Reducing the Integration Complexity of Software Transactional Memory with TBoost.STM

Vicente J. Botet Escribá  
Wireless Network Product Division  
Alcatel-Lucent France  
vicente.botet@alcatel-lucent.com

Justin E. Gottschlich  
Department of Electrical and Computer Engineering  
University of Colorado at Boulder  
gottschl@colorado.edu

Dwight Y. Winkler  
Department of Computer Science  
California State Polytechnic University, Pomona  
dywinkler@csupomona.edu

ABSTRACT

Transactional memory (TM) is a concurrency control paradigm that reduces the difficulty of writing parallel programs and supports the efficient execution of some concurrent workloads. While TM offers advantages over other abstractions, it can sometimes require complex hardware, programming language extensions, specific compiler support, or enforce impractical software design, making it unrealistic for early adopters.

In this paper, we explore recent advances in TBoost.STM, a software TM library, that support a wider array of C++’s control flow statements and simplify the integration of transactions into programs without transactions.

Categories and Subject Descriptors

D.3.3 [Programming Languages]: Language Constructs and Features – Concurrent programming structure

General Terms

Performance, Design, Languages.

Keywords

Software Transactional Memory, C++ Library, API, Boost

1. INTRODUCTION

Due to computer manufacturers transitioning from unicomputers to multi and manycore computers, parallel programming is now arguably the predominant form of programming [5]. This shift has brought about a research model where investigation into parallel programming models that are efficient and easy to program are paramount [6]. While many parallel programming techniques exist, few, aside from transactional memory (TM), offer both efficiency and simplicity [7, 8].

Transactional memory promotes simplicity by removing named bindings, such as named locks or monitors, to data that is shared across threads. By removing these name bindings, programmer errors, in the form of race conditions, can be reduced or, in some cases, eliminated. Furthermore, TM can execute concurrent threads efficiently by allowing threads to optimistically execute transactions, the synchronization mechanism of TM [9]. The optimistic execution of transactions provides a means for TM to concurrently execute as many operations as possible. When optimistically executed transactions conflict, one or more transactions are undone, so that program consistency is maintained. All of the management of transactions is handled within the TM subsystem which reduces the programmer’s software complexity.

In this paper we concentrate on the simplification of TM from a library viewpoint with a specific emphasis on the challenges regarding, and recent improvements of, TBoost.STM. Because libraries are limited in the amount of code transformation they can apply, common techniques that are available to compiler-based software TMs (STMs) [8] cannot be applied in TBoost.STM, thereby increasing the challenges of simplifying its interface.

2. LANGUAGE-LIKE INTERFACE

In this section, we explore the transformation of a sequential program into a concurrent program using transactions. We show how transactional code is simplified through the recent advances in TBoost.STM’s language-like application programming interface (API). We then detail TBoost.STM’s recent control flow support.

2.1 Transaction Blocks

In [3], Crowl et al. propose a new control construct to introduce transactions into C++:

```cpp
transaction statement
```

As with other control constructs in C++, `statement` defines a new local scope (and is generally a compound statement). The transaction context is initialized upon entering the transaction through the top of the control structure. When the control structure is exited, the transaction ends by either committing or aborting the transaction.

In an approach analogous to Crowl et al.’s, TBoost.STM provides a set of language-like macros which encapsulate the protocol needed to make a transaction atomic. The TBoost.STM `atomic/end_atom` [2] pair has been updated to more closely conform to the Boost coding guidelines as shown below.

```cpp
BOOST_STM_ATOMIC(_)
{
    // transaction block
}BOOST_STM_END_ATOMIC
```
2.2 Transactional Control Flow

If we look more closely into the BOOST_STM_ATOMIC/BOOST_STM_END_ATOMIC preprocessor macros we see they are equivalent to:

```cpp
for (boost::stm::transaction _;
    !_committed();
    && _check_throw_before_restart();
    && _restart_if_not_inflight();
    _no_throw_end()) try{
    // transaction block
    } catch (boost::stm::aborted_tx &){}
```

Observe that the transaction’s context is instantiated in the for loop initializer. When the flow of control exits the for loop, the instantiated transaction object’s destructor will be invoked, causing the transaction to abort if it had not already committed; otherwise, it does nothing.

