimistic critical sections and is demonstrated in Section 4.1.1.

A counterargument to this scenario is that such optimistic concurrency only allows one critical section to commit, while one must be aborted. Because mutual exclusion only allows one conflicting critical section execution at a time, and because mutual exclusion does not support failure atomicity (i.e., rollbacking of the critical section), mutual exclusion's pessimistic behavior is superior in terms of energy and efficiency. Mutual exclusion, unlike TM, suffers no wasted work because conflicting critical sections are limited to a single thread of execution, reducing the energy it uses. Furthermore, because mutual exclusion does not require original data to copied, as needed for TM's direct or deferred update, it executes faster.

While there is merit to this counterargument, an important scenario is not captured by it: truly optimistic critical sections can support multiple reader / single write executions which, if executed so the readers commit before the writer, all critical sections will succeed. This scenario is impossible to achieve using pessimistic critical sections. Although mutual exclusion can use read/write locking, as soon as a writer thread begins execution on a conflicting critical section, all readers must be stalled. TM's truly optimistic concurrency does not suffer from this overly pessimistic limitation of throughput and is therefore capable of producing an immeasurable amount of concurrent throughput under such conditions.

From a theoretical perspective, given L memory locations and P processes, mutual exclusion can support the consistent concurrent execution of P*L number of readers or L writers. TM can support the consistent concurrent execution of P*L number of readers and L writers. Using the above variables, the mathematical expression of the performance scalability of mutual exclusion (S(ME)) is:

```
S(ME) = (P*L) + L
```

Using the same variables, the mathematical expression of the performance scalability of transactional memory is:

```
S(TM) = (P * L) + L
```

As should be clear from the above equations, mutual exclusion cannot achieve the same performance scalability of TM. This is because TM supports truly optimistic concurrency and mutual exclusion is confined to pessimistic concurrency. While other examples exist that demonstrate optimistic concurrency can increase throughput via contention management, the above equations capture the indisputable mathematical limitations in mutual exclusion's performance scalability.

Modularity

Software modularity is an important aspect of software that is necessary for its reuse. Formally, software is modular if it can be composed in a new system without altering its internal implementation. Informally, software is modular if it can be used, in its entirety, through its interface.

By making software modular, it can be freely used in an unlimited number of software systems. Without software modularity, software can only be used in the original system where it was written. Clearly, without software modularity, software cannot be reused. Because most software developments are based on extensive library use, software reuse is an integral part of software development. As such, limiting software reuse, would result in severely hampered development capabilities and overall development time. For these reasons, software modularity is vital for any software paradigm to be practical. Software paradigms that do not support software modularity are, in short, impractical.

Mutual Exclusion and Software Modularity

In this section, we show that mutual exclusion, regardless of its implementation, fails to deliver software modularity. We demonstrate this through a running example started in Figure 7 which implements inc(), mult() and get(); these functions use lock G to respectively implement an increment, multiply and get operations for the shared data.

```
1   void inc(int v) {
2       lock(G); g += v; unlock(G);
3   }
4
5   void mult(int v) {
6       lock(G); g *= v; unlock(G);
7   }
8
9   int get() {
10      lock(G); int v = g; unlock(G);
11      return v;
12  }
```

Figure 7. Mutual Exclusion for Increment, Multiply and Get of Shared Variable.

Now suppose a programmer wants to increment and multiply by some values within the same atomic operation. The initial implementation may look like the following.

```
1   inc(a);
2   mult(-b);
```

An unwanted side-effect of such an implementation is the exposure of the intermediate state of g between inc() and mult(). A second thread performing a get() may read an inconsistent value of g; the value of g between inc() and mult(). This is demonstrated in the timing diagram of Figure 8 .

Figure 8. Example of Exposed State of Mutual Exclusion.

If the programmer needs the inc() and mult() operations to be executed together, without an intermediate state being revealed, he or she could make lock G reentrant. Reentrant locks are locks that can be obtained multiple times by a single thread without deadlocking. If G is made reentrant, the following code could be used to make inc(a) and mult(-b) atomic. A basic implementation of a reentrant lock is to associate a counter with its lock and increment the counter each time the lock() interface is called and to decrement the counter each time the unlock() interface is called. The reentrant lock is only truly locked when a call to lock() is made when its associated counter is 0. Likewise, the reentrant lock is only truly unlocked when a call to unlock() is made when its associated counter is 1.

```
1   lock(G);
2   inc(a);
3   mult(-b);
4   unlock(G);
```

If the above code uses reentrant locks, it will achieve the programmer's intended atomicity for inc() and mult(), isolating the state between inc() and mult(), which disallows the unwanted side-effect shown in Figure 8. While the atomicity of the operations is achieved, it is only achieved by exposing the implementation details of inc() and mult(). In particular, if the programmer had not known that lock G was used within inc() and mult(), making an atomic operation of inc() and mult() would be impossible.

An external atomic grouping of operations is impossible using embedded mutual exclusion without exposing the implementation details because the heart of mutual exclusion is based on named variables which the programmer specifies to guard their critical sections. Because

these variables are named, they cannot be abstracted away and any programmer wishing to reuse the mutually exclusive code must be able to access and extend the implementation details.

```
1   void inc(int v) { atomic { g += v; } }
2
3   void mult(int v) { atomic { g *= v; } }
4
5   int get() { atomic { return g; } }
```

Figure 9. TM of Increment, Multiply and Get of Shared Variable.

Summary of Mutual Exclusion Modularity

As we presented at the beginning of this section, software modularity can be informally understood as a component's ability to be used entirely from its interface. Therefore, components that cannot be used entirely from their interface, components that must expose their implementation details to be extended, are not modular. As such, the paradigm of mutual exclusion does not support software modularity.

Transactional Memory and Software Modularity

Transactional memory works in a fundamentally different manner than mutual exclusion, with regard to its interface and implementation. To begin, as demonstrated in Section 3, TMs do not generally expose any of their implementation details to client code. In fact, in many TMs, client code is more versatile if it knows and assumes nothing about the active implementation of the TM. By abstracting away details of TM implementation, a TM subsystem can adapt its behavior to the most efficient configuration for the program's current workload, much like the algorithms used for efficient operation of processes controlled by operating systems. TM uses such abstractions to optimize the performance of concurrent programs using various consistency checking methods, conflict detection times, updating policies, and contention management schemes.

Achieving TM Software Modularity

TM achieves software modularity by allowing transactions to nest. With transactional nesting, individual transactions can be wrapped inside of other transactions which call the methods where they reside, resulting in a transaction composed of both the parent and child transaction. Furthermore, this is achieved without altering or understanding the child transaction's implementation. To best demonstrate transactional nesting, we reuse the prior mutual exclusion example shown in Figure 7 and implement it using transactions as shown in Figure 9.

As before, the programmer's goal is to implement a combination of inc() and mult() executed in an atomic fashion. The basic, and incorrect implementation is demonstrated below:

```
1   inc(a);
2   mult(-b);
```

Even with transactions, this approach fails because the transactions within inc() and mult() begin and end inside their respective functions. However, to make the above operations atomic, the programmer need only make the following modification shown in Figure 10.

```
1   atomic { // atomic {
2       inc(a); // atomic { g += a; }
3       mult(-b); // atomic { g *= -b; }
4   } // }
```

Figure 10. Modularity: Transaction of Increment and Multiply.

In effect, the TM system subsumes the transactions that are nested inside of the inc() and mult() operations. 6 The left side of Figure 10 shows the actual code of the transaction, while the right side shows the child transactions that are subsumed by the parent transaction.

Because transactions are isolated and atomic, the resulting state of g, from operations inc() and mult(), is invisible to outside observers until the transaction is committed. As such, outside threads cannot view any intermediate state constructed by partial transaction execution. The result of such isolated behavior is the guaranteed consistent concurrent execution of interleaved accesses to shared memory from in-flight transactions. This is demonstrated in Figure 11; let g=0 and assume deferred update is the active updating policy, as explained in Section 4.2.

```
Figure 11. Example of Isolated and Consistent State of TM.
```

As shown in Figure 11, multiple concurrently executing threads can read and write to the same shared memory in a consistent and isolated fashion when using TM. In the example, thread T2 performs x = get() after T1 has already executed inc(a). However, since T1 has not yet committed its transaction, T2's view of shared data g is consistent (g=0). When T2 begins the commit phase of its transaction, the TM subsystem verifies that shared data g has not been updated since it initially read it. Since no other transaction has updated shared data g, T2's transaction is permitted to commit. Thread T1 then continues with its mult() operation and then enters its commit phase. The TM subsystem also verifies the consistency of T1's transaction before it is allowed to commit. Again, since no one transaction has updated shared data g between its reads and writes to it, T1's transaction is permitted to commit.

The above analysis demonstrates that software modularity can be achieved in TM through transactional nesting (Figure 10). In this case, the specific software modularity achieved is extension to an existing critical section. Critical section extension was also possible with mutual exclusion, as demonstrated in Section 5.1, but only through exposing the details behind the mutual exclusion implementation. Due to this, mutual exclusion fails to deliver a practical level of software modularity.

Summary of Transactional Memory Modularity

TM supports software modularity by allowing transactions to nest, to any depth, while logically grouping the shared data accesses within the transactions into an atomic, consistent and isolated (ACI) operation. Transactional nesting is natural to the programmer because nested transactions behave in the same manner as unnested transactions. TM's ACI support ensures transactions will behave in a correct manner regardless of if the transaction is used by itself or subsumed into a larger transaction.

C++ library language-like solution

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. The next version of C++ will incorporate library support for threads, while numerous researchers are exploring ways to extend C++ to support transactional memory (TM).

A strength of C++ is its support for automatic objects. Rather than requiring that parallel primitives be added directly to the language, automatic objects in C++ can be used to implement much of their necessary infrastructure. 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.

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.

TBoost.STM present such a language-like solution for C++ using generic library coupled with a deliberate use of the preprocessor. The culmination of these components facilitate a simple, yet powerful, parallel programming interface in C++.

[/

Users'Guide

[/

Getting Started

Installing STM

TBoost.STM is currently in an alpha release stage. This means our STM library is downloadable, but is undergoing some fundamental changes. Furthermore, our alpha release of TBoost.STM may exhibit some stability issues. While our internal tests have shown TBoost.STM to be fairly stable, your usage may vary.

TBoost.STM's current alpha download is mainly intended for other researchers who are curious about TBoost.STM's implementation and transactional memory. While we encourage non-TM experts to download the library and explore writing parallel code within its framework, we want such users know the system is in its early stages. As such, your usage of TBoost.STM may reveal some weaknesses or stability issues. Until we are into a fully released version, these issues are to be expected. However, please let us know if you encounter any problems using our library and we will do our best to resolve such issues immediately.

Thank you for considering TBoost.STM and we hope you enjoy exploring transactional memory!

Getting Boost.STM

Visit our Toward Boost.Stm home page be aware of the last news.

You can get the last stable release of from the Download page.

You can also access the latest (unstable?) state from the Boost Sandbox.

Building Boost.STM

TBoost.STM is not a header only library. You need to compile it before use.

```
cd libs/stm/build
bjam
```

Requirements

The POSIX threads (pthreads) library is needed to use TBoost.STM. Pthreads is part of the standard deployment for almost all Unix / Linux flavors, however, it is not standard in Windows. If you are doing Windows development you can find the POSIX threads library at the below link: POSIX threads (pthreads) for Windows

In order to be more portable TBoost.STM is migrating to Boost. You should use either Boost version 1.39.x or the version in SVN trunk (even if Boost version 1.35.x should works also). In particular, TBoost.STM will depends on:

Boost.DynamicBitset from Jeremy Siek and Chuck Allison	dynamic_bitsets (replacement of the current bit_vector)
Boost.Thread from Anthony Williams	threads and synchronization primitives (making the lib portable to other platforms)
Boost.Pool from Stephen Cleary	Memory pool management (replacement of the current memory_pool)

In addition TBoost.STM will uses the following libraries on the Boost Sandbox

Boost.BloomFilters from Dean Michael Berris	bloom_filters (replacement of the current bloom_filter)
Boost.Chrono from Beman Dawes	Standard Chrono library (making the lib portable to other platforms)
Boost.Containers from Ion Gaztanaga	flat containers (replacement of the current vector_map and vector_set)
Boost.InterThreads from Vicente J. Botet Escriba	Thread specific shared pointer (replacement of the current maps having thread_id as domain)
Boost.Move from Ion Gaztanaga	Move semantics (replacement of the current Draco_move)
Boost.Synchro from Vicente J. Botet Escriba	Exception based timed locks synchronization primitives & Language-like Synchronized Block (making the lib portable to other platforms)

Exceptions safety

All functions in the library are exception-neutral and provide strong guarantee of exception safety as long as the underlying parameters provide it.

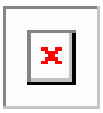
Thread safety

All functions in the library are thread-unsafe except when noted explicitly.

Tested compilers

Currently, **Boost.STM** has been tested in the following compilers/platforms:

- Visual Studio 6 - Windows
- GCC 3.4.4 - Cygwin



Note

Please send any questions, comments and bug reports to boost <at> lists <dot> boost <dot> org

Hello World!

The below example gives a basic introduction into TBoost.STM's transactional framework and demonstrates TBoost.STM's ACI conformance.

```
tx_obj<int> counter;

int increment() {
    int val = 0;
    use_atomic(_) {
        (*counter)++;
        val = *counter;
    }
    return val;
}
```

In the above example, (A) both the write on counter and the 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 TBoost.STM's conformance to the ACI principles. The above example also gives a basic introduction into TBoost.STM's transactional framework.

Tutorial

A number of example transactions are presented in this section using the TBoost.STM 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 TBoost.STM's extensible contention manager and compositional framework is provided. The final example demonstrates a number of important aspects of TBoost.STM's implementation, such as, differing priority inversion mechanics for different consistency models, transactional attribute enrichment via composition and threaded memory sharing amongst transactions.

While most of the examples are intuitive and a complete understanding of the TBoost.STM API is not needed for a high-level understanding of its functionality, a complete description of all interfaces used below can be found in the referenced section.

A Simple Transaction

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

Client Invoked Inserts

```
tx_ptr<linked_list<int> > llist;
...
for (int i = 0; i < 100; ++i) {
    use_atomic(_) {
        llist->insert(i);
    }
}
```

After inspecting the above client invoked insert code it is apparent that the code itself shows no deep signs of being transactional. The two exceptions are the declaration of the list using a tx_ptr smart pointer and the fact that the insertion of the 100 elements is atomic. 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.

Linked list declaration

A clasical linked list use a list node chained following the memer next_. On a transactional context we nedd to state that the pointer to the next node is a transactional one. This is reached using the smart pointer tx_ptr<>

```
template <typename T>
struct list_node {
    tx_ptr<list_node<T> > next_;
}

template <typename T>
class linked_list {
// ...
private:
    tx_ptr<list_node<T> > head_;
};
```

Insert retry transaction

```
void insert(T const &val) {
    for (boost::stm::transaction _; !_.committed() && _.restart(); _.no_throw_end())
        try {
            // ... see below
        } catch (aborted_transaction_exception&) {}
}
```

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 TBoost.STM 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.

