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

Vicente J. Botet Escribá[1] Justin E. Gottschlich[2]

[1]Wireless Network Product Division
Alcatel-Lucent France

[2]Department of Electrical and Computer Engineering
University of Colorado at Boulder

12/05/2010 / BoostCon 2010

Who We Are?

V. Botet
Software Engineer,
Alcatel-Lucent

J. Gottschlich
Ph.D. Student, CU-Boulder

Outline

- Part I: Brief Overview of TM and TBoost.STM Yesterday
- Part II: TBoost.STM Today
- Part III: What else?

Outline

- Part I: Brief Overview of TM and TBoost.STM Yesterday
- Part II: TBoost.STM Today
- Part III: What else?

Outline

- Part I: Brief Overview of TM and TBoost.STM Yesterday
- Part II: TBoost.STM Today
- Part III: What else?

Part I

Brief Overview of TM and TBoost.STM Yesterday

Locks Do Not Compose

```
class Account {
    tx::int_t balance_;
    mutex mtx_;
public:
    void Deposit(int amount) {
        synchronize(mtx_) { balance_ += amount; }
    }
    void Withdraw(int amount) {
        synchronize(mtx_) { balance_ -= amount; }
    }
};
void Transfer(Account& from, Account& to, int a) {
    from.Withdraw(a); to.Deposit(a);
}
```

Locks Expose Details

```
class Account {
// ...
public:
   mutex& get_mutex() {return mtx_;}
// ...
};
void Transfer(Account& from, Account& to, int a) {
  synchronize(from.get_mutex())
    synchronize(to.get_mutex()) {
        from.Withdraw(a); to.Deposit(a);
    }
}
```

Locks Can Deadlock

Thread A

```
Transfer(a,b,10);
```

Thread B

```
Transfer(b,a,20);
```

1. Thread A lock a mutex
2. Thread B lock b mutex
3. Thread B waits to lock a mutex
4. Thread A waits to lock b mutex
5. DEADLOCK

Transaction Can Help Us

```
class Account { tx::int_t balance_;
public:
    void Deposit(int amount) {
        transaction { balance_ += amount; }
    }
    void Withdraw(int amount) {
        transaction { balance_ -= amount; }
    }
};
void Transfer(Account& from, Account& to, int a) {
  transaction {
    from.Withdraw(a);
    to.Deposit(a);
  }
}
```

Transaction in Memory Aspects

- Transaction Memory **ACI** properties
  - Atomic: all or nothing.
  - Consistent: only legal memory states.
  - Isolated: other transactions cannot see until committed.
- Granularity: Word based versus **Object** based.
- Concurrency control: **Optimistic** versus pessimistic.
- Updating policy: **Direct**/**Deferred**.
- Conflict detection: **Validation/Invalidation**.
- **User Defined** Contention Management

STM in four interfaces

- Language-like macros

```
atomic(<transaction_name>)
{ <compound> } end_atom
```

- Built-in wrapper

```
template <class T> native_trans;
```

- Explicit read/write access
  - Perform tx write; returns ref of object to write

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

  - Perform tx read; returns const ref of object to read

```
template <typename T> T const& transaction::r(T&)
```

Using native_trans<int>

Yesterday

```
native_trans<int> C = 0;
void inc(int c) {
  int res;
  atomic (tx) {
    tx.w(C) += c;
    res = tx.r(C);
  } end_atom
  return res;
}
```

Goal

```
int C = 0;
void inc(int c) {

  transaction {

    return C+=c;
  }

}
```

Part II

TBoost.STM Today's

What a STM Library Must Provide to Make Easier Users Adoption?

- A Simple, Intuitive and Non-Intrusive API
- Concurrent throughput
- Interaction with non-transaction world
- Portable implementation

What a STM Library Must Provide to Make Easier Users Adoption?

- A Simple, Intuitive and Non-Intrusive API
- Concurrent throughput
- Interaction with non-transaction world
- Portable implementation

What a STM Library Must Provide to Make Easier Users Adoption?

- A Simple, Intuitive and Non-Intrusive API
- Concurrent throughput
- Interaction with non-transaction world
- Portable implementation

What a STM Library Must Provide to Make Easier Users Adoption?

