Transactifying a Computer Game

Exploring the use of Software Transactional Memory with a Multiplayer Computer Game

MESFIN ZEWDE

Master of Science Thesis
Stockholm, Sweden 2009

TRITA-ICT-EX-2009:200
Transactifying a Computer Game

Exploring the use of Software Transactional Memory with a Multiplayer Computer Game

Mesfin Zewde
September, 2009

Master’s Thesis in Computer Science

Home University Examiner: Assoc. Prof. Vladimir Vlassov
Department of Software and Computer Systems (SCS),
Royal Institute of Technology (KTH), Sweden

Host University Supervising Professor: Prof. Rachid Guerraoui
Host University Supervisor and Project Coordinator: Mr. Aleksandar Dragojevic
Distributed Programming Laboratory (LPD),
School of Computer and Communication Sciences (I&C),
École Polytechnique Fédérale de Lausanne. (EPFL), Switzerland
Abstract

One of the latest concurrent programming technologies is Software Transactional Memory (STM). This degree project studied the use of STM by taking the large open-source computer game Globulation2 and modifying it from a non-concurrent version to several concurrent versions – a lock-based version and a STM version with finer granularity, as well as an additional STM version with coarser granularity. The different game versions were to be compiled with a STM compiler, which resulted in an evaluation of existing STM compilers. The first choice LLVM and Tanger turned out to be unable to compile the game versions because Tanger lacked an irrevocable mode and support for exceptions inside transactions as well as basic C++ support needed by the game, including memory operators new and delete and the C++ STL. Together with an instability that was detected while using the LLVM compiler and Tanger, LLVM and Tanger were finally considered too unstable to use as the STM compiler for this project. Instead the Intel C++ STM compiler was chosen as the STM compiler for the project, and could successfully be used to compile the different game versions.

Performance data from the game versions was gathered by timing different parts of the code, including the simulation part of the game’s main loop where most of the game computation is done. Using the collected data a comparison of the game versions’ performance and how well they scaled when increasing the number of threads was made. The results showed that the STM versions of the game performed worse than the lock-based version and did not scale well when the number of threads was increased. The coarser-grained STM version did however have better performance and scaled better than the finer-grained STM version. Switches to irrevocable mode, transaction overhead and to some extent transaction retries were identified as possible reasons for the bad performance and scaling of the STM version. An attempt was also made to use an experimental version of the Intel C++ STM compiler that integrated the SwissTM STM library, but it was not ready to use, and SwissTM could not be used or evaluated in this project.
Sammanfattning


Preface

This degree project has been carried out for the Distributed Programming Laboratory at the École Polytechnique Fédérale de Lausanne (EPFL) in Switzerland as part of their ongoing research about Software Transactional Memory. The practical work of the project was mainly carried out at the EPFL in Switzerland.

This report is a degree project report in Software Engineering done for the Department of Software and Computer Systems (SCS) of the Royal Institute of Technology (KTH) in Sweden. Examiner of this thesis is Associate Professor Vladimir Vlassov at KTH, Supervisor and Project Coordinator is Mr. Aleksandar Dragojevic at EPFL and Supervising Professor at EPFL is Professor Rachid Guerraoui.
Acknowledgements

I would like to start by thanking my Project Coordinator and Supervisor Aleksandar Dragojevic (Distributed Programming Laboratory) for his help and support during the course of this project and for always being available for guidance. Thanks to System Manager Fabien Salvi (Distributed Programming Laboratory) for the support and help with setting up the project environment. Thank you also to my examiner Associate Professor Vladimir Vlassov (KTH) for guiding me through the academic process of this thesis.

Last but not least, I would like to thank my family and friends for your support and encouragement. Pedro Lopez, for all those early mornings, lunch brakes and late nights, thank you for your company and consideration. A special thank you goes to my father Ayele Zewde, who constantly and unconditionally encouraged me and gave me guidance during this project. Your support and wisdom helped me reach my goals.
# Table of Contents

ABSTRACT ............................................................................................................................ I  
SAMMANFATTNING ............................................................................................................. II  
PREFACE ............................................................................................................................... III  
ACKNOWLEDGEMENTS ........................................................................................................ IV  
TABLE OF CONTENTS ......................................................................................................... V  
LIST OF FIGURES ................................................................................................................ VIII  
LIST OF ACRONYMS AND ABBREVIATIONS ..................................................................... IX  

1 INTRODUCTION .................................................................................................................. 1  
1.1 TRANSACTIONAL MEMORY ............................................................................................. 1  
1.2 OBJECTIVES .................................................................................................................. 2  
1.3 OVERVIEW OF TASKS ................................................................................................... 3  
1.4 THESIS OUTLINE ......................................................................................................... 4  

2 BACKGROUND .................................................................................................................. 5  
2.1 THE NOTION OF A TM TRANSACTION .......................................................................... 5  
2.1.1 Memory accesses in transactions .............................................................................. 5  
2.1.2 Transaction modifier instructions ........................................................................... 6  
2.1.3 A simple transaction example ................................................................................ 6  
2.2 WELL WRITTEN PROGRAMS WITH TM ........................................................................ 7  
2.2.1 Deadlocks ................................................................................................................ 7  
2.2.2 Priority Inversion ..................................................................................................... 7  
2.2.3 Convoying ............................................................................................................... 8  
2.3 TM TRANSACTIONS VS DATABASE TRANSACTIONS .................................................. 8  
2.4 TM TRANSACTIONS VS MONITORS ............................................................................ 9  
2.5 DESIGNING A STM MECHANISM ................................................................................ 9  
2.5.1 Lock-based vs Obstruction-free ............................................................................... 10  
2.5.2 Word-based vs Object-based................................................................................. 10  
2.5.3 Optimistic vs Pessimistic conflict detection ........................................................... 10  
2.5.4 Timid vs Greedy contention manager .................................................................... 10  
2.5.5 Invisible vs Visible readers .................................................................................... 11  
2.5.6 Design choices of SwissTM ..................................................................................... 11  

3 RELATED WORK ................................................................................................................. 12  
3.1 SWISSTM ..................................................................................................................... 12  
3.2 LLVM ........................................................................................................................... 12  
3.3 TANGER ......................................................................................................................... 12  
3.4 ATOMIC QUAKE .......................................................................................................... 13  
3.5 STM BENCHMARKS ....................................................................................................... 13  
3.5.1 STM Bench7 ............................................................................................................ 13  
3.5.2 STAMP .................................................................................................................... 14  
3.5.3 Lee-TM ..................................................................................................................... 14  
3.5.4 Red-black tree ....................................................................................................... 14  
3.5.5 Conclusion .............................................................................................................. 14  

4 PROBLEM FORMULATION .............................................................................................. 15  
4.1 THE BENCHMARK COMPUTER GAME ......................................................................... 15  
4.2 ANALYZING THE GAME CODE .................................................................................... 15  
4.3 WRITING A CONCURRENT MUTEX LOCK GAME VERSION ........................................ 15  
4.4 Compiling the Game ...................................................................................................... 16  
4.5 WRITING A CONCURRENT STM GAME VERSION ....................................................... 16
8 SUMMARY AND CONCLUSION ............................................................................................................ 75
  8.1 SUMMARY OF RESULTS .............................................................................................................. 75
  8.2 LESSONS LEARNED .................................................................................................................... 76
  8.3 FUTURE WORK ............................................................................................................................. 78
9 REFERENCES ................................................................................................................................... 80
List of Figures

Figure 1: Example of a transaction ........................................................................................................ 6
Figure 2: Two opposing players in battle ................................................................................................. 20
Figure 3: Advanced base with units and buildings ................................................................................ 20
Figure 4: Main loop of Globulation2 ...................................................................................................... 23
Figure 5: Call stack overview of the main loop. ....................................................................................... 25
Figure 6: Parallel sections of the simulation part .................................................................................... 28
Figure 7: Thread creation in the Map class. ............................................................................................... 29
Figure 8: Initiation of variables in the ThreadData structure ..................................................................... 30
Figure 9: Critical section with macros for the mutex lock ......................................................................... 30
Figure 10: Pseudo code example of problems with partitioning tasks to threads. ................................... 31
Figure 11: Pseudo code example of a solution to the problem with partitioning tasks to threads. ............... 32
Figure 12: Barrier example of thread synchronization. ............................................................................ 33
Figure 13: Tanger transaction example. .................................................................................................... 35
Figure 14: Pure wrapper functions for puts and printf. ............................................................................. 39
Figure 15: Problem with STL containers and Tanger .............................................................................. 41
Figure 16: List with custom allocator and List.erase() inside a transaction. ........................................... 42
Figure 17: Strange behavior with LLVM and Tanger ............................................................................... 43
Figure 18: Intel STM transaction example................................................................................................ 44
Figure 19: Example of how to annotate a function as tm_callable in Linux syntax .............................. 45
Figure 20: Map topologies of test cases.................................................................................................... 49
Figure 21: Strange statistics with the Intel statistics mechanism ................................................................. 50
Figure 22: Partial output of a statistics file with time measurements of the simulation part. .................... 52
Figure 23: Analysis file created by the StatsAnalyzer program. ................................................................. 52
Figure 24: Total average of all four test cases, with 10 repetitions of each test case. ................................. 57
Figure 25: Average mean time for the four test cases, with 10 repetitions of each test case. ...................... 57
Figure 26: Total average of all four test cases, comparing the Intel C++ STM compiler with g++ for the mutex version .............................................................................................................. 60
Figure 27: Total average of all four test cases on a single core machine .................................................... 61
Figure 28: Total average of all four test cases with up to 8 threads on 4-core machine ............................... 61
Figure 29: Analysis file with evidence of extreme values. .......................................................................... 62
Figure 30: Comparison between total average mean and median times. ................................................... 63
Figure 31: Fine-grained timing of the four parallel thread sections in test case G2-1min. ......................... 64
Figure 32: Total average mean time in simulation part of test case G2-1min with fine-grained timing turned on. ........................................................................................................................................... 65
Figure 33: Fine-grained timing of the four parallel thread sections in test case strange2 ......................... 66
Figure 34: Total average mean time in simulation part of test case strange2 with fine-grained timing turned on. ........................................................................................................................................... 67
Figure 35: Counting transaction retries .................................................................................................... 67
Figure 36: The average mean time for all test cases where long time measurements have been filtered out with different thresholds. ................................................................................................. 70
Figure 37: Comparison between 0.0 and 0.75 thresholds for filtering measured times in the simulation part .......................................................... 71
Figure 38: Comparison between coarse-grained locking and fine-grained locking. ................................. 71
Figure 39: Simple program that could not be compiled with SwissSTM and the Intel compiler. ............... 73
## List of Acronyms and Abbreviations

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABI</td>
<td>Application Binary Interface</td>
</tr>
<tr>
<td>AI</td>
<td>Artificial Intelligence</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>HTM</td>
<td>Hardware Transactional Memory</td>
</tr>
<tr>
<td>HyTM</td>
<td>Hybrid Transactional Memory</td>
</tr>
<tr>
<td>IR</td>
<td>Intermediate Representation</td>
</tr>
<tr>
<td>JIT</td>
<td>Just-In-Time compiler</td>
</tr>
<tr>
<td>LLVM</td>
<td>Low Level Virtual Machine</td>
</tr>
<tr>
<td>LT</td>
<td>Load-transactional</td>
</tr>
<tr>
<td>LTX</td>
<td>Load-transactional-exclusive</td>
</tr>
<tr>
<td>Mutex</td>
<td>Mutual exclusion</td>
</tr>
<tr>
<td>RTS</td>
<td>Real Time Strategy</td>
</tr>
<tr>
<td>ST</td>
<td>Store-transactional</td>
</tr>
<tr>
<td>STL</td>
<td>Standard Template Library</td>
</tr>
<tr>
<td>STM</td>
<td>Software Transactional Memory</td>
</tr>
<tr>
<td>TM</td>
<td>Transactional Memory</td>
</tr>
<tr>
<td>TX</td>
<td>Transaction</td>
</tr>
<tr>
<td>YOG</td>
<td>Ysagoon Online Game (server)</td>
</tr>
</tbody>
</table>
1 Introduction

The recent trend in computer hardware architecture has been to shift the focus from improving computer performance\(^1\) by increasing the clock frequency of single processors to improving computer performance by designing multi-core computers, that is, computers with multi-core processors. These are processor architectures with multiple independent processors (or cores) on a single chip, where the processors are connected through a shared memory [1].