Conflicts with other threads are arbitrated by the call to _check Throwable before restart() and _restart_if_not_inflight(); and also by the catch of the aborted tx exception. In this subsection, we will show how TBoost.STM handles the general control flow in to and out of the transaction control structure, excluding the somewhat more complex case of an exception being thrown (which is discussed in a following section).

2.2.1 Entering the Transaction Control Structure

The single way to begin a transaction is by instantiating a transaction object, which invokes its constructor. Thus, any jump into the middle of the block where the transaction is declared will not properly initialize a transaction and cause undefined behavior.

However, C++ provides two mechanisms to jump directly into a control block: goto [10] and longjmp. Attempting to jump into the BOOST_STM_ATOMIC control block through the use of goto should result in, at least, a compiler warning. By encapsulating the transaction control structure within a for loop that contains the declaration of a non-POD transaction variable, as is done within the BOOST_STM_ATOMIC macro, a compiler warning will be generated if a goto is used to enter it because Standard C++ forbids such jumps. Unfortunately, the use of longjmp in the same situation does not generate a compiler warning and, therefore, its behavior is undefined for TBoost.STM.

2.2.2 Exiting the Transaction Control Structure

Just as entering a transaction is started in the transaction constructor, exiting it is associated to the transaction variable destructor. When we reach the end of the transactional block, the call to _no_throw_end() will commit the transaction, while the test !._committed() will exit the for loop and destroy the transaction variable. However, as noted by Crowl et al., there are situations where a non-standard exit of a transaction block may be called for, as can achieved through return, break, continue, or goto keywords. Thus, TBoost.STM now provides a number of language-like macros to support this necessary functionality.

Returning a value from a function which contains a transaction structure presents a small challenge in that care must be taken to ensure that the transaction commits before calling return. One straightforward example is when a value must be returned within a transaction, but cannot be returned until the transaction commits. The below example illustrates how the prior version of TBoost.STM handled the above scenario, requiring the programmer to write somewhat awkward code:

```cpp
int inc_c() {
    int res;
    BOOST_STM_ATOMIC(_) {
        res = (c+=i);
        return res;
    }
}
```

In the above example, the programmer must move the return statement until after the transaction structure has exited, ensuring the transaction commits. If we attempt to return instead from within the transaction atomic block, the transaction will not be committed implicitly, as the statement _no_throw_end() is not called.

```cpp
int inc_c(int i) {
    BOOST_STM_ATOMIC(_) {
        return c+=i;
    } BOOST_STM_END_ATOMIC
}
```

However, we can force a commit explicitly before the return statement:

```cpp
int inc_c(int i) {
    BOOST_STM_ATOMIC(_) {
        return c+=i;
    } BOOST_STM_END_ATOMIC
}
```

Unfortunately, this approach still requires the explicit construction of an unnecessary temporary variable to store the return value, which increases the code footprint and could increase the execution time if invoking the temporary object’s constructor is operationally expensive.

To avoid these issues, TBoost.STM provides a language-like macro, BOOST_STM_RETURN, which allows the transaction to commit before returning.

```cpp
int inc_c(int i) {
    BOOST_STM_ATOMIC(_) {
        BOOST_STM_TX_RETURN(_, c+=i);
    } BOOST_STM_END_ATOMIC
}
```

The macro BOOST_STM_TX_RETURN is defined as follows:

```cpp
template <typename T>
T commit_and_return(transaction &t, T expr) {
    t.commit();
    return expr;
} #define BOOST_STM_TX_RETURN(TX, EXPR) \
    return commit_and_return(TX, EXPR)
Similarly, break, continue, and goto are commonly used to direct the flow of control within a single function body [10]. Most C++ programmers would expect the functionality of these statements to be available and to have them treated as transaction successful exits. Thus, TBoost.STM provides a full set of language-like macros which automatically commit the transaction before exiting the control structure per the statement.

The macro BOOST_STM_GOTO can be used to goto to a label in a transactional context. Their general structure is briefly outlined below.