- A Simple, Intuitive and Non-Intrusive API
- Concurrent throughput
- Interaction with non-transaction world
- Portable implementation

Using object<int>

Now

```
object<int> C = 0;
void inc(int c)
{
  BOOST_STM_TRANSACTION
    return C+=c;
  BOOST_STM_END_TRANSACTION
}
```

Goal

```
int C = 0;
void inc(int c)
{
  transaction {
    return C+=c;
  }
}
```

A Simple, Intuitive and Non-Intrusive API

1. Language-like Transactional Blocks
2. Built-in Transactional Objects
3. Transactional Objects and Object Orientation

A Simple, Intuitive and Non-Intrusive API

1. Language-like Transactional Blocks
2. Built-in Transactional Objects
3. Transactional Objects and Object Orientation

A Simple, Intuitive and Non-Intrusive API

1. Language-like Transactional Blocks
2. Built-in Transactional Objects
3. Transactional Objects and Object Orientation

Outline

1. Language-like Transactional Blocks
   - Transactional blocks
   - Entering Transactional Blocks
   - Exiting Transactional Blocks
   - Aborts Due to Conflicting Transactions
   - Handling User Exceptions
2. Built-in Transactional Objects
   - Transparency
   - Fine Grained Structures
   - Coarse Grained Structures
3. Transactional Objects and Object Orientation
   - Single Inheritance
   - Multiple Inheritance

Outline

1. Language-like Transactional Blocks
   - Transactional blocks
   - Entering Transactional Blocks
   - Exiting Transactional Blocks
   - Aborts Due to Conflicting Transactions
   - Handling User Exceptions

2. Built-in Transactional Objects
   - Transparency
   - Fine Grained Structures
   - Coarse Grained Structures

3. Transactional Objects and Object Orientation
   - Single Inheritance
   - Multiple Inheritance

Behind the Scenes of Transactional blocks

Previous implementation

```
// atomic(__txn)
for (boost::stm::transaction __txn;
        should_try(__txn);
       __txn.no_throw_commit())
try
    <transactional compound statements>
// end_atom
catch (boost::stm::aborted_tx &) {}
<other exception catches>
```

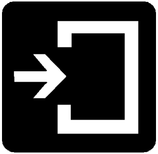
Outline

1. Language-like Transactional Blocks
   - Transactional blocks
   - Entering Transactional Blocks
   - Exiting Transactional Blocks
   - Aborts Due to Conflicting Transactions
   - Handling User Exceptions

2. Built-in Transactional Objects
   - Transparency
   - Fine Grained Structures
   - Coarse Grained Structures

3. Transactional Objects and Object Orientation
   - Single Inheritance
   - Multiple Inheritance

Entering Transactional Blocks



```
for (stm::transaction __txn;
     <CND>,
     <ACT>)
{...}
```

- The single way to begin a transaction is by instantiating a transaction object.
- Any jump into the middle of the transaction block will not properly initialize a transaction and causes undefined behavior.
- C++ provides two ways to jump into a block
  - `goto` a label declared in the transaction block.
  - `longjmp` to a context set by `setjmp` in the transaction block.

Entering Transactional Blocks with goto

```
goto label;
BOOST_STM_TRANSACTION
  // ...
  label:
  // ...
BOOST_STM_END_TRANSACTION;
```

Warning

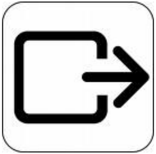
As `BOOST_STM_TRANSACTION` contains the declaration of a non-POD variable, a compiler warning will be generated if a goto is used to enter it.

Toward boost STM

Outline

Exiting Transactional Blocks

```
for (stm::transaction __txn;
        should_try(__txn);
       __txn.no_throw_commit())
```



- Exiting is associated to the transaction destructor.
- The call to _.`no_throw_commit()` commits the transaction
- If the transaction is committed `should_try(__txn)` returns false, exits the for loop and destroy the transaction variable.
- When commit fails, there are two behaviors of this function depending on
  - nested: throws an abort_tx exception
  - root: restart

Exiting Transactional Blocks

Liabilities

- The user expects successful non-standard exit of a transaction block through:
  - `return` from a function,
  - `goto` to a label declared out of the transaction block or
  - `break/continue` a possible user loop.
- Issues:
  - Need to call commit before leaving.
  - Introduction of a for loop .
  - Need to nest a try-catch block to manage with the user exceptions.