Increasing the clock frequency of single processors has proved to become more and more difficult due to power and cooling limitations [2]. The more advanced the processors become the more transistors they have and the more power they require, thus generating more heat. But limitations in the cooling capacities of current cooling technologies prevent processor manufactures from increasing the power requirements of processors at the same rate as in the past, thus stalling the development of conventional single core processors. In multi-core processors, however, the multiple processors allow more instructions to be executed per second than with a single processor computer since instructions can be executed in parallel, but without increasing the clock frequency. The parallel computation capabilities of multi-core computers thus provide the ability to improve performance of certain well written parallel programs (see Section 2.2), and the shift in computer hardware architecture from sequential computers to parallel computers has since sparked the interest and research of different parallel programming techniques.

1.1 Transactional Memory

When using the full parallel computation capabilities of a multi-core computer in a concurrent program there needs to be some synchronization mechanism between threads of execution that concurrently access shared memory, to ensure that concurrent computations do not interfere [3]. Transactional Memory (TM) is a parallel programming mechanism for synchronizing concurrent accesses to shared memory data in multi-threaded programs [4], and as such TM highly benefits being used with multi-core computers compared to single processor computers. TM is also appealing from a programmer's perspective since it simplifies the design of concurrent multi-threaded programs by providing the illusion that shared objects are protected by some global lock, while maintaining high performance but avoiding many of the pitfalls and difficulties related to designing concurrent multi-threaded programs based on mutual exclusion (mutex) locks [5; 6; 7]. In such programs when a thread needs access to a shared object or a critical section\(^2\), it first has to acquire the mutex lock for that shared object or region of code. It can only do so if no other thread holds the lock at that moment, in which case it needs to try at a later point in time and see if the lock has been released. When it is able to acquire the lock all other threads are excluded from accessing the code protected by the lock. Threads thus mutually exclude one another from accessing code protected by the lock when they acquire the lock.

When writing mutex lock-based concurrent programs the programmer is required to think about how different, sometimes seemingly unrelated parts of code overlap and interact.

---

1 Computer performance is a general term meaning the amount of useful work accomplished by a computer system compared to the time and resources used. Some examples of metrics used to measure computer performance are response time, availability, speedup, scalability and throughput.

2 A piece of code that accesses shared data, but must not be concurrently executed by more than one thread.
Typically, the programmer must use a locking policy that avoids deadlocks (see Section 2.2.1) and other common problems experienced during contention for a shared resource. Ensuring that a lock-based concurrent program does not exhibit these problems can be quite a complicated and difficult task, especially when the program is made up of very intricate concurrent interactions. Therefore, from a programmer's point of view, TM seems to be a very attractive and promising mechanism because it takes care of ensuring that these problems do not occur by providing e.g. freedom from deadlocks and priority inversion [8]. TM thus greatly simplifies programming concurrent programs by relieving the programmer from the responsibility and headache of thinking about how to avoid many of these problems [7].

The Transactional Memory mechanism can be implemented either as a pure Hardware Transactional Memory (HTM) scheme, as first proposed in [4], as pure Software Transactional Memory (STM) schemes like those in e.g. [5; 6; 9; 10] or as a Hybrid Transactional Memory (HyTM) scheme combining both hardware and software as in e.g. [11]. STM provides the semantics for Transactional Memory in a software runtime library, as opposed to HTM which provides the semantics in hardware chips, and today there exist many different suggestions for STM mechanisms. No TM mechanisms have yet had any wide commercial success, in part because TM is still a very active research area where many different proposals and strategies exist and new ideas and improvements are still being proposed, and also because the research community has not yet reached a consensus of the characteristics of the ultimate Transactional Memory scheme. Another reason is that so far TM has not seemed to be able to outperform mutex locks in commercial applications, as was shown with one STM scheme in [12].

1.2 Objectives

The main objective of the thesis is to modify an existing sequential open source computer game to two new multi-threaded versions. The first, a parallel version using traditional mutual exclusion locks on shared memory accessed by threads. The second will be an attempt to write a parallel version that uses STM on shared memory accessed by threads. Then the two parallel versions and the original sequential version are evaluated and compared performance wise and how they scale when increasing the number of threads on a multi-core computer, to find out how the STM version performs in comparison to the other versions.

Another goal of this thesis is to investigate the possibilities of using SwissTM as the underlying STM mechanism in the parallel STM version of the game. SwissTM is a new and promising STM proposal that has been shown to outperform other STM systems on some commonly used STM benchmarks [5]. In this thesis a computer game will be used as benchmark to evaluate the performance because it is a real world application that might differ in complexity and diversity from previous experiments with common STM benchmarks.

Large applications with complex data-structures, such as computer games and business software, have been identified as applications that can greatly benefit from the new wave of multi-core computers [13]. Computer and video games have for a long time been in the forefront of pushing computer hardware technology to its limits. For decades each generation of game consoles has come with more and more powerful hardware to increase gaming performance, and it is not uncommon that newly released computer games have such high requirements that available computers on the market cannot run them smoothly on the highest settings. Therefore it seems fitting that to investigate and evaluate the performance and scalability of STM, a large real world application such as a computer game is used as a benchmark.
To make use of a STM library, a compiler that recognizes STM calls must be used to compile the program, unless one manually instruments the STM specific code which can be quite a tedious and time consuming task. Another major aspect of this thesis is therefore to find or configure a compiler that is able to compile both the computer game and calls to a STM library. There are a couple of potential compilers, but a compiler framework that is explored in this thesis and that has been shown to be efficient in comparison with other compilers like e.g. gcc, is the Low Level Virtual Machine (LLVM) compiler [14], which with Tanger is capable of compiling programs making use of STM calls. Tanger is a module (or pass) for the LLVM compiler that statically replaces regular load and store instructions inside transactions with STM specific function calls and adds STM calls for the start and end of transactions [15]. These STM calls are implemented by the underlying STM library TinySTM, but could be changed to for example SwissTM if the Tanger module is modified. Other interesting compiler options explored in this thesis are the Intel C++ STM compiler and the gcc-tm compiler. The Intel C++ STM compiler and its STM implementation have successfully been used in a similar study with the Quake computer game, but the STM version of Quake showed reduced performance and poor thread scalability compared to an implementation with traditional mutual exclusion locks [12].

The best compiler options are compilers which give access to the full source code and can be changed if needed, and with which various options easily can be experimented with. At the time of writing the only compiler providing the full source code is LLVM and the Tanger module, but the Application Binary Interface (ABI) for the Intel C++ STM compiler is freely available making it possible to change the underlying STM in theory. However, it is not a goal of this degree project to write a SwissTM compatible compiler, to rewrite an existing compiler or to write a compiler module that translates STM calls for a STM compiler so that SwissTM can be used with the compiler. This is a different project being simultaneously worked on at the Distributed Programming Laboratory at the EPFL, but existing solutions will be tried in an attempt to use SwissTM as the STM library in the parallel STM version of the game.

This degree project is a novel approach for studying the possibilities of using the SwissTM library with a complex real world application as a benchmark. In a combination of state-of-the-art tools for implementing STM, it is investigated whether current technologies allow the use of STM to simplify concurrent programs, but without affecting performance negatively.

1.3 Overview of tasks

- Find and choose an open source computer game to work with.
- Identify the simulation part of the main loop of the game.
- Write a concurrent version of the simulation part of the game using threads and a single global mutual exclusion lock.
- Configure and make sure the game can be compiled with an available STM compiler.
- Write another concurrent version with threads using STM calls to a STM library.

---

3 When STM instrumenting code, one replaces regular load and store instructions inside transactions with STM specific function calls and adds STM calls for the start and end of transactions. This can be done either manually where the programmer reviews the code and performs the necessary changes, or it can be done automatically using a compiler that performs the necessary changes.

4 The GNU Compiler Collection, a commonly used compiler for C and C++ programs. http://gcc.gnu.org/

5 This thesis project is part of an ongoing research about STM at the École Polytechnique Fédérale de Lausanne.
- Try using SwissTM as the underlying STM library by using some pre-existing solution for interoperability between SwissTM and the STM compiler.
- Formulate test cases that can be reproduced multiple times.
- Measure performance and scalability of the three game versions.
- Analyze the results and try to explain why they were obtained.
- Suggest ways to improve the performance and scalability of the STM version, and if practically possible implement these improvements.

1.4 Thesis Outline

The rest of this thesis is organized as follows. In chapter two the notion of a TM transaction is explained and in what way it differs from other similar constructs, and the different design choices of STMs are discussed. Chapter three presents related work that is relevant for this thesis. Chapter four describes in detail the requirements of the thesis as well as the steps involved in the thesis. Chapter five describes the full implementation process and what actually was done, including how the different game versions were implemented and what issues arose when doing so, as well as the test cases and tests that were created. In chapter six the way in which experiments were conducted is explained and the obtained results are presented and discussed. Chapter seven analyzes the obtained results and describes additional experiments that were performed to better understand the results as well as presents improvements to the obtained results. Finally, chapter eight summarizes the project and the obtained results. Conclusions from the experiences working with this project and what can be learned from the project, as well as recommendations for the future of STM and this project are given.
2 Background

In this chapter the notion of a transactional memory transaction is explained and an overview of how transactions are implemented is given. TM makes it easier for programmers to write well written parallel programs since some of the problems commonly associated with lock-based parallel programs are taken care of by the TM mechanism, and these problems are presented here. TM transactions are further compared to database transactions and monitors. Finally there are many design choices a designer of an STM has to make, and these are also presented here. The design choices of state-of-the-art STM implementations such as RSTM, Intel STM, TinySTM and TL2 are compared with SwissTM, the STM in focus in this thesis.

2.1 The notion of a TM transaction

So far TM transactions have been mentioned several times without a clear definition of the notion. The core concept of Transactional Memory is the transaction, which Larus and Rajwar describe as “a sequence of actions that appears indivisible and instantaneous to an outside observer [1]”. The first practical implementation of TM was a HTM mechanism proposed by Herlihy and Moss in 1993 [4], and they defined a transaction as a finite sequence of machine instructions, executed by a single process (or thread) that satisfies two properties: Serializability and Atomicity.

- **Serializability** - Transactions appear to be executed in order, i.e. it appears as if there is no mixture or interleaving between steps of different transactions, and all processors (or threads) see committed transactions (see Section 2.1.2) to have been executed in the same order.

- **Atomicity** - When a transaction has finished its changes to shared memory and wants to complete the transaction, it will either commit and instantly make the changes visible to other transactions, or it will abort and discard all of its changes.

2.1.1 Memory accesses in transactions

Apart from regular non-transactional load and store instructions Herlihy and Moss introduced three new primitive instructions used by transactions to access memory [4].

- **Load-transactional (LT)** – Reads the value of a shared memory location into a private register.

- **Load-transactional-exclusive (LTX)** – Does the same as LT, but additionally marks or “hints” that the memory location is likely to be updated by the transaction.

- **Store-transactional (ST)** – Speculatively writes a value from a private register to a shared memory location. The new value does not become visible to other processors (or threads) until the transaction successfully commits (see Section 2.1.2).