```c++
BOOST_STM_ATOMIC(_) { 
  if (condition())
    BOOST_STM_GOTO(_, label1);
  ...
} BOOST_STM_END_ATOMIC

label1:
...
```

The definition of BOOST_STM_TX_GOTO is quite simple

```c++
#define BOOST_STM_TX_GOTO(TX, LABEL) \
 BOOST_JOIN(__boost_stm_break_, TX)
```

The successful management of the statements break and continue are a little bit more difficult to emulate. This is due to the fact the macro BOOST_STM_ATOMIC introduce already a loop and thus we need to exit from this internal loop, to be able to break or continue the user loop.

In order to exit from the internal loop and then take the appropriate action we have used a trick, we have prefixed the internal loop with the following code

```c++
if (bool BOOST_STM_VAR_STOP = 
  boost:stm::detail:no_opt_false())
{
  BOOST_STM_LABEL_CONTINUE(TX): 
  ... continue;
  BOOST_STM_LABEL_BREAK(TX): 
  break; 
} else
  for // as before
```

where

```c++
#define BOOST_STM_LABEL_CONTINUE(TX) \ 
  BOOST_JOIN(__boost_stm_continue_, TX)
#define BOOST_STM_LABEL_BREAK(TX) \ 
  BOOST_JOIN(__boost_stm_break_, TX)
#define BOOST_STM_VAR_STOP __boost_stm_stop_
```

As the if condition is always false, this code is never executed other than jumping to the labels BOOST_STM_LABEL_CONTINUE(TX) or BOOST_STM_LABEL_BREAK(TX). There is a bemoj with this code. It compiles only if included on a loop. We have preferred to include in on a specific macro BOOST_STM_ATOMIC_IN_LOOP. There are yet two minor liabilities: As these labels could not be used if the user don’t break or continue the compiler could generate a warning. The second is that as we use the name of the transaction to generate the labels we cannot have two atomic blocks with the same transaction name in the same function.

The language-like macros are defined as follows

```c++
#define BOOST_STM_BREAK(TX) \ 
  if (!TX.commit());
  else goto BOOST_STM_LABEL_BREAK(TX)
#define BOOST_STM_CONTINUE(TX) \ 
  if (!TX.commit());
  else goto BOOST_STM_LABEL_CONTINUE(TX)
```

Exiting the transaction structure when an exception is thrown is a more subtle situation, which we cover in a following section.

### 2.2.3 Aborts Due to Conflicting Transactions

When a transaction is aborted due to a conflict with another transaction, an aborted tx exception is thrown. If a transaction is aborted, the default behavior associated to the macro BOOST_STM_END_ATOMIC is to absorb the conflict exception and retry the transaction. By doing this, transactional conflicts are mitigated seamlessly by TBoost.STM and no additional programmer involvement is required to retry the transaction.

However, to avoid livelock, some method of contention management is necessary to ensure that a single thread’s transactions are not starved, as explained by Gottschlich and Connors [1]. One of the recent additions to TBoost.STM’s interface is its programmer-controlled BOOST_STM_BEFORE_RETRY macro, which allows the programmer to control the transaction’s retry behavior without building a new contention management policy. The following code presents a high-level view of the new interface.

```c++
BOOST_STM_ATOMIC(_)
```

```c++
// ... 
} BOOST_STM_BEFORE_RETRY {
  // ... 
}
```

The programmer can, through the use of BOOST_STM_BEFORE_RETRY, provide a hook with the desired conflict resolution functionality. For example, the number of transactional retries can be managed by the macro, the priority of the transaction can be altered proportionally to the current number of executing threads, the transaction could be delayed temporarily preventing further transactional collisions, and so on.

### 2.3 Handling Exceptions

When a non-transaction exception is thrown, the default TBoost.STM behavior is to abort the transaction and propagate the exception upward (inherited from DracoSTM [12]). In
particular, if a non-transaction exception were thrown in the below sample transaction, the exception would abort the transaction and propagate the exception to the first matching catch clause.

