

---

# Software Transactional Memory

## C++14 STM

---

### Final Report

Zoltan Fuzesi

April 14, 2018

---

Student ID: C00197361

Supervisor: Joseph Kehoe

Institute of Technology Carlow

Software Engineering



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>About the project</b>	<b>1</b>
<b>3</b>	<b>Problems encountered and resolved</b>	<b>2</b>
3.1	Storing unknown objects . . . . .	3
3.2	Template class and functions . . . . .	3
3.3	RAW pointers with polymorphic objects . . . . .	4
3.4	Segmentation fault, deadlock, live-lock . . . . .	5
3.5	Synchronizing commit function without locking code segment . .	5
3.6	Resolved problems . . . . .	6
3.6.1	Storing unknown objects . . . . .	6
3.6.2	Template class and function . . . . .	6
3.6.3	RAW pointers with polymorphic objects . . . . .	6
3.6.4	Segmentation fault . . . . .	6
<b>4</b>	<b>Project description/achievement</b>	<b>8</b>
4.1	Achieved . . . . .	8
4.2	Not achieved . . . . .	9
<b>5</b>	<b>What learned</b>	<b>11</b>
5.1	Technical learning . . . . .	11
5.2	Personal learning . . . . .	12
<b>6</b>	<b>What I would do differently if starting it again</b>	<b>12</b>
<b>7</b>	<b>Report on document update</b>	<b>13</b>
7.1	Additional research . . . . .	14
<b>8</b>	<b>Module description</b>	<b>14</b>
8.0.1	Transaction Manager . . . . .	15
8.0.2	Transaction object . . . . .	16
8.0.3	Base object . . . . .	17
8.1	Public API functions for transactions . . . . .	18
8.1.1	Register . . . . .	18
8.1.2	Load . . . . .	18
8.1.3	Store . . . . .	18

8.1.4	Commit . . . . .	18
8.2	Private API and Transaction Manager functions . . . . .	19
8.2.1	Transaction Manager . . . . .	19
8.2.2	Transaction object . . . . .	20
<b>9</b>	<b>Data structure</b>	<b>20</b>
9.1	Transaction Manager . . . . .	20
9.2	Transaction object (TX) . . . . .	21
<b>10</b>	<b>Testing</b>	<b>21</b>
<b>11</b>	<b>Benchmark</b>	<b>22</b>
11.1	Page Fault . . . . .	22
11.2	Transactions . . . . .	23
11.2.1	Low number of threads transaction test . . . . .	23
11.2.2	Two objects threaded transaction test . . . . .	25
11.2.3	Six objects threaded transaction test . . . . .	26
11.2.4	Collection of objects in transaction . . . . .	26
<b>12</b>	<b>Conclusion</b>	<b>27</b>
<b>13</b>	<b>Acknowledgements</b>	<b>28</b>
<b>14</b>	<b>Bibliography</b>	<b>29</b>

## **1 Introduction**

This document summarizes all of the related works done, problems, errors, tests and learning outcomes, while developing Software Transactional Memory (STM) throughout the project lifetime.

The Software Transactional memory based on a concept to use memory locking at software level to increase the concurrent programming performance without changing the computer or CPU architecture. This concept was implemented in many programming languages, and the most of the implementations use primitive type (int, double, char etc..) shared memory space locking to protect the shared data in front of other threads (light weight processes).

This project intend to produce a STM solution in C++ language that can be used by any C++ application without changing or customize the library code. Since OOP (Object Oriented Programming) is very popular, I decided to implement object based STM library to protect complex objects instead of primitive type memory protection. This decision causes lots of unknown problems such as working with any type of objects without to know anything of the connected application. If I choose the traditional memory storing option, then the experiment can go much further, since the primitive types of variables are known types and much easier to work with them.

This document describing the library, the data structures and the related testing carried out to develop the Software Transactional Memory library.

## **2 About the project**

The purpose of this project to implement Software Transactional Memory, concurrency control solution as a Shared and Static library, that can be included in any C++ software application. Most of the STM (Software Transactional Memory) libraries works with a single primitive variable type, as it stores and protect the memory location of a single variable. Those type of STM solution doesn't need any extra implementation in order to use the library. This project produced a STM solution, that can store object instead of primitive variable memory location. Because the Shared or Static library able to take any type of object, and the library doesn't know anything about the client application objects, it uses poly-

morphic class hierarchy to achieve the required functionalities.

It's required extra implementation to the client application to inherit from the library base object. This inheritance gives a huge flexibility to my library to work with any type of unknown class instance. By the inheritance, the base object supports all the required fields and functions to the library, to carry out all the STM library (Software Transactional Memory) specific computations.

To achieve Transactional Memory implementation, we can use programming language extensions, specific compiler support, that required to install or replace/upgrade the gcc/g++ compiler on the developer system. My project not using this compiler specific features, the logic was built from scratch, to implement the Software Transactional Memory functionalities. I chose this way of implementation, because I want to learn as much as possible from this project and from the transactions.

With the previous testing and the short experience with the library, I can say the library can handle any number of threads with any number of objects. Of course, if we increasing the number of objects and the complexity of the transactions, the running time and the roll-back will significantly increasing. The library supporting the nested transactions as well.