Exiting Transactional Blocks With return/goto

The Problem

```
int increase_counter(int i) {
  atomic {
    return c+=i;
  } end_atom
}
```

Because `commit()` has not been called when exiting the block.

Solutions:

- Do not use return on transaction blocks.
- Commit before return.
- Integrate a commit on destruction that will do the commit automatically.

Committing Before Leaving

Language-like solution

Declare a inner variable that will commit on destruction

```
// BOOST_STM_TRANSACTION
for (boost::stm::transaction __txn; CND; )
try {
  commit_on_destruction __comm(__txn);
  try{
      <transaction statement sequence>
// BOOST_STM_END_TRANSACTION
  } catch(...) { __comm.release(); throw; }
} catch (boost::stm::aborted_tx &) {}
<other exception catches>
```

Committing Before Leaving

Committing on destruction

commit_on_destruction

```
struct commit_on_destruction {
    transaction* tx_;
    commit_on_destruction(transaction& tx)
        : tx_(&tx) {}
    ~commit_on_destruction() {
        if (tx_ != 0) commit();
    }
    void release() { tx_= 0;    }
};
```

Exiting Transactional Blocks with break/continue

The problem

```
for(int i=0; i<N; ++i) {
  atomic {
    if (cnd) break;
  } end_atom
}
```

Because `break` will leave the for controlling the transaction block, not the outer user loop.

Solution: Commit before using break/continue. 3 phases

- break/continue the internal loop
- Commit the transaction
- break/continue the user loop

Leaving the Internal Loop with break/continue

Language-like solution

```
{control_flow __ctrl_=none;
  for (transaction __txn; ; ) {
    try { commit_on_destruction __comm(__txn);
      try {
        do { __ctrl_=break_;
         <transaction statement sequence>
          ___ctrl_=none;
          break;
        } while ((__ctrl_=continue_),false);
      } catch(...) { __comm.release(); throw;  }
    } catch (aborted_tx &) <retry compound statement>
  }
  if (___ctrl_==continue_) continue;
  if (___ctrl_==break_) break;
}
```

Outline

1. Language-like Transactional Blocks
   - Transactional blocks
   - Entering Transactional Blocks
   - Exiting Transactional Blocks
   - Aborts Due to Conflicting Transactions
   - Handling User Exceptions

2. Built-in Transactional Objects
   - Transparency
   - Fine Grained Structures
   - Coarse Grained Structures

3. Transactional Objects and Object Orientation
   - Single Inheritance
   - Multiple Inheritance

Aborts Due to Conflicting Transactions

- When a transaction is aborted due to a conflict with another transaction, an `aborted_tx` exception is thrown.
- The behavior associated to the macro `BOOST_STM_END_TRANSACTION` is to ignore the conflicting exception and retry the transaction.
- To avoid live-lock, some method of contention management is necessary to ensure that a single thread's transactions are not starved.

Previous approach liabilities

The retry block was not executed if `no_throw_commit` fails

```
for (stm::transaction __txn; CND;
        __txn.no_throw_commit())
try {...} catch (boost::stm::aborted_tx &)
<RETRY>
```

Partial solution: explicit commit

```
atomic (tx) {
  tx.commit();
} before_retry {
  tx.raise_priority();
};
```

Current approach retry

- `BOOST_STM_RETRY` macro, allows the programmer to control the transaction's retry behavior without building a new contention management policy.

No more need to commit the transaction explicitly

```
BOOST_STM_TRANSACTION
    ...
BOOST_STM_RETRY
  BOOST_STM_CURRENT.raise_priority();
BOOST_STM_END_RETRY;
```

Outline

Abort or commit

- Two ways to manage with user exceptions by default:
  - exceptions abort on exit
  - exceptions commit transactions
- It is desirable to let the programmer decide how each individual exception is handled.

Previous approach liabilities

The retry block was not be executed if transaction aborted during exception handling, as the catch was at the same level than the retry

```
atomic {
  <transactional block>
} before_retry {...
} catch(Ex& ex) {
// transaction aborted
}
```

Exceptions Abort on Exit

Explicit try-block to Commit Transactions

How to commit the transaction on user exception?

```
BOOST_STM_TRANSACTION
  try {
    // transaction block
  } catch(Ex &ex) {
    BOOST_STM_COMMIT();
    throw;
  }
BOOST_STM_END_TRANSACTION;
```

Exceptions Commit Transactions

How to?