The set of shared memory locations read by LT instructions inside a transaction were called the transaction’s *read set*. Similarly the set of shared memory locations accessed by LTX and ST instructions inside a transaction were called the transaction’s *write set*. The unified set of the read and write set was called the transaction’s *data set*. 

- 5 -
2.1.2 Transaction modifier instructions

Further Herlihy and Moss described three different instructions to change the state of a transaction. To understand the mechanics of TM and STM in particular here is a brief overview of them. A more detailed description of how these are implemented can be found in [4].

- **Commit (COMMIT)** – Attempts to make the speculative changes of a transaction permanent. The COMMIT instruction can either **succeed** or **fail**, and returns an indicator of either success or failure depending on the outcome. A COMMIT succeeds only if no location in the transaction’s data set has been updated by other transactions, and if no location in the transaction’s write set has been read by other transactions. If it succeeds, the transaction’s changes to its write set become visible to other processes (and transactions). If it **fails**, all changes to the write set are discarded (the read set has not been modified and there is no changes to discard).

- **Abort (ABORT)** – Discards all updates to the transactions write set.

- **Validate (VALIDATE)** – Tests the current transaction status. A successful VALIDATE returns True, indicating that the current transaction has not aborted (although it may do so later). An unsuccessful VALIDATE returns False, indicating that the current transaction has aborted, and discards the transaction’s speculative updates.

In today’s STMs, a transaction will execute its instructions and then either commit or abort. If it commits, all of the transaction’s actions are applied atomically to shared memory, but if it aborts the effect of all of the transaction’s actions are rolled back and will never be visible to other transactions [5].

2.1.3 A simple transaction example

To mark a critical section of code to be executed in a transaction, typically the critical code section is separated from the rest of the code with some markers and a transaction keyword is used to identify the beginning of the transaction. Figure 1 shows an example where the critical code section is enclosed within curly brackets and the transaction keyword `atomic` is used to specify the beginning of the transaction.

```plaintext
atomic { //Start of the transaction
    shared_memory_counter++; //critical section of code
} //End of the transaction
```

Figure 1: Example of a transaction. The atomic keyword is used to specify the beginning of the transaction and curly brackets are used to enclose the critical section of code protected by the transaction. In this simple example a counter in shared memory is incremented, and makes up the critical section.

The exact implementation details of the steps a transaction goes through from the beginning of the transaction to the end of the transaction vary between different STM implementations. In e.g. SwissTM [5] the first thing a transaction does is to read a global `commit-counter`. Then it will read memory locations and attempt to write to memory locations (validation of the transaction’s read set can occur during reads and writes), in clever and
implementation specific ways for SwissTM. If conflicts are detected during validation the memory operations are rolled back. Finally after executing all memory operations the transaction can commit. The way in which a transaction commits usually differs depending on what memory operations it has performed, as with SwissTM, where a read-only transaction can commit immediately whereas a read-write and a write-only transaction re-validates their read set before committing. A successful re-validation will cause the transaction to finish the commit instruction, while an unsuccessful re-validation causes the transaction to roll back and restart the transaction. In either case, the transaction will have updated the global commit-counter before committing so that other transactions can determine whether they are working with inconsistent data or not.

2.2 Well written concurrent programs

Writing lock-based concurrent programs can be difficult and programmers need to make sure to avoid common problems associated with locks to ensure the correctness of their programs. For a program to fully make use of the parallel computation capabilities of multi-core or multi-processor computers, the programs need to be correct and well written. Multi-threaded programs need to use a locking policy that avoids the common problems associated with concurrent lock-based programs. Shared data must be protected with locks from concurrent accesses by threads to avoid inconsistent views of memory, and the acquisition and release of locks has to be done in a consistent manner so that special execution conditions do not change the program behavior in an unexpected manner, which is especially important when using multiple locks. If the locking policy is bad or poorly thought through when using more than a single thread, the program might perform poorly or the forward progress of the program might stop altogether. Instead of increasing performance by introducing threads to the program and computing in parallel, one could regretfully decrease the performance and thus have the opposite effect than what was wished for. However, in TM a programmer does not need to think about many of these problems, because many implementations of the TM mechanism avoid problems such as deadlocks, priority inversion and convoying all together [8], and writing a correct and well written program becomes much easier.

2.2.1 Deadlocks

Using locks one has to be careful to avoid a deadlock, a situation where threads wait for conditions that will never occur [16]. E.g. if there are two threads that hold one lock each but both need the lock held by the other thread to be released in order to continue executing, then there is a circular waiting between the threads. All forward progress of the threads stop because they are all waiting and neither thread can release the resource they have already obtained [17]. Deadlocks can occur if threads or processes attempt to lock the same set of objects in different orders [4]. A similar problem to a deadlock is livelock, where no progress of any task is made because the thread or process is kept busy handling an input overload or continuous aborts and retries [18; 19].

2.2.2 Priority Inversion

In priority inversion a process (or thread) with lower-priority is preempted while holding a lock needed by higher-priority processes, and therefore hinders the higher-priority process to perform all of its actions [4].
2.2.3 Convoying

Convoying occurs when a process (or thread) holding a lock is descheduled and other processes capable of running are unable to progress because they need the same resource, resulting in high contention for the resource which causes performance degradation. This can happen if e.g. the process exhausts its scheduling quantum, a page fault occurs or some other kind of interrupt occurs [4].

2.3 TM transactions vs Database transactions

TM transactions might seem conceptually similar to database transactions, but they are different since they are implemented differently and executed in different environments. Database transactions satisfy the ACID properties which stand for atomicity, consistency, isolation and durability. In [1], Larus and Rajwar describe these properties as:

- **Atomicity** – All actions of a transaction have to complete successfully, or it has to appear like none of these actions ever started executing. When a transaction completes successfully it is said to *commit*, and otherwise, when a transaction fails, it is said to *abort*. Larus and Rajwar like to make a clearer distinction and call this property *failure atomicity*, not to be confused with *atomic execution* which involves elements from the other ACID properties.

- **Consistency** – Transactions modify the state of the “world”. Database transactions modify the state of the database, while TM transactions modify the state of memory. The consistency property is application dependent (usually consisting of a collection of invariants) and says that a transaction should never leave the state of the world in an inconsistent state. Transactions start with the assumption that the state is consistent (invariants hold), and should also leave it in a consistent state after they modify the state, because succeeding transactions start executing from this modified state.

- **Isolation** – The isolation property says that each transaction should produce a correct result no matter what other transactions execute simultaneously.

- **Durability** – The durability property says that once a transaction commits, its results should be permanent and available to all other succeeding transactions.

The main difference between database transactions and TM transactions is that database transactions store data on disk while TM transactions store data in memory (e.g. RAM), hence the term transactional *memory*. Disk access in database transactions takes much longer time than memory access in TM transactions and databases can use this time much more efficiently for computation. TM transactions cannot perform much computation during memory access, which impacts the way TM mechanisms are designed compared with databases.

Another important difference is that in TM all transactions observe a consistent state and are not allowed to view an inconsistent state at all, they need to abort immediately. However, database transactions might observe an inconsistent state and keep executing, even though they eventually would abort.
Another difference is that TM transactions only need to adhere to the ACI properties. Durability is not important in TM because data in memory is short-lived, meaning that data in memory does not survive program termination and is not permanently stored on disk.

2.4 TM transactions vs Monitors

A somewhat higher level concept than mutual exclusion locks is the monitor used in e.g. Java to synchronize concurrent accesses to shared data. Similarly to mutex locks, Java’s monitor implementation only allow one thread at a time to access the critical section of code protected by the monitor lock, and the lock over such a synchronized region of code is automatically acquired and released [20]. However, monitors also take care of avoiding many deadlocking situations that can potentially arise when using mutex locks. For example, monitors avoid deadlocks due to failed unlocking, a situation where a thread that acquires a lock fails to release the lock and there are other threads wanting the lock that therefore blocks, thus halting the program execution. In Java’s monitor implementation a lock is released automatically after returning from the region of code supervised by the monitor even if the return is caused by an uncaught exception. The release of a lock cannot be forgotten by an unmindful programmer.

Using the monitor scheme a programmer has to be careful not to assume that a synchronized critical region executes atomically, even though all reads and writes to memory are synchronized. Neither monitors nor mutex locks offer atomicity over synchronized memory accesses, they simply synchronize critical sections with read and write operations to memory. In monitors only critical sections are synchronized, but in TM the memory is synchronized and the atomicity property does hold. This is one of the main differences between monitors and TM.

Another important difference is that with locks and monitors shared data cannot be accessed by concurrent threads as long as it is in use. This can be a waste of concurrency in certain cases and decrease performance. E.g. if a large data structure is locked by a thread and another threads wants to access a totally different portion of it, it has to wait for the lock to be released even though the threads could have proceeded concurrently without conflicting. In TM, shared memory is accessed by transactions without waiting for exclusive access, increasing concurrency, and conflicting accesses are usually not detected until the end of the transaction, the validation phase.

2.5 Designing a STM mechanism

Many different STM proposals exist today, for example SwissTM, TL2, TinySTM, RSTM and Intel STM [5; 6; 9; 10; 21], and even more proposals will likely emerge in the future. When designing STMs there are many design choices to be made. To better understand the difference between different STM implementations and what decisions designers of STMs face, this Section will present the main design choices that can be made when constructing STMs. Since part of this thesis is to investigate whether the STM library SwissTM can be used together with a STM compiler, this Section will also describe the main design choices of SwissTM and highlight the similarities and dissimilarities with previous well performing STMs.

---

6 In Java a critical section of code is protected with the synchronized keyword, and the synchronization is based on the monitor mechanism first proposed by C.A.R Hoare in 1974 [56].
2.5.1 **Lock-based vs Obstruction-free**

STMs can be lock-based (blocking), meaning that they internally use mutual exclusion for some stages of a transaction [22]. Most lock-based STMs use a two-phase locking protocol\(^7\) like the one proposed by Ennals in [7], in which the two-phase locking protocol is used to lock objects that the transactions update (writes to), giving a transaction exclusive permission to the objects. Examples of lock-based STMs include TinySTM [9], Intel STM [21] and TL2 [6]. STMs can also be obstruction-free (or nonblocking) STMs, such as RSTM [10], meaning that they do not use any locks or other blocking mechanisms in transactions and guarantee transaction progress if there are no other conflicting transactions.

2.5.2 **Word-based vs Object-based**

STMs also use some sort of *transactional logging* where data read by a transaction is recorded in order to validate the transaction’s read set before the transaction commits [5; 6]. Validating the read set is a process where the transaction makes sure that data read by the transaction is still valid (has not been updated by other transactions since the beginning of the transaction) so that it never views or operates on inconsistent data.

When data is read in the logging phase of a transaction, the two main classes of locking granularity are memory words and objects (cache-line level locking also exists in e.g. McRTSTM [23]). In word-based STMs like TL2, Intel STM and TinySTM the data recorded in the log are memory words, and in object-based STMs like RSTM the data recorded are language-level objects. Object-based STMs are usually used with object-oriented languages [10].

2.5.3 **Optimistic vs Pessimistic conflict detection**

The *contention manager* in STMs is responsible for deciding what to do when a transaction detects a conflict with a concurrent transaction. [24]. A conflict occurs when concurrent transactions access the same data, and at least one of the accesses updates the data (writes to it). If the detection of conflicts is *optimistic* (also called *lazy*), then the conflict detection is delayed until the transaction attempts to commit. If the detection of conflicts is *pessimistic* (also called *eager*), the conflict detection is performed when the data is accessed by the transaction. RSTM supports a contention manager with both eager and lazy conflict detection, Intel’s STM runtime library implements both lazy and eager concurrency control, TL2 only lazy conflict detection and TinySTM only eager conflict detection.