To use my STM library with any c++ application, we need to follows some rules to achieve the inheritance. The tutorials of this implementation and from the API functionality can be found on the project website. <http://glasnost.itcarlow.ie/soft-eng4/C00197361/index.html>

### 3 Problems encountered and resolved

The most significant problem caused by the theory such as use the library, with any c++ client application, as it store and modify the instances of any class instance within the transactions, without changing or customize the library code. To achieve this functionality it need a different way of thinking to developing the library. When the programmers building a new library for a given application, that compatible only with that application, then they can use a reference of any classes in the library solution. It makes the development easier, but the code must be redesign/customize to work with other software as well.

The second biggest problem was to synchronize objects state with multiple threads, without locking the code segments. Because if we locking code segments, we narrowing down our application to sequential execution, where all transaction will be dependent on other transaction, however they are not using the same objects and they should running concurrently instead of sequentially.

**These theoretical questions caused the following problems:**

1. Storing unknown class instances.
2. Template class and functions.
3. RAW pointers with polymorphic objects.
4. Segmentation fault.
5. Synchronizing specific library functions without locking code segment.

### 3.1 Storing unknown objects

During the research I have found the boost c++ library, as a best solution to store unknown objects. However, I didn't planned to use third party libraries, everywhere I have faced to the same answer. So, the boost library does the job, however in my solution without knowing the client application object to store it, the boost library doesn't helps at all it turned out. If I store primitive type rather than object type, the boost library could be a better solution. I tried to implement the solution, but it's required to use template class and functions, that caused the next problem. *3.2 Template class and function*

### 3.2 Template class and functions

In c++ programming language, if we intend to use shared or static library with the client application, the header files must be include in the developed code directory/compilation process. If the code contains a template class or function, that means the code will be checked in runtime, and must be placed in the header file. There is a way to hide the template functions into the cpp (source) files, but that required to know the target object reference to work with.

So, if all the code relying on template classes or/and functions, the static or shared library can not be build, because only the cpp (source) files will be build into the library file and the header files requires to include into the client application as a entry point to the library. To solve this problem, I need to remove the boost library, and redesign the code with standard c++ solution, provided by the available libraries in the language, and providing only the functions declaration in the c++ header files.

### 3.3 RAW pointers with polymorphic objects

During the research, all of the documentation found used RAW pointers. For this reason, and because the RAW pointers the fastest in c++ language, I used as a first solution. The RAW pointers did not cause any problem during the development, until the memory management test. The memory management test happened with **Valgrind** programming tool for memory debugging, memory leak detection, and code profiling.