- To make exceptions commit transaction by default, we should
  - release the committer only if the transaction is forced to abort
  - avoid commit to throw when during unwinding exception.

```
commit_on_destruction __comm(__txn);
try{
  <statement sequence>
} catch (...) {
  if (__txn_.forced_to_abort()) __comm.release();
  else __comm.commit();
  throw;
}
```

- We have defined a family of macros `BOOST_STM_C_` on which exception commit transactions.

Exceptions Commit Transactions

Explicit try-block to Abort Transactions

How to abort the transaction on user exception?

```
BOOST_STM_C_TRANSACTION
  try {
    // transaction block
  } catch(Ex&) {
    BOOST_STM_ABORT();
    throw;
  }
BOOST_STM_C_END_TRANSACTION;
```

Outline

Outline

Transparency

- We need a `tx::object` wrapper of any built-in type that allow to use them transparently.

Transparent

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

BOOST_STM_TRANSACTION
    if (i==0) i += 5;
BOOST_STM_END_TRANSACTION
```

Implementation

tx::mixin

- Built-in types are encapsulated in a mixin class.
- Conversion from types convertible to T are implicitly convertible to `mixin<F,T>`

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

Implementation

tx::mixin

- Opens for write for every operator modifying the underlying type.

```
template<typename F, typename U>
mixin& operator =(mixin<F,U> const& rhs) {
  if(this!=rhs) ref()=rhs.value();
  return *this;
}
template<typename F, typename U>
mixin& operator +=(mixin<F,U> const& rhs) {
  if(this!=rhs) ref()+=rhs.value();
  return *this;
}
...
```

- Opens for read on conversion operator .

```
operator T() const { return value(); }
```

Implementation

tx::mixin

- ... where:

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

User types

- `mixin` is not a class that can be instantiated.
- We need to derive from it and forward the constructors.
- `tx::object<T>` is a final class
- TBoost.STM provides some shortcuts for the fundamental types on the namespace `stm::tx::int_t`, ...
- TBoost.STM provides a smart pointer type that in addition defines the pointer operators `->, *, ....`

Outline

Fine Grained Structures

Usage

Sequential code

```
struct A {
  int i;
};
struct B {
  A* a_ptr;
};
A a; B b;
{
  b.a_ptr=&a;
  b.a_ptr->i =1;
}
```

Transactional code

```
struct A {
  int_t i;
};
struct B {
  pointer<A> a_ptr;
};
A a; B b;
BOOST_STM_TRANSACTION
  b.a_ptr=&a;
  b.a_ptr->i =1;
BOOST_STM_END_TRANSACTION;
```

Fine Grained Structures

Extreme case

- Consider the following string class:

```
struct String1 {
    String1() : ptr(0) {}
    pointer<chat_t> ptr;
    // other specific member
};
```

- The size of the string will be 8 times bigger than a classic string.
- the change of the complete string would mean the opening for write of N transactional objects
- decreasing the STM internal efficiency.

Fine Grained Structures

Advantages/Liabilities

- Advantages:
  - Transparency.
  - Only the visited fields are cached reducing the copy to the minimum
  - Reduce the chances of conflicts.
- Liabilities:
  - Increase the size of the structure as each leaf transactional object needs some meta-data.
  - When a lot of the fields are visited it would be more efficient to have a more coarse structure.
  - Introduce alias to the same field.

Outline

Coarse Grained Structures

Usage

- Coarse-grained structures are no transparent. The preceding code ... could be written on a transaction context as:

```
struct A : transaction_object<A> {int i;};
struct B : transaction_object<B> {A* a_ptr;};
```

... but the direct use of the fields accesses the shared object.

```
A a; B b;
BOOST_STM_TRANSACTION
    b.a_ptr=&a;
    b.a_ptr->i =1;
BOOST_STM_END_TRANSACTION;
```

Coarse Grained Structures

Explicit access

- The user needs to inform the TM system which memory locations are being accessed by the transaction.