```cpp
BOOST_STM_ATOMIC(_) {  
    // transaction block
} BOOST_STM_END_ATOMIC
```

However, as explained in [3, 11], specific exceptions may require specific mitigation. As such, it may be desirable to let the programmer decide how each individual exception is handled. In this section, we discuss how our recent additions to TBoost.STM enable programmer-controlled exception handling. Note that because TBoost.STM’s default behavior is to terminate a transaction when a non-transaction exception occurs, and to propagate such an exception upward, we do not discuss it further in the below subsections.

### 2.3.1 Exceptions Commit Transactions

The most straightforward transaction behavior when a non-transaction exception is emitted is to have the transaction commit. In order to achieve this behavior, the programmer must explicitly state that this is the desired behavior by closing the transaction control structure with the `BOOST_STM_RETRY` macro as shown below:

```cpp
BOOST_STM_ATOMIC(_) {  
    BOOST_STM_TX_RETURN(_, c+=i);
} BOOST_STM_RETRY BOOST_STM_RETHROW(_, Ex)
```

TBoost.STM now provides the following two macros:

```cpp
#define BOOST_STM_RETHROW(TX, E)\  
catch (...) {(TX).commit(); throw;}
#define BOOST_STM_RETHROW_ANY(TX)\  
catch (E&) {(TX).commit(); throw;}
```

### 2.3.2 Exceptions Abort on Exit

Sometimes, when a transaction is terminated by a non-transaction exception, the programmer may want the transaction to abort, but the exception to be absorbed, allowing the transaction to be retried immediately. This behavior is similar to TBoost.STM’s default behavior, except that it applies to all exceptions, not just the ones specific to transactions.

A natural way to implement this behavior is to abort the transaction when the exception leaves the transaction control, as is done in the `BOOST_STM_END`. In order to achieve this behavior, the programmer must explicitly state this is the desired behavior by using the following closing control flow macro:

```cpp
BOOST_STM_ATOMIC(_) {  
    BOOST_STM_TX_RETURN(_, c+=i);
} BOOST_STM_RETRY
    BOOST_STM_ABORT_ON_EXCEPTION(_, Ex)
```

The library provides these two macros:

```cpp
#define BOOST_STM_TX_ABORT_ON_EXCEPTION(TX, E)\  
catch (...) {(TX).force_to_abort();}
```

```cpp
#define BOOST_STM_TX_ABORT_ON_ANY_EXCEPTION(TX)\  
catch (...) {(TX).force_to_abort();}
```

It is important to reiterate that the above behavior is slightly different from the default TBoost.STM behavior in that all exceptions, not just those related to transactions, are absorbed by the transaction’s control flow structure before retrying the aborted transaction.

### 3 BUILT-IN TRANSACTIONAL OBJECTS

Our recent extensions to TBoost.STM add a pallet of transactional objects that wrap many of the C++ built-in types. These added transactional objects improve the TM subsystem’s programming transparency by allowing common logical operations, such as assignment, equivalence, and so forth, to be performed on transactional objects in the same manner they would for built-in types, thereby increasing the intuitiveness of TBoost.STM. To use TBoost.STM with built-in types, the user need only replace the C++ built-in type with the corresponding `tx::` type at the point of variable declaration.

#### 3.1 TBoost.STM’s `tx::object`

Before we explain the `tx::` type design and implementation, we must first explore the transactional base class `tx::object`. TBoost.STM requires that `tx::object` be used as the template wrapper class for any C++ built-in type that the programmer wishes to use directly for reads or writes within a transaction. In addition, `tx::object` uses a mixin class to extend its implementation. Below, we present a brief outline of `tx::object` and the mixin it uses.