2.5.4 **Timid vs Greedy contention manager**

When a transaction induces a conflict with some other non-suspecting transaction, the contention manager has to decide whether to abort the non-suspecting transaction, abort the conflict inducing transaction or to delay the conflict inducing transaction and retry it at a later point in time. The *timid* contention manager always aborts the conflict inducing transaction [5], and favors thus short transactions. The *greedy* contention manager is more suitable for large transactions and avoids starvation of transactions [5]. It decides which transaction to

---

\(^7\) A transaction handles its locks in two distinct phases; the growing phase where locks are acquired but never released and the shrinking phase where locks are released but never acquired [57].
abort by starting each transaction with assigning it a unique and monotonically increasing timestamp, and in case of a conflict between transactions, the transaction with the lowest timestamp is allowed to progress. More advanced contention managers also exist like e.g. Polka which gives transactions priorities based on the number of objects they have accessed so far, and backs off lower priority transactions exponentially when encountering conflicting transactions with higher priority [25]. RSTM uses the Polka policy and Intel STM also uses a variant of Polka, while TinySTM and TL2 use timid contention management.

2.5.5 Invisible vs Visible readers

Transactions that only read data (read-only transactions) can either be visible or invisible to concurrent transactions that write to the same data [10]. Invisible readers (invisible read-only transactions) cannot be detected by writers (read-write and write-only transactions), and the invisible readers are the only ones responsible for detecting conflicts with the writers, i.e. validating their read sets. With visible readers however, writers can detect readers and therefore both the writers and the readers can detect conflicts. RSTM supports both visible and invisible readers, Intel STM supports visible readers, while TL2 and TinySTM only support invisible reads.

2.5.6 Design choices of SwissTM

In [5] it is believed that at the time of writing no other STM proposal existed with the same combination of design choices as SwissTM. It is described as a lock- and word-based STM implementation using optimistic conflict detection for read/write conflicts and pessimistic conflict detection for write/write conflicts, something which is called mixed invalidation, and has never been used in any previous lock-based or word-based STM implementation before. SwissTM is also described as using a new two-phase contention manager that ensures progress for long and complex transactions which are managed greedily, but with no overhead for short and read only transactions which are managed timidly.
3 Related Work

In this chapter a quick review is presented of what has been done before in the area and how it directly relates to this project. SwissTM, the STM library of interest is presented as well as studies about the main compiler candidate. A similar study to this project that used a computer game to benchmark STM code is also presented, along with some other benchmarks that have previously been used to benchmark STMs.

3.1 SwissTM

In the paper "Stretching Transactional Memory" by Dragojevic, Guerraoui and Kapalka [5], the STM library SwissTM is shown to outperform other STM implementations in experiments on different common STM benchmarks. Performance is measured by execution time of the benchmarks, and scalability is measured by varying the number of threads. The design choices of SwissTM are explained and why they are thought to be the best combination of design choices for a STM implementation.

Part of this thesis is to investigate whether SwissTM can be used as the underlying STM library in the parallel STM version of the game, and it is therefore useful to understand the difference between other STMs and SwissTM. What they did not do in this paper is to compare SwissTM with Intel’s STM library because they did not have access to the low-level API of Intel’s STM library, which they needed access to for their experiments. Intel’s C++ STM compiler is one of the compiler options for this thesis, and to compare the performance between Intel’s STM implementation and SwissTM would have been interesting because it would indicate whether the Intel C++ STM compiler can be made to perform better or not by changing the underlying STM library from Intel’s STM library to SwissTM.

3.2 LLVM

In "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation" by Lattner and Adve [14], the LLVM compiler framework is described, and evaluated by size and effectiveness. The compiler performance is evaluated with different C benchmarks, and the results show that the generated code size is comparable in size to x86 machine code, while optimization time is significantly less than compiling with gcc. Since LLVM is open source, fully available and has been shown to have good performance, it is together with Tanger the prime compiler candidate because it can be changed if needed and with which various options can easily be experimented with. It is also, as far as the author is aware of, the only compiler that has been previously used with SwissTM as the underlying STM library.

3.3 Tanger

In "Transactifying Applications using an Open Compiler Framework” by Felber et al. [15], the LLVM compiler and Tanger, a module for the LLVM compiler, are used to generate efficient concurrent applications using word-based STM libraries. Tanger transactifies intermediate code produced by the LLVM compiler by statically instrumenting it and replacing regular load and store instructions inside transactions with stm_load and stm_store function calls and adding calls for the start and end of transactions. These STM calls are
implemented by the underlying STM library TinySTM. However, one goal of the Tanger project is that it should require little effort for STM designers to configure Tanger so that their own STM implementation can be plugged in [9]. The process of compiling involves first to compile with LLVM to obtain intermediate code and apply optimizations, then to transactify the intermediate code with Tanger, and finally to apply optimizations again and use LLVM back-ends to produce native binary code or C source code. Comparisons were made with manually instrumented and optimized code, compiled with both LLVM and gcc. LLVM code was typically as good as manually optimized code even when scaling up the number of threads. LLVM and Tanger is a strong candidate in the selection of compiler for compiling this project. If they are able to compile the parallel versions of the computer game an attempt will be made to use an existing rewritten version of Tanger that uses SwissTM as the underlying STM library instead of TinySTM.

### 3.4 Atomic Quake

In "Atomic Quake: Using Transactional Memory in an Interactive Multiplayer Game Server" by Zyulkyarov et al. [12], a parallel implementation of the Quake multi-player game server using traditional mutual exclusion locks, condition variables and barriers was converted to one that uses transactional memory for all synchronization between threads. The Intel C++ STM compiler (Prototype Edition 2.0) was used, but with optimizations disabled due to compile problems with the Intel compiler. The STM version of Quake was most effective in simplifying parallel code that previously used fine-grain locking. However, the resulting STM version that implemented critical sections with transactions scaled poorly when increasing the number of threads due to many transaction aborts, and thus a waste of computation time. The Intel C++ STM compiler is another candidate for compiling this project, but a weaker candidate due to these previous poor results and the fact that there was no existing solution that used SwissTM as the underlying STM library at the start of this degree project.

### 3.5 STM benchmarks

In this thesis an attempt is made to use a complex real world computer game as benchmark to evaluate the performance of a parallel mutex lock and STM version of the game. Previously, often much simpler benchmarks have been used to evaluate STM implementations. These benchmarks might not use all the features of a programming language that a complex real world application might do. Their transactions might not represent the type of transactions that non-expert programmers would use, or their workloads may not represent well the workloads of complex real world applications. Furthermore, they might not invoke third party libraries which is common with real world applications, and these libraries might only be available in binary form that have not been compiled with STM. Below is a short description of some of the most popular STM benchmarks.

#### 3.5.1 STMBench7

One of the STM benchmarks that comes closest to a realistic real world application is the STMBench7 [26] implemented in C/C++, which aims to be comparable to a realistic object-oriented application with a range of workloads and different concurrency patterns. It mixes read and write operations of various lengths, uses various workloads from read-only
transactions to write-dominated ones and uses large non-uniform data structures with long transactions that access a large number of objects [5]. Even so, being a complex benchmark as it is does not mean that it covers the full C/C++ programming language, and therefore does not necessarily represent a complex real world application making use of advanced C++ features found in e.g. the Standard Template Library (STL)\(^8\).

3.5.2 STAMP

A benchmark with slightly smaller workloads than STMBench7 which also aims to represent real world applications is the STAMP benchmark [27]. It is a collection of medium sized workloads from various fields such as bioinformatics, engineering, computer graphics and machine learning. However, it does not involve very long transactions, and might therefore not be representable of code produced by a non-expert programmer or code automatically generated by a compiler [5].

3.5.3 Lee-TM

Another benchmark that aims to offer large and realistic workloads is Lee-TM, which is a benchmark based on Lee’s circuit routing algorithm [28]. It does have large transactions, but a drawback is that they have very regular access patterns where they first read many memory locations and then write to a few memory locations [5]. Another constraint is that it uses small objects that can be represented with single words. Therefore it might not be representative of real world applications which might have transactions involving much larger objects and with more irregular access patterns.

3.5.4 Red-black tree

One of the most often used STM microbenchmarks is the Red-black tree [24] which consists of lookups, insertions and removals from a red-black tree structure using short and simple transactions. Though being a less realistic benchmark than the previously mentioned ones, it can still be useful for testing the mechanics of STMs and to compare low-level implementation details of an STM [5]. There are also other data structures than the Red-black tree that have been used as micro-benchmarks to evaluate STMs like e.g. linked lists, skip lists and hash-tables [10; 7].

3.5.5 Conclusion

The four benchmarks described ovan were all used in experiments evaluating SwissTM in [5], comparing it to other STMs RSTM, TinySTM and TL2. Together they represent many types of different workloads, and in most cases SwissTM outperformed the other STMs. In this degree project a computer game will be used as benchmark, and it will be a novel benchmark since a computer game will be chosen that has never been used to evaluate STMs before.

\(^8\) A generic collection of algorithms and class templates that makes it easier for programmers to implement standard data structures such as lists, queues and stacks etc. [49].
4 Problem formulation

This Section describes the requirements that were identified at the beginning of the project for the different steps involved in this degree project from choosing the benchmark computer game and writing different parallel versions of it, to how to evaluate the performance of the parallel versions.

4.1 The benchmark computer game

The first part of this thesis is to find and choose a computer game of suitable scale and complexity to work with that can produce meaningful results as a benchmark for the study of STM. The game should not be too small or too simple (like e.g. Tic-Tac-Toe or Connect Four), so that when writing the parallel versions the solutions will not be too trivial and the performance measurements meaningless. It should not either be too big or complex so that it would be unreasonable to finish rewriting it to parallel versions in the time allotted for the project. There needs to be a working version of the game that can be compiled and is open source so that the source code is freely and fully available to rewrite to a parallel multi-threaded game. It should also be written in C or C++ so that it is compatible with SwissTM and the candidate STM compilers. Further it should have sufficient Artificial Intelligence (AI) so that it is possible to test the performance without too much of human interaction, and so that the simulation part of the game is large enough to have non-trivial solutions.

4.2 Analyzing the game code

After finding an appropriate game the next step is to identify the simulation part of the main loop of the game, and what parts of the loop make sense to parallelize by conducting a thorough code review to understand the control flow of the code. A useful tool would be to perform an investigative walkthrough of the call stack of the main loop to discover what data would become shared data after parallelizing the game and therefore needs to be protected with mutual exclusion, and what code dependencies are inherent. Code dependencies are important to identify so that the code will continue to execute correctly after parallelization, and therefore the need for thread barriers has to be examined. That is, if in the parts that will be parallelized there are code sections that depend on the completed execution of other code segments before they can proceed executing, and thus requires barriers where threads have to rendezvous before continuing execution.

4.3 Writing a concurrent mutex lock game version

The chosen computer game needs to be rewritten to a coarse-grained concurrent version that uses threads and a single global mutual exclusion lock to protect shared data. Since focus in this thesis is on evaluating the STM version of the game and comparing it to the mutex version, too much effort is not required to be spent on analyzing how to parallelize the game in the most efficient manner using intricate fine-grained locking. As long as the mutex and STM versions of the game protect the same critical code sections, they will be comparable. Therefore a coarse-grained concurrent version with a single global mutual exclusion lock on all shared data is sufficient for the purpose of this study. Only the simulation part of the game
should be parallelized where the state of the game is advanced, and not for example the
network communication part for multiple players or the graphical processing.

4.4 Compiling the game

At the start of project only three different STM compilers were known to the author:
LLVM with the Tanger module, the Intel C++ STM compiler and the gcc-tm compiler.
Depending on the availability and compatibility of these STM compilers, an investigation of
how to make a STM compiler compile the game and what configurations have to be made
will be performed. This also entails deciding how to handle eventual exceptions being thrown
inside transactions, if they cannot be handled by the compiler (Section 5.5 covers
irrevocability).

