
Software Transactional Memory

C++14 STM

Final Report

Zoltan Fuzesi

April 1, 2018

Student ID: C00197361

Supervisor: Joseph Kehoe

Institute of Technology Carlow

Software Engineering



Contents

1	Introduction	1
2	About the project	1
3	Problems encountered and resolved	2
3.1	Storing unknown objects	2
3.2	Template class and function	3
3.3	RAW pointers with polymorphic objects	3
3.4	Segmentation fault, deadlock, live-lock	3
3.5	Synchronizing commit function without locking code segment . .	4
3.6	Resolved problems	4
3.6.1	Storing unknown objects	4
3.6.2	Template class and function	5
3.6.3	RAW pointers with polymorphic objects	5
3.6.4	Segmentation fault	5
4	Project description/achievement	6
4.1	Achieved	6
4.2	Not achieved	7
5	What learned	8
5.1	Technical learning	8
5.2	Personal learning	9
6	What I would do differently if starting it again	9
7	Report on document update	9
7.1	Additional research	10
8	Module description	10
8.0.1	Transaction Manager	10
8.0.2	Transaction object	11
8.0.3	Base object	11
8.1	Public API functions for transactions	11
8.1.1	Register	11
8.1.2	Load	11
8.1.3	Store	11

8.1.4	Commit	12
8.2	Private API and Transaction Manager functions	12
8.2.1	Transaction Manager	12
8.2.2	Transaction object	12
9	Data structure	13
9.1	Transaction Manager	13
9.2	Transaction object (TX)	14
10	Testing	14
11	Conclusion	15
12	Bibliography	16

1 Introduction

This document summarizes all of the related works done, problems, errors and learning outcomes while developing Software Transactional Memory throughout the project lifetime. It will describing the library, the data structures and the related testing carried out.

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 the 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. Because the Shared or Static library should take any type of object, and the library doesn't know anything about the client application objects, it uses polymorphic class hierarchy to achieve the required functionalities.

It's required extra implementation to the client application to inherit from the STM 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 support all the required fields and functions to the library, to carry out all the STM (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 implementation. 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. We can nesting our transactions to any level as we want, the library can handle it as well.

To use my STM library with any c++ application, we need to follow some rules to achieve the inheritance. The tutorial of this implementation and from the API functionality can be found on the project website.

3 Problems encountered and resolved

The most significant problem caused by the theory, such as use my library with any c++ client application, as it store and use the instances of any client application object within the transactions without modifying 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 program, they can use and reference all the client application classes in the library/functions.

The second biggest problem was to synchronize objects state with multiple threads, without using mutex lock in 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.

These theoretical questions caused the following problems:

1. Storing unknown objects.
2. Template class and function.
3. RAW pointers with polymorphic objects.
4. Segmentation fault.
5. Synchronizing commit function 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. If I store primitive type rather than object,

the boost library could be a good 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 function

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. So, if all the code relying on template class or function, the static or shared library can not be build, because only the *.cpp files are build into the library file. 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 the documentation provided by the internet 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. In the project I used the **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 method in the base class to copying the objects within the transactions, and it is provided by the polymorphic class behaviour. This function cause 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 impassible without the target class reference.

3.4 Segmentation fault, deadlock, live-lock

This are the more frequent problems when working with multi threaded programming environment. Since the project not using mutex locks to lock code sections, apart from registering the the objects and transactions, it is a big 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, that depends the threads sequence execution in the background and the number of threads use 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 of threads. As a faster debugging solution I start printing out messages to the console from the different code segments, to narrowing down to the possible line of code where is the error raised by the thread.

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 in 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 transaction. So, if we have five transactions, they are not using any object from another transaction, then all the transactions can make changes on the objects at the same time.

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 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. Because, c++ programming language supporting multiple inheritance, then the client application classes are not restricted to achieve inheritance, polymorphic Object Oriented behaviour implementation from any other classes in the client application, but it give me lots of

freedom to control unknown objects in the library.

3.6.2 Template class and function

Because, the library store the unknown objects as a base class reference, there is no more required to use any template class or function in the project. All the code has moved into the cpp 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 pointer, that give me a freedom to use the polymorphic classes in the target class to copying the object from the external library. In this way the smart pointer get destroyed when goes out of the scope, and the memory leak was 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, the possible permutations and combinations of the concurrent execution of transactions/threads are really big. If we running the same code twenty times might the code will faced with the segmentation fault if not well designed. Every time, when my code was changed at least I did execute the same combination hundred time to check the percentage of the code failure. During the development I mixed up the transactions as complicate as possible to causing segmentation fault. In this phase of the project life time, I have to say there was no segmentation fault in the last two iteration. But, because the possible complexity 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 are not visible to other (successful) transactions.