```cpp
template <typename T>  
class mixin  
:: public transaction_object< Final > {  
protected:  
    T val_;  
    public:  
        object() : val_() {}  
        template<typename F, typename U>  
            mixin(mixin<F, U> const &r)  
                :val_(r.value()) {}  
        template<typename U>  
            mixin(U v) : val_(v) {}  
        /...  
    }  

template <typename T>  
struct object : mixin< object<T> >, T > {  
    typedef mixin< object<T> >, T > base_type;  
    object() : base_type() {}  
    template<typename U>  
        object(object<U> const &r) : base_type(r) {}  
        // constructor from a convertible to T
        template<typename U>  
            object(U v) : base_type(v) {}  
    };
```

While we encourage programmers to use the `tx::` type typedefs (as will be explained shortly), programmers can use the `tx::object` template by simply supplying a C++ built-in type
as the template parameter for `tx::object<T>` upon variable declaration, as shown below.

```cpp
tx::object<int> i(0);
```

Using `tx::object` directly, the above declaration creates an `int` instance of a transactional object type, named `i`, which can be used inside of a transaction, as shown below.

```cpp
BOOST_STM_ATOMIC() {  
  if (i++==0) i += 5;  
} 
BOOST_STM_END_ATOMIC
```

Once the transaction successfully commits, `i`'s value becomes 5, even though operations performed on `i` are seemingly devoid of transactional operations. This works because `tx::object<T>` provides an implicit conversion to a reference to `T` as shown below.

```cpp
operator T() {  
  transaction* tx=current_transaction();  
  if (tx!)=0) return tx->write(*this).val_;  
  else return val_;  
}
```

This operator opens the transactional object for writing within the current transaction and returns the reference to the specific transaction cache. By doing this, TBoost.STM’s transactions can be written in such a way that they are completely free of transactional annotations within the transaction itself.

In addition, we also provide a transactional read wrapper for built-in types that opens a transactional object for reading:

```cpp
operator T() const {  
  transaction* tx=current_transaction();  
  if (tx!)=0) return tx->read(*this).val_;  
  else return val_;  
}
```

The read-only conversion operator opens a wrapped object for reading, returning the current transaction’s value for the object as a constant:

```cpp
bool is_even(tx::object<int> const& i) {  
  BOOST_STM_ATOMIC() {  
    BOOST_STM_RETURN (i%2==0);  
  }  
  BOOST_STM_END_ATOMIC
}
```

### 3.2 Fundamental types: `tx::numeric`

While the above discussion of `tx::object<T>` provides insight into how the implementation of TBoost.STM reduces the necessary annotations of transactional accesses of built-in types, here we provide the concrete typedefs that further simplify using built-in types, by abstracting away template parameters entirely. By using `tx::numeric<T>` the programmer needs only specify which built-in type he or she wishes to use by referencing the correct `tx::numeric<T>`. typedef `tx::numeric<T>` differs from `tx::mixin<T>` in that it is a final class and the default constructor initializes its value to 0.

```cpp

```

```cpp

template <typename T>  
class numeric: public mixin< numeric<T>, T > {  
typedef object< numeric<T>, T > base_type;  
public:  
  numeric() : base_type(0) {}  
  template<class U>  
  numeric(numeric<U> const & r) : base_type(r) {}  
  template <typename U>  
  numeric(U v) : base_type(v) {}  
};
```

We provide the following numeric typedefs which we recommend be used directly.

```cpp
typedef numeric<bool> boolean;  
typedef numeric<char> char_t;  
typedef numeric<unsigned char> uchar_t;  
typedef numeric<short> short_t;  
typedef numeric<signed short> sshort_t;  
typedef numeric<int> int_t;  
typedef numeric<signed int> sint_t;  
typedef numeric<long> long_t;  
typedef numeric<signed long> ulong_t;  
typedef numeric<signed long> llong_t;  
typedef numeric<signed long long> llulong_t;  
llulong_t;  
typedef numeric<double> double_t;  
```

### 3.3 Compound Types

The following subsections briefly discuss TBoost.STM’s recent simplification of some compound types in C++. Informally, compound types are those types which are made up by using other types. As such, specific mechanisms are needed to gain access to each of the instances of the types contained within an instance of a compound type.

#### 3.3.1 Pointers

Pointers are one of the main compound types found in C++. The following declarations (cp, cpc, ppc, p and cp) are examples of pointers in C++. Note that `ci` and `i` are instances of the built-in int type in C++.