```
A a; B b;
BOOST_STM_TRANSACTION
    stm::write(b).a_ptr=&a;
    stm::write((stm::read(b).a_ptr)->i =1;
BOOST_STM_END_TRANSACTION;
```

- The user code is overloaded with transaction specifics.
- Each access results in a lookup on the cache tables.

Coarse Grained Structures

Getters/Setters

- One way to simplify the code is to use accessors that will open for read/write the embedding object. We can even provide a macro that simplifies the user code.

```
struct A : transaction_object<A> {
 BOOST_STM_FIELD(int, i);
};
struct B : transaction_object<B> {
 BOOST_STM_FIELD(A*, a_ptr);
};
```

- The usage is much more simple, but the lookup is needed yet.

```
A a; B b;
BOOST_STM_TRANSACTION
      b.a_ptr()=&a;
      b.a_ptr()->i()  =1;
BOOST_STM_END_TRANSACTION;
```

- Use accessors when you need to access once the transactional object.

Coarse Grained Structures

Smart Pointer

- An alternative is to use a smart pointer that will open transparently for read/write the pointed structure when the pointer is dereference.

```
BOOST_STM_TRANSACTION
    tx_ptr<B> bPtr(&b);
    bPtr->a_ptr=&a;
    tx_ptr<A> aPtr(bPtr->a_ptr);
    aPtr->i =1;
BOOST_STM_END_TRANSACTION;
```

- While the lookup is avoided if there are multiple accesses, the user code is infected with transactional specifics.
- Use smart pointer when you need to access multiple times to the same transactional object.

Coarse Grained Structures

Advantages/Liabilities

- Advantages:
  - Adapted to objects that are seen as whole..
  - Reduce the size of the transactional object as the meta-data is needed only once.
- Liabilities:
  - Transparent use results in access to the shared object, not the transactional specific one.
  - Transparency is obtained through the use of auxiliary smart pointers.
  - Copy of the whole object is done even if only one field has been visited
  - Increase the probability of conflicts.

Outline

Outline

The Problem

- Earlier versions of TBoost.STM used `new/delete` to create the cache. For example, consider the below inheritance hierarchy.

```
struct B : transaction_object<B>    {
  void mod_fct();
};
struct D : B {};
```

`write_ptr` did a backup copy of `B` instead of `D`

```
B* ptrB = new D();
BOOST_STM_TRANSACTION
  BOOST_STM_CURRENT.write(ptrB)->mod_fct();
BOOST_STM_END_TRANSACTION;
```

Dynamic Polymorphism

- `base_transaction_object` must have a virtual factory function `make_cache` that creates the correct instance.
- For coherency we need to add a `delete_cache` virtual function and rename the `copy_state` function.

```
struct base_transaction_object {
  virtual void copy_cache(
        base_transaction_object const & rhs) = 0;
  virtual base_transaction_object* make_cache(
        transaction& t) const = 0;
  virtual void delete_cache()=0;
    ...
};
```

Toward boost STM

Defining Derived Transactional Objects

Explicit

- As the definitions of the virtual functions provided by `transaction_object<B>` don't work for `D` instances these functions must be overloaded by the final class `D`.

```
struct D : B {
  base_transaction_object* make_cache() {
    return new D(*this);
  }
  void copy_state(
       base_transaction_object const & rhs)
  {
    static_cast<D*>(this)->operator=
       ((D&)*(static_cast<D const*>(&rhs)));
  }
  ...
};
```

Defining Derived Transactional Objects

Macro

- Unfortunately defining these functions for derived classes is cumbersome.
- What about using a helper macro?

```
struct D : B {
  BOOST_STM_DEFINE_VIRTUAL_FUNCTIONS(D)
  // other
};
```

- Could we do better?

Defining Derived Transactional Objects

Parameterizing the Base Class

- An alternative is to extend the `transaction_object` class to have the base class as template parameter.

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

- With this new interface we can define D as follows:

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

- However, there is an additional issue with this approach: The constructors forwarding problem.

Outline

Multiple Inheritance

The problem

- If `D` must inherits from two transactional objects, how `D` can be declared?

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

Use the variadic version of `transaction_object`

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

Multiple Inheritance

Virtual inheritance

- TBoost.STM requires that a transactional object has a single `base_transaction_object`, so virtual inheritance is needed.
- We do not want to force virtual inheritance of Base every time.