Insert specific

```
upgrd_ptr<list_node<T> > prev(_, head_);
upgrd_ptr<list_node<T> > curr(_, head_->next_);
while (curr) {
    if (curr->value_ == val) return;
    else if (curr->value_ > val) break;
    prev = curr;
    curr = curr->next_;
}
if (!curr || (curr->value_ > val)) {
    make_wr_ptr(_, prev)->next_ =
        BOOST_STM_NEW(_,transactional_object<list_node<T> >(val, curr));
}
```

The above example illustrates the simplicity of TBoost.STM transactions and their interfaces. The transactional implementation is nearly identical to a non-transactional implementation with the exception of some annotations. The templatized 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

make_wr_ptr(), daisy-chained method invocation can be performed allowing streamlined usage. These aspects help make TBoost.STM transactions small and easy to understand.

upgrd_ptr is a read only smart pointer that can be upgraded to a read/write smart pointer. Once the smart pointer is constructed (associating it to the current transaction), the user can access any member as if it was non transactional.

make_wr_ptr is a read/write smart pointer factory. The returned wr_ptr<> allows to modify the pointee object.

One minor, but vital, detail is in the way new objects are created on a transactional context. 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.

In order to simplify the retry mechanism the macro `use_atomic` is provided so the user can just write

```
void insert(T const &val) {
    use_atomic(_) {
        upgrd_ptr<list_node<T> > prev(_, head_);
        upgrd_ptr<list_node<T> > curr(_, head_->next_);
        while (curr) {
            if (curr->value_ == val) return;
            else if (curr->value_ > val) break;
            prev = curr;
            curr = curr->next_;
        }
        if (!curr || (curr->value_ > val)) {
            make_wr_ptr(_, prev)->next_ =
                BOOST_STM_NEW(_,transactional_object<list_node<T> >(val, curr));
        }
    }
}
```

We could also write the following, but this is less efficient as tx_ptr is a smart pointer that associated the pointer to the current transaction for writting. The use of the smart pointer upgrd_ptr and the smart pointer factory function make_wr_ptr allows to make the difference kind of access.

```
void insert(T const &val) {
    use_atomic(_) {
        tx_ptr<list_node<T> > prev(head_);
        tx_ptr<list_node<T> > curr(head_->next_);
        while (curr) {
            if (curr->value_ == val) return;
            else if (curr->value_ > val) break;
            prev = curr;
            curr = curr->next_;
        }
        if (!curr || (curr->value_ > val)) {
            prev->next_ =
                BOOST_STM_NEW(_,transactional_object<list_node<T> >(val, curr));
        }
    }
}
```

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. TBoost.STM'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.

Client Invoked Inserts / Moves.

```
tx_ptr<linked_list<int> > llist;
...
atomic(_) {
    for (int i = 0; i < 100; ++i) {
        llist->insert(i);
    }
    for (int j = 0; j < 100; ++j) {
        llist->move(j, -j);
    }
}
```

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.

Remove

```
void remove(const T& val) {
    use_atomic(_) {
        // find the node whose val matches the request
        upgrd_ptr<list_node<T> > prev(_,head_);
        upgrd_ptr<list_node<T> > curr(_,prev->next_);
        while (curr) {
            // if we find the node, disconnect it and end the search
            if (curr->value_ == val) {
                make_wr_ptr(_, prev)->next_=curr->next_;
                delete_ptr(_,curr);
                break;
            } else if (curr->value_ > val) {
                // this means the search failed
                break;
            }
            prev = curr;
            curr = prev->next_;
        }
    }
}
```

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

Composed External Move

```
void move(T const &v1, T const &v2) {
    use_atomic {
        remove(v1);
        insert(v2);
    }
}
```

The move() implementation do not requires any overhead required in non-transactional implementations, other than using the atomic guard. The remainder of the code is preserved.

A Dynamically Prioritized, Composed Transaction

The following subsection discusses how priority inversion is handled within TBoost.STM 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 TBoost.STM 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.

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 TBoost.STM is performing invalidation.

The permission_to_abort() interface is called from TBoost.STM'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 (lefthand 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:

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. TBoost.STM 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 imple-

mented using a size_t type. Additionally, TBoost.STM 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.

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 TBoost.STM'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 TBoost.STM's internal priority processing which requires a additional amount of explanation, as follows.

TBoost.STM'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 TBoost.STM'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. TBoost.STM'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 TBoost.STM'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 TBoost.STM 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 TBoost.STM, 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 TBoost.STM built-in a default priority inversion handler. Considering these factors, as well as many others, TBoost.STM 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.

Can a transactional object embeed another transactional object?

Let me show what happens when we emmend transactional objects: class C : public transaction_object<C> {...};

```
class E : public transaction_object<E> {
    C c;
    int i;
};
```

E e;

When a thread T1 modifies e (and possibly the b part) on a transaction another thread T2 can modify e.c on another transaction. When the two transactional objects overlap the STM system is unable to detect this efficiently, having as consequence an inconsistency.

The following pseudo-code shows the

```
    T1                  T2
1   atomic {            atomic {
2       e.i=0;              e.c=A;
3                       }
4   }
```

When T2 commits on (3) e.c will be modified, but T1 will modify the complete E independent of the e.c modification doen by T2, overwriting the e.c with its old value. T1 will be coherent, but not T2. If we want to allow such a scheme, we have several possibilities

- implicit: take care of overlaping transactional objects, so we can detect when T1 commits that an overlaping TO e.c has been modified, so the transaction on T1 is aborted and retried.
- explicit: the developper of T1 must notify the STM system that the transactional object e constains a transactional object e.c
- declarative: the developper of E would declare that it contains an emmbeed transactional object C at E::c, so the STM system will use this explicit information when an E is accesses using the STM interface.

The implicit approach is the more transparent for the user but it is also the less efficient. It need to add a virtual function quering the size of the class, and to modify the conflic detection algorithm (maybe the write_list must be a write_set, ordered by the address)

The explit approach let the user with a lot of burden and risk of error when access to an E is done on a transaction. This needs to add the size and the algorithm which adds transactional object on the write_list (maybe the write_list must be a write_interval_set, ordered by the address).

The declarative concentrate on a single place, so the user will be not too concerned, and this open possible optimizations.

Supose now that we are able to embeed transactional objects. and that the T1 not only opens for writing e, but also e.c either on the same block on on a block (or nested transaction). The STM will see both address to be written, and even if there should be no risk of incoherence the object e.c will be copied twice.

Returning values from a fucntion

Returning from outside the transaction context

The simgle way consists is using a local variable declared outside the transaction context.

```
int inc_c() {
    int res;
    atomic(_) {
        res = *(_.read(c))+i;
    } end_atom;
    return res;
}
```

Returning from inside

The following attempt does not work as the transaction will not be commited.

```
int inc_c(int i) {
    atomic(_) {
        write_ptr<tx_int> tx_c(_,c);
        return *(tx_c)+=i;
    } end_atom;
}
```

We need to commit before returning. SO we will need any way a local variable storing the result before commiting.

```
int inc_c(int i) {
    atomic(_) {
        write_ptr<tx_int> tx_c(_,c);
        int res = *(tx_c)+=i;
        _.commit();
        return res;
    } end_atom;
}
```

Pointer to transactional objects

Access to a pointer to a transaction object works, but this do not take care of modifications of the pointer itself.

Using the mixin transaction_object<>

Let B the TO made using the mixin transaction_object.

```
class B : public transaction_object<B> {
public:
    void virtaul fct();
};
```

How a variable pointing to B must be declared. The fact that B is a transactional object do not means that a pointer to it is one. We need a class that wraps the pointer

```
template <typename TO>
class transaction_object_ptr : public transaction_object<transaction_object_ptr<TO> > {
public:
    TO* ptr_;
    typedef transaction_object_ptr<TO> this_type;
    typedef transaction_object<transaction_object_ptr<TO> > base_type;
    transaction_object_ptr() :  base_type(), ptr_(0) {
    }
    transaction_object_ptr(const transaction_object_ptr & rhs) :  base_type(rhs),  ptr_(rhs.ptr_) {
    }
    transaction_object_ptr(transaction_object_ptr & rhs) :  base_type(rhs), ptr_(rhs.rhs) {
    }
    transaction_object_ptr(TO* ptr) :  base_type(), ptr_(ptr) {
    }
    ~transaction_object_ptr() {
    }
    this_type& operator=(TO* rhs) {
        ptr_=rhs;
        return *this;
    }

    TO* get() const {
        std::cout << "get" << std::endl;
        return ptr_;
    }

    inline TO& operator*() const { return *get(); }
    inline TO* operator->() const { return get(); }
};
```

Let declare a pointer to B

```
transaction_object_ptr<B> ptr_b;


atomic(_) {
    write_ptr< pointer<B> > tx_ptr_b_ptr(_, ptr_b);
    *tx_ptr_b_ptr=BOOST_STM_NEW(_, B());
}
```

Polymorphic

How the user can defien a transactional class D inheriting from a transactional class B

:mixin Using the mixin transaction_object<>

Let B the base class

```
class B : public transaction_object<B> {
public:
    void virtaul fct();
};
```

The derived class must declare D as follows:

```
class D : public transaction_object<D,B> {
    ...
};
```

How a variable pointing to B must be declared. The fact that B is a transactional object do not means that a pointer to it is one.

```
pointer<B> ptr_b = 0
```

How can we assign a pointer to D?

```
write_ptr<B*> tx_ptr_b_ptr(_,ptr_b);
*ptr_b = BOOST_STM_NEW(_, D())
```

::wrapper Using the wrapper transactional_object<>

Can non transactional objets participate in a transaction?

Non transactional objects are the objects that do not inherit from base_transaction_object.

Lock aware

References

For an introduction to TM, please refer to

Shifting the Parallel Programming Paradigm Justin E. Gottschlich, Dwight Y. Winkler, Mark W. Holmes, Jeremy G. Siek, and Manish Vachharajani - *Raytheon Information Systems and Computing Symposium (ISaCTN), March 2009*

For details on the underlying software architecture of TBoost.STM, including explanations of its API, please refer to our summary paper.

DracoSTM: A Practical C++ Approach to Software Transactional Memory Justin E. Gottschlich and Daniel A. Connors - *Proceedings of the ACM SIGPLAN Symposium on Library-Centric Software Design (LCSD), October 2007*

TBoost.STM uses a novel method of consistency checking called invalidation. Invalidation can assist in both high performance and strict contention management control. For more information on how invalidation can improve overall system performance and ensure true user-defined contention management policies, see our below research papers.

A Consistency Checking Optimization Algorithm for Memory-Intensive Transactions Justin E. Gottschlich, Daniel A. Connors and Jeremy G. Siek - *Proceedings of the ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC) (brief announcement), August 2008*

Extending Contention Managers for User-Defined Priority Based Transactions Justin E. Gottschlich and Daniel A. Connors - *Proceedings of the ACM Workshop on Exploiting Parallelism with Transactional Memory and other Hardware Assisted Methods (EPHAM), April 2008*

TBoost.STM supports transaction-lock interaction. This support allows existing lock-based parallel code to interact safely with transactions. In addition, we provide three types of transaction-lock interaction with varying degrees of performance. Please see our research paper for more details.

An Intentional Library Approach to Lock-Aware Transactional Memory — Justin E. Gottschlich, Daniel A. Connors, Dwight Y. Winkler, Jeremy G. Siek and Manish Vachharajani -

TBoost.STM uses the last C++ features to provide a friendly and efficient interface. See the following papers to see how.

Toward Simplified Parallel Support in C++ — Justin E. Gottschlich, Jeremy G. Siek, Paul J. Rogers, and Manish Vachharajani - *BoostCon'09, the Third Boost Libraries Conference, May 2009*

C++ Move Semantics for Exception Safety and Optimization in Software Transactional Memory Libraries — Justin E. Gottschlich, Jeremy G. Siek and Daniel A. Connors - *International Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS), July 2008*

Check for the last updates.

TBoost.STM (or Related) Peer-Reviewed Publications

Glossary

ACI	Atomic, Consistent and Isolated
LATM	lock aware transaction memory
LiT	Lock inside Transactions
LoT	Lock outside Transactions
STM	Software Transaction Memory
TM	Transaction Memory
TX	Transaction
abort	.
Conflict Detection	.
commit	.
Consistency Checking	.
Deferred Updating	.
Direct Updating	.
Full-Lock protection	.
In fligh transaction	.
Irrevocable transaction	.

Isolated transaction .

TM-Lock protection .

TX-Lock protection .

commit .

commit .

commit .

Reference

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

Header <boost/stm/base_contention_manager.hpp>

```
namespace boost { namespace stm {
    class base_contention_manager;
}}
```

Abstract Class base_contention_manager

```
class base_contention_manager {
public:
    virtual void abort_on_new(boost::stm::transaction const &t) = 0;
    virtual void abort_on_delete(
                boost::stm::transaction const &t,
                boost::stm::base_transaction_object const &in) = 0;
    virtual void abort_on_read(
                boost::stm::transaction const &t,
                boost::stm::base_transaction_object const &in) = 0;
    virtual void abort_on_write(
                boost::stm::transaction &t,
                boost::stm::base_transaction_object const &in) = 0;
    virtual bool abort_before_commit(boost::stm::transaction const &t) = 0;
    virtual bool permission_to_abort(
                boost::stm::transaction const &lhs,
                boost::stm::transaction const &rhs) = 0;
    virtual bool allow_lock_to_abort_tx(
                int const & lockWaitTime,
                int const &lockAborted,
                bool txIsIrrevocable,
                boost::stm::transaction const &rhs) = 0;
    virtual int lock_sleep_time();
    virtual void perform_isolated_tx_wait_priority_promotion(
                boost::stm::transaction &) = 0;
    virtual void perform_irrevocable_tx_wait_priority_promotion(
                boost::stm::transaction &) = 0;
    virtual ~base_contention_manager() {};
};
```

Virtual destructor base_contention_manager

```
virtual ~base_contention_manager();
```

Virtual function abort_on_new

```
virtual void abort_on_new(boost::stm::transaction const &t) = 0;
```

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 function `abort_on_delete`

```
virtual void abort_on_delete(
            boost::stm::transaction const &t,
            boost::stm::base_transaction_object const &in) = 0;
```

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 function `abort_on_read`

```
virtual void abort_on_read(
            boost::stm::transaction const &t,
            boost::stm::base_transaction_object const &in) = 0;
```

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 function `abort_on_write`

```
virtual void abort_on_write(
            boost::stm::transaction &t,
            boost::stm::base_transaction_object const &in) = 0;
```

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 function `abort_before_commit`

```
virtual bool abort_before_commit(boost::stm::transaction const &t) = 0;
```

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 function `permission_to_abort`

```
virtual bool permission_to_abort(
            boost::stm::transaction const &lhs,
            boost::stm::transaction const &rhs) = 0;
```

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.