With this test in the project I have faced with memory leaks. The library use virtual methods in the base class to copying the objects within the transactions, and these methods are implemented in the child classes, and it can be used by the polymorphic class behaviour. This functions causing memory leaks during copying the child class instance. I have tried to use smart pointer in this function to bridge the memory leaks, but because the library doesn't know anything about the connected classes, the casting in the library from smart pointer to RAW pointer is impossible without knowing the target class reference.

```
==26335== LEAK SUMMARY:
==26335==  definitely lost: 0 bytes in 0 blocks
==26335==  indirectly lost: 0 bytes in 0 blocks
==26335==  possibly lost: 0 bytes in 0 blocks
==26335==  still reachable: 72,704 bytes in 1 blocks
==26335==  suppressed: 0 bytes in 0 blocks
==26335== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
==26335== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

*No memory leaks with smart pointers using 300 threads.*

The **==26335== still reachable: 72,704 bytes in 1 block** part of the report is

a common problem when using Valgrind with different version of compiler and Linux distributions.

### 3.4 Segmentation fault, deadlock, live-lock

The segmentation fault, deadlock and live-lock are the more frequent problems when working with multi threaded programming environment. The project not uses mutex locks to lock code sections, apart from registering the the objects and transactions, it is a huge challenge to find where is the memory corruption deadlock or live-lock happens. Sometimes, the code executing correctly, but sometimes throws an error or just hanging forever, it depends the threads sequence execution in the background by the CPU scheduler, and the number of threads used in the process. The code might throws an error with hundred threads but might not if the process uses only ten threads. Even, if the code executing correctly, it should be executed multiple times, to make sure there is no corruption or other locking events happen.

I started to use the gdb default C++ debugger, but it is a bit difficult and time consuming when the process uses hundreds or thousands of threads. As a faster debugging solution I start printing out messages to the console from the different code segments, to narrowing down the error area, to the possible line of code or segment where is the error raised by the thread(s).

### 3.5 Synchronizing commit function without locking code segment

To synchronize the library commit function execution, the project use object base locking. If the code segment locked by a mutex lock, then all the transactions are dependent on the other transactions even if they are using different objects, and only one transaction can working at the same time. To solve the concurrent transactional execution, all the functions are completely lock free, and the transactions are locking the objects itself. This design blocking those transactions only, in which using any object that is participating in any another transactions. So, if we have five transactions, they are not using any object from another transaction, then all the transactions can make changes on their objects at the same time. The concurrency dependent on the computer architecture. More core better concurrency.



## **3.6 Resolved problems**

### **3.6.1 Storing unknown objects**

To store unknown objects the project use inheritance, polymorphic behaviour as it stored the objects as a base class reference type. All the classes in the client application we intend to use with the library, must inherit from the base class. If the project written in other programming language, we can argue about this solution, because might the language not supporting multiple inheritance, and the client application classes will be restricted to inherit from the base class only.

The c++ programming language support multiple inheritance, in this way the client application classes are not restricted to the library supported base class polymorphic behaviour and they can inherit from another classes as well if that is required. This programming design gives me lots of freedom to control unknown objects in the library.

### **3.6.2 Template class and function**

The library store the unknown objects as a base class reference (OSTM), the use of template class or/and function no more required in the library implementation. All the function codes has moved into the cpp (source) files, and only the class and function declaration left in the header files that is visible to the client application.

### **3.6.3 RAW pointers with polymorphic objects**

The RAW pointers are replaced with smart pointers, that give me a freedom to use the polymorphic behaviour in the target classes to control the object copying mechanism from my external library without knowing the client application. In this way the smart pointer get destroyed when goes out of the scope, and the memory leak has solved.

### **3.6.4 Segmentation fault**

The segmentation fault was the most tricky part. If the application use for instance hundred transactions and six hundred objects in every transaction, the possible permutations and combinations of the concurrent execution of transactions/threads are really big. If we running the same combination twenty times

might the code will faced with the segmentation fault if not well designed. Every time, when the code was changed, it was at least hundred times executed with the same transactions combination, to check the percentage of the code failure.

During the development, the transactions was mixed up as complicated as possible, to try to causing segmentation fault early at code change. In this phase of the project life time, I have to say there was no segmentation fault in the last two iteration. But the possible complexity with high number of objects and transactions might can not be tested by one person, the segmentation fault might can be raised by other person who thinking a bit different way of the problem.

## 4 Project description/achievement

The initial goal was to implement a Software Transactional Solution in C++ language, where a piece of code executes a series of reads and writes to shared memory, and these logically occur at a single instant in time. The intermediate states should not be visible to other (successful) transactions.

### **Mandatory:**

1. Full Software Transactional Memory implementation.
2. Full documentation of the library with Doxygen.
3. Tutorial of the library usage.

### **Discretionary:**

1. Library is tested across multiple platforms(Windows, Linux, MAC OSX).