The choice of compiler boils down to if it is possible to compile the game when it makes
use of STM calls and what support it has for using SwissTM as the underlying STM compiler.
If at the end it is found that a compiler is not compatible with SwissTM not much can be done
but to try another compiler, since it is not an objective of this thesis to write a new STM
compiler. To make the game compile when using STM calls, it might be necessary to change
the build configuration of the game. Ultimately the goal is to use SwissTM as the underlying
STM library in the STM version of the game as it is the STM library of interest in this thesis.
However, if existing solutions do not allow compatibility between a STM compiler and
SwissTM, the original underlying STM library of the compiler will be evaluated. It is enough
if the game compiles with one of the available STM compilers, but it is not required to
compile with all the above mentioned.

4.5 Writing a concurrent STM game version

When a suitable and compatible STM compiler has been found or configured, a second
concurrent version based on the first parallel multi-threaded mutex lock version should be
written. Instead of using mutex locks on shared data, the STM version will use transactions to
protect shared data. The transactions will use STM calls to the underlying STM library to
transactify the critical sections of the code. It will be enough to only have a compiler
instrumented version of the game, instead of having both a manual and compiler instrumented
versions. If however, there is room for further improvements and tests at the end of the
project, a manually instrumented version of the game can possibly be developed to make
further comparisons with the compiler instrumented version.

4.6 SwissTM as underlying STM

One aspect of this thesis is to investigate the possibilities of using SwissTM as the
underlying STM runtime library with the STM compiler used to compile the game. It is not
part of this thesis to write a new compiler or to rewrite an existing compiler so that it is
compatible with SwissTM. At the beginning of the project, the only known existing solution
was the LLVM compiler and a rewritten version of the Tanger compiler module that uses
SwissTM instead of TinySTM as the underlying STM library.
4.7 Benchmarking the game

To perform performance and scalability evaluations of the parallel versions of the game, it is necessary to formulate a test suite with test cases that can be reproduced multiple times. The AI is important here to ease the creation and execution of test cases, because with adequately complex AI less human interaction is necessary, making it easier to repeat the same steps each time a test is run. A complex AI will most likely also have a large computational simulation part that leads to a non-trivial solution for critical sections of the parallelized game, giving the test results more reliability and importance since more computation is performed in the parallelized parts.

4.8 Performance measurements and comparison

The performance should be measured on a computer with a multi-core processor to improve the benefits of parallelism in the two multi-threaded versions of the game. All three versions shall be compared against each other: the original single-threaded version, the multi-threaded version with a single global lock, and the multi-threaded version using STM. The number of threads shall be varied in the tests to measure scalability of the game up to the number of cores that the processor has (beyond worsens benefits of parallelism). Timing mechanisms of the underlying CPU will be used to perform these measurements, and possibly also other useful data such as number of transactions and the abort rate of the transactions (if the compiler supports the gathering of such data).

4.9 Analysis of results

Finally the results obtained from the performance tests should be analyzed to try to explain why they were obtained and what they mean. With the understanding of how the results were obtained and what they mean, it is meaningful to suggest ways to improve the performance and scalability of the STM version of the game. Only if there is enough time and it is practically possible, identified improvements could be implemented to further improve the performance and scalability of the STM version.

4.10 Expected results

A similar study comparing a mutex lock and STM version of a computer game has previously been done using the Intel C++ STM compiler and its STM implementation on the Quake computer game. The results unfortunately showed reduced performance and poor thread scalability of the STM version of the Quake computer game compared to an implementation with traditional mutual exclusion locks [12]. Therefore it was possible that the same results would be obtained in this degree project which aims to transactify a computer game and compare the mutex lock version with a STM version of the game, and attempt to use SwissTM as the underlying STM library

However, there are some differences that perhaps might lead to different results. In the previous study using the Quake computer game, when compiling the game, optimizations had to be disabled due to compile problems with the Intel C++ STM compiler. The resulting STM version that implemented critical sections with transactions scaled poorly due to many transaction aborts and thus a waste of computation time. In this degree project, the hope is that full optimizations can be turned on when compiling the computer game which might lead
to better results. Furthermore, SwissTM is a different STM implementation than the Intel STM and it might perform better than Intel’s STM implementation if SwissTM can be used together with the chosen STM compiler. SwissTM has been shown to outperform other state-of-the-art STM implementations RSTM [10], TL2 [6] and TinySTM [9] in experiments on different benchmarks aiming to cover much of the complexity space that real world applications might use. It was not compared to Intel’s STM library, however, because Intel’s STM did not expose a low-level API that would have allowed manual instrumentation, and the benchmarks could not be transactified using Intel’s STM [5].

The best case scenario would be that the results of the study conducted in this thesis show, contrary to similar previous studies, that the parallel STM version of the game performs better than both the original sequential version and the parallel version with mutex locks, and that it is possible to use SwissTM with the computer game and compile it with the chosen STM compiler. The favored outcome is that the STM version scales nicely and performs better than the mutex lock version, but if it does not and it is possible to explain why, it is also very good. Even better is if it is possible to improve performance and scalability of the STM version after identifying why at first it did not perform or scale well.
5 Implementation

In this Section the implementation phase of the project is described. The game chosen for this project is described and an overview of the game code is given. The way in which the game was parallelized to a new parallel mutex lock version by introducing threads is explained. When transactifying the game to a STM version, many difficulties arose such as compiling the game, and the way these were handled are described along with what the actual process of transactifying the game to the final STM version involved. Finally the way in which the different tests were written is described and the way in which experiments with these tests were conducted is explained.

5.1 The computer game Globulation2

The choice of what computer game to use in this project was more a choice of preference and availability, but the game had to obey certain criteria. It needed to be open source so that the source code is freely and fully available so that it can be rewritten to parallel versions, and there needed to be a working version of the game that compiles. It should be written in C or C++ for compatibility with the STM tools used in this project, and the game should have an adequate AI to simplify testing and it should be reasonably large and complex to produce meaningful results.

After a rather quick survey on the internet was performed for suitable games, the Real Time Strategy (RTS) computer game Globulation2 [29], an open source multiplayer computer game written in C++, was chosen as the game to be used for this project. To evaluate if Globulation2 was of suitable scale and complexity the source code of the game was inspected, and with 110 053 lines of code in 380 files in total for the Globulation2 project, the expected amount of work was considered sufficient for one person to work with during the amount of time given. Beta version 0.9.4 (released 2009-02-01) of Globulation2 was used for this project.

Globulation2 features both a single player mode and a multi-player mode either through a Local Area Network (LAN) or through the internet via a server called Ysagoon Online Game (YOG). It has advanced AI and supports gameplay against both human and computer controlled teams, some limited support for writing your own computer controlled teams as well as an integrated map editor. Globulation2 is similar to other popular RTS games such as Command & Conquer and Warcraft, but takes an innovative approach to minimizing micro-management by automatically assigning tasks to units. Individual units cannot be controlled, but directives can be given to your whole unit population to e.g. build buildings or attack certain areas. The game will automatically assign units to these directives based on availability and priority, and the units will automatically follow the directives. Figure 2 and Figure 3 below give an example of what the gameplay of Globulation2 is like.
Figure 2: Two opposing players in battle. Screenshot from Globulation2.

Figure 3: Advanced base with units and buildings. Screenshot from Globulation2.
5.2 Code analysis of Globulation2

Globulation2 is an open source computer game and the code is available to download for free from the Globulation2 projects website [29].

5.2.1 Required packages

To compile the project it is also necessary to install some additional packages with libraries used by the game (download links can be found on the Globulation2 website):

- **scons** - SCons is an open source software construction tool (build tool) [30]. It is not a compiler, one still compiles e.g. C/C++ code with the default compiler (usually gcc), but it is just a tool for specifying build commands and building software in an easier way. It is a cross-platform substitute for the classic *make* utility.

- **SDL 1.2** - Simple DirectMedia Layer or SDL is a cross-platform multimedia library and API that provides low level access to audio, keyboard, mouse, joystick, 3D hardware via OpenGL, and 2D video framebuffer [31].

- **SDL_net 1.2** - A small cross-platform networking library, with a sample chat client and server application [32]. It provides a programmer network functionality compatible with many different platforms. It also makes it easier to handle network connections and data transfers.

- **SDL_image 1.2** - An image file loading library that loads images as SDL surfaces [33]. It includes loading and conversion algorithms that a programmer can utilize to make it easier to work with multiple image formats.

- **SDL_ttf 2.0** - A font rendering library that makes it possible to use TrueType fonts in SDL applications [34]. It provides font rendering routines that allow programmers to use multiple TrueType fonts and to easily create high quality text output using e.g. outline fonts and antialiasing.

- **speex** – An open source voice compression codec with an audio compression format designed for speech [35].

- **libogg and libvorbis** – The Ogg Vorbis audio compression format which is an open source audio compression format similar to e.g. mp3 [36].

- **zlib** – A free cross-platform lossless data compression library similar to zip and gzip [37].

- **Boost C++ Libraries** – Open source extension libraries to the standard C++ libraries including libraries for linear algebra, pseudorandom number generation, multi-threading, image processing, regular expressions, unit testing etc. [38].

- **Fribidi (optional)** - An implementation of the Unicode Bidirectional Algorithm (bidi) which is an algorithm for determining the directionality for bidirectional Unicode text [39]. Most text is displayed with characters from left to right, but not e.g. Arabic or Hebrew, and this algorithm is used to determine this directionality.

9 A utility to automatically compile and link computer programs from multiple source code files, often used with C or C++ programs. It can determine what files of large programs need to be recompiled and what do not [54].
5.2.2 The Boost libraries

Since the Globulation2 project requires the Boost.thread library and Boost.shared_ptr class (amongst other) to be installed in order to build the project, it was important to find out in what way they were used and if they would have any impact on how the game was going to be parallelized. The Boost.thread library enables multi-threading with shared data by providing classes and functions for handling threads and synchronizing data between threads. The Boost.shared_ptr is a template class that stores a pointer to a dynamically allocated object (usually with the C++ new expression). The shared_ptr guarantees that the object pointed to is deleted when the last shared_ptr that points to the object is destroyed or reset. None of the Boost libraries used in the Globulation2 project needed any special treatment when compiling because most Boost libraries are header-only, meaning that they are only header files with templates and inline functions so they do not need any separately compiled library binaries and can be linked without any special requirements.

Inspecting the source code showed that the game is multi-threaded to a small degree and uses the Boost.thread libraries in the four classes FertilityCalculatorThread, IRCThread, NetConnectionThread and OverlayAreas. These are used to help compute certain graphic related parts and handle network communication, and as such are not part of the simulation part of the game which was the only part of interest to parallelize. The shared_ptrs in the game are mainly used by the network communication parts and are used to pass around network messages, player orders and game events, and do not impact how the simulation part of the game should be parallelized.

5.2.3 The main loop of Globulation2

Globulation2 has, like most other computer games, a main loop (core engine) and the main loop of Globulation2 works sequentially (synchronously) meaning that there is no multi-threading involved apart from what was mentioned in Section 5.2.2. A brief description of the core engine of Globulation2 can be found in a small document published on the game’s website [40] as a resource for developers. In one tick of the game loop the main steps are to handle input from the Graphical User Interface (GUI), send and receive orders from the user and AI controlled players through the network, execute new orders, synchronize game objects by advancing them one step, update the map, check winning conditions, draw and update the screen and finally to adjust the frame rate. The main loop is found in the Engine.run() method in the Engine class, and a simplified overview is shown in Figure 4.