```
template <class B=base_transaction_object>
struct virtually : virtual B {
  // forward constructors
};
```

- Virtual inheritance from `base_transaction_object`.

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

Multiple Inheritance

Smart Cast

- The use of `static_cast<T*, base_transaction_object*>` is not allowed when `T` inherits virtually from `base_transaction_object`.
- Need to use `dynamic_cast` in this case.
- To not decrease the performance on the general case we use a variation of `smart_cast`, which will use either `static_cast` or `dynamic_cast` depending on whether the `base_transaction_object` is a virtual base class of `T`.
- `smart_cast` needs its class parameters to be completely defined, as it depends on `is_virtual_base_of`.

Multiple Inheritance

Smart Cast

- `smart_cast` cannot be used for the definition of `transaction_object`.
- `Final` is not completely defined.

Compile Error

```
template <class Final, class Base>
class transaction_object {
public:
  void copy_cache(
      base_transaction_object const & rhs) {
    *static_cast<Final *>(this) =
        *smart_cast<Final const *>(&rhs);
  }
  ...
```

Multiple Inheritance

Incomplete Smart Cast

- As `Final` inherits from `transaction_object<Final>` that inherits from `Base`, the check could be limited to see if `base_transaction_object` is virtual base of `Base`.
- We need an `incomplete_smart_cast` that in addition has an intermediate class used for the tests.

```
void copy_cache(
    base_transaction_object const & rhs) {
  *static_cast<Final *>(this) =
      *incomplete_smart_cast<Final const*,
                             Base const*>(&rhs);
}
```

Part III

What else?

Other Non-Intrusive Aspects

- Deep versus shallow copy semantics.
  - Allows non constructible object to be transactional.
- Separating transactional from user memory management.
  - TBoost.STM does not use any more the specific new and delete class operators internally.

Improving Concurrent Throughput

- Improving the data organization and algorithms
  - Commit time invalidation.
  - Separating the data specific to a transaction or shared by several transactions.
  - Use of transaction specific memory managers (monotonic)
  - Use of movable objects and shallow copy semantics.
- Providing abstractions that allow the users to make their code more efficient.
  - Smart pointers.
  - Use of specific contention managers

Interaction With Non-Transaction World

- Calling non-transactional code.
- Irrevocable transactions.
- Lock aware TM (LATM) synchronization mechanisms

Portable Implementation

- Most of the STM libraries are tied to a specific platform if not a compiler.
- The use of Boost makes our implementation portable on a large set of platforms and compilers.
- TBoost.STM dependencies
  - Thread, Config, TypeTraits, EnableIf, Fusion
  - TBoost.Move
  - TBoost.Synchro

Part IV

Conclusion

Summary

- TBoost.STM provides a simple, intuitive and non-intrusive C++ API for fine grained transactional objects
- Writing efficient transactional programs can not be completely transparent when coarse grained are used.

Features Summary

Feature	Past	Today
Control flow statements exiting transaction commits	NO	YES
Transaction name is anonymous	NO	YES
Abort on internal can be retried	NO	YES
Abort on user exceptions handlers can be retried	NO	YES
Transparent fine grained TOs	NO	YES
Support dynamic array of TOs	NO	YES
Support structures of TOs	NO	YES
Support polymorphic TOs	NO	YES
Support multiple inheritance	NO	YES
Support non CopyConstructible classes	NO	YES
Separated transactional from user memory mngmt	NO	YES

Outlock

1. Reducing Nested Performance Degradation.
2. Analyze the impact of allowing embedded transactional objects.
3. Reduce the meta-data size up to 4 bytes for invalidation and 8 bytes for validation consistency checking.
4. Explore the separation of meta-data from the user object.

Availability

- **Website** :
  http://eces.colorado.edu/ gottschl/tboostSTM/aboutTBoost.STM.html
- **Documentation** :
  http://svn.boost.org/svn/boost/sandbox/stm/branches/vbe/libs/stm/doc/index
- **Download** :
  http://eces.colorado.edu/ gottschl/tboostSTM/downloads.html
- **Sandbox** : http://svn.boost.org/svn/boost/sandbox/stm/branches/vbe/

Questions?

Thanks for your attention.