2. Website with downloadable libraries and demonstrate the usage.

### **Exceptional:**

1. Benchmarks: The library is fully benchmarked.
2. Comparison: Compare with other STM approaches.

### 4.1 Achieved

**Mandatory :** The STM implementation has achieved using any number of threads/transactions, and any number of objects/complexity. To prove that the transactions are not visible to other transactions, the test cases using multiple threads with the same objects, to cause race conditions. The Transaction Manager responsible to handle the threads as it assign new transaction to each thread. In this way, every transaction has it's own copy of the working objects and sharing the original global memory spaces. This programming design implementation ensure that the intermediate states in the transactions are not be visible to other transactions, but the final changes on the global shared memory spaces will be visible only when the transactions has finish with it's changes.

The objects/memory is shared between transactions, to prove it we can display the object values/state after all transactions finish with the computation on them,

and if all the transactions accessed the same objects the final value/state should increased and consistent. To prove the STM not corrupting the objects within the transactions, we can execute the same code repeatedly, and the object state after the transactions finished should be same all the time.

The library fully documented with the Doxygen document generating tool and linked, available through the project website alongside the tutorials and the downloadable multi platform (Windows, Linux, MAC OSX) library files.

As an extra the transaction nesting is added to the library, where we can nesting our transactions to any level. It means, if the transaction dependent on other logical inner transaction, all the nested transaction will be executed as a single unit.

**Discretionary :** The library is developed on the three main platform, Windows, MAC OSX and Linux. On Windows I used Visual Studio, because the MVSC (Microsoft Visual Studio Compiler) supports only the `std::mutex` in Windows platform, otherwise the `std::mutex` need to be change. On the Unix platforms, Linux and MAC OSX, I used Netbeans 8.2 as the main development IDE. The library development and testing processes happened on Linux platform. The project website has tutorials to demonstrate how to compile and use the library on the different platforms.

**Exceptional:** To compare the library with other similar STM solution is not the easiest task, since my library using objects, and the most STM solution using primitive type pointer based implementation. The document contains a section called "**Benchmark**", where I tried to explorer the library specific possible memory faults, performance issues and using a diagram from a Master Thesis to compare the library performance with other libraries from the past decade.

## 4.2 Not achieved

The Software Transactional Memory theory was a very research intensive area, and most of the papers I found during my research to support my project are from Master and PHD level. All of this papers uses the primitive type memory space protection, what is a bit different way as my project handling the protected memory spaces. My library use polymorphism, since it can handle any type of unknown objects with C++ applications. As an extra I planned to implement the

primitive type memory protection as well, but due in lack of time the implementation behindhand.

It is not an easy subject, and I should try using different data structure like binary tree, Red-Black tree, linked list and many other data structures to compare the library performance with them.

## 5 What learned

During the development I have learning a lots from transactions, concurrency control and C++ language features.

### 5.1 Technical learning

#### Transactions:

1. What is transaction at all.
2. How to implement atomic expression with my own code.
3. Have a fear idea how to implement transaction in any application in the future to protect shared memory space, where multiple user can access shared data with write access.

#### Concurrency control:

1. Create lock free implementation of protecting critical section.
2. Using object base locking, to prevent other threads to access same object at the same time.
3. How complicated can be working with multi threaded environment.

#### C++ language features:

1. Difference between C++11, C++14, C++17.
2. Where to use RAW pointers and smart pointers.
3. Better knowledge/experience of C++ programming language.
4. Building code as using polymorphic programming features to handle unknown objects.

#### Static and Shared library implementation:

1. Control the build of c++ programming language using Make file instead let the IDE to control the build process.
2. What is the difference between Static and Shared libraries.
3. Why is important to build libraries instead reuse, copy and paste the same code.

## 5.2 Personal learning

**I have learned from my project:**

1. *Time management.*  
Deliver planned portion of the project for a given deadline.
2. *Setting more specific targets for iterations.*  
There was no weakly plan, I only had a big picture what I want to achieve to the end of the iterations.
3. *Why good research is very important.*  
If I spend a bit more time on a given problem/question at research phase, what caused big problem later in the development, that can save lots of time and I don't need to rewrite the already achieved part of the project.
4. *Design efficient code in productive way.*  
The first versions of the library contains huge amount code, where the single *load* function was much bigger then the whole transaction class at the end of the project.
5. *Design part of the solution first in theory on paper, before jump into the code.*  
Before this project I had a bad habit, as I jumped into the code and design the the application on the fly. The structure of the project shows me a different way, how to think about the problem, before start coding.

## 6 What I would do differently if starting it again

If I've to do this project again, I would spend bit more time with research on the object base locking and not spending that much time to find out all the compiler (g++/gcc) supported transactional syntaxes. My library store object spaces rather than primitive type variable spaces, that gives an advantage to lock the objects themselves.

The area where I would spend more time is the pointers, and not relying on the opinion by other programmers and on my personal preference. If I use smart pointers from the begging, I don't need to redesign my code from RAW pointers