The developers’ solution to playing games over a network is to use a shared network system, as opposed to a client-server architecture, in order to minimize network bandwidth. In the shared network system every computer in a Globulation2 game has to compute everything and all the computers have to compute everything exactly in the same way. That is, all data will be used in the same way and every single condition is evaluated the same way in the game’s main loop in all computers of a network game. The main loop therefore only uses deterministic algorithms and uses the special random function Utilities::syncRand() which provides pseudo-random numbers such that each computer gets the same pseudo numbers and can make deterministic choices in the main loop.
Another clarification that needs to be made is the distinction between the Team class and the Player class, because these are handled separately in the game engine. Team is the “colony” one belongs to and has a unique color, and it is the different Teams that battle each other in Globulation2. Player is like an interface and can be controlled by either the user or an AI, and one Team can have up to 32 Players but a Player can only belong to one Team. A Team that has more than one Player will execute all orders from all of its Players. This becomes obvious when a user and an AI controls one Player each in the same Team, and the user and AI give conflicting orders in which case the order that is received first is executed.

---

```java
//Illustrates one iteration of the game’s main loop
Engine.run()
{
    //Handle GUI input such as mouse clicks and keystrokes
    gui.step();

    //Transmit orders from user and AI’s to network interface
    net.pushOrder(gui.getOrder(), localplayer);
    for each AI
    {
        net.pushOrder(ai.getOrder(), ai);
    }

    //Send and receive orders through network
    net.step();

    //Execute orders from players
    for each player
    {
        game.executeOrder(net(player).getOrder());
    }

    //Advance each game object with one step by calling .step()
    for each team
    {
        for each Unit
        {
            unit.step();
        }
        for each Building
        {
            building.step();
        }
        for each Bullet
        {
            bullet.step();
        }
    }

    updateMap();
    checkWinCondition();

    //Update screen and redraw everything
    draw();

    //Sleep the right time to have 40[ms] per frame.
    sleep();
}
```

Figure 4: Main loop of Globulation2. A simple overview of the main loop in Engine.run(). This is a modified version of the description of the core engine given in [55].
first and carried out on screen, but when the second order is received and executed, the first order will be disrupted and the conflicting second order will be carried out on screen instead. Therefore multiple Players in one Team are not in an alliance since the Team is like a “hive” and there is no way of separating Players or their orders from each other. The Teams are in fact the “logical players” in the game.

5.2.4 The simulation part of the main loop

Since the main loop of Globulation2 is not multi-threaded and one objective was to parallelize the simulation part (which is executed in the main loop), first the simulation part had to be identified. This is the part where the state of the game, the game world and all its objects, are updated and synchronized to represent the most current view of the game world. E.g. a new building that a Player has chosen to construct must also logically be created in the state of the game as a game object, or a unit that has died must have its game object deleted from the game world. To identify the simulation part, the game code of the main loop was studied in detail to understand how the control flow is directed through the game engine and what methods are called and what these methods do. Figure 5 shows an illustrative overview of the call stack of the main loop and the simulation part.
Figure 5: Call stack overview of the main loop.
The main loop in `Engine.run()` calls the method `GameGUI.executeOrder()` to execute orders after processing GUI input and transmitting and receiving orders through the network. The orders represent any interaction one can have with the game, and for each Player one order is processed by the `GameGUI`, which handles orders such as e.g. sending text or voice messages, pausing or quitting the game etc. All orders, processed or not, are eventually passed on to `Game.executeOrder()` which handles gameplay related orders such as e.g. creating/removing buildings or management of directives to the units. `Game.executeOrder()` will call the appropriate method to handle the order before the control flow returns back to the main loop in `Engine.run()`. An example of an order processed by `Game.executeOrder()` is `ORDER_CONSTRUCTION` (the order can be used to create new buildings or to repair existing buildings) for which the method `Building.launchConstruction()` is called. The method will attempt to remove units going to or working with the building, then reassign worker units to the building and finally attempt to get a building site for the building. In this way the appropriate method is called to handle the specifics of the order being processed. The execution of orders was not included in the simulation part that was to be parallelized, because the nature of the orders are such that they rarely update the state of the game, and if they do they only modify it to a small degree, and also because the code is not well suited for parallelization because it contains few and short loops.

The code section identified as the simulation part of the main loop is where `Game.SyncStep()` is called because it is the part that does the actual work of updating the state of the game and basically advances the game with one tick. It first calls `Team.SyncStep()` which performs one step for each building, unit and bullet in the Team by calling their respective `syncStep()` method. Up to 1024 units and buildings are allowed per Team and much of the computation in `Game.SyncStep()` and the methods it calls is carried out in different loops that iterate through a Team’s containers with units and buildings, and therefore lends itself well for parallelization.

The `Unit.SyncStep()` which is called by `Team.SyncStep()` does everything related to advancing the units (called globs) in the game with one step, like e.g. checking if the unit is under attack, performing its actions (moving, healing, fighting etc.) and determining if the unit dies during the tick.

After handling the units `Team.SyncStep()` moves on to update the state of buildings by deleting destroyed buildings, constructing new buildings and updating the amount of units the buildings need. Swarms are a special type of building that needs special treatment when updating them because they are the “base” building which spawns new units and also “heals” itself. Another building that needs special treatment is the turret, which resembles a “gun tower” since it finds enemy targets, computes the best target and shoots a bullet at the target.

The next thing updated by `Team.SyncStep()` is the clearing flags for buildings which states what resources the buildings need. Flags are a way of communicating directives to your units, since individual units cannot be controlled in the game. The units are directed to the flags by gradients in the cells of the map, and a unit will always move to the adjacent map cell with the highest gradient using a “hill-climbing” algorithm. So e.g. a unit that is flagged for finding food and is in search for the nearest food area will move to the adjacent cell with highest food gradient until it reaches a cell with food.

The final proceedings of `Team.SyncStep()` is to check if the Team died during the tick, to update the statistics of the Team such as total Defense, Attack and Health Points and to update the game event list by removing old and processed events. The game gathers statistics about

---

10 The majority of which are C++ std::list containers.
the Team because it is possible to set win conditions based on the amount of “prestige” a Team has, such that the first Team to reach a certain prestige threshold wins the game. The total prestige of a Team is calculated in many ways e.g. how advanced the Team’s buildings are, how much food and resources the Team has and how happy the Team’s units are.

After all the Teams have been updated, `Game.SyncStep()` calls the `Map.SyncStep()` which grows new resources on the map (food, stone etc.), calls the `sector.step()` for each map cell, and updates the map’s resource gradients. The `sector.step()` method handles bullets “flying” through the map cell and determines if they hit units and buildings in the cell and creates events for these such as explosions and death animations.

The final steps in `Game.SyncStep()` are to update the visibility around buildings on the map for each building in each Team, and to synchronize buildings that are waiting for a map area to be cleared of units so that it can be constructed by calling `buildProjectSyncStep()`. The statistics and prestige of teams is also updated with `prestigeSyncStep()`, the map script (for tutorials, campaigns etc.) is updated with `script.SyncStep()` and finally the game engine checks whether any team has won or lost.

Now `Game.SyncStep()` has finished updating the state of the game world and the control flow returns from the simulation part to the main loop in `Engine.run()` which redraws everything on screen and sleeps until the next frame. The frame rate is adjusted to about 40 [ms] at the end of the main loop, so that in the simulation part all game objects are synchronized with the state of the game world roughly every 40 [ms].

5.3 Parallelizing the game

Arbitrary non-concurrent code, like the code in the simulation part of the main loop, can be parallelized in some different ways using different techniques. Some of the techniques one could use are automatic parallelization and manual programming with pthreads or OpenMP.

Automatic parallelization is a technique where sequential code is automatically converted to multi-threaded parallel code. Using perfect automatic parallelization a programmer would not need to worry about the difficult and error-prone parallelization process of writing concurrent code. However, perfect and fully automatic parallelization does not exist yet (or TM would not be considered) because of the difficulties of the complex program analysis it requires and its need to handle unknown factors such as input size. An example of an approach with automatic parallelization of programs is the Sieve Multicore Programming System where programs are written in Sieve C++ (a C++ extension) and compiled and run with the Sieve compiler and runtime [41]. As automatic parallelization does not work perfectly and parallelizing code with e.g. a compiler like Sieve excludes the use of a STM compiler, this technique was not suited for the task of parallelizing the simulation part of Globulation2.

Turning to manual parallelization techniques the choice lay between using OpenMP or pthreads (POSIX threads). OpenMP is a parallel programming API and model that is directive-based, where a code section that is to be run in parallel is marked with a preprocessor directive [42]. Pthreads is a thread package and API for creating, managing and synchronizing threads defined for the C programming language [43]. Often it can be difficult to choose between the two approaches, but the difference between them is that OpenMP is a more high-level approach than pthreads and can be easier to program with, whereas pthreads offers a more fine-grained control over threads [44]. However, in this case pthreads were
chosen because it was uncertain if the STM compiler that would be used would support OpenMP and because pthreads would allow a better control over threads.

5.3.1 Introducing threads

Once pthreads was chosen as the approach for introducing threads and parallelizing the game, the code in the simulation part had to be inspected to determine exactly what code should run in parallel and what should run sequentially. Not all code is suited to run in parallel, e.g. if the code does not contain any loops or there are strict code dependencies such that the order of the code execution has to remain intact. In these cases the potential performance gain of parallelizing would heavily be outweighed by the performance penalties of creating threads for code that anyway would be run sequentially by one of the threads, or creating many thread barriers\footnote{A thread barrier is a point in the code where a thread has to stop and wait for all other threads to reach the barrier before proceeding.} where threads have to synchronize not to break the execution order of the code. In contrast, code that has many and long loops and no code dependencies is well suited for parallelization because work inside the loops can be distributed to multiple threads and there is no need to maintain the execution order of the code by synchronizing threads with thread barriers. The code sections of the simulation part that were considered reasonable to parallelize are shown in Figure 6. These are parts of code with loops that are long and/or perform much computing during each loop iteration, and that has few code dependencies.

![Simulation part overview](image)

The four classes where threads were introduced are Game, Map, Team and TeamStat which use somewhat coarse-grained locking on shared data with a single global mutual exclusion lock that is accessible to all created threads. Without going too deep in to the specifics of working with pthreads – there are many books and online tutorials covering the matter much
better e.g. [43] – instead a short explanation of how the pthreads package was used for parallelizing the game will be given.

All four of the classes Game, Map, Team and TeamStat create their own threads when their syncStep() method is called in the simulation part, and all use the same scheme for creating threads. The simplest of the classes is the Map class which will be used as an example, and the code for creating threads in the Map class is shown in Figure 7.

```c
/** Start thread section **/
void *status;
pthread_attr_t attr;
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
pthread_mutex_init(&mutexLock, NULL);

/* Create threads as joinable*/
for(int i=0; i<NUMTHRDS; i++)
{
    pthread_create(&thrds[i], &attr, stepMapSectors, (void *)i);
}
pthread_attr_destroy(&attr);

/* Wait on the other threads */
for(int i=0; i<NUMTHRDS; i++)
{
    pthread_join(thrds[i], &status);
}
pthread_mutex_destroy(&mutexLock);
/** End thread section**/
```

Figure 7: Thread creation in the Map class.

In the example, a thread attribute attr used to set attributes for threads is created, initiated and the “detached status” is set to “joinable”, meaning that if threads are created with this attribute they can join with the calling thread when they exit. A mutex lock mutexLock is also initiated before all threads are created, and this is the global mutex lock used to protect shared data. The variable NUMTHRDS specifies the number of threads that are to be created, and the game code was rewritten in such a way that the NUMTHRDS variable could be specified as a parameter on the console using the -threads command when starting the game. The stepMapSectors() is the name of the C function threads will execute once they have been created. The variable i is used to send an argument to the start routine stepMapSectors() and is used to pass on the thread’s id. The program will wait for all threads to finish executing and exit the stepMapSectors() routine with pthread_join(), and the void pointer status can be used by a programmer to specify the termination status of an exiting thread. Note also that the thread attribute attr and the mutex lock mutexLock are destroyed when they are not needed anymore.