Virtual function `allow_lock_to_abort_tx`

```
virtual bool allow_lock_to_abort_tx(
            int const & lockWaitTime,
            int const &lockAborted,
            bool txIsIrrevocable,
            boost::stm::transaction const &rhs) = 0;
```

Virtual function `lock_sleep_time`

```
virtual int lock_sleep_time();
```

Virtual function `perform_isolated_tx_wait_priority_promotion`

```
virtual void perform_isolated_tx_wait_priority_promotion(
            boost::stm::transaction &) = 0;
```

virtual void perform_isolated_tx_wait_priority_promotion(boost::stm::transaction &) = 0;

Virtual function `perform_irrevocable_tx_wait_priority_promotion`

```
virtual void perform_irrevocable_tx_wait_priority_promotion(
            boost::stm::transaction &) = 0;
```

Header `<boost/stm/base_transaction_object.hpp>`

```
namespace boost { namespace stm {
    class base_transaction_object;
}}
```

Abstract Class `base_transaction_object`

This is the base class of all the transactional objects. It tracks:

- `transactionThread_`: the thread identifier holding the write acces to this transactional object
- `transaction_`: the pointer to the transaction
- `version_`: the version when performing validation
- `newMemory_`: states whether this object is a new object

Transactional objets must specialize the pure virtual functions

- `copy_state(base_transaction_object const * const rhs)`
- `move_state(base_transaction_object * rhs)` if `BUILD_MOVE_SEMANTICS`
- `cache_deallocate()` if `BOOST_STM_USE_UNASIGNED_COPY`

`copy_state` is used to copy the backup/working copy to the shared transactional object when the roolback/commit is done direct/defered policy is used.

`move_state` is used to move the backup/working copy to the shared transactional object when the roolback/commit is done direct/defered policy is used.

`cache_deallocate` is used to release the backup/working copy when the transaction ends if direct/defered policy is used.

When USE_STM_MEMORY_MANAGER is defined this class provides two functions (`retrieve_mem` and `return_mem`) and to manage a pool of memory.

```
class base_transaction_object
{
public:

    base_transaction_object();
    virtual ~base_transaction_object() {};

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

    void transaction_thread(thread_id rhs) const;
    thread_id transaction_thread() const;

    void new_memory(bool rhs)
    bool new_memory() const;

#if BOOST_STM_PERFORMING_VALIDATION
    void version(std::size_t rhs);
    std::size_t version() const;
#endif

#if BOOST_STM_USE_MEMORY_MANAGER
    static void return_mem(void *mem, size_t size);
    static void* retrieve_mem(size_t size);
    static void alloc_size(size_t size);
#endif

#ifdef BOOST_STM_USE_UNASIGNED_COPY
    virtual void cache_deallocate()=0;
#endif

};
```

Constructor `base_transaction_object()`

```
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 Destructor ~base_transaction_object()

```
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 function copy_state()

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

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.

Derived classes usually simply override this method and perform an `operator=()` function call for the specific derived type.

Virtual function move_state()

```
virtual void move_state(base_transaction_object * rhs) = 0;
```

The move_state() method is internally called at deferred updating commit time and at direct updating abort time and invokes the user-defined derived transaction class's move assignment (e.g. operator=(type &&)).

Static function retrieve_mem()

```
static void* retrieve_mem(size_t size);
```

Static interface into TBoost.STM'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.

Static function return_mem()

```
static void return_mem(void *mem, size_t size);
```

Static interface into TBoost.STM'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.

Static function alloc_size()

```
static void alloc_size(size_t size);
```

Static interface into TBoost.STM'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 TBoost.STM's default value, currently 8192. The alloc_size() interface can be reconfigured at runtime and is used upon the next buffer increase.

Virtual function `cache_deallocate()`

```
virtual void cache_deallocate()=0;
```

Header <boost/stm/cache_fct.hpp>

```
namespace boost { namespace stm {
    template <class T> T* cache_clone(const T& val);
    template <class T> void cache_copy(const T* const ori, T* target);
    void cache_release(base_transaction_object* ptr);

    template <class T> T* cache_allocate();
    template <class T> void cache_deallocate(T*);

}}
```

Template Function `cache_clone<>()`

```
template <class T>
T* cache_clone(const T& val);
```

Makes a new copy of the parameter. Allocates an object of type T using the `cache_allocate` function and copy from the given parameter `val` using the free function `cache_copy`.

The user can overload this function, as TBoost.STM uses ADL to access it.

When `BOOST_STM_NO_PARTIAL_SPECIALIZATION` is defined, i.e. on compilers not supporting partial template specialization, the function calls to `partial_specialization_workaround::cache_clone<T>::apply(val)`. So the user will need to overload the class `partial_specialization_workaround::cache_clone<T>` defining the function

```
static T* apply(const T& val);
```

Template Function `cache_copy<>()`

```
template <class T>
void cache_copy(const T* const source, T* target);
```

Copy from souce to target using memcpy by default.

The user can overload this function, as TBoost.STM uses ADL to access it.

When `BOOST_STM_NO_PARTIAL_SPECIALIZATION` is defined, i.e. on compilers not supporting partial template specialization, the function calls to `partial_specialization_workaround::cache_copy<T>::apply(source, target)`. So the user will need to overload the class `partial_specialization_workaround::cache_copy<T>` defining the function

```
static T* apply(const T* const source, T* target);
```

Function `cache_release<>()`

```
void cache_release(base_transaction_object* ptr) {
```

Release the given base_transaction_object by calling to the virtual function `cache_dealocate`.

Template Function `cache_allocate<>()`

```
template <class T>
T* cache_allocate();
```

Allocates an instance of calls T. Depending on whether BOOST_STM_CACHE_USE_MEMORY_MANAGER, BOOST_STM_CACHE_USE_MALLOC or BOOST_STM_CACHE_USE_TSS_MONOTONIC_MEMORY_MANAGER are defined this function will use the static pool of the class T, malloc or the thread specific monotonic allocator.

Template Function `cache_deallocate<>()`

```
template <class T>
void cache_deallocate(T*ptr) {
```

Deallocates an instance of calls T. Depending on whether BOOST_STM_CACHE_USE_MEMORY_MANAGER, BOOST_STM_CACHE_USE_MALLOC or BOOST_STM_CACHE_USE_TSS_MONOTONIC_MEMORY_MANAGER are defined this function will use the static pool of the class T, free or the no-op.

Header <boost/stm/data_types.hpp>

```
namespace boost { namespace stm {
    enum transaction_type;
    enum latm_type;
    enum transaction_state;
    typedef thread_id_t;
}}
```

Enum transaction_type

```
 enum transaction_type
 {
     eNormalTx,
     eIrrevocableTx,
     eIrrevocableAndIsolatedTx
};
```

Enum latm_type

```
enum latm_type
{
    eFullLatmProtection,
    eTmConflictingLockLatmProtection,
    eTxConflictingLockLatmProtection
};
```

Enum transaction_state

```
enum transaction_state
{
   e_no_state = -1, // no state is -1
   e_aborted,       // ensure aborted = 0
   e_committed,     // committed is positive
   e_hand_off,       // so is handoff in case bool conversion
   e_in_flight
};
```

Typedef thread_id_t

```
typedef platform_specific thread_id_t;
```

Header <boost/stm/exceptions.hpp>

```
namespace boost { namespace stm {
    class aborted_transaction_exception;
}}
```

Class aborted_transaction_exception

Header <boost/stm/language_like.hpp>

The 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. 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 TBoost.STM's API end() does. This non-throwing behavior deviates from the prior TBoost.STM 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() ensures any operations within the before_retry block are executed if the transaction is aborted.

```
#define atomic(T)
#define try_atomic(T)
#define use_atomic(T)
#define catch_before_retry(E)
#define before_retry
#define end_atom
```

Macro `atomic`

Use 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

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

Macro `try_atomic`

```
#define try_atomic(T) \
    for (transaction T; \
        !T.committed() && T.restart(); \
        T.no_throw_end())
    try
```

Macro `use_atomic`

```
#define use_atomic(T) \
    for (transaction T; \
        !T.committed() && T.restart(); \
        T.end())
```

Macro `catch_before_retry`

```
#define catch_before_retry(E) catch (aborted_tx &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).

To be used pairwise with `try_atomic` or `atomic`.

Macro `before_retry`

```
#define before_retry catch (aborted_tx &)
```

Same as catch_before_retry(E) except the exception is discarded.

To be used pairwise with `try_atomic` or `atomic`.

Macro `end_atom`

```
#define end_atom catch (aborted_tx &) {}
```

Same as before_retry except the exception is discarded and {} are automated.

To be used pairwise with `try_atomic` or `atomic`.

Header <boost/stm/transaction.hpp>

```
namespace boost { namespace stm {
    class transaction;
    struct thread_initializer;
}}
```

Class transaction

TBoost.STM defines transactions as stackbased 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.

```
class transaction {
public:
    // initialization
    static void initialize();
    static void initialize_thread();
    static void terminate_thread();

    // contention manager
    static void contention_manager(base_contention_manager *rhs);
    static base_contention_manager* get_contention_manager();

    static bool enable_dynamic_priority_assignment();
    static bool disable_dynamic_priority_assignment();
    static bool doing_dynamic_priority_assignment();

    // bookeeping
    static void enableLoggingOfAbortAndCommitSetSize();
    static void disableLoggingOfAbortAndCommitSetSize();
    static const transaction_bookkeeping & bookkeeping();

    // conflict detection
    static bool early_conflict_detection();
    static bool late_conflict_detection();
    static bool do_early_conflict_detection();
    static bool do_late_conflict_detection();

    // consistency checking
    static std::string consistency_checking_string();
    static bool validating();
    static bool invalidating();

    // updating
    static bool direct_updating()
    static bool deferred_updating();
    static bool do_direct_updating();
    static bool do_deferred_updating();
    static std::string update_policy_string();

    // Lock Aware Transactional Memory support methods
    static latm_type const latm_protection();
    static std::string const latm_protection_str();
    static void do_full_lock_protection();
    static void do_tm_lock_protection();
    static void do_tx_lock_protection();
    static bool doing_full_lock_protection();
    static bool doing_tm_lock_protection();
    static bool doing_tx_lock_protection();

    //in flight transactions
    static InflightTxes const & in_flight_transactions();

    // transaction
    transaction();
    ~transaction();

    bool committed() const;
    bool aborted() const;
    bool in_flight() const;
```

```
    void begin();
    bool restart();
    bool restart_if_not_inflight();
    void end();
    void no_throw_end();
    void force_to_abort();
    void unforce_to_abort();

    std::size_t const &priority() const;
    void set_priority(std::size_t const &rhs) const;
    void raise_priority();

    void make_irrevocable();
    void make_isolated();
    bool irrevocable() const;
    bool isolated() const;

    thread_id_t const & thread_id() const;


    template <typename T> T& find_original(T& in);

    template <typename T> T const & read(T const & in);
    template <typename T> T const * read_ptr(T const * in);
    template <typename T> T const & r(T const & in);

    template <typename T> T& write(T& in);
    template <typename T> T* write_ptr(T* in);
    template <typename T> T& w(T& in);

    template <typename T> T* get_written(T const & in);

    template <typename T> void delete_memory(T &in)
    template <typename T> T* new_memory();
    template <typename T> T* new_memory_copy(T const &rhs);

};
```

Static Function `initialize()`

```
static void initialize();
```

This method must called before any transaction objects are constructed. The initialize method initializes the overall transaction locking framework.

Static Function `initialize_thread()`

```
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 Function `terminate_thread()`

```
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.

Static Function `contention_manager`

```
static void contention_manager(base_contention_manager *rhs);
```

Sets TBoost.STM's contention manager to point to the passed in contention manager. When a new CM is passed in to TBoost.STM, 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 TBoost.STM then takes ownership of these CMs.

Static Function `get_contention_manager`

```
static base_contention_manager* get_contention_manager();
```

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

Static Function enable_dynamic_priority_assignment

```
static bool enable_dynamic_priority_assignment();
```

Static Function disable_dynamic_priority_assignment

```
static bool disable_dynamic_priority_assignment();
```

Static Function doing_dynamic_priority_assignment

```
static bool doing_dynamic_priority_assignment();
```

Static Function enableLoggingOfAbortAndCommitSetSize

```
static void enableLoggingOfAbortAndCommitSetSize();
```

Static Function disableLoggingOfAbortAndCommitSetSize

```
static void disableLoggingOfAbortAndCommitSetSize();
```

Static Function transaction_bookkeeping

```
static const transaction_bookkeeping & bookkeeping();
```

Static Function early_conflict_detection

```
static bool early_conflict_detection();
```

Returns true if direct updating is active and early conflict detection, otherwise returns false.

Static Function late_conflict_detection

```
static bool late_conflict_detection();
```

Returns true if deferred updating is active or if direct updating and late conflict detection are active, otherwise returns false.

Static Function do_early_conflict_detection

```
static 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.

Static Function 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.

```
static bool do_late_conflict_detection();
```

Static Function consistency_checking_string

```
static std::string consistency_checking_string();
```

Static Function validating

```
static bool validating();
```

Static Function invalidating

```
static bool invalidating();
```

Static Function direct_updating

```
static bool direct_updating()
```

Returns true if direct updating is active, otherwise returns false.

Static Function deferred_updating

```
static bool deferred_updating();
```

Returns true if deferred updating is active, otherwise returns false

Static Function do_direct_updating

```
static bool do_direct_updating();
```

Attempts to switch to direct updating. Returns false if inflight transactions are found and TBoost.STM is unable to switch updating models. Otherwise, returns true and enables direct updating.

Static Function do_deferred_updating

```
static bool do_deferred_updating();
```

Attempts to switch to deferred updating. Returns false if inflight transactions are found and TBoost.STM is unable to switch updating models. Otherwise, returns true and enables deferred updating.

Static Function update_policy_string

```
static std::string update_policy_string();
```

Static Function latm_protection

```
static latm_type const latm_protection();
```

Static Function latm_protection_str

```
static std::string const latm_protection_str();
```

Static Function do_full_lock_protection

```
static void do_full_lock_protection();
```

Static Function do_tm_lock_protection

```
static void do_tm_lock_protection();
```

Static Function do_tx_lock_protection

```
static void do_tx_lock_protection();
```

Static Function doing_tx_lock_protection

```
static bool doing_tx_lock_protection();
```

Static Function doing_full_lock_protection

```
static bool doing_full_lock_protection();
```

Static Function doing_tm_lock_protection

```
static bool doing_tm_lock_protection();
```

Static Function in_flight_transactions

```
static InflightTxes const & in_flight_transactions();
```

Constructor transaction()

```
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.

Destructor `~transaction()`

```
~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.

Member Function `committed`

States if the state of the transaction is e_committed or e_hand_off

```
bool committed() const;
```

Member Function `aborted`

States if the state of the transaction is e_aborted

```
bool aborted() const;
```

Member Function `in_flight`

States if the state of the transaction is e_in_flight

```
bool in_flight() const;
```

Member Function `begin`

```
void begin();
```

Member Function `restart`

```
bool restart();
```

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

Member Function `restart_if_not_inflight`

```
bool restart_if_not_inflight();
```

Member Function `end`

```
void end();
```

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 transaction 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.

Member Function `no_throw_end`

```
void no_throw_end();
```

Do the same as end() by caching every exception.