to smart pointers.

The another area to spend more time is the way of store the unknown objects. When I found the third party boost library, I've finished the research and made my decision to use it. With a bit more deeper research on boost library, I could save lots of time, because most of the code needed to be redesign when it was replaced with standard map collection. Otherwise, the project was out of my comfort zone with the concurrent programming, and everything about Software Transactional Memory concepts was completely new to me.

## 7 Report on document update

Between the design document and the final implementation there are some changes.

1. **There is no start transaction API function.**

The library API design was similar to TinyStm (STM library) API implementation, where is the client application need to notify the library to start the transaction. In my library solution, the singleton Transaction Manager manages all the transactions and the management is automatically behind the scene.

2. **Object names and API function declarations.**

In the design document, perhaps in the first iterations the process id, and the thread id was part of the functions parameters in the API connection. These function parameters makes the library usage very difficult, and I tried to remove all the extra requirements from the API function parameters, and keep them as simple as possible. The lots of unnecessary parameters between the client application and the library API makes the developer job very awkward, because have to pay attention on details that is not important to the developer. The object and function names are changed as well, now they are shorter and more meaningful.

3. **There is no functions to receive collections.** In the design document I have separated functions to work with collections, but those ideas are removed from the final product. The client application can iterate through the collections to register and make changes on the objects. It gives full control on the objects to the client application rather than trusting in the library.



4. **Double version checking.** The library use object based locking mechanism, that close out other transactions while a single transaction has ownership on the objects. With this design, the other transactions cannot access the same objects what is already used by another transaction, and the double checking mechanism not necessary, only makes the execution time longer.
5. **Domain model.** Actually the Domain model has change a lot about the API functions, for this reason I have added the new updated domain model to the Design Document with the connected sample classes. I found better way to manage the transactions, if I leave only few available options to the client application to use, and manage everything by the library automatically. In this way the library has less API function, that makes it easier to use, but has better self management in my opinion .

## 7.1 Additional research

A small additional research required to the nested transactions implementation, because it was not in the description and the research was narrowed down to the main functionalities.

The another part that needed research, is unit testing for C++ programming language.[3] As everything else, the unit testing for c++ multi-threaded environment is more difficult then sequential programming.

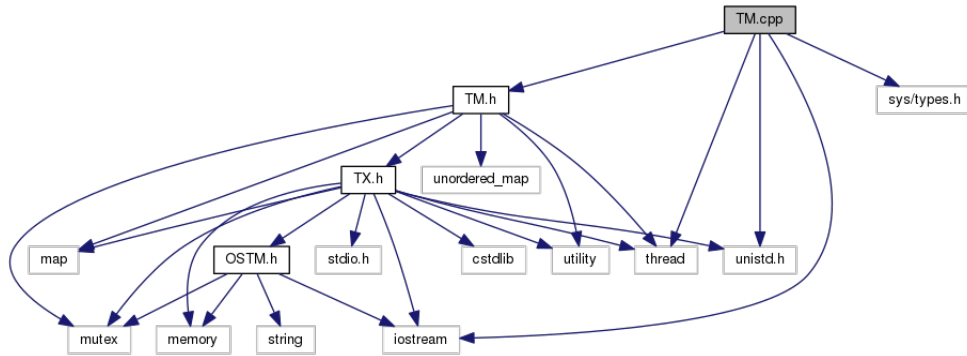
## 8 Module description

The Software Transactional Memory C++ library, is a concurrency control mechanism, analogous to the database transactions. The library provide lock free programming solution to the client application, to manage shared memory spaces between transactions. The library built up from three main parts, Transaction Manager, the transaction object and the base object. If the client application using my library, then the way of programming in the client application is exactly same, as to programming a sequential coded application. The library take care of the thread synchronization, race-condition and memory corruption protection. The client application only need to create and use the threads, but doesn't need to worry about locking any function or code segment.

### 8.0.1 Transaction Manager

The Transaction Manager responsible to manage all the transaction associated with the connected client applications. When a Transaction Manager requested from the library, then a singleton object provided to the application to manage the transactional environment. The Transaction Manager is the only object who has permission to create a transaction object. The client application cannot create or request a transaction. This restriction was necessary to implement, because the library need full control on the created transactions with a single controller. Every single public API function in the library, could throw a runtime error, if the client application try to use an illegal or null pointer of an object.

**"TX EXIT"** public function is providing the memory clearing management to the library. If the library shared, then the used memory by the main process should be cleaned up before the process exit. However, the library using smart pointer, that get destroyed when goes out of the scope, the process associated collections are still in the memory. The **"TX EXIT"** function should be called by the client application, to delete all process maintained values. The Transaction Manager in this function deletes all the main process specific stored values, and calls the **"ostm exit"** transactional function to clean up the global map from the main process specific memory spaces as well. In this way the shared library still running in the background in the operating system, but not accumulating the memory usage.

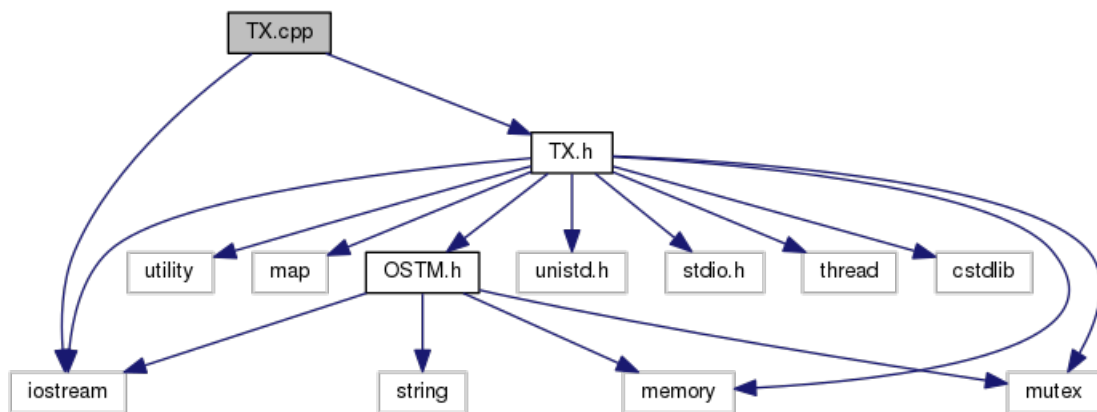


*Transaction Manager.*

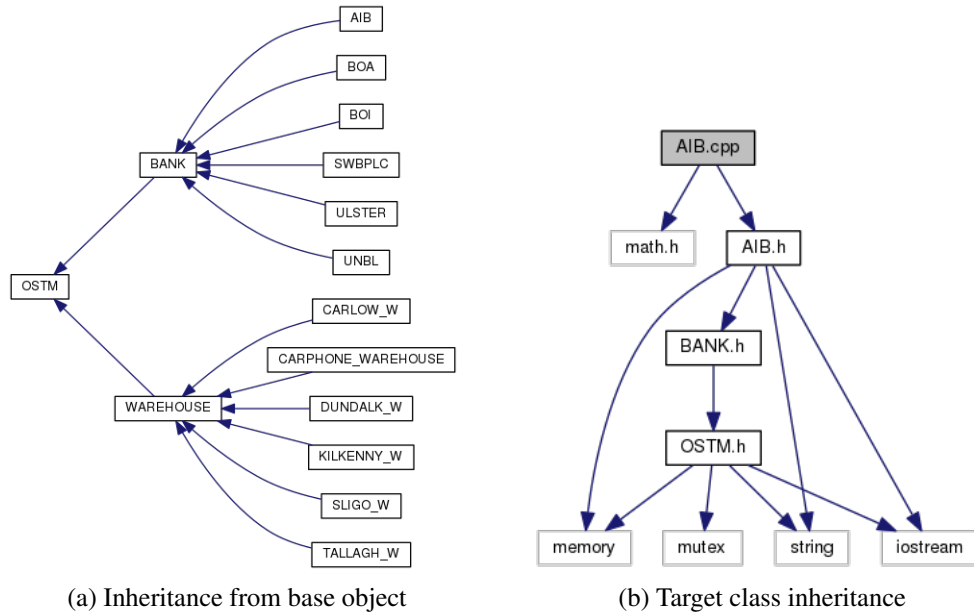
### 8.0.2 Transaction object

The transaction objects created by the Transaction Manager can access the library API functions, to manage polymorphic objects throughout the transaction lifetime. It's including to register, load, store and commit the objects accessed by the transaction. There are more functions in the library, but those functions are not accessible to the client application. Those functions are private and automated within the transactions. Transaction can handle all process specific objects. The process can be the main process or light-weight process as well. To use transactions, doesn't need to use threads at all.