```cpp
const int ci = 10, *pc = &ci,  
*const cpc = pc, **ppc;  
int i, *p, *const cp = &i;
```

Within the TBoost.STM transactional context, the above are written as:

```cpp
using namespace boost::stm::tx;  
int_t const ci = 10;  
int_t const cj = 10;  
pointer<int_t const> pc = &ci;  
pointer<int_t const> pc2 = &cj;  
pc=pc2; // this must not compile  
pointer<pointer<tx::int_t const> > ppc;  
int_t i;  
pointer<int_t> p;  
pointer<int_t const> cp = &i;
```

Note that only the declarations have been changed. More details into the transactional pointer implementation is provided shortly.
3.3.2 Arrays

TBoost.STM’s arrays are also nearly transparent with the exception of the declaration. The following sequential code

```cpp
int v[2];
int *p;
p = &v[0];
p = v;
++p;
```

could be written transactionally in the following way:

```cpp
tx::int_t v[2];
tx::pointer<tx::int_t> p;
p = &v[0];
p = v;
++p;
```

Note that only the declarations have been changed.

3.3.3 Structures and classes

When the user needs a transactional structure, the fields could be declared as built-in transactional objects.

```cpp
struct A {
    A() : i(0), ptr(&i) {}  
    tx::int_t i;
    tx::pointer<tx::int_t> ptr;
};
struct B {
    B() : a_ptr() {}  
    stm::tx::pointer<A> a_ptr;
};
```

We say that this structure is fine-grained, as all of its leave members are transactional objects. The access to the members of this structure are transparent to the client code because the members’ types inherit from tx::mix in.

A simple example of struct A and B’s usage is demonstrated below:

```cpp
A a;
B b;
BOOST_STM_ATOMIC(_) {
    b.a_ptr=&a;
    b.a_ptr->i = -1;
}  
BOOST_STM_END_ATOMIC
bool res;
BOOST_STM_ATOMIC(_) {
    res =(b.a_ptr->i==1) && (*b.a_ptr->ptr==1);
}  
BOOST_STM_END_ATOMIC
```

Now that TBoost.STM’s transactional pointer, arrays and structures have been explained, we can combine them and demonstrate how they are used within a transaction. Below is an example using an array of B’s.

```cpp
B b3[3];
BOOST_STM_ATOMIC(_) {
    b3[0].a_ptr=&a;
    b3[0].a_ptr->i = -1;
}  
BOOST_STM_END_ATOMIC
```

Below is a similar example using a pointer as the referential type, rather than an array.

```cpp
pointer<B> bp;
BOOST_STM_ATOMIC(_) {
    bp->a_ptr=&a;
    bp->a_ptr->i = -1;
}  
BOOST_STM_END_ATOMIC
```

Notice that the accesses within the transactional blocks in the above examples look identical to those that would be used in non-transactional code.

4 COARSE-GRAINED STRUCTURES

Making every leaf of the data type tree into a transactional object has a transparency advantage, but it also has disadvantages, such as incompatibility with legacy code that cannot be changed. In addition, each transactional object uses additional internal memory for transactional rollbacking and conflict resolution. If the number of fine-grained transactional resources grow sufficiently, its performance may be suboptimal.

To demonstrate this, consider the fine-grained transactional string class shown below:

```cpp
struct String1 {
    String1() : ptr(0) {}  
    char* ptr;
};

struct String2 : transaction_object< String2 > {
    String2() : ptr(0) {}  
    char* ptr;
    // other specific member
};

```

With the above design every character of the string will be a transactional object, increasing the number of transactional objects to check during conflict detection. An alternative design is to implement the string class using a single transactional object. By doing this, the number of internal conflict detection operations performed will be reduced and, depending on its usage, performance may improve.