Member Function `force_to_abort`

```
void force_to_abort();
```

Member Function `unforce_to_abort`

```
void unforce_to_abort();
```

Member Function `priority`

```
std::size_t const &priority() const;
```

Member Function `set_priority`

```
void set_priority(std::size_t const &rhs) const;
```

Member Function `raise_priority`

```
void raise_priority();
```

Member Function `make_irrevocable`

```
void make_irrevocable();
```

Member Function `make_isolated`

```
void make_isolated();
```

Member Function `irrevocable`

```
bool irrevocable() const;
```

Member Function `isolated`

```
bool isolated() const;
```

Member Function `thread_id`

```
thread_id_t const & thread_id() const;
```

Template Member Function `find_original<>`

```
template <typename T> T& find_original(T& in);
```

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 Function `read<>`

```
template <typename T> T const & read(T const & in);
template <typename T> T const & r(T const & in);
```

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 Function `read_ptr<>`

```
template <typename T> T const * read_ptr(T const * in);
```

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

Template Function `write<>`

```
template <typename T> T& write(T& in);
template <typename T> T& w(T& in);
```

The templatized 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 Function `write_ptr<>`

```
template <typename T> T* write_ptr(T* in);
```

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

Template Function `get_written<>`

```
template <typename T> T* get_written(T const & in);
```

Template Function `delete_memory<>`

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

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

Template Function new_memory<>

```
template <typename T> T* new_memory();
```

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 Function new_memory_copy<>

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

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.

Struct thread_initializer

Scoped thread initializer calling transaction::initialize_thread() in the constructor and transaction::terminate_thread() in the destructor.

Declare an object of this class on each thread participating on a transaction.

```
struct thread_initializer {
    thread_initializer();
    ~thread_initializer();
};
```

Header <boost/stm/transaction_object.hpp>

```
namespace boost { namespace stm {
    template <class Derived> class transaction_object;
    template <typename T> class native_trans;
}}
```

Template Class transaction_object<>

To further simplify the usage of TBoost.STM, 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.

With the templatized 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 TBoost.STM's memory manager for all user-defined types derived from it.

```
template <class Derived>
class transaction_object : public base_transaction_object {
public:

    virtual base_transaction_object* clone() const;
    virtual void copy_state(base_transaction_object const * const rhs);
    virtual void move_state(base_transaction_object * rhs);
};
```

Virtual function `clone()`

```
virtual base_transaction_object* clone() const;
```

The clone() method calls to the free function cache_new_copy which allocates enough memory for the new object and then do a memcpy by default.

Virtual function `copy_state()`

```
virtual void copy_state(base_transaction_object const * const rhs);
```

The copy_state() method call to the free function cache_copy which do a memcpy by default.

Virtual function `move_state()`

```
virtual void move_state(base_transaction_object * rhs);
```

The move_state() method call to the free function cache_copy which do a memcpy by default.

Virtual function `cache_deallocate()`

```
virtual void cache_deallocate();
```

The cache_deallocate() method call to the free function cache_deallocate which do a memcpy by default.

Template Class native_trans<>

```
template <typename T> class native_trans : public transaction_object< native_trans<T> > {
public:
   native_trans(){}
   native_trans(T const &rhs);
   native_trans(native_trans const &rhs);
   native_trans(native_trans &&rhs);
   native_trans& operator=(native_trans &&rhs);

   native_trans& operator=(T const &rhs);

   native_trans& operator--();
   native_trans operator--(int);
   native_trans& operator++();
   native_trans operator++(int);
   native_trans& operator+=(T const &rhs);
   native_trans operator+(native_trans const &rhs);

   operator T() const;
   T& value();
   T const & value() const;
};
```

Header <boost/stm/transactional_object.hpp>

```
namespace boost { namespace stm {
    template <class T> class transactional_object;

    template <typename T>
    static transactional_object<T>* tx_up_cast(T* ptr);

    template <typename T, typename U>
    static transactional_object<T>* tx_static_cast(transactional_object<U>* ptr);

    template <typename T, typename U>
    static transactional_object<T>* tx_dynamic_cast(transactional_object<U>* ptr);
}}
```

Template Class transactional_object<>

Transactional object wrapper. A `transactional_object<T>` is a base_transaction_object wrapping an instance of type T. Provides the definition of the virtual functions

- forward constructors to the wrapped type
- `copy_state`: relaying on the cache_copy<T> generic function
- `move_state` and
- `cache_deallocate`: relaying on the cache_copy<T> generic function

Defines in addition the functions `new` and `delete` when `BOOST_STM_USE_MEMORY_MANAGER` is defined

If a class D inherits from B we have that `transactional_object<D>` dont inherits from `transactional_object<B>`, but we can static/dynamic cast them robustly.

Evidently the `std::static_cast`/`std::dynamic_cast` do not works. We need to define the specific cast

```
transactional_object<D>* d=...;
transactional_object<B>* b=tx_static_cast<B>(d);

transactional_object<B>* b=...;
transactional_object<D>* d=tx_dynamic_cast<B>(b);
```

Synopsis

```
template <typename T>
class transactional_object : public base_transaction_object {
public:
    T value;

    transactional_object();
    transactional_object(const T*ptr);
    transactional_object(transactional_object<T> const & r);
    template <typename T1>
    transactional_object(T1 const &p1);

    template <typename T1, typename T2>
    transactional_object(T1 const &p1, T2 const &p2);

    transactional_object & operator=(transactional_object const & r);

    virtual void copy_state(base_transaction_object const * const rhs);

#ifdef BOOST_STM_USE_UNASIGNED_COPY
    virtual void cache_deallocate();
#endif

#if BOST_STM_USE_MEMORY_MANAGER
   void* operator new(size_t size) throw ();
   void operator delete(void* mem);
#endif

};
```

Virtual function `copy_state()`

```
virtual void copy_state(base_transaction_object const * const rhs);
```

Virtual function `move_state()`

```
virtual void move_state(base_transaction_object * rhs);
```

Virtual function `cache_deallocate()`

```
virtual void cache_deallocate()=0;
```

Header `<boost/stm/tx_ptr.hpp>`

```
namespace boost { namespace stm {
    template <typename T> class read_ptr;
    template <typename T> class write_ptr;
}}
```

Template Class `read_ptr<>`

```
template <typename T> class read_ptr {
public:
   inline read_ptr(transaction &t, T const &tx_obj);
   const T* get() const;
   inline T const & operator*() const;
   inline T const * operator->() const;

   inline transaction &trans();
   T* write_ptr();
};
```

Template Class `write_ptr<>`

```
template <typename T> class write_ptr {
public:
   inline write_ptr(transaction &t, T & tx_obj);

   T* get() const;
   inline T& operator*();
   inline T* operator->();
```

Header `<boost/stm/tx_smart_ptr.hpp>`

```
namespace boost { namespace stm {

    template <typename T> class tx_obj;
    template <typename T, typename U>
    bool operator==(const tx_obj<T>& lhs, const tx_obj<U>& rhs);
    template <typename T, typename U>
    bool operator==(const T& lhs, const tx_obj<U>& rhs);
    template <typename T, typename U>
    bool operator==(const tx_obj<T>& lhs, const U& rhs);
    template <typename T, typename U>
    bool operator!=(const tx_obj<T>& lhs, const tx_obj<U>& rhs);
    template <typename T, typename U>
    bool operator!=(const tx_obj<T>& lhs, const tx_obj<U>& rhs);
    template<class T> inline void swap(tx_obj<T> & a, tx_obj<T> & b);

    template <typename T> class tx_ptr;
    template <typename T, typename U>
    bool operator==(const tx_ptr<T>& lhs, const tx_ptr<U>& rhs);
    template <typename T, typename U>
    bool operator!=(const tx_ptr<T>& lhs, const tx_ptr<U>& rhs);
    template<class T> inline void swap(tx_ptr<T> & a, tx_ptr<T> & b);

    template <typename T>
    void delete_ptr(tx_ptr<T> ptr);
    template <typename T>
    void delete_ptr(transaction& tx, tx_ptr<T> ptr);

    template <typename T, typename U>
    static tx_ptr<T> tx_static_cast(tx_ptr<U> ptr);
    template <typename T, typename U>
    static tx_ptr<T>* tx_dynamic_cast(tx_ptr<U> ptr);

    template <typename T> class rd_ptr;
    template <typename T>
    rd_ptr<T> make_rd_ptr(transaction& tx, tx_ptr<T> ptr);
    template <typename T>
    rd_ptr<T> make_rd_ptr(transaction& tx, tx_obj<T> const & ref);

    template <typename T>
    rd_ptr<T> make_rd_ptr(tx_ptr<T> ptr);
    template <typename T>
    rd_ptr<T> make_rd_ptr(tx_obj<T> const & ref);
    template <typename T>
    void delete_ptr(rd_ptr<T> ptr);
    template <typename T>
    void delete_ptr(transaction& tx, rd_ptr<T> ptr);

    template <typename T> class upgrd_ptr;
    template <typename T>
    void delete_ptr(upgrd_ptr<T> const& ptr);
    template <typename T>
    void delete_ptr(transaction& tx, upgrd_ptr<T> const& ptr);

    template <typename T> class wr_ptr;
    template <typename T>
    wr_ptr<T> make_wr_ptr(transaction& tx, tx_ptr<T>& ptr);
    template <typename T>
    wr_ptr<T> make_wr_ptr(tx_ptr<T>& ptr);
```

```
    template <typename T>
    void delete_ptr(wr_ptr<T> ptr);
    template <typename T>
    void delete_ptr(transaction& tx, wr_ptr<T> const& ptr);
}}
```

Template Class `tx_obj<>`

tx_obj is a kind of smart pointer to a wrapped transactional_object T providing builting operators

```
template <typename T> class tx_obj {
public:
    tx_obj();
    template<ctypename Y> tx_obj(tx_obj<Y> const& r);

    template <typename T1> tx_obj(T1 p1);

    T* operator->();
    const T* operator->() const;
    T& operator*();
    const T& operator*() const;
    T* get();
    const T* get() const;
    T& ref();
    const T& ref() const;

    tx_obj& operator--();
    T operator--(int);

    tx_obj& operator++();
    T operator++(int);

    tx_obj& operator+=(T const &rhs);
    T operator+(T const &rhs) const;
};
```

Template Class `tx_ptr<>`

a tx_ptr<T> is an smart pointer to a transactional_object<T> (which contains an instance of T). Reference fields in linked structures should always be tx_ptrs. The result of derreferencing it will be the pointer to the T instance When this pointer is derreference on a transaction the transactional_object<T> is set a written and the transaction specific storage will be used. Otherwise the shared storage is used.

Used to initialize a rd_ptr<T>, wr_ptr<T>, or upgrd_ptr<T>.

```
template <typename T>
class tx_ptr {
public:
    typedef tx_ptr<T> this_type;

    tx_ptr();
    template<class Y> explicit tx_ptr(Y * ptr);
    explicit tx_ptr(transactional_object<T>* ptr);
    tx_ptr(const tx_obj<T>& r);

    template<class Y> tx_ptr(tx_ptr<Y> const& r);
    template<class Y> tx_ptr(rd_ptr<Y> const & r);
    template<class Y> tx_ptr(wr_ptr<Y> const & r);
    template<class Y> tx_ptr(upgrd_ptr<Y> const & r);

    template<class Y> tx_ptr& operator=(transactional_object<Y>* ptr);
    template<class Y> tx_ptr & operator=(tx_ptr<Y> const & r);

    T* operator->() const;
    T& operator*() const;
    T* get() const;

    typedef transactional_object<T>* this_type::*unspecified_bool_type;

    operator unspecified_bool_type() const;
    void swap(tx_ptr & other);
};
```

Template Class `rd_ptr<>`

A rd_ptr<T> ("read pointer") points to an object that the current transaction has opened for read only access. You can only call a const method through a read pointer.

A rd_ptr<T> is constructed from an tx_ptr<T> through an explicit constructor. Once a rd_ptr<T> has been constructed, an tx_ptr<T> can be opened for reading simply by assignment (operator=()) into the constructed rd_ptr<T>.

It is not safe to derreference a rd_ptr<T> after having assigned the same tx_ptr<T> to a wr_ptr<T>. If this is the case the readen value do not match the writen one. If it is possible to write on the same transaction use upgrd_ptr instead which is safe.

```
template <typename T>
class rd_ptr {
public:
    rd_ptr(transaction &t, tx_ptr<T> tx_obj);
    rd_ptr(transaction &t, tx_obj<T> const& tx_obj);

    template<class Y> rd_ptr & operator=(tx_ptr<Y> r);

    template<class Y> rd_ptr& operator=(tx_obj<Y> const & r);

    const T* get() const;

    inline const T & operator*() const;
    inline const T* operator->() const;

    operator unspecified_bool_type() const;

};
```

Template Class upgrd_ptr<>

A upgrd_ptr<T> ("upgradable pointer") points to an object that the current transaction has opened for read only access.

You can only call const method of the wrapped type through a upgradable pointer.

A upgrd_ptr<T> is constructed from an tx_ptr<T> through a constructor having also the transaction as parameter. Once a rd_ptr<T> has been constructed, an tx_ptr<T> can be opened for reading simply by assignment (operator=()) into the constructed rd_ptr<T>.

It is safe to derreference a rd_ptr<T> after having assigned the same tx_ptr<T> to a wr_ptr<T>.

A upgrd_ptr<T> can be upgraded to a wr_ptr<T> through a constructor.

```
template <typename T>
class upgrd_ptr {
public:

    inline upgrd_ptr(transaction &t, tx_ptr<T> tx_obj);
    template<class Y>
    upgrd_ptr & operator=(tx_ptr<Y> const&  r);

    const T* get() const;

    inline T const & operator*() const;
    inline T const * operator->() const;

    operator unspecified_bool_type() const;

    void write_ptr(transactional_object<T>* ptr);
    T* write_ptr();

};
```

Template Class wr_ptr<>

A wr_ptr<T> ("write pointer") points to a shared object that the current transaction has opened for writing.

A wr_ptr<T> is initialized explicitly from an tx_ptr<T>.

A wr_ptr<T> can also be explicitly constructed from a upgrd_ptr<T> as an upgrade-to-writable operation.

```
template <typename T>
class wr_ptr {
public:
    wr_ptr(transaction &t, tx_ptr<T> tx_obj);

    wr_ptr(transaction &t, upgrd_ptr<T> tx_obj);

    T* get();

    inline T& operator*();
    inline T* operator->();

    operator unspecified_bool_type() const;
};
```

Header <boost/stm/non_tx_smart_ptr.hpp>

```
namespace boost { namespace stm { namespace non_tx {
    template <typename T> class rd_ptr;
    template <typename T> rd_ptr<T> make_rd_ptr(transaction& tx, T* ptr);
    template <typename T> rd_ptr<T> make_rd_ptr(transaction& tx, T& ref);
    template <typename T> rd_ptr<T> make_rd_ptr(T* ptr);
    template <typename T> rd_ptr<T> make_rd_ptr(T& ref);
    template <typename T> void delete_ptr(rd_ptr<T> ptr);

    template <typename T> class upgrd_ptr;
    template <typename T> void delete_ptr(upgrd_ptr<T> const& ptr);
    template <typename T> void delete_ptr(transaction& tx, upgrd_ptr<T> const& ptr);

    template <typename T> class wr_ptr;
}}}
```

These smart pointers points to an unspecified cache type which inherits from base_transactional object. This cache is obtained from an internal thread safe cache map.

As the the non_tx smart pointers need to lookup on a cache they must be used only when you need to access to a non transactional variable on the context of a transaction. Otherwise, please use tx_ptr<> or the mixin transaction_object<>.

Template Class rd_ptr<>

A rd_ptr<T> ("read pointer") points to an cache that the current transaction has opened for read only access. You can only call a const method through a read pointer.