All transaction has private objects and collections, and sharing some class level field and collection too. The class level sharing necessary to use, because the library must let communicate the transactions when they needed. These communications are not controlled, since the thread sequence executions are completely random, controlled by the kernel. The communication happens in the register function, because all the transaction accessing the shared collection with write access. The sharing happens on the original objects between all the executing transactions when ever they needed, and its controlled by the objects locking mechanism itself.



*Transaction Object.*

*OSTM based object types*

### 8.0.3 Base object

The base object provides all the required fields and functions to the inherited classes to work with the library. It is creating the uniqueness to the inherited objects, that required by the library to identify every single objects used, registered within the library. The base object providing the locking and the copying mechanism to the library as well, through polymorphic behaviour.

## **8.1 Public API functions for transactions**

### **8.1.1 Register**

The register function receiving a base class type smart pointer to register into the library. If the object already registered, the library will ignore the register request. When the object registered first time within the library, goes into the global shared map, and a copy of the object( working copy ) get registered in the transaction as well. This object will be used to make temporary changes associated with the target object, memory space during the transaction what is invisible to other transactions.

### **8.1.2 Load**

The client application can request for an temporary/working object using the original object reference. With this request the library returning the working object, to use by the client application. If the function receiving a illegal object, null pointer, or any object inherit from the base object, but not registered, that will throw a runtime error.

### **8.1.3 Store**

When the client application wish to store the working copy of the object, it can use the store API function to save the changes during the transaction lifetime. If the object registered in the library, the object reference will update with the new changed object reference. If the function receiving a illegal object, null pointer, or any object inherit from the base object, but not registered, that will throw a runtime error.

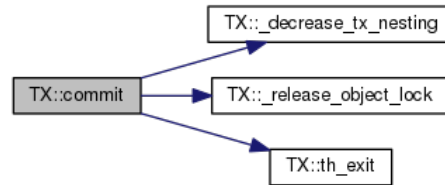
### **8.1.4 Commit**

The commit API call is the most complex function in the transaction. First the library need to check if the transaction is nested or not. If the transaction nested, then will call the private function to decrease the transaction associated value, and return to the client application to finish the outer transactions. If the transaction not nested, then need to find all the global shared object to lock them from the other transactions. If any of the transaction objects are locked in the global collection, then the transaction must wait in busy waiting, until the objects get unlocked

buy the another transaction.

When the transaction get access/ownership to the object, then will lock it, and compare the version number associated with working copy and the original object. If any of the object within the transaction collection has different version number than the global shared original object, the transaction should finish and get the new updated values for all the working copies. When the transaction finishes, it will call the private function in the library, to unlock all the global objects associated with the transaction.

If the transaction committed all the changes on all the objects, then return true to the client application to indicate the transaction successful finished. If any version conflict was detected during the transaction, the function returns false to the client application to indicate, it need to restart the transaction. Since, the commit function doesn't take any object reference, because it knows all the associated working copies and global shared objects with the transaction, we cannot cause any programmed error from outside in the commit phase with illegal object or null pointer.



*Automated functions.*

## 8.2 Private API and Transaction Manager functions

### 8.2.1 Transaction Manager

When the singleton Transaction Manager request for a transaction object, the transaction manager class checking the actual thread is registered in the manager or not. If the thread registered, the transaction associated with the thread is returned with increased the nesting counter. If the thread not registered, it will call the register private function, to register the thread within the transaction manager. The register function including an another private function call, that returns a new map to store the thread within the transaction manager.

### 8.2.2 Transaction object

During the register process, the transaction request a new map from a private function to every single application, to store all the associated transactions with the main process.

If the transaction finish or need to restart, will call the private function to release all the global shared objects, give access to the other transactions make changes on them.

When the transaction finish with the changes on the objects, it calls the exit private function to clean up all the associated working copies from the memory.

The transaction manger can call a function to clean up the global shared map from all the associated memory spaces before the application exists.

The transaction manager can call a function to increase a value within the transaction to indicating the nesting level. To decrease a nesting level, the transaction need make a call from the commit function, if the transaction wants to commit the changes, but the transaction is nested.

## 9 Data structure

There are many different collection type can be use to store the objects within the library. Actually, the collection type doesn't make different in the library functionality, but it can make different between the library performance. I have used the `std::map`, that is a sorted associative container, that contains key-value pairs with unique keys. I chose the map, because it has the fastest performance to access a unique object by the key in the collection without need to iterate through. However, the binary tree is a very fast data structure, but still needs to check the leaves from the root to find the requested object.

The library need to create good few collections to keep the association between processes and transaction.

### 9.1 Transaction Manager

To store all transactional objects created with Transaction Manager:

```
std::map<std::thread::id, std::shared_ptr<TX>>txMap;
```

To store all process (threads) associated keys to find when deleting transactions.

```
std::map<int, std::map< std::thread::id, int >> process_map_collection;
```

Function used to return a map to insert to the process map collection as an inner value.

```
std::map< std::thread::id, int > get_thread_Map();
```

## 9.2 Transaction object (TX)

To store parent based pointers to make invisible changes during isolated transaction.

```
std::map< int, std::shared_ptr<OSTM> > working_Map_collection;
```

To store parent based pointers to control/lock and compare objects version number within transactions.

```
std::map<int, std::shared_ptr<OSTM> > main_Process_Map_collection
```

To store all process associated keys to find when deleting transactions.

```
std::map<int, std::map< int, int >> process_map_collection;
```

Function used to returning and map to insert to the process map collection as an inner value

```
std::map< int, int > get_thread_Map();
```

## 10 Testing

As part of the developer tools I used CppUnit, unit testing framework to test the API functions and the library functionality. *CppUnit is the C++ port of the famous JUnit framework for unit testing.*[2][3]. I have described few testing methods in the Design Manual document to prove the library correct functionality. All those testing methods are implemented, but because those methods are not so advanced testing perspectives, I added more complex test to prove that it works as it suppose to.

### Main tests:

1. Multi threaded multiple object exchange test.
2. Multi-threaded single Object test.



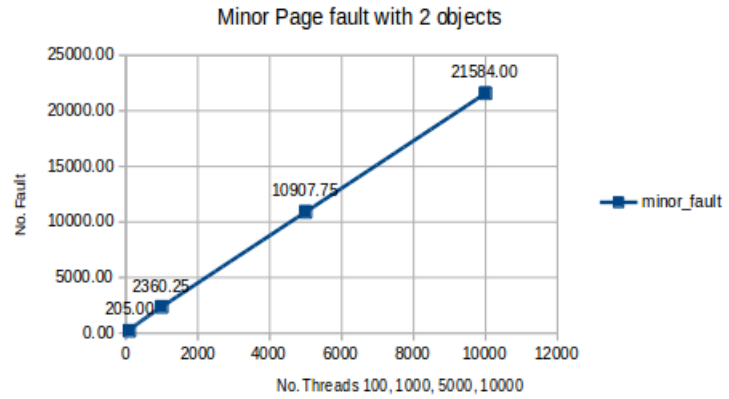
- 

## 11 Benchmark

A **major page** fault occurs when disk (I/O) access required from an application. When an application loaded into the memory, and an instruction need to be execute, the Linux kernel will search in the physical memory and CPU cache. If data/instruction doesn't exist in the searched memory, the Linux issues a major page fault.

If the application uses too much memory that cannot be handled by the main memory, that can cause major page faults which will slow down the application.

The following diagram shows the Minor Page faults during thread creations. There was no Major Page fault in the test.



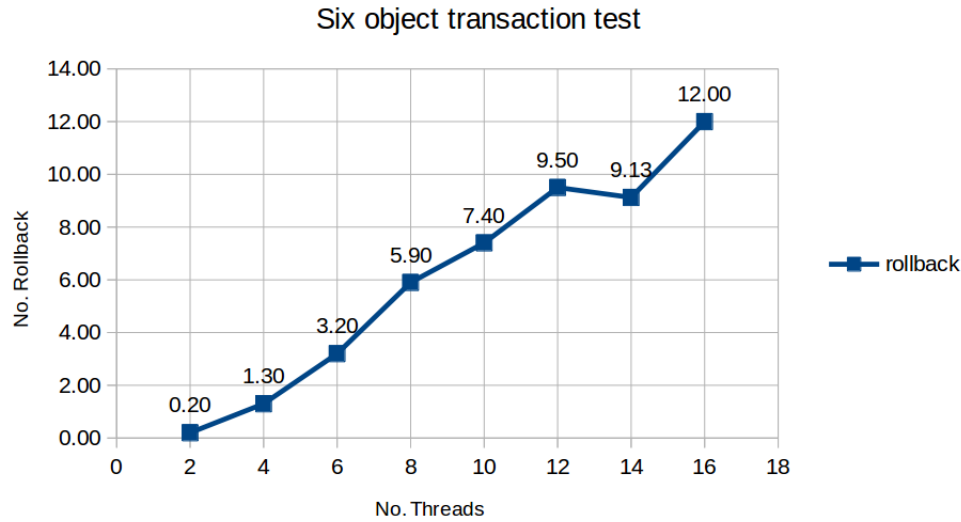
*Minor Page fault 10 to 10000 threads.*

## 11.2 Transactions

At this part in the benchmarking, I'll use the same objects between the threads, to force the library to restart the transactions as much as possible. First, I'll use low number of threads to compare the library with other Software Transactional Memory solutions, from the past decades. All the test will be done with the average calculation in a way to executing the same transactions more times and the average will be used in the test output.

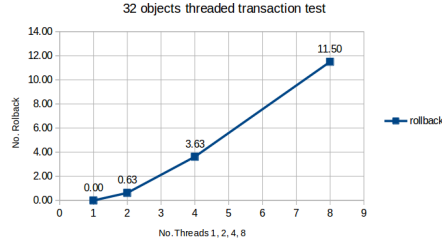
### 11.2.1 Low number of threads transaction test

In the past the Software Transactional Memory solution used up to 16 threads to test the performance of the library. In this test I'll use the same six objects in all of the transactions to cause rollback as much as possible.

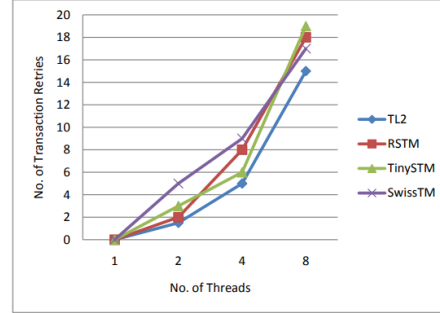


*Low number of the reads 2 - 16.*

To comparing my Software Transactional Memory library with other similar solutions need to have some information how the test was carries out. The best resource to compare the number of transaction roll-backs between the libraries found in a Master Thesis[1] from 2010. According to the Thesis they used Bayes algorithm, with **32 variable**[1] or memory spaces in the test. My library only managing the memory spaces, so the algorithm might irrelevant in my test, because every time when we want to make change on the memory space, we need communicate between the algorithm and the library.



(a) My library

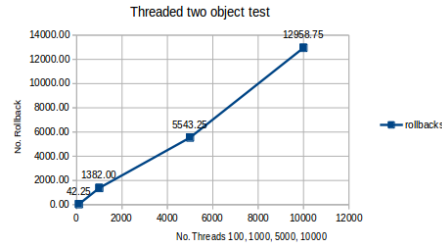


(b) Other solutions

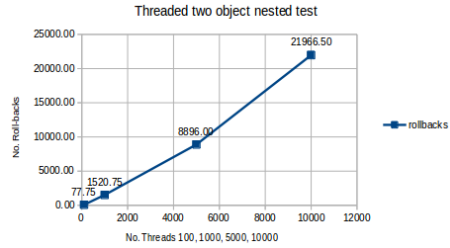
*Compare with other libraries (32 objects) from Master Thesis[1]*

### 11.2.2 Two objects threaded transaction test

The following diagrams shows the number of roll-backs with two objects accessed by various number of threads in simple transactions and nested transactions.