```cpp
struct String2 : transaction_object< String2 > {
    String2() : ptr(0) {}  
    char* ptr;
    // other specific member
};

```

In the case of String2, access to the member ptr is not transparent because its implementation uses the C++ built-in type int*.

The below code shows how String1 is used within a transaction.

```cpp
String1 s;
BOOST_STM_ATOMIC(_) {
    // access the shared not the tx specific
    if (s.ptr==0) {
        // ... delete ptr
        s.ptr=0;
    }
}  
BOOST_STM_END_ATOMIC
```

Next we show the additional code that is necessary to create a transaction using String2. Since String2 needs to inform the TM system which memory locations are being accessed by the
transaction, the below code uses TBoost.STM’s read and write operations on s.

```cpp
String2 s;
BOOST_STM_ATOMIC(t) { 
  if (t.read(s).ptr!=0) { 
    // _ delete ptr
    t.write(s).ptr=0;
  }
} BOOST_STM_END_ATOMIC
```

Notice that we use BOOST_STM_ATOMIC(t) instead of BOOST_STM_ATOMIC( ) in the above transaction. This is because
we access the transaction object (named t) in this transaction. As such, we provide it with a meaningful name. In the prior
examples, the transaction object was not accessed directly, so an underscore was used as a transaction object placeholder.

### 4.1 Using Non-Transactional Types Within a Transaction

There are many non-transactional libraries. In particular, all of the
STL [13, 14] and Boost [15] libraries are currently non-
transactional. TBoost.STM provides a simple way to use non-
transactional types within transactiona: the tx::object. Non-
transactional types can be wrapped by the tx::object template so
they can be used within transactions. For example, consider the
std::vector as shown below:

```cpp
std::vector<int> v;
v[i]=0;
```

The below minor modification would allow such a vector to be used within TBoost.STM’s transactions.

```cpp
tx::object<std::vector<int>> > v;
BOOST_STM_ATOMIC(_) { 
  v[i]=0;
} BOOST_STM_END_ATOMIC
```

TBoost.STM’s tx::object can be used with many of the STL
classes. A full list of STL compatibility is provided on the
TBoost.STM website.

### 4.2 Polymorphic Transactional Objects

Preceding versions of TBoost.STM had a shortcoming where the
copy of the transactional pointer was not polymorphic. That is, the
implementation used operator new T instead of using a
clone operation. While this was sufficient for a single derivation,
it did not handle longer inheritance trees gracefully. Transactional
objects that are polymorphic need to define a clone virtual function
in order to be copied. For example, consider the below
inheritance hierarchy.

```cpp
class B : public transaction_object<B> 
{ public:
  void mod_fct();
};
```

If we have a pointer to B that is initialized by a pointer to D

```cpp
B* ptrB = new D();
```

The following write_ptr call

```cpp
transaction t;
t.write_ptr(ptrB)->mod_fct();
```

will copy construct a new B while it should copy construct a new
D. What is really needed is a write pointer that would clone
instead of copy construct. To address this, we have changed
TBoost.STM’s transaction::write() implementation in
the following way. Instead of performing:

```cpp
base_transaction_object* returnValue =
  new T(in);
```

it now performs:

```cpp
base_transaction_object* returnValue =
  in.clone();
```

The clone function is a virtual function of base_transaction_object. If the clone function is overloaded by a derived class, it will invoke the derived
implementation. If not, the base implementation, shown below,
will be invoked.