A rd_ptr<T> is constructed from an T pointer or reference. Once a rd_ptr<T> has been constructed, the associated cache is opened for reading.

It is not safe to derreference a rd_ptr<T> after having assigned the same T to a wr_ptr<T>. If this is the case the readen value do not match the writen one. If it is possible to write on the same transaction use upgrd_ptr instead which is safe.

```
template <typename T>
class rd_ptr {
public:
    rd_ptr(transaction &t, T const * ptr);

    rd_ptr(transaction &t, T const & obj);

    template<class Y>
    explicit rd_ptr & operator=(Y const* ptr);

    template<class Y>
    explicit rd_ptr & operator=(Y const& ref);

    const T* get() const;

    const T & operator*() const;
    const T* operator->() const;

    operator unspecified_bool_type() const;
};
```

Template Class `upgrd_ptr<>`

A upgrd_ptr<T> ("upgradable pointer") points to an cache that the current transaction has opened for read only access. You can only call const method of the wrapped type through a upgradable pointer.

A upgrd_ptr<T> is constructed from an T pointer or reference through a constructor having also the transaction as parameter. Once a upgrd_ptr<T> has been constructed, an a cache of T can is opened for reading.

It is safe to derreference a upgrd_ptr<T> after having assigned the same T to a wr_ptr<T>.

A upgrd_ptr<T> can be upgraded to a wr_ptr<T> through a constructor.

```
template <typename T>
class upgrd_ptr {
public:
    upgrd_ptr(transaction &t, T* ptr);
    upgrd_ptr(transaction &t, T& ref);

    template<class Y> upgrd_ptr & operator=(Y* ptr);

    template<class Y> upgrd_ptr & operator=(Y& ref);

    const T* get() const;
    inline T const & operator*() const;
    inline T const * operator->() const;

    operator unspecified_bool_type() const;

    void write_ptr(detail::cache<T>* ptr);
    T* write_ptr();
};
```

Template Class `wr_ptr<>`

A wr_ptr<T> ("write pointer") points to a cache that the current transaction has opened for writing.

A wr_ptr<T> is initialized explicitly from an T pointer or reference.

A wr_ptr<T> can also be explicitly constructed from a upgrd_ptr<T> as an upgrade-to-writable operation.

```
template <typename T>
class wr_ptr {
public:
    wr_ptr(transaction &t, T* ptr);
    wr_ptr(transaction &t, T& obj);

    T* get();
    inline T& operator*();
    inline T* operator->();

    operator unspecified_bool_type() const;
};
```

Header <boost/stm/transaction_bookkeeping.hpp>

```
namespace boost { namespace stm {
    class transaction_bookkeeping
}}
```

Class transaction_bookkeeping

```
class transaction_bookkeeping
{
public:

   typedef std::map<uint32, uint32> thread_commit_map;
   typedef std::map<ThreadIdAndCommitId, uint32> CommitHistory;
   typedef std::map<ThreadIdAndCommitId, uint32> AbortHistory;

   transaction_bookkeeping();

   uint32 const & lockConvoyMs();
   uint32 const & commitTimeMs();
   uint32 const & readAborts();
   uint32 const & writeAborts();
   uint32 const & abortPermDenied();
   uint32 const totalAborts() const;
   uint32 const & commits() const;
   uint32 const & handOffs() const;
   uint32 const & newMemoryAborts();
   uint32 const & newMemoryCommits();
   uint32 const & deletedMemoryAborts();
   uint32 const & deletedMemoryCommits();
   uint32 const & readChangedToWrite() const;
   uint32 const & readStayedAsRead() const;

   void inc_read_aborts();
   void inc_write_aborts();

   void inc_thread_commits(uint32 threadId);
   void inc_thread_aborts(uint32 threadId);

   thread_commit_map const & threadedCommits() const;
   thread_commit_map const & threadedAborts() const;

   void inc_lock_convoy_ms(uint32 const &rhs);
   void inc_commit_time_ms(uint32 const &rhs);
   void inc_commits();
   void inc_abort_perm_denied(uint32 const &threadId);
   void inc_handoffs();
   void inc_new_mem_aborts_by(uint32 const &rhs);
   void inc_new_mem_commits_by(uint32 const &rhs);
   void inc_del_mem_aborts_by(uint32 const &rhs);
   void inc_del_mem_commits_by(uint32 const &rhs);
   void incrementReadChangedToWrite();
   void incrementReadStayedAsRead();

   CommitHistory const& getCommitReadSetList() const;
   CommitHistory const& getCommitWriteSetList() const;
   AbortHistory const& getAbortReadSetList() const;
   AbortHistory const& getAbortWriteSetList() const;

   void pushBackSizeOfReadSetWhenAborting(uint32 const &size);

   void pushBackSizeOfWriteSetWhenAborting(uint32 const &size);

   void pushBackSizeOfReadSetWhenCommitting(uint32 const &size);
   void pushBackSizeOfWriteSetWhenCommitting(uint32 const &size);
```

```
   bool isLoggingAbortAndCommitSize() const;
   void setIsLoggingAbortAndCommitSize(bool const &in);

   friend std::ostream& operator<<(std::ostream& out, transaction_bookkeeping const &that);
};
```

Examples

This section includes complete examples using the library.

[/

Appendices

Appendix A: History

Version 0.1, XX YY, 2009 *Announcement of STM*

Features:

•

Toolsets:

- Tested with static library.
- Tested on cygwin gcc 3.4.6.

Tickets:

v0.1#1: .

Appendix B: Rationale

TM-Specific Concepts

Optimistic concurrency

ACI transactions

Transactional memory was founded on the database ACID principle (atomic, consistent, isolated and durable), except without the D (durability) because unlike database transactions, TM transactions are not saved to permanent storage (e.g., hard drives).

- 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 TBoost.STM's transactional framework and demonstrates TBoost.STM's ACI conformance.

```
native_trans<int> global_int;
int increment_global() {
    atomic(t) {
        t.write(global_int)++;
        val = t.read(global_int);
    } end_atom
    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 TBoost.STM's conformance to the ACI principles. The above example also gives a basic introduction into TBoost.STM's transactional framework.

STM Synchronization Types

There are two ways STM systems synchronize memory:

1. using non-blocking mechanisms (lock-free) 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.

TBoost.STM is a lock-based STM system. At its core, TBoost.STM 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. TBoost.STM's lockbased strategy allows it to gain the performance benefits of a nonblocking system, such that when transactions are not committing, the transactions do not block each other and are guaranteed to make forward progress. Yet TBoost.STM maintains the benefits of a lockbased system, enabling it to perform commit-time invalidation, its primary consistency model mechanism.

Recent research shows lock-based STM systems outperform non-blocking systems. Our own research shows that through TBoost.STM'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.

Updating policies

In any STM 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 TBoost.STM'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).

Table 1. Comparaison with other STM systems

Features	after write	after read
write	**YES**	**YES**
read	**YES**	**NO**

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 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, TBoost.STM implements a run-time configurable early and late conflict detection mechanism. For deferred updating, TBoost.STM only implements late conflict detection. The decision to have TBoost.STM 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 policies

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 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.

TBoost.STM currently implements consistency checking only through invalidation. One of the next goals of TBoost.STM 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, TBoost.STM is unique in that it is the first STM system to implement commit-time invalidation. While other systems, such as RSTM, have implemented invalidation, no other system implements commit-time invalidation.

We believe TBoost.STM 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).

- 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 TBoost.STM. 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.

- 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 inverted 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.

Consistency versus Updating policies composition

While TBoost.STM 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 TBoost.STM 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 direct updating, deferred updating, validation and invalidation, all of which should be configurable at run-time and compile-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.

Table 2. Consistency versus Updating policies composition

Features	Direct	Deferred
Validation	**YES**	**YES**
Invalidation	**Not Yet Implemented**	**YES**

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. Type memory granularity allows transactions to read and write at the type level, usually controlled by implementation of a transactional object cache. 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. TBoost.STM 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 rollbacking 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;

- updates to global,
- allocated memory and

- deallocated memory.

TBoost.STM 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.

Composable transactions

Composition is the process of taking separate transactions and adding them together to compose a larger single transaction. 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.

TBoost.STM implements composition via subsumption and is a closed nested system. 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. TBoost.STM's closed 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.

Future versions of TBoost.STM will implement closed nested transactions.

Contention management

Priority-Based Tasks

Real-time systems or systems that have strict requirements for task behavior, such as deadline-drive systems, usually guarantee such behavior through priority scheduling. While significant prior TM contention management (CM) research has been done, its attention has been primarily focused on preventing starvation through fairness. In many systems preventing starvation may be sufficient, yet some systems (e.g. deadline-driven or real-time systems) require stronger behavioral guarantees. In these cases, user-defined priority-based transactions are necessary.

Approach

This work extends the prior TM contention management research by concretely implementing user-defined contention managers as determined consequential. We approach this by first presenting a brief background of TM aspects important to understanding the complexities of contention management. Next, we review and expand upon prior contention management work. We then show how consistency models play a significant role in the correctness and capability of priority-based transactional scheduling. We then build user-defined priority-based transactions with TBoost.STM, demonstrating how contention management frameworks work with different consistency checking models. Last, we present our experimental results.

Attacking & Victim Transactions

We refer to transactions which can identify a memory conflict in another in-flight transaction as attacking transactions. Transactions which are targeted by attacking transactions are referred to as victim transactions. The attacking and victim transaction terms help simplify the invalidation process. An example of attacking and victim transactions is as follows. Consider transaction Tv, a victim transaction and transaction Ta, an attacking transaction. Tv writes to memory location L0. Ta then tries to write to L0. If our STM system only allows single-writers, both Ta and Tv cannot write to L0 at the same time. As Ta is the second transaction attempting to write to L0, the single-writer semantics require it to deal with the conflict located at L0. Handling this conflict makes Ta the attacking transaction as it decides if Tv is aborted. Tv is the victim transaction since Ta may abort it.

User-Defined Priority-Based Transactions

Validation and invalidation

consistency checking are both practical solutions to transactional memory for different classes of problems. When inflight transactions are low and memory usage is high, invalidation may be preferred. When transactional traffic is high and transactions are small, validation may be preferred. Neither type of consistency checking seems to perform universally better than the other. Though certain classes of problems perform better under different consistency checking models, the ramifications of these consistency checking schemes performing within strict user-defined priority-based transactional systems is unclear from prior research. Our work clarifies the advantages and disadvantages of validation and invalidation in user-defined priority-based transactional environments.

Extensible Polymorphic Contention Management Interface

Due to the innumerable possibilities of consistency checking model combinations, for the remainder of this work, we narrow our scope of consistency checking to commit-time validation and committime invalidation. We do this for a number of reasons. First, validation and invalidation are near opposites and commit-time consistency checking is straight forward in that it is only performed once per transaction. Due to this, we can see the two primary views of consistency checking in an understandable fashion. Second, early TM systems tended toward validation while newer TM systems tend toward invalidation. This makes considering both viewpoints practical. Third, both validation and invalidation perform well under different circumstances. By considering both types, we can help extend their current understanding to user-defined priority-based systems as well. Finally, TBoost.STM supports both commit-time validation and commit-time invalidation.

TBoost.STM extensible contention manager supports two types of management: abort behavior management and conflict resolution management. The abort behavior interfaces are somewhat novel compared to prior work, in that they allow TBoost.STM's contention manager to decide how the system should behave when a transaction must abort. These interfaces, shown in bellow, were added as it was determined that not all aborted transactions should behave in the same manner. For example, some transactions which have extremely large read and write sets, should perhaps sleep for some period of time before running again. Other transactions which have already high priorities, should perhaps receive larger priority boosts.

```
class base_contention_manager {
public:
    virtual void abort_on_new(transaction const &t) = 0;
    virtual void abort_on_delete(transaction const &t, base_transaction_object const &in) = 0;
    virtual void abort_on_read(transaction const &t, base_transaction_object const &in) = 0;
    virtual void abort_on_write(transaction &t, base_transaction_object const &in) = 0;
    // conflict interfaces removed
};
```

The interfaces shown above do not manage any memory contention, instead they only handle how required aborts behave.

The second type of TBoost.STM contention management interfaces is shown in below.

```
class base_contention_manager
{
public:
    // abort interfaces removed
    virtual bool abort_before_commit(transaction const &t) = 0;
    virtual bool permission_to_abort(transaction const &lhs, transaction const &rhs) = 0;
};
```

These interfaces implement true management of memory conflicts. The two interfaces shown: abort_before_commit() and permission_to_abort() enable two fundamentally different types of consistency checking. abort_before_commit() is called when TBoost.STM is performing validating consistency checking, while permission_to_abort() is called when TBoost.STM is performing invalidating consistency checking. Both APIs are called internally at different points in the transaction's commit phase.

Lock-aware transaction

Transactional memory (TM) has garnered significant interest as an alternative to existing concurrency methods due to its simplified programming model, natural composition characteristics and native support of optimistic concurrency. Yet, mutual exclusion locks are ubiquitous in existing parallel software due to their fast execution, inherent property for irreversible operations and long-standing underlying support in modern instruction set architectures (ISAs). For transactions to become practical, lock-based and TM-based concurrency methods must be unified into a single model.

TBoost.STM presents a lock-aware transactional memory (LATM) solution with a deliberate library-based approach. Our LATM system supports locks inside of transactions (LiT) and locks outside of transactions (LoT) while preserving the original programmer-intended locking structure. Our solution provides three policies with varying degrees of performance and required programming. The most basic LATM policy enables transactionlock correctness without any programming overhead, while improved performance can be achieved by programmer-specified conflicts between locks and transactions. The differences in LATM policy performance are presented through a number of experimental benchmarks on .

Introduction

Yet, for all of the benefits of transactional programming, its native inoperability with mutual exclusion locks is a major obstacle to its adoption as a practical solution. Mutual exclusion (implemented through locks) is arguably the most predominant form of thread synchronization used in parallel software, yet its native incompatibility with transactions places a considerable restriction on the practical use of TM in real-world software.

A TM system that is cooperative with locks can extend the lifetime and improve the behavior of previously generated parallel programs. TBoost.STM C++ library is a lock-aware transactional memory (LATM) system supporting the simultaneous execution of transactions and locks. Of critical importance is that the extended TBoost.STM LATM system naturally supports lock-based composition for locks placed inside of transactions (LiT), a characteristic previously unavailable in locks. The LATM policies present in TBoost.STM are implemented with software library limitations as a central concern. While novel operating system (OS)-level and language-based transaction-lock cooperative models have been previously found, these implementations use constructs not available in library-based STM systems. While useful, the previously identified OS-level and language-based solutions do not address the critical need for a transaction-lock unified model expressed entirely within the limitations of a software library. In languages that are unlikely to be extended, such as C++, and development environments bound to specific constraints, such as a particular compiler or OS, library based solutions are paramount as they present practical solutions within industry-based constraints. Our approach presents a novel library-based LATM solution aimed at addressing these concerns. TBoost.STM makes the following contributions:

1. Extension of the TBoost.STM library for support of locks outside of transactions (LoT) and locks inside of transactions (LiT).

2. Proof that an LATM LiT system naturally enables lock composition. Analytical examples are examined that show LiT composable locks demonstrating atomicity, isolation and consistency as well as deadlock avoidance.

3. Introduction of three novel LATM policies: full lock protection, TM-lock protection, and TX-lock protection. Each LATM policy provides different programming / performance trade offs. Experimental results are provided which highlight the performance and programming differences of the three LATM policies.

Background

Transaction-lock interaction does not natively exhibit shared memory consistency and correctness due to differences in underlying critical section semantics. Strongly and weakly isolated transactional memory systems are susceptible to incorrect execution when used with pessimistic critical sections.

Mutual exclusion locks use pessimistic critical section semantics; execution of a lock-controlled critical section is limited to one thread and guarantees the executing thread has mutually exclusive (isolated) access to the critical section code region. Transactions use optimistic critical section semantics; transaction controlled critical sections support unlimited concurrent thread execution. Shared memory conflicts arising from simultaneous transaction execution are handled by the TM system during the commit phase of the transaction. Optimistic critical sections and pessimistic critical sections are natively inoperable due to their contradictive semantics as demonstrated in the below example:

Lock and transaction violation (code).