Since there was a need to pass multiple arguments to the threads’ start routines, but the pthread_create() routine only allows one argument to be passed, a header file called Concurrency.h was created with a global struct called ThreadData. The ThreadData struct represents a collection of pointers to shared data needed by the threads to do their work. In the case of the routine stepMapSectors(), the threads executing it need access to an array
sectors and the variable sizeSector with the size of the array, so a pointer to the array sectors and a copy of the array size sizeSector is initiated in the ThreadData struct every time before threads are created in the Map class. Figure 8 shows the code in the Map class that does this.

```c
/** start init ThreadData struct **/
  thrdData.sectors = sectors;
  thrdData.sizeSector = sizeSector;
/** end init ThreadData struct **/

/** Start thread section **/
void *status;

Figure 8: Initiation of variables in the ThreadData structure. This is done right before threads are created.

In addition, the header file Concurrency.h also declares for all the classes that use threads, all the parallelized methods that are called after creating threads like e.g. stepMapSectors(), and also all thread specific variables such as the number of threads to use, a condition variable and a mutex lock for synchronization between threads etc.

5.3.2 Partitioning work amongst threads

The shared data accessed by threads was protected with the mutex lock mutexLock so that whenever a critical section of code was executed by a thread, it would call the pthread methods pthread_mutex_lock (&mutexLock) to lock and pthread_mutex_unlock (&mutexLock) to unlock the mutex lock. In total there are 27 regions of code that are protected and locked with the mutex lock in one iteration of the simulation part of the main loop - three in the Game class, one in the Map class, 16 in the Team class and seven in the TeamStat class.

To ease the switch in the future from mutex locks to transactions when writing the STM version of the game, macros tx_start and tx_end were defined in Concurrency.h for the locking and unlocking of a critical section of code. In this way the macros in Concurrency.h could easily be changed to the appropriate STM syntax for transactions without having to replace individual lines in the code. An example of how the macros were used is shown in Figure 9 for the stepMapSectors() method in the Map class.

```c
void *stepMapSectors(void *arg)
{
  int id = (int) arg;

  tx_start; //macro for pthread_mutex_lock (&mutexLock)
  for (int i = id; i < thrdData.sizeSector; i+=NUMTHRDS)
    thrdData.sectors[i].step();
  tx_end; //macro for pthread_mutex_unlock (&mutexLock)

  pthread_exit((void*) 0);
}
```

Figure 9: Critical section with macros for the mutex lock. A coarse-grained locking is used for the entire for loop, even though in this case a more fine-grained locking would have been possible on only the invocation of the step() routine. Tasks in the for loop are evenly divided amongst threads.
To make efficient use of the parallel computation capabilities of working with threads, work had to somehow be distributed amongst threads. One possible solution would be to create a “thread pool” from where idle threads perform tasks taken from a task queue. This solution was, however, not chosen because the interest lay in comparing a mutex and STM version of the game and a thread pool approach is not needed to make that comparison, and also because it would have required a large restructuring of the code that would have resulted in less time to work with the more important parts of the thesis. The solution that was chosen was to divide work evenly amongst threads in for loops. As can be seen in Figure 9, the for loop in `stepMapSectors()` is protected with the mutex lock and work is distributed evenly to threads based on their `id` number. The `id` numbering ranges from 0...n, where n = NUMTHRDS – 1, and a thread will always start working with a task equal to its `id`. If there are fewer tasks then a thread’s `id`, that thread will not do any work and let other threads with smaller `id`’s do the work. There will always be some thread that does some work, unless the for loop’s test expression always evaluates to false and there are no tasks, because thread 0 will always get the first task 0, thread 1 the second, thread 2 the third and so on. That is, thread with `id` = 0,1,2 ... n gets task `t` such that {t = id + NUMTHRDS * i, i = 0,1,2 ..., t < size} where `size` is the size used to determine the number of iterations in the for loop’s test expression, in this case `sizeSector`. Since there are usually much more tasks than threads, all threads get to do approximately the same amount of work.

However, some caution had to be taken with this approach which illustrates some of the difficulties of writing parallel code. There were some cases were a mutex locked for loop iterates over a list of objects using the size of the list in the loop test, but where objects were removed from the list inside the lock-protected code section, thus changing the size of the list. If a partition of tasks to threads is based on the size of the list (as in Figure 9), then this will result in faulty code where not all tasks are handled. There were four such situations in the game code (one in the `Game` glass and three in the `Team` class), and a shortened pseudo code example of such a situation is shown in Figure 10.

```cpp
void *removeObjects(int id) {
   tx_start;
   for (int i = id; i < objList.size(); i+=NUMTHRDS) {
      iterator it = objList.begin();
      advance(it,i);
      if ((*it)->shouldErase())
         objList.erase(it);
   }
   tx_end;
}
```

Figure 10: Pseudo code example of problems with partitioning tasks to threads.

Consider a very simple example with only one thread running the program and two objects in the `objList` in Figure 10, where both objects’ `shouldErase()` method evaluates to true and they should be erased. Thread 0, the only thread, is responsible for erasing objects 0 and 1 in the `objList`, but when it erases object 0 the size of `objList` changes to 1 and then `i` is incremented to 1 in the counting expression so that the loop’s test expression is no longer true and the for loop ends. This means that object 1 that was also supposed to be erased is not erased from `objList` because the code is faulty.

The solution that was used to handle this problem was to make an additional temporary list with references to the objects that needed to be erased, and then let the last thread that
executes the code erase the objects from the original list using the stored references in the temporary list. Figure 11 shows the pseudo code for this solution where references to the objects that should be deleted from the `objList` are stored in the temporary list `erasables`. When a thread has finished all work related to `objList` in the for loop it increments the `thrdsRdy` counter, and when all threads have finished working with `objList` the last thread that finished will be responsible of erasing the objects in `ObjList` by using the references stored in the temporary list `erasables`.

```c
void *removeObjects(int id) {
    tx_start
        for (int i = id; i < objList.size(); i+=NUMTHRDS) {
            iterator it = objList.begin();
            advance(it,i);
            if ((*it)->shouldErase()) {
                erasables.push_back(it);
            }
        }
    tx_end

tx_start
    thrdsRdy.push_back(id);
    if (thrdsRdy.size() == NUMTHRDS) {
        for (iterator it = erasables.begin(); it != erasables.end(); ++it)
            objList.erase(*it);
    }
    //clear for next round
    thrdsRdy.clear();
    erasables.clear();
    tx_end
    return 0;
}
```

Figure 11: Pseudo code example of a solution to the problem with partitioning tasks to threads.

5.3.3 Barriers

To ensure the correctness of the program after parallelizing it, all code dependencies in the parallel parts had to be detected and managed. These are code sections run in parallel that depend on the completed execution of other code segments before they can proceed executing, that is, code that depends on the results of earlier code and has to be executed in a particular order. The call stack walkthrough performed when investigating the game code had shown that there were four different parts within the parallel code that needed threads to synchronize in order to keep a particular order of execution and ensure the safe and correct execution of the program. Therefore to prevent incorrect execution, barriers for threads to synchronize were implemented with pthreads for these four parts (two in the Team class and two in the TeamStat class).

An example of how barriers were used is shown in Figure 12 for one of the barriers in the Team class. The `updateAllBuildingTasksConcurrently()` method, which attempts to assign units to buildings that have tasks that need to be performed by units, always needs to
be executed after the `stepBuildingsWaitingForRoom()` method which removes buildings from a waiting list of buildings that do not have enough space yet on the game map to be constructed. The strict execution order of these methods needs to be preserved so that if a building is removed from the waiting list, it can be assigned units and its construction can start. If there were no barrier for thread synchronization, it could be the case that a fast thread finished with the `updateAllBuildingTasksConcurrently()` method before another slower thread frees a building from the waiting list in `stepBuildingsWaitingForRoom()` for which the faster thread should have assigned worker units to. This leads to an incorrect execution of the program due to the broken order of the code execution.

```c
stepBuildingsWaitingForRoom(id);

// rendezvous with barrier
tx_start;
++readyThrds;
if (readyThrds<NUMTHRDS) {
    pthread_cond_wait(&barrier, &mutexLock);
}
else if(readyThrds == NUMTHRDS) {
    readyThrds = 0; // reset barrier
    pthread_cond_broadcast(&barrier);
}

// continue execution
updateAllBuildingTasksConcurrently(id);
```

Figure 12: Barrier example of thread synchronization.

The way in which this was solved was to use the pthread condition variable `barrier` to synchronize the threads. Whenever a thread reaches the barrier point it will lock `mutexLock` and check if all other threads have finished or not. If all threads have not finished it will wait with the `pthread_cond_wait()` routine which unlocks `mutexLock` and waits for a “wake up” signal to `barrier`. If all threads have finished it will reset the barrier and broadcast a wake up signal to all other threads waiting with the `barrier` condition variable, and all threads can continue executing the program.

### 5.4 Transactifying with LLVM and Tanger

Now that the parallel mutex version of the game was finished, a STM compiler needed to be found that could compile the rewritten game with its introduction of threads using the pthreads API. The prime candidate was the LLVM compiler since it has been shown to have good performance [14], but primarily because the LLVM compatible STM compiler module Tanger had been rewritten to use SwissTM (instead of TinySTM) at the Distributed Programming Laboratory at EPFL, and they were interested in trying it out on a larger scale application. The Intel C++STM compiler was the second best choice since it had already been used in a similar study, even though it did not do well performance wise [12]. The gcc-tm compiler was finally not considered as an option since there was no finished implementation that could be experimented with [45].
5.4.1 Modifying the scons script for LLVM

When building the game, the scons build tool is used to build all of the files and libraries needed by the game (as opposed to the classic make utility). The game uses a script file called SConstruct in the project’s main directory as the main building script, and in each of the subdirectories a subsidiary file SConscript is used by the scons build tool to determine what needs to be built. All of the scons files are written with the Python programming language, and understanding the scons files requires knowledge of both scons scripting and Makefile scripting.

When compiling the game with scons it by default uses gcc/g++ to compile, and in order to change the underlying compiler to LLVM the SConstruct file had to be modified. Before changing the scons script, however, the LLVM compiler had to be installed which can be downloaded from the LLVM website\(^\text{12}\). The version used in this project was LLVM 2.5. Important to note here is that the LLVM compiler and Tanger module are separate entities and are not bundled, so installing LLVM does not install Tanger as well.

To change the underlying compiler of scons to LLVM, a construction environment used to communicate build information to scons needed to be created with the scons utility’s \texttt{Environment()} method. The construction environment needed knowledge of the path to the LLVM compiler, and some of the default construction variables had to be changed from gcc commands to LLVM specific commands so that the LLVM build tools where used instead of gcc. The \texttt{CXX} compiler variable had to be changed from \texttt{g++} to \texttt{llvm-g++} and the \texttt{LINK} linker variable from \texttt{ld} to \texttt{llvm-ld}. The object file suffix \texttt{OBJSSUFFIX} was also changed from \texttt{o} to \texttt{.bc} to resemble the most common usage of LLVM. Changing the SConstruct file in this way made it possible to compile and build the Globulation2 project with LLVM and the scons software construction tool using the \texttt{scons} command in the main directory of the project containing the SConstruct file. The switch from \texttt{g++} to LLVM was successful, and the source files were now compiled with \texttt{llvm-g++} and linked to an executable with \texttt{llvm-ld}, and the game could be played without any problems.

5.4.2 Compiling with LLVM and Tanger

As mentioned earlier, LLVM and Tanger are not bundled and to use Tanger it has to be separately installed on the underlying system. The Tanger version that was used in this project was Tanger-20090304, and to install Tanger some configuration of paths to libraries, the LLVM compiler etc. needs to be made. Included with the Tanger package are some test examples that use STM, and to compile these or any other projects that use Tanger, first of all the Tanger header file \texttt{tanger-stm.h} has to be included in the source file so that the Tanger commands for transactions are detected by Tanger and the application can be transformed with the Tanger compiler module.

The Tanger STM commands can be grouped in three groups where the first and most high-level group consists of \texttt{tanger_init()} and \texttt{tanger_shutdown()}. The \texttt{tanger_init()} routine initializes the STM for the whole application, and must be called before any individual threads are initialized. The \texttt{tanger_shutdown()} routine shuts down the STM and should be called when STM is not needed anymore by the application. The second and middle-level group consists of \texttt{tanger_thread_init()} and \texttt{tanger_thread_shutdown()}. The \texttt{tanger_thread_init()} routine initializes the STM for the current thread and has to be called whenever a thread uses transactions, and the \texttt{tanger_thread_shutdown()} shuts down

\(^{12}\) http://llvm.org/releases/download.html
the STM for the current thread and should be called when the thread has finished executing transactions. The third and most low-level group consists of `tanger_begin()` and `tanger_commit()` which together constitutes a transaction. The `tanger_begin()` routine marks the start of a transaction and Tanger will look for a matching call to `tanger_commit()` which marks the end of the transaction at which point an attempt to commit the transaction will be made. There must be a corresponding call to `tanger_commit()` for each `tanger_begin()` call, so e.g. if a transaction is started and then a return is made from within the transaction, the programmer must make sure that `tanger_commit()` is called before returning. Using `tanger_begin()` and `tanger_commit()` is equivalent to the transactional code in the example of Figure 1, where a transaction is written as `atomic { statements; }`. Figure 13 shows the equivalent way of writing this with the Tanger commands.