Mandatory:

1. Full STM implementation.
2. Full documentation of library with Doxygen.
3. Tutorial of 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 I used a roll-back counter in the library, to display the conflicts/roll-backs happening between transactions. The memory of course is shared between transaction, to prove it we can display the object after all transactions finish with the computation, and if all the transactions accessed the object the final value should be increased and consistent. Of course, to prove the STM not corrupting the objects within the transaction we can execute the same code repeatedly, and the object state after the transactions finished should be the same all the time.

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

As an extra the nesting is added to the library, where we can nesting the transactions. It means, if the transaction dependent on other logical inner transaction, the library can handle the nesting to any level.

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) supporting the `std::mutex` that was used in the library, and the main c base language developer tool is the Visual Studio on Windows platform. The project website has tutorials to demonstrate the usage of the library.

Exceptional: Benchmarks, is not implemented yet, but hopefully it will be ready to the end of the project lifetime. To compare with other similar STM solution is not the easiest task, since my library using objects, and the most STM solution with available code using primitive type pointer based implementation.

4.2 Not achieved

The Software Transactional Memory is a very research heavy area, and most of the papers I found during my research are from Master and from PHD level. It is not an easy subject, and I could try using different collection like binary tree, Red-Black tree, linked list and many other structure to compare the performance between 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, 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 the same code.

5.2 Personal learning

I have learned from my project:

1. Time management.
2. Setting targets for iterations.
3. Why good research is very important.
4. Design efficient code in productive way.

6 What I would do differently if starting it again

If I've to do this project again, I would spend more time on pointers and not relying on the opinion by other programmers and on my personal preference. If I use smart pointers from the begging, I should not redesign my code from RAW 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 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 using inheritance. Otherwise, the project was out of my comfort zone, and everything about STM concepts was completely new to me.

7 Report on document update

Between the design document and the final implementation there is 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 starting the transaction. In my library, 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 parameters in the API functions. Because, it makes the library usage very difficult, I tried to remove all the extra requirements in the API function parameters, and keep them as simple as possible. The object and function names are changed as well, changed them shorter and more meaningful.

3. 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.

7.1 Additional research

A small additional research required to the nested transactions implementation.

Unit testing for C++ programming language.[2]

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.

8.0.1 Transaction Manager

The Transaction Manager responsible to manage all the transaction associated with the connected client application. When a Transaction Manager requested from a Shared library, then a singleton object provided to the application to manage the transactional environment. The Transaction Manager is the single object 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. Every single public API function, which receiving a reference of an object, using could throw a runtime error if the client application want to use or store an illegal or null pointer.

8.0.2 Transaction object

The returned transaction objects can access the library API functions to manage the 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.

8.0.3 Base object

The base object provides all the required fields and functions to the inherited classes to work with the library.

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 into the global shared map, a copy of the object/working object get registered within the transaction as well. This object will be used to make temporary changes associated with the target object, memory space.

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.

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.

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 the from the other transactions. If any of the transaction objects are lock 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 to the object, it will lock it and compare the version number associated with working copy and the original object. If any of the object within the transaction 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 finis, it will call the private function it the library to unlock all the global object associated with the transaction.

If the transaction committed all the changes on the objects, then return true to the client application to indicate the transaction successful finished with the changes. 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.

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 increasing the nesting counter. If the thread not registered, it will call the register private function to register the thread in the manager. The register function including an another private function call, that returns a new map to store the thread within the 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 want 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 an object in the collection without need to iterate through. However, the binary tree is a 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:

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

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

```
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.


```
: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.

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

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

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

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

```
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

```
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.*[1][2]. I have described few testing methods in the Design Manual document to use 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.
3. Single-threaded multiple object test.
4. Multi-threaded multiple Objects test.

5. Testing the API store, load and register functions.
6. Complex transaction with multiple threads and object collections.

The documentation with the test code from the CppUnit test, can be found on the project website under CppUnit test.

11 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 with 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 GIU 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 and create the client application to use the library, that is not part the final product, but without that implementation the library development is impossible.

I have learned from this project, how to implement transactional environment without using any framework. 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.

12 Bibliography

References

- [1] S. Media. CppUnit - C++ port of JUnit. <https://sourceforge.net/projects/cppunit/>. 2018 (Accessed : 31/03/2018).
- [2] Sourceforge. CppUnit Cookbook. http://cppunit.sourceforge.net/doc/cvs/cppunit_cookbook.html. N/A (Accessed : 31/03/2018).