```
1   native_trans<int> x;
2
3   int lock_dec() {
4       lock(L1);
5       int val = --x;
6       unlock(L1);
7       return val;
8       }
9
10  void tx_inc() {
11      for (transaction t;;t.restart())
12      try {
13          ++t.write(x);
14          t.end(); break;
15      } catch (aborted_tx &) {}
16  }
```

Without an intervening system, thread T1 executing lock_dec() and thread T2 executing tx_inc() are not guaranteed to operate consistently, as they would if each function were run in a lock-only or transaction-only system, respectively. The following example demonstrates how the correctness of the above code is violated from a deferred update and direct update standpoint.

Deferred Update Lock/Transaction Violation. A deferred update TM system stores transactional writes off to the side, updating global memory once the transaction commits. A deferred update system can exhibit an inconsistency in the above code in the following way. Given: x = 0, thread T1 executes lock_dec() and thread T2 executes tx_inc(). T2 executes line 10 - the first half of line 13 (storing a transactional value for x = 0, but not performing the increment). T1 executes lines 3-5 (global x = -1). T2 executes the remainder of line 13 and line 14 (++x on its stored reference of 0, setting its local x = 1) and then commits. T1 executes lines 6-7. The resulting global state of x = 1 is incorrect; x = 0 is correct as one thread has incremented it and one thread has decremented it.

Direct Update Lock/Transaction Violation. A direct update TM system stores transactional writes directly in global memory, storing an original backup copy off to the side in the event the transaction must be unwound. In a direct update system, threads T1 and T2 can exhibit inconsistencies using the code shown above in a variety of ways; T1 executes line 3 - the first half of line 5 (decrementing x, but not setting val). T2 executes lines 10-13 (incrementing the global x and creating a restore point of x = -1). T1 executes the remainder of line 5 (val = 0). T2 executes line 14, but is required to abort, restoring x to -1. T1 completes and returns val = 0 which is incorrect. T2 never committed its x = 0 and has not successfully committed, therefore, x has never correctly been set to 0.

Overcoming Transaction-Lock Inoperability

In order for transactions and locks to cooperate, transactions must adhere to a single mutual exclusion rule: Mutual exclusion semantics require that instructions within a mutex guarded critical section be limited to <=1 simultaneous thread of execution.

Our LATM implementation adheres to this rule and is discussed in two high-level views:

1. locks outside of transactions (LoT) and

2. locks inside of transactions (LiT).

TBoost.STM supports lock-aware transactions by supplying a pass-through interface that is used in place of prior locking calls.

The TBoost.STM locking API is used to perform the additional transaction-lock communication before or after the pthreads interface is called. In all cases, the TBoost.STM pass-through interface results in at least one corresponding call to the appropriated interface to lock, try to lock or unlock the mutual exclusion lock. Further details are provided in later sections.

Locks Outside of Transactions (LoT)

Locks outside of transactions (LoT) are scenarios where a pessimistic critical section of a lock is executed in a thread T1 while an optimistic critical section of a transaction is executed in a thread T2, simultaneously. Thread T1's lock-based pessimistic critical section is entirely outside of thread T2's transaction, thus the term locks outside of transactions or LoT. Figure 5 sets up a running LoT example, used throughout this section, by constructing six functions which are simultaneously executed by six threads. Three of the functions in the below example, tx1(), tx2() and tx3(), are transaction-based, while the other three functions, lock1(), lock2() and lock3(), are lock-based. The functions tx1(), tx2() and lock3() do not have any memory conflict with any other transaction-based or lock-based function and should therefore be able to run concurrently with any of the other functions. However, certain LoT policies inhibit the execution of these nonconflicting functions; details of such inhibited behavior is explained in the following subsections. The below example is used throughout this section to illustrate the differences in the LoT policies.

```
1   native_trans<int> arr1[99], arr2[99];
2
3   void tx1() { /* no conflict */ }
4   void tx2() { /* no conflict */ }
5   void tx3() {
6       for (transaction t;;t.restart())
7       try {
8           for (int i = 0; i < 99; ++i)
9           {
10              ++t.w(arr1[i]).value();
11              ++t.w(arr2[i]).value();
12          }
13          t.end(); break;
14      } catch (aborted_tx&) {}
15  }
16
17  int lock1() {
18      transaction::lock(L1); int sum = 0;
19      for (int i = 0; i < 99; ++i) sum += arr1[i];
20      transaction::unlock(L1); return sum;
21  }
22  int lock2() {
23      transaction::lock(L2); int sum = 0;
24      for (int i = 0; i < 99; ++i) sum += arr2[i];
25      transaction::unlock(L2); return sum;
26  }
27  int lock3() { /* no conflict */ }
```

LoT Full Lock Protection

The most basic implementation of transaction-lock cooperation, what we call full lock protection, is to enforce all transactions to commit or abort before a lock's critical section is executed. All locks outside of transactions are protected from transactions violating their critical section execution by disallowing transactions to run in conjunction with locks. Transactions are stalled until all LoT critical sections are complete and their corresponding locks are released.

An example of full lock protection is shown in Figure 6 using the previously described six threaded model. Full lock protection has no programmer requirements; no new code is required, aside from alteration of existing locking code to use the LATM passthrough interfaces. Additionally, an understanding of how the locks behave within the system to enable transaction-lock cooperation is also not needed. However, full lock protection suffers some performance penalties. As seen in Figure 6, T1 ?? T3 are blocked for the entire duration of the critical sections obtained in T4 ?? T6, since full lock protection prevents any transactions from running while LoTs are obtained. Although T3 does conflict with T4 ?? T5, T6's critical section does not interfere with any of the transactions and should therefore not prevent any transactions from running concurrently; TM-lock protection, the next level of lock protection, is able to avoid such unnecessary stalling.

LoT TM-Lock Protection

TM-lock protection is slightly more complex than full lock protection, yet it can yield better overall system performance. TMlock protection works in the following way: locks which can conflict with transactions are identified by the programmer at startup. Once a conflicting LoT is acquired, all in-flight transactions are either committed or aborted. Transactions are then blocked until the conflicting lock-based critical sections are completed and released. Locks that do not conflict with transactions do not cause any transactions to stall.

The identification of locks that can conflict with transactions requires the programmer to (1) write new code and (2) have a basic understanding of the software system. Due to this, the requirements of TM-lock protection are greater on the end programmer. The trade-off for higher programmer requirements is greater overall system performance.

TM-lock protection addresses the problem of transactions unnecessarily stalled when T6 is executing. When using TM-lock protection, the end programmer must explicitly express which locks can conflict with the TM system. In this example, locks L1 and L2 from threads T4 and T5 conflict with tx3 in thread T3. The end programmer would explicitly label these locks as conflicting in the following way:

```
1 transaction::do_tm_conflicting_lock_protection();
2 transaction::add_tm_conflicting_lock(L1);
3 transaction::add_tm_conflicting_lock(L2);
```

As shown in the six threaded example, TM-lock protection shortens the overall TM run-time by allowing T1 ?? T3 to restart their transactions as soon as L2's critical section is completed. Yet, there still exists unnecessary stalls in threads T1 and T2 as their associated transactions do not conflict with any of the lock-based critical sections of T4 ??T6. The remaining unnecessary stalls are resolved by using TX-lock protection, the third lock protection policy.

LoT TX-Lock Protection

TX-lock protection enables maximum performance throughput by identifying only true conflicts as they exist per transaction. TXlock protection is similar to TM-lock protection except rather than requiring conflicting locks be identified at a general TM-level, conflicting locks are identified at a transaction-level. While this level of protection yields the highest level of performance, it also requires the greatest level of familiarity of the locks within the system and the most hand-crafted code.

An example of TX-lock protection is in Figure 8. By using TX-lock protection and explicitly identifying conflicting locks per transaction, the system only stalls for true conflicts, increasing overall system performance. The code required for correct TX-lock protection in the prior six threaded example is shown below by extending the original tx3() implementation:

```
1   void tx3() {
2       for (transaction t;;t.restart())
3       try {
4           // identify conflicting locks
5           t.add_tx_conflicting_lock(L1);
6           t.add_tx_conflicting_lock(L2);
7           for (int i = 0; i < 99; ++i)
8           {
9               ++t.w(arr1[i]).value();
10              ++t.w(arr2[i]).value();
11          }
12          t.end(); break;
13      } catch (aborted_tx&) {}
14  }
```

Using TX-lock protection, threads T1 and T2 are no longer stalled when threads T4 and T5 lock their associated locks, L1 and L2. In fact, only thread T3 (the only true conflict) is stalled while the critical sections created by L1 and L2 are executing, resulting in the highest transaction-lock cooperative performance while still adhering to the rule.

Locks Inside of Transactions (LiT)

Locks inside of transactions (LiT) are scenarios where a lock-based pessimistic critical section is executed partially or completely inside a transaction. Only two of the three possible LiT scenarios are supported by our work: (1) pessimistic critical sections are encapsulated entirely within a transaction or (2) pessimistic critical sections start inside a transaction but end after the transaction has terminated.We do not support LiT scenarios where pessimistic critical sections start before a transaction begins, having the front-end of the transaction encapsulated by the pessimistic critical section. The reason to disallow such behavior is to avoid deadlocks.

Consider the following scenario. Thread T1 has an in-flight irrevocable transaction Tx1 and thread T2, after obtaining lock L2, starts a transaction Tx2. Tx2 is not allowed to make forward progress until it is made irrevocable (details to follow). Tx1 is already in-flight and irrevocable. Since two irrevocable transactions cannot run simultaneously as they are not guaranteed to be devoid of conflicts with one another, Tx2 must stall until Tx1 completes. If irrevocable transaction Tx1 requires lock L2 to complete its work the system will deadlock. Tx1 cannot make forward progress due to its dependency upon L2 (currently held by Tx2) and Tx2 cannot make forward progress as it requires Tx1 to complete before it can run. As such, LiT scenarios where locks encapsulate the front-end of a transaction are disallowed; our implementation immediately throws an exception when this behavior is detected.

Irrevocable and Isolated Transactions

The LiT algorithms use the same policies as the LoT algorithms: full lock protection, TM-lock protection and TX-lock protection. Locks inside of transactions have the same characteristics as normal mutual exclusion locks, and Lemma 1 must be followed in order to ensure correctness. Since the LiT algorithms use locks acquired inside of transactions and these locks are not guaranteed to have failure atomicity as transactions do, the containing transactions must become irrevocable (see Lemma 2). Irrevocable transactions, characterized by T(irrevocable) = true, are transactions that cannot be aborted. The concept of irrevocable (or inevitable) transactions is not new; Welc et al. and Spear et al. have shown these types of transactions to be of significant importance as well as having a variety of practical uses [17, 19]. We extend the prior work of Welc et al. and Spear et al. by using irrevocable transactions to enable pessimistic critical sections within transactions, as well as to create composable locks within transactions. In addition to irrevocable transactions, the LiT full lock protection and TM-lock protection require a new type of transaction, one that we term as an isolated transaction. Isolated transactions, characterized by T(isolated) = true, are transactions that cannot be aborted and require that no other type of transaction run along side it simultaneously. Isolated transactions can be viewed as a superset of irrevocable transactions; isolated transactions have the properties of irrevocable transactions and must be run in isolation. To demonstrate how the LiT algorithms work, consider the six threaded example shown in Figure 12.

3 LiT Transaction Threads, 3 Locking Threads.

```
native_trans<int> g1, g2, g3, g4;

void tx1() { /* no conflict */ }
void tx2() {
    transaction::add_tm_conflicting_lock(L2);
    for (transaction t;;t.restart())
    try {
        t.add_tx_conflicting_lock(L2);
        inc2();
        t.end(); break;
    } catch (aborted_tx&) {}
}
void tx3() {
    transaction::add_tm_conflicting_lock(L3);
    for (transaction t;;t.restart())
    try {
        t.add_tx_conflicting_lock(L3);
        inc3();
        t.end(); break;
    } catch (aborted_tx&) {}
}