```cpp
virtual base_transaction_object*
D::clone() {
  return new D(*this);
}
```

This allows the original behavior to be called when non-derived
copy construction is needed and cloned behavior when derived
cases are needed, satisfying the requirements of both cases
seamlessly.

#### 4.2.1 Using a Macro to Define Virtual Functions

While the above TBoost.STM update handles new memory
construction for multiple levels of derivation, another open
problem exists in ensuring that transactions commit and abort
their updated memory correctly. The updated version of
TBoost.STM provides new mechanisms for copy_state() and
move_state() which are used to ensure transactional
memory is committed and aborted correctly.

TBoost.STM’s current design requires that copy_state() be
implemented within transaction_object. The user then can
redefine copy_state() and move_state(), virtually
within the user-defined transactional class. An example of such
an implementation of copy_state() is shown below.

```cpp
class D : public B {
  ...
  virtual void copy_state{
      base_transaction_object const * const rhs) {
        static_cast<D*>(this)->operator=
          ((D*)&(static_cast<D const*const>(rhs)));
      }
```

The following macro implement the copy_state() method

```cpp
TBoost.STM_LIBRARY_API BOOST_STM_ATOMICнструмент
```
Unfortunately defining these functions for derived classes is cumbersome. To alleviate some of this labor, the library provides a macro that will define all the virtual functions needed and let the user choose when to use it as shown below.

```cpp
class B : public transaction_object<B> {  
public:  
  void mod_fct();  
};  
class D : public B {  
public:  
  //  
  BOOST_STM_DEFINE_VIRTUAL_FUNCTIONS(D)  
};
```

### 4.2.2 Separating Base and Derived Classes

There is an alternative design that, instead of inheriting solely from `transaction_object`, inherits from `base_transaction_object` and `transaction_object` using a new `Base` template parameter which defaults to `transaction_object`. An example is provided below.

```cpp
template <class Derived,  
  typename Base=base_transaction_object>  
class transaction_object  
  : public Base {...};
```

Using this design, users can derive from `transaction_object` in such a way that they are able to specify the final derived class as the mixin parameter to `transaction_object` as shown below.

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

In the above example, `B` passes itself as the final derived class to `transaction_object`. Yet, `D` passes class `B` as its base and itself (`D`) as the final derived class. This ensures that the correct `copy_state()` and `move_state()` mixins are implemented for the final derived class.

However, there is an additional issue with this approach. The forward constructors from `transaction_object<D, B>` to `B` cannot be implemented completely with C++98. This is known as the forwarding problem [4]. As there is no complete solution with C++98, TBoost.STM also provides the `DEFINE_VIRTUAL_FUNCTION` macro to the user.

### 4.2.3 Managing Multiple Inheritance

The preceding scheme works well for a linear inheritance hierarchy. Yet, if the user needs a transactional object that inherits from two transactional objects, it will need a mixin with multiple base classes.

```cpp
struct B1 : transaction_object<B1> {...};  
struct B2 : transaction_object<B2> {...};  
struct D : transaction_object<D, B1, B2> {...};
```

However, as transactional objects require a single base `base_transaction_object`, virtual inheritance is used to guarantee only one copy of `base_transaction_object` is generated. An example of this virtual inheritance is provided below.

```cpp
template <class Derived,  
  typename Base=base_transaction_object>  
class transaction_object  
  : public virtual Base {...};
```

### CONCLUSION

In this paper we have shown how developers can write transactional programs with minimal effort due to the increased degree of abstraction within the latest version of TBoost.STM. In many of these cases, minimal effort means that the only required change to make non-transactional code transactional is to alter variable declarations. With these simplifications, TBoost.STM’s transactions can sometimes be as transparent as full compiler-based transactions. As such, to the best of our knowledge, TBoost.STM is the first STM library that provides this level of transactional transparency. We also explained some deficiencies we found and corrected in earlier versions of TBoost.STM.

### REFERENCES

http://eecs.colorado.edu/~gottschl/boostSTM/pubs/epham08-stm.pdf


[3] Lawrence Crowl, Yossi Lev, Victor Luchangco, Mark Moir and Dan Nussbaum - Integrating Transactional Memory into C++ - TRANSACT 2007 -  


(http://www.upcrc.illinois.edu/documents/UPCRC_Whitepaper.pdf)
(http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf)