(a) Simple transaction



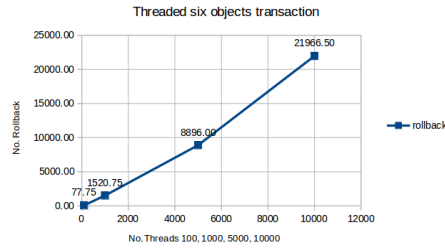
(b) Nested Transaction

*Number of rollbacks*

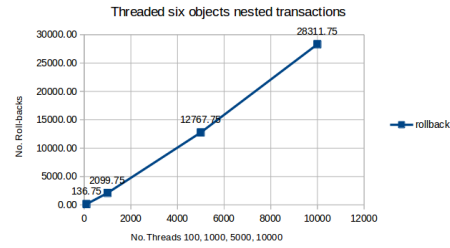
As we can see between the two diagram, the nested transactions are more expensive than the simple transaction. While the transactions are starting a new inner transaction, and the same time the other transactions are commits the changes with the same objects, it causing lots of starvation, and required restart the transactions more the once, as we increasing the thread number.

### 11.2.3 Six objects threaded transaction test

The following diagrams shows the number of roll-backs with six objects accessed by various number of threads in simple transactions and nested transactions.



(a) Simple transaction



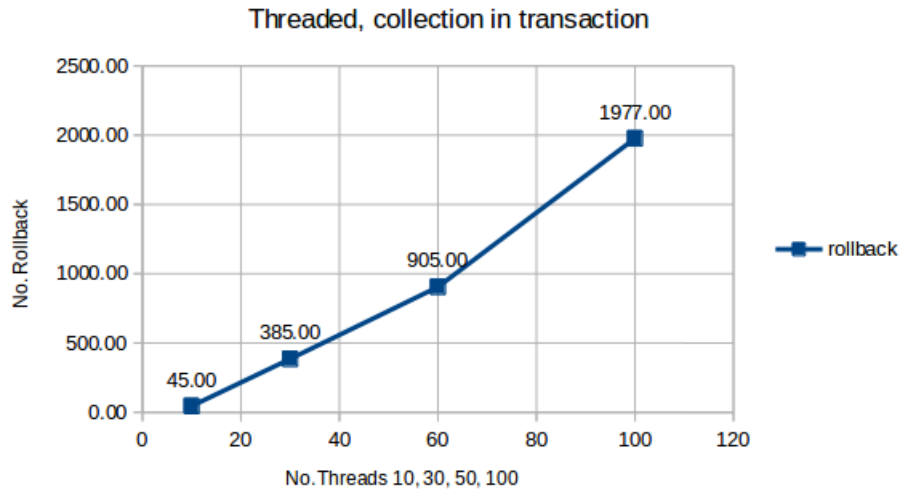
(b) Nested Transaction

*Number of rollbacks*

The transaction having higher number of same objects (six in this case) causing similar starvation like the nested transactions with two objects. Even nesting these transactions causing high number of roll-back.

### 11.2.4 Collection of objects in transaction

When transaction have a collection of objects, the execution are exactly the same like the other transactions with single object, but the complexity can cause very long execution time, specially if more transaction accessing the same collection. For this reason I'll test only the transaction without nesting with high number of objects and low number of threads.



*Transactions with 600 object in the collection.*

## 12 Conclusion

This project was out of my comfort zone, with the concurrent programming and the Transactional environment. During my internship I have met with the transactions in Java framework, and now I had chance to learn how to implement it in C++ programming language. This project is really different from another project or from previous experience, where we need to create communication between programming components, create GUI interface to interact with the system and store values in the database.

This project required to produce a library(s), that can be used by other C++ applications. The most time consuming part of the project was to find out the best base class and create the client application to use the library, with the polymorphic objects/classes, that is after all not part of the final product, but without that development and implementation the library development is impossible to carry out.

I think it is not an easy subject, since we need to work with multi threaded environment, without any visual feedback. The only visual feedback is the console could be, where is the threads can printing out the feedback/string in any order, since we are not using lock in the code segment. If we using lock in the code segment to control the threads visual feedback, then the code is behaving differently

then it suppose to be.

I have learned a lot from this project, how to control multiple threads without locking, and implement transactional environment without using any framework or someone else library. However, Software transactional Memory obsolete over the past few years, I think it is a very good experience/knowledge to know how transactions are implemented in the applications or libraries.

## **13 Acknowledgements**

I would like to express my very great appreciation to Dr Joseph Kehoe for his valuable and constructive suggestions during the planning and development of this research work. I would also like to thank the other lecturers for the help throughout the year.

## 14 Bibliography

### References

- [1] N. A. Gulfam Abbas. Performance Tradeoffs in Software Transactional Memorys . <https://pdfs.semanticscholar.org/5ca0/50378c9e45bb0215a558ebd18d9fad7cc8bc.pdf>. 2010 (Accessed : 12/04/2018).
- [2] S. Media. CppUnit - C++ port of JUnit. <https://sourceforge.net/projects/cppunit/>. 2018 (Accessed : 31/03/2018).
- [3] Sourceforge. CppUnit Cookbook. [http://cppunit.sourceforge.net/doc/cvs/cppunit\\_cookbook.html](http://cppunit.sourceforge.net/doc/cvs/cppunit_cookbook.html). N/A (Accessed : 31/03/2018).