void inc2() {
    lock(L2); ++g2.value(); unlock(L2);
}
void inc3() {
    lock(L3); ++g3.value(); unlock(L3);
}
void inc4() { /* no conflict */ }
```

In Figure 12 thread T1 executes tx1(), T2 executes tx2(), T3 executes tx3(), T4 executes inc2(), T5 executes inc3() and T6 executes inc4(). Threads T1 (tx1()) and T6 (inc4()) do not conflict with any other thread, yet their execution can be inhibited by other threads based on the LiT policy employed. Thread T2 has a true conflict with thread T4 (both threads call inc2()) and thread T3 has a true conflict with thread T5 (both threads call inc3()). A staggered start time is used in the coming diagrams: T1 starts, followed by T2, T3, T4, T5 and finally T6.We label the LiT threads based on the locks they acquire: tx1 acquires lock L1, thread tx2 acquires lock L2 and thread tx3 acquires lock L3. The same taxonomy is used for locking threads: thread lockL2 acquires lock L2, thread lockL3 acquires lock L3 and thread lockL4 acquires lock L4.

In Figure 12, both add_tm_conflicting_lock() and add_tx_conflicting_lock() are present. If the system is using TM-lock protection only the add_tm_conflicting_lock() is necessary, whereas if the system is using TX-lock protection only the add_tx_conflicting_lock() is necessary. If neither TMlock or TX-lock protection is in use, neither call is needed. These interfaces are supplied for the completeness of the example.

LiT Full-Lock Protection

Figure 13 demonstrates how the six threads interact using the full lock protection policy, as well as showing policy conflicts in comparison to true conflicts. The LiT full lock protection algorithm requires that any transaction that has a lock inside of it be run as an isolated transaction. Prior to a lock inside the transaction being obtained, all other in-flight transactions are aborted or committed and all currently held locks must execute through release. Future attempts to obtain locks outside of the isolated transaction are prevented until the transaction commits. This behavior is required as the system must assume (1) all external locks can conflict with the isolated transaction, so no external locks can be obtained; and (2) all external transactions can conflict with the LiT transaction, and therefore no external transactions can execute.

For Figure 13, once tx2 begins, tx1 is stalled as tx2 must run as an isolated transaction. Due to tx2's isolation, tx3 is also stalled. Both lockL2 and lockL3 are also stalled because full lock protection disallows transactions from running while LoT locks are obtained; as tx2 is an isolated transaction, the threads attempting to lock L2 and L3 are stalled until tx2 completes. When tx2 completes, tx3 is started as

it has stalled for the longest amount of time. The thread executing lockL4 is stalled until tx3 completes. When tx3 completes, tx1, lockL2, lockL3 and lockL4 are all allowed to resume.

LiT TM-Lock Protection

Like full lock protection, LiT TM-lock protection runs transactions encapsulating locks in isolation. However, LiT TM-lock protection also requires the end programmer to identify locks obtained within any transaction prior to transaction execution (just as LoT TM-lock protection). Unlike LiT full lock protection, TM-lock protection allows non-conflicting LoT locks to execute along side LiT locks, increasing overall system throughput.

As shown in Figure 14 TM-lock protection reduces the overall policy-induced conflicting time to a range closer to the true conflicting time. Since tx2 and tx3 are true conflicts with lockL2 and lockL3, lockL2 and lockL3 must stall while tx2 and tx3 are executing. However, lockL4 does not conflict with either tx2 or tx3 and as such, should not be stalled while the LiT transactions are inflight. TM-lock protection correctly identifies this conflict as false, allowing lockL4's execution to be unimpeded by tx2 and tx3's execution.

Three problems still exist in the LiT example: (1) tx1 is stalled by tx2 and tx3, (2) lockL2 is stalled by tx3 and (3) lockL3 is stalled by tx2. Ideally, none of these stalls should occur as none represent true conflicts. All three false conflicts are avoided by using the following LiT protection policy, LiT TX-lock protection.

LiT TX-Lock Protection

Like LoT TX-lock protection, the Lit TX-lock protection algorithm allows for the highest throughput of both transactions and locks while requiring the highest level of programmer involvement and system understanding. Unlike both prior LiT algorithms, TX-lock protection allows LiT transactions to run as irrevocable transactions, rather than isolated transactions. This optimization can increase overall system throughput substantially if other revocable transactions can be run along side the LiT transaction.

With LiT TX-lock protection, the programmer specifies locking conflicts for each transaction. In Figure 12's case, the programmer would specify that tx2 conflicts with L2 and tx3 conflicts with L3. By specifying true transactional conflicts with locks, the TM system can relax the requirement of running LiT transactions in isolation and instead run them as irrevocable. While no two irrevocable transactions can be run simultaneously, as they may conflict with each other (resulting in a violation of their irrevocable characteristic), other non-irrevocable transactions can be run along side them, improving overall system throughput.

The run-time result of using LiT TX-lock protection is shown in Figure 15. Transaction tx1 is able to run without being stalled as it has no conflicts with other transactions or locks. Transaction tx2 is run as an irrevocable transaction, rather than as an isolated transaction, allowing tx1 to run along side it. Irrevocable transaction tx3 is prevented from starting as irrevocable transaction tx2 is already in-flight. Likewise, lockL2 cannot lock L2 since tx2 conflicts with L2 and allowing lockL2 to proceed would require tx2 to abort. Since tx2 is irrevocable (e.g. unabortable), lockL2 is stalled. However, lockL3 and lockL4 start immediately, since neither conflict with any in-flight transaction. When tx2 completes, both tx3 and lockL2 can try to proceed. Transaction tx3 is stalled by lockL3, but lockL2 executes immediately as its conflict with tx2 has passed. When lockL3 completes tx3 begins and runs through to completion.

Lock Composition

Any TM system that supports locks inside of transactions must ensure the pessimistic critical sections of the locks inside of transactions are not violated. This is achieved by making the containing transactions either isolated or irrevocable once the lock inside the transaction is obtained. Lemma 2 proves the necessity of the irrevocability characteristic for LiT transactions.

Locks Inside of Transactions Lemma 2. Any lock L obtained during an in-flight transaction Tif requires Tif be immediately and permanently promoted to an irrevocable transaction, characterized by Tif (irrevocable) = true, which cannot be aborted.

Proof. (Contradiction) Given: threads T1 and T2 execute inc() and get() from Figure 19, respectively and variables x = 0, y = 0. ++x and ++y operations within inc() are unguarded direct access variable operations that perform no transactional undo or redo logging operation; these operations are irreversible (e.g. normal pessimistic critical section operations). The atomic property of any transaction T requires all memory operations of T are committed or none are committed.

Execution: T1 starts transaction Tx1 (Tx1(irrevocable) = false) and completely executes lines 3-7, setting x = 1. T2 executes 13- 14, obtains lock L1, released by Tx1 and flags Tx1 to abort due to its identified lock conflict. T2 reads x = 1 and y = 0, unlocks L1 and returns.

Tx1 tries to lock L1, but instead is required to abort, causing the atomic transactional property of Tx1 to be violated since ++x has been performed and will not be undone when Tx1 is aborted.

```
1   int x = 0, y = 0;
2
3   void inc() {
4       for (transaction t;;t.restart())
5       try {
6           t.add_conflicting_lock(L1);
7           lock(L1); ++x; unlock(L1);
8           lock(L1); ++y; unlock(L1);
9           t.end(); break;
10      } catch (aborted_tx&) {}
11  }
12
13  void get(int &retX, int &retY) {
14      lock(L1);
15      retX = x; retY = y;
16      unlock(L1);
17  }
```

Figure 19. Violating LiT Lemma 2.

Referring again to Figure 19, now consider the correct execution (following Lemma 2) of threads T1 and T2 of inc() and get(), respectively, with variables x = 0, y = 0. T1 starts transaction Tx1 and executes lines 3-7, setting x = 1 and Tx1(irrevocable) = true. T2 executes 13-14, but fails to obtain lock L1 even though it is released by Tx1. T2 fails to obtain lock L1 because in order to do so would require Tx1 be aborted as it has flagged itself as conflicting with lock L1 via t.add_conflicting_lock(L1). Yet, Tx1 cannot be aborted since Tx1(irrevocable) = true. T2 stalls until Tx1 completes, setting x = 1, y = 1. T2 then executes, obtaining the necessary locks and returning x = 1 and y = 1 the atomically correct values for x and y (equivalent values) given transaction Tx1.

By incorporating both Lemma 1 and 2, locks inside of transactions naturally compose and emit database ACI characteristics; atomic, consistent and isolated. They are atomic (all operations commit): once a lock is obtained inside a transaction, the transaction becomes irrevocable. This ensures that all operations will commit, irrespective of how many locks are obtained within the LiT transaction. They are consistent: no conflicting locks or transactions can run along side the irrevocable LiT transaction, ensuring memory correctness. They are isolated: conflicting locks or transactions are disallowed from running in conjunction with an LiT transaction, preventing its state from being visible before it commits. Even when a lock is released inside an LiT transaction, other threads remain unable to obtain the lock until the transaction commits.

Criticality of LiT Lock Composition

The composable nature of LiT locks is critical to the incremental adoption of transactions into pre-existing, lock-based parallel software. To understand this, consider a multi-threaded linked list implemented using mutual exclusion locks. The linked list software is mature, thoroughly tested and contains lookup(), insert() and remove() methods. A software designer wishes to extend the linked list's behavior to include a move() operation. The move() operation behaves in the following way: if element A exists in the list and element B does not exist in the list, element A is removed from the list and element B is inserted into the list.With LiT lock composition, the move() operation could be implemented entirely from the previously built locking implementation. Figure 20 demonstrates how this is achieved within the TBoost.STM LATM extension.

```
1   bool move(node const &A, node const &B) {
2       // compose locking operations: lookup(), remove()
3       // & insert() in an irrevocable & indivisible tx
4       // to make a new transaction-based move() method
5       for (transaction t;;t.restart())
6       try {
7           t.add_tx_conflicting_lock(list_lock_);
8           // lookup() uses lock list_lock_
9           if (lookup(A) && !lookup(B)) {
10              remove(A); // uses lock list_lock_
11              insert(B); // uses lock list_lock_
12          }
13          else return false;
14
15          t.end(); return true;
16      } catch (aborted_tx&) {}
17  }
```

Figure 20. Move Implemented with LiT Lock Composition.

The move() method in Figure 20 is viewed by other threads as an indivisible operation whose intermediate state is isolated from other threads, even though it contains four disjoint pessimistic critical section invocations (e.g. two lookups, a remove and an insert). Even when listlock is released intermittently during the transaction execution, other threads are prevented from obtaining it until the transaction commits. This behavior ensures each pessimistic critical section within the transaction is prevented from being viewed independently. Once the transaction commits other threads can obtain listlock and view the cumulative affects of the move() operation.

Understanding LiT Lock Composition

Figure 21 shows a transfer function implemented using LiT lock composition. This example was chosen because the code sample is small enough to be presented in full (not possible with the prior linked list move() example). Figure 21 uses LiT TX-lock protection and contains two locks inside of the same transaction that are subsumed by the transaction. The transaction composes the two separate lock-based critical sections into an atomically viewed and isolated operation.

Consider threads T1 executing lit_transfer(1), T2 executing get1() and T3 executing get2() with x1 = 0 and x2 = 0 in the time line shown in Figure 22. The dotted vertical lines in the conflicting lock time in Figure 22 demonstrate when T1's transaction tx1 obtains and releases locks L1 and L2 with regard to T2 and T3. Thread T1 starts transaction tx1 and adds L1 and L2 to tx1's conflicting lock list. Next, tx1 locks L2 and becomes irrevocable (Lemma 2). Thread T2 then attempts to lock L1, however, since L1 conflicts with tx1 and tx1 is irrevocable, thread T2 is disallowed from aborting tx1 and is therefore prevented from obtaining L1, stalling instead. After tx1 sets x2 = ??1 and unlocks L2, thread T3 tries to lock L1, however, it is disallowed because L1 is on tx1's conflicting lock list and tx1 is irrevocable. Thus, thread T3 is stalled. Transaction tx1 then locks L1, sets x1 = 1 and unlocks

```
1   int x1 = 0, x2 = 0;
2
3   void lit_transfer(int transfer) {
4       for (transaction t;;t.restart())
5       try {
6           t.add_tx_conflicting_lock(L1);
7           t.add_tx_conflicting_lock(L2);
8
9           lock(L2); x2 -= transfer; unlock(L2);
10          lock(L1); x1 += transfer; unlock(L1);
11
12          t.end(); break;
13      } catch (aborted_tx&) {}
14  }
15
16  void get1and2(int& val1, int& val2) {
17      lock(L1); lock(L2);
18      val1 = x1; val2 = x2;
19      unlock(L2); unlock(L1);
20  }
21  void get1(int& val) {
22      lock(L1); val = x1; unlock(L1);
23  }
24  void get2(int& val) {
25      lock(L2); val = x2; unlock(L2);
26  }
```

Figure 21. LiT Transaction and Two Locking Getters.

L1. When tx1 commits, threads T2 and T3 are unstalled and read x1 = 1 and x2 = ??1, respectively, the correct atomic values for the lit_transfer(1) operation with lock composition.

Figure 22. Composed LiT Example using TX-Lock Protection.

In the above scenario, both T2 and T3 tried to acquire the unlocked locks L1 and L2 but failed due to tx1's irrevocability even though the locks themselves were available. The characteristic of disallowing lock acquisition even when locks are available is a primary attribute of LiT transactions. This characteristic is not present in systems which use locks alone, and is a key attribute to enabling lock composition. As demonstrated earlier with LoT transactions, a lock and a transaction conflicting with the same lock cannot both execute simultaneously. LiT transactions use this same behavior and extend it to favor transactions over locks once a lock is obtained inside of a transaction, making it irrevocable. By restricting access of locks to in-flight LiT transactions which have acquired (and then released) them, the system ensures any remaining behavior not yet performed by the transaction will occur prior to other threads obtaining such released locks. This ensures

all the pessimistic critical sections within an LiT transaction are executed in isolation and consistently without memory interference from outside locking threads. Note that this was the case when T3 tried to acquire lock L2 even after transaction tx1 had completed its operations on its shared memory x2.

LiT Lock Identification

LiT TX-lock protection requires that all transaction conflicting locks, those locks used directly inside of transactions, be identified prior to any lock being obtained within the transaction; failure to do so can lead to deadlocks. A simple example of how adding conflicting locks as they are obtained inside of transactions can lead to deadlocks can be derived from Figure 21. If lit_transfer() is executed without adding conflicting locks (e.g. no calls to add_conflicting_lock()), the following deadlock is possible. Thread T1 executes lines 3-5, skips lines 6-7 (the conflict calls) and executes line 9 locking L2 and adding it to its conflicting lock list. Thread T2 then executes lines 16 and part of line 17 of get1and2(), locking L1.Without the transactional code identifying L1 as a conflict, the TM system would not disallow

T2 from locking L1. The system is now deadlocked. T2 cannot proceed with locking L2 as L2 has been added to T1 conflicting lock list, yet T1 cannot proceed as lock L1 is held by T2.

These deadlocks scenarios are overcome by requiring the LiT TM-lock or TX-lock protection to list all conflicting locks prior to obtaining any locks within transactions. Attempting to use locks inside of transactions without first identifying them as conflicts causes TBoost.STM to throw an exception informing the programmer of the missed conflicting lock and how to correct the error. Recycling the same scenario as above with conflicting locks identified at the start of the transaction avoids deadlocks. For example, thread T1 adds locks L1 and L2 to its conflicting lock list. It then executes lines 3-8 and part of line 9, locking L2 (but not unlocking it). Thread T2 executes lines 16 and part of 17, trying to lock L1, and sees T1's transaction as conflicting. Thread T2 tries to abort T1's transaction, but is disallowed as the transaction is irrevocable (see Lemma 2). T2 is therefore stalled prior to obtaining lock L1. T1 continues and executes the remainder of line 9-12, obtaining and releasing lock L1 and finally committing. Once T1's transaction has committed, T2 is allowed to resume and runs to completion.

Unrecoverable transactions to manage with I/O

C++ and Library-Specific Concepts

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

Native Language Compatibility

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.

TBoost.STM 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. 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, TBoost.STM provides a builtin templatized user-configurable memory manager which generally yields 20% performance improvement over direct calls to the default 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. TBoost.STM 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 TBoost.STM's memory manager to perform faster than C++'s native new and delete operations.

TBoost.STM 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 TBoost.STM are similar to those discussed in Bulka and Mayhew's, Efficient C++, Chapter 7, Multi-threaded Memory Pooling. In C++ semantics, the performance gains within TBoost.STM'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. TBoost.STM achieves this by implementing transactions as objects using the Resource Acquisition Is Initialization (RAII) principle.

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 TBoost.STM'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.

Mixin versus wrapper helpers

TBoost.STM's provides some helper classes to make transactional objects. The user should use each one on a per case, taking in account the advantages and liabilities of each one.

1. mixin; transaction_object
2. intrinsic wrapper: transactional_object
3. extrinsic wrapper: transactional_reference_cache

Next follows the features the user should question

1. Allows to use an existing class, for example std::pair class. Does the helper class allows the smart pointers to return an reference to an existing class?
   - mixin: There is no mean
   - intrinsic: the associated smart pointers to a transactional_object<std::pair> returns a std::pair reference
   - extrinsic: This is exacly the role of the extrinsec approach
2. Allows to use an existing instance variable, for example a variable of type std::pair class. Does the helper class allows the smart pointers to return an reference to an existing class?
   - mixin: There is no mean
   - intrinsic: There is no mean
   - extrinsic: This is exacly the role of the extrinsec approach
3. Works for class hierarchies. if D isa B, then helper<D> is a helper<B>
   - mixin: This is exactly the role of the extrinsec approach since we have added the base parameter (class B: mixin<B>{} class D: mixin<D,B>{}).

- intrinsic: Not directly, we need to use some form of cast. clas D:B{}. intrinsic<B>* ptr=new intrinsic<D>(); ????
- extrinsic: N/A because the helper is hidden

4. Allows to avoid the forwarding problem for base classes
   - mixin: As the user defines its own class he can define the exact constructors avoiding the problem added the base parameter
   - intrinsic: the wrapper must define generic constructors having the forwarding problem.
   - extrinsic: N/A because the helper is hidden
5. Allows to avoid the forwarding problem for derived classes
   - mixin: As the inheritance of the base class is done using the helper the base mixin must define generic constructors having the forwarding problem.
   - intrinsic: the wrapper must define generic constructors having the forwarding problem.
   - extrinsic: N/A because the helper is hidden
6. Performs optimally.
   - mixin: optimal, only one allocation/copy
   - intrinsic: optimal, only one allocation/copy
   - extrinsic: instead of a single allocation/copy we need two, so no optimal

The following table is a compilation of the preceding analysis:

Table 3. Comparaison with other STM systems

feature	Mixin	Intrinsic Wrapper	Extrinsic Wrapper
existing class	[No		
	*Yes		
	*Yes		

```
[
    [[*existing variable]]                                  [[No]]      [[No]]                  [[*Yes]]
]
[
    [[*existing class hierarchy]]                           [[No]]      [[No]]                  [[*Yes]]
]
[
    [[*avoid the forwarding problem for base classes]]      [[Yes]]     [[No]]                  [[*Yes]]
]
[
    [[*avoid the forwarding problem for derived classes]]   [[No]]      [[No]]                  [[*Yes]]
]
[
    [[*Works for class hierarchies]]                        [[*Yes]]    [[No]]                  [[*Yes]]
]
[
    [[*Performs optimaly]]                                  [[*Yes]]    [[*Yes]]                [[No]]
]
```

[[**existing variable**] [[No]] [[No]] [**Yes**]] [[**existing class hierarchy**] [[No]] [[No]] [**Yes**]] [[**avoid the forwarding problem for base classes**] [[Yes]] [[No]] [**Yes**]] [[**avoid the forwarding problem for derived classes**] [[No]] [[No]] [**Yes**]] [[**Works for class hierarchies**] [**Yes**] [[No]] [**Yes**]] [[**Performs optimaly**] [**Yes**] [**Yes**] [[No]]]]

Why TBoost.STM's cache uses now memcpy instead of copy-constructor and assignement

Until version 0.3 TBoost.STM's used the copy-constructor to clone a new object which will be used directly by the derreed update policy and as a backup by the direct update policy. This has some drawbacks:

1. Domain: Transactional object are limited to CopyConstructible objects
2. Exception safety: As the user copy-constructor and assignement can throw exceptions
3. Performances: Whole copy, either copy-constructor and assignement, of transactional objects could be expensive

The following sections explain deeply this issues.

Domain

It seems a hard limitation to force a STM user to make all its classes CopyConstructible and Assignable.

Exception Safety

With copy-constructor and assignement TBoost.STM 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. 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, TBoost.STM 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 memory 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.

Performances

Another problem with the copy is that we will need to copy the whole logical object when what we need is just the phisical part, e.g. if copy is used to clone the transactional object, each time we modify a linked list, we will copy the entire list, even if only one node is modified.

A pure transactional object is a transactional object that do not points to any non transactional object. For example, a transactional object including a std::string, is not a pure transactional object, because std::string has a pointer to a dynamically allocated memory which is not a transactional object. For these non pure transactional objects a copy will be needed. One way to limit this is to place every non pure transactional object on a separated class and add a pointer to it. So for example instead of doing

class X { std::string str; // ... other fields

};

which will make X non pure TO, we can do

class X { tx_ptr<std::string> str_ptr; */ owned pointer which is cloned when making a copy but not by the STM system. /* ... other fields

};

In this way X will be a pure TO poining to a non pure TO.

If memcpy is not portable we can consider specializing the clone and copy_state virtual functions.

The user must state explicitly the TO that are not pure by overloading the cacheclone, cache ... . We can also define a trait.

Move semantics

When copying is needed we solve the problem of commit-time and abort-time exceptions by using move semantics in place of copy semantics when the class provided it. The idea of moving is new to C++ and will be available in the next version of the standard.

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 and thusly, not used in TBoost.STM.

TBoost.STM 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 TBoost.STM's functionality, inheritance is used. All of these factors considered, TBoost.STM is a research library that requires type-safety to be a foremost concern, as its usage would 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.

Language-like macro blocks

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

Table 4. Comparaison with other STM systems

Keyword	Behavior
use_lock(L)	**Acquires lock L and executes critical section. Supports single operations without {} scope construction such**
use_timed_lock(MS, L)	**Acquires timed lock L and executes a critical section based on MS millisecond timer. If the timer expires wi**
try_timed_lock(MS, L)	**Same as use_timed_lock(L), but exceptions are caught locally. Requires catch_lock_timeout(L) or lock_tim**
catch_lock_timeout(E)	**Catches timed lock exception E thrown from try_timed_lock(L). Required to follow all try_timed_lock(L),**
lock_timeout	**Catches and discards failed timer exception thrown from try_timed_lock(L). Required to follow all try_tim**

Table 1. TBoost.STM Mutual Exclusion Locking Parallel Constructs.

Library-based Lock Implementations

Mutual exclusion locks, or just locks, ensure a programmerspecified 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.

```
1 // mutual exclusion with lock()/unlock().
2 pthread_mutex_lock(l);
3 int ret = foo();
4 pthread_mutex_unlock(l);
5 return ret;
```

or

```
1 // mutual exclusion with automatic objects.
2 {
3 scoped_lock<pthread_mutex_t> lock(l);
4 return foo(); // lock released
5 }
```

or

```
1 // language-like mutual exclusion.
2 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.

Pitfalls in Scoping of Automatic Object Locks

A closer examination of the automatic object and language-like approaches to locking reveal differences in potential programmerinduced 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_locks and the use_locks are changed. In the case of the scoped_locks, 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_locks, 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.

```
1   // mutual exclusion with automatic objects.
2   {
3       scoped_lock<pthread_mutex_t> lock1(L1);
4       scoped_lock<pthread_mutex_t> lock2(L2);
5       scoped_lock<pthread_mutex_t> lock3(L3);
6       operation_A();
7       operation_B();
8       operation_C();
9   }
```

or 1 // language-like mutual exclusion. 2 use_lock(L1) use_lock(L2) use_lock(L3) 3 { 4 operation_A(); 5 operation_B(); 6 operation_C(); 7 }

```
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:

```
1   grammatical-phrase
2   {
3       operations ...
4   }
```

A number of control structures, specifically in C++, follow this format, such as, if statements, for loops, switches, classes, structs and so forth. While the use_lock structure follows this common practice, scoped automatic objects (e.g., scoped_locks) do not. Because scoped_locks 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.

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. The code below demonstrates three different implementations for transactions from a library-based approach where x is shared memory that must be synchronized.

```
begin_transaction(t);
tx_write(t, x) = val;
end_transaction(t);

transaction with begin()/end():
```

or

```
{
    transaction t;
    t.write(x) = val;
    t.end();
}

transaction with automatic object:
```

or

```
atomic { x = val; }

language transaction:
```

Inspection of the above code reveals that begin_transaction() and end_transaction() are susceptible to the problem when 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 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.

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. The code below illustrates the client code necessary to implement a basic retry behavior for automatic objects and language-like transactions.

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

automatic object transaction with retry
```

and

```
atomic(t) {
    t.write(x) = val;
} end_atom

language-like transaction with retry
```

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

```
// 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();
}

automatic object:
```

and

```
// 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
}

language-like transaction:
```

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 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.

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 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.

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 multithreaded applications today. Without wide TM availability, the primary parallel programming construct used today is locks. Parallel programming research experts unanimously agree that finegrained 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.

Parallel Constructs for Mutually Exclusive Locks

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

```
1   // single-line sequential locking
2   use_lock(L1) use_lock(L2) use_lock(L3) foo();
1   // multi-line sequential locking
2   use_lock(L1) use_lock(L2) use_lock(L3) {
3   foo();
4   }
1   // multi-line sequential single 100 ms timed lock
2   while (polling)
3   try_timed_lock(100, L1) {
4       use_lock(L2) use_lock(L3)
5       {
6           foo();
7           polling = false;
8       }
9   } lock_timeout { execute_on_failed(); }
```

Figure 9. Mutual Exclusion Locks with TBoost.STM.

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.

Exception-based Timed Locks

A notable feature of our TBoost.STM 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.

```
1   // unoptimized timed locking
2   while (polling) {
3       timed_lock tl1(100, L1);
4       timed_lock tl2(100, L2);
5       timed_lock tl3(100, L3);
6
7       if (tl1.has_lock() && tl2.has_lock() &&
8           tl3.has_lock()) {
9           foo();
10          polling = false;
11      }
12      else execute_on_failed();
13  }
```

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 un-optimized 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.

```
1   // optimized timed locking
2   while (polling) {
3       timed_lock tl1(100, L1);
4       if (tl1.has_lock()) {
5           timed_lock tl2(100, L2);
6           if (tl2.has_lock()) {
7               timed_lock tl3(100, L3);
8               if (tl3.has_lock()) {
9                   foo();
10                  polling = false;
11              }
12              else execute_on_failed();
13          }
14          else execute_on_failed();
15      }
16      else execute_on_failed();
17  }
```

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 TBoost.STM 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 TBoost.STM.

Parallel Constructs for Transactional Memory

In this section we describe the behavior, and give examples, of the TBoost.STM language-like parallel constructs for transactional memory. The TBoost.STM 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 TBoost.STM 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

```
void inc_x(int val)
{
    atomic(t) { t.write(x) += val; } end_atom
}
void inc_y_and_z(int val)
{
    atomic(t) {
        t.write(y) += val;
        t.write(z) += val;
    } end_atom
}
void transfer_x_to_y_z(int val)
{
    atomic(t) {
        inc_x(-val);
        inc_y_and_z(val / 2);
    } end_atom
}
```

Figure 13. Transactions with TBoost.STM.

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.

TBoost.STM 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.

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, TBoost.STM 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 languagelevel [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 nonnested 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 TBoost.STM. 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].

Comparaison with other STM systems

Table 5. Comparaison with other STM systems

Features	TBoost.STM	TL2
Lock-free	NO	YES
Lock-based	YES	NO
Validation	YES	??
Invalidation	YES	NO
Direct-Updating	YES	YES
Deferred-Updating	YES	YES
Word memory granulatity	NO	YES
Object memory granulatity	NO	YES

Appendix C: Implementation Notes

Language-like macro blocks

Our language-like lock and transaction parallel constructs are implemented in TBoost.STM 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.

Locking Macros

The lock macros use an unoptimized if statement to ensure the variables inside their local for loop are terminated for nonstandard 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.

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 TBoost.STM API end() does. This non-throwing behavior deviates from the prior TBoost.STM 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.

Correcting Non-Compliant Compilers

The `if (0 == rand()+1) {} else` expression in the preprocessor atomic 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 trys are wrapped within if statements. The if statements and for loops naturally nest without delineated scope (e.g., f, g) allowing programmers to execute single or multiple operations based on their preference.

(0 == rand()+1) always returns false and cannot be optimized away by an optimizing compiler. By using an always false 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_locks and transactions is necessary to release acquired locks, terminate transactions and release transactional memory.

Cache

Dispersed

Compact

Appendix D: Acknowledgements

TBC

Appendix E: Tests

XXX

Name	kind	Description	Result	Ticket
XXX	compile	XXX	Pass	#

Appendix F: Tickets

Kind	Identifier	Description	Resolution	State	Tests	Version
feature	v0.0#1	boostify	XXX	Open	See array_locker_tests	v1.0

Appendix E: Future plans

Tasks to do before review

Interface

- **DONE** Adding transactional smart pointers.
- Allows to have non transactional_object participating on transactions (separate the information related to a transactional object from the object itself and add two pointers to transactional_object_cache one to the object itself and the other to the transactional_object.
- **DONE** Managing Movable and non CopyConstructible types.

Boostifying STM

- **DONE** set boost directory architecture
- **DONE** name files in lowercase
- **DONE** Add a config file with all the configuration macros
- **DONE** Add a stm file at the boost level including all the STM interfaces
- **DONE** Replace bit_vector by std::bitset or boost::dynamic_bitset (BOOST_STM_BLOOM_FILTER_USE_DYNAMIC_BITSET)
- **DONE** Replace Sleep by boost::this_thread::sleep
- **DONE** Replace pthread_mutex by boost::mutex
- Replace THREAD_ID by boost::thread_id
- **DONE** Provide a unique array/tuple locker (Boost.Synchro)
- Replace var_auto_lock by boost::synchro::unique_array_locker
- Replace auto_lock by boost::synchro::unique_locker and redefine use_lock macros
- use lock_guard when lock/unlock
- Adapt the pool to Boost.Pool
- **DONE** Provide a thread specific shared pointer (Boost.Interthreads)
- **DONE** Provide a transparent initialization (Boost.Interthreads)
- Replace thread specific access using the thread id by boost:interthreads::thread_specific_shared_ptr
- Replace the initialization to the library Boost.Interthreads (decorations and decorators)

- Replace draco_move by boost::move and its emulation
- Replace vector_set and vector map by the respectives Boost.Container flat_set and flat_map
- Replace blom_filter by the Boost.BlomFilter blom_filter

Implementation

- **DONE** Separate the data that is global, thread specific but shared to other threads using a lock, thread local or specific to a transaction.
- **DONE** Define access to these data using functions
- Separate the interface from the implementation
- Group all the cache containers (Read,Write,Delete,New) in only one cache in order to improve the lookup performance when using smart pointers.

Tests

- Add unit tests

Benchmarks

- Add some specific benchmarks.

Documentation

- **DONE** Create the empty files and set the doc generation environement
- **DONE** Write the Motivation section
- Write the User'Guide section
  - **DONE** Write the Getting started section
    - **DONE** Write the Installation started section
    - **DONE** Write the Hello World example
  - **DONE** Write the Installation section
  - Write the Tutorial section
    - Write the Preamble/Initialization section
    - **DONE** Write the Simple transaction section
    - **DONE** Write the Composable transaction section
    - **DONE** Write the A Dynamically Prioritized, Composed Transaction section
    - Write the Lock aware section
    - Write the Lock aware section
  - **DONE** Write the References section
  - Write the Glossary section

- Write the Examples section
- Write the Reference section
- Write the Appendix section
  - Write the History section
  - Write the Rationale section
  - Write the Implementation notes section
  - Write the Acknowledgements section
  - Write the Tests section
  - Write the Tickets section
  - **DONE** Write the Future plans section

For later releases

- Integrate with STM test benchmarks as STAMP or STMBench7.
- Add close nested transactions.
- Allows configuration at compile-time and run-time.

More recherch needed

- Mixing STM updating policies.
- Mixing STM consistency checking.