```
tanger_init();
tanger_thread_init();
tanger_begin();  //Start of the transaction
    statements;
    tanger_commit();  //End of the transaction
    tanger_thread_shutdown();
tanger_shutdown();
```

Figure 13: Tanger transaction example.

The act of building an application with LLVM and Tanger is a multi-step process where the first step is to compile all source files to the Intermediate Representation (IR) used by LLVM. Commonly the `.bc` suffix is used for files compiled to LLVM IR bitcode to differentiate them from machine code files with an `.o` suffix compiled with e.g. gcc. To compile an application that uses transactions and Tanger with LLVM to LLVM IR, the `llvm-g++` command (alternatively `llvm-gcc`) is used. The appropriate directory to the include path of the Tanger header file `tanger-stm.h` has to be provided with the `-I` command-line option since it normally is not a part of the default gcc include path, and also the `-emit-llvm` and `-c` command-line options has to be provided which compile to LLVM IR.

After compiling all files to LLVM IR, they need to be linked to a single bitcode file which can be done using the LLVM linker `llvm-ld`. To link the files to a library, and not an executable, the `-link-as-library` linker option has to be provided because STM transformations with Tanger are yet to be done. Any libraries that are needed by the application should also be linked now.

When everything has been linked to a single bitcode file, the LLVM optimizer `opt` is used to transform and optimize the bitcode file in two different passes. The `libtanger.so` shared library needs to be loaded with the `-load` option to add new Tanger command-line options that enables Tanger to perform various STM optimizations and transformations. Recall that Tanger statically replaces regular load and store instructions inside transactions with STM specific function calls, and adds STM calls for the start and end of transactions. These transformations are done in the first optimization pass using the `-tanger` command option. Also the additional optimizations `-internalize` `-globaldce` `-raiseallocs` and `-mem2reg` were used when compiling Globulation2. These are the optimizations used in the Tanger Makefile example in the `tests/performance` directory of the Tanger project, and is recommended as a reference by the readme file provided with the Tanger project. The

---

13 LLVM defines a common language-independent type system in low-level code that is slightly richer than assembler.
optimization \texttt{-internalize} internalizes global symbols, \texttt{-mem2reg} promotes memory to registry, \texttt{-raiseallocs} raises allocations from calls to instructions and \texttt{-globaldce} performs global dead elimination. The second step of optimizations with \texttt{opt} only uses \texttt{-deadargelim} which eliminates dead arguments and is also used by the Makefile recommended by the Tanger readme file.

When all the transformations and optimizations have been performed it is time to link in the STM library used by Tanger. Recall that Tanger is written to allow a change of the underlying STM implementation, so it is necessary to explicitly specify precompiled bytecode files of the STM library you want to use. So once again \texttt{llvm-ld} is used with command-line option \texttt{-link-as-library} to link the transformed and optimized bytecode file with the precompiled Tanger STM bytecode files \texttt{libtanger-stm.bc} and \texttt{stmsupport.bc} to a library. The default STM of the Tanger project is a “dummy” STM which uses a single lock.

Now that the STM transformations have been carried out, and STM bytecode files have been linked, the resulting bytecode library should be compiled to assembly language for the specific architecture with the LLVM static compiler \texttt{llc}.

The final step is then to take the assembly language output produced by \texttt{llc} and pass it through a native assembler and linker like e.g. gcc to generate a native executable. However, the \texttt{libtanger-stm.a} also needs to be linked in as a binary to the application, which can contain support code for anything else that the LLVM compiler cannot compile. After this final step, to execute the application is now no different from executing a normal application compiled with e.g. gcc. This multi-step process of compiling with LLVM and Tanger was successfully used for the test examples provided in the \texttt{tests/performance} directory of the Tanger project, so the attention was turned towards compiling Globulation2 with the same procedure.

5.4.3 Compile problems with LLVM and Tanger

In order to compile Globulation2 with LLVM and Tanger, some modifications to the SConstruct build script had to be made in order to go through the multi-step compilation process. The path to the Tanger header file \texttt{tanger-stm.h} was added to the construction environment’s path variable \texttt{CPPPATH}, the \texttt{-emit-llvm} option was added to the construction environment’s \texttt{CXXCOM} command line variable used to compile C++ programs, and finally the option \texttt{-link-as-library} was also added to the \texttt{LINKFLAGS} link variable in the construction environment. Now the source files could be compiled to LLVM IR bytecode files, instead of the usual machine code output.

Since the scons script only executed the compile command once (and compiled to LLVM IR), the rest of the compilation steps had to be added manually to the SConstruct file. Scons provides the possibility to execute \texttt{Actions} through an \texttt{Execute(Action)} command. An \texttt{Action} can be as simple as a string with a command to run, and by creating command strings for each of the steps of the compilation process and using the \texttt{Execute} command to execute these, the full compilation process of compiling with LLVM and Tanger could be represented.

The next steps of the compilation process did not go as smoothly as the first one, and the linkage of the game’s bytecode files with \texttt{llvm-ld} did not work. After some experimentation with different linker options it was found that linking with the \texttt{-g} option which includes debugging symbols did not work, and the files had to be linked without it. The original build script of Globulation2 included the \texttt{-g} option during linkage which for the gcc linker \texttt{ld} is supposed to be ignored, and the \texttt{llvm-ld} linker should have behaved the same, but for some
reason it did not. Although the linkage without the \texttt{-g} option worked, a few ambiguous LLVM warnings concerning anonymous\footnote{The function names are modified when compiled to LLVM IR.} boost functions where produced saying that arguments were dropped, and these warnings could not be resolved.

After the linkage had been resolved it was time to run the two optimization passes and transform the bitcode file with Tanger. At this point the game code that was compiled was simply the mutex version and did not contain any transactions, but running the first optimization pass with \texttt{opt} that transforms the code with Tanger still produced thousands of LLVM warnings for anonymous functions of the type: \textit{function is a potential indirect call target, but has no txnal version}. Although the warnings could not be resolved, no errors occurred and the second optimization pass could be run without producing any warnings or errors.

The second linkage step where \texttt{libtanger-stm.bc} and \texttt{stmsupport.bc} are linked with the transformed bitcode file worked fine and did not produce any warnings or errors, but when trying to compile to assembly language with \texttt{llc}, errors in the LLVM libraries occurred that prevented assembly language output, and thus a native executable from being produced. The LLVM compiler infrastructure also includes the \texttt{lli} Just-In-Time compiler (JIT) which can be used to directly execute programs in LLVM bitcode, so the \texttt{lli} was tried on the bitcode file after the second linkage. However, the \texttt{lli} was not able to execute the program, and some other solution had to be found. Many different things and compile combinations were tried including skipping the optimization passes and compiling without Tanger, and then using \texttt{llc} to produce assembly language output, but without any success. The LLVM debugger \texttt{bugpoint} as well as the open source debugging and profiling tool Valgrind, were also used in order to find the problem but without much useful results. At this point it was not clear whether it was the \texttt{llvm\_g++} that generated LLVM IR code that was buggy, or if the \texttt{llvm\_ld} had bugs that did not allow it to link the generated LLVM IR code files.

In order to better track down what was causing the compilation to fail, and to localize the problem, subsets of Globulation2 where experimented with. There were some indications that the problem was related with the SDL packages, and by removing the part of the code that handled sound, voice and music in the game it could be compiled through the whole compilation process, including producing assembly language output with \texttt{llc}. Starting the application however, only the main menu of the game was working, and as soon as a game was loaded or started from the main menu the game would crash.

While conducting further experiments and trying out different compile options, the \texttt{-g} option was omitted when compiling the source files to LLVM IR bitcode files in the first step with \texttt{llvm\_g++}, and suddenly the game could be compiled with the full LLVM tool chain and Tanger compilation procedure (compilation, linkage, optimization, transformation etc.). The conclusion was that it was the \texttt{llvm\_g++} that was generating buggy LLVM IR code, but why the \texttt{-g} option was causing such troubles was never found out. Nonetheless, after omitting it, all original features could be turned on (including the sound and music parts) and the game could be compiled and played without crashing (albeit the first optimization pass still generated thousands of warnings). The troubles of compiling with the \texttt{-g} option using LLVM and Tanger was the first indication of LLVM and/or Tanger being more unstable than the much more widely used gcc compiler.
5.5 Irrevocability with LLVM and Tanger

Now that the game could be compiled with LLVM and Tanger, the next step was to start introducing transactions to the game and replacing the mutex locks with Tanger STM transactions (still using the default “dummy” STM of Tanger). Simple test transactions were introduced and the game could still be compiled, so an attempt to replace a mutex lock with a “real” transaction was made. However, compiling now did not work anymore because Tanger was not able to create transactional versions of some functions. After some investigation using Tanger debugging messages it became clear that this was due to Tanger’s lack of support of irrevocability.

When a transaction is aborted, all actions it has taken and all changes it has made have to be undone in adherence to the Atomicity property. However, in this lies a problem since not all changes and actions can be undone. There are irreversible operations (sometimes referred to as unsafe operations) such as I/O, system calls, exceptions and calls to “black box” libraries that cannot be undone once their effects have been made visible to the outside world. E.g. an I/O operation such as printing a message to the player cannot be taken back after the text is displayed, and a transaction that prints output but later has to abort and rollback its changes before it has the chance to commit, cannot reverse the effect of displaying the text. To cope with these types of irreversible operations, STMs can either not allow irreversible operations at all inside transactions or they can support an irrevocable mode (sometimes called inevitable mode). If a STM switches to irrevocable mode this means that only one irrevocable transaction is allowed to execute at a time, and all other irrevocable or conflicting transactions have to wait for it to finish, temporarily reducing concurrency. Irrevocability typically does not scale well when increasing the number of threads, but there are suggestions where irrevocable transactions can run in parallel with non-irrevocable non-conflicting transactions without significant overhead for the non-irrevocable transactions [46]. The transaction running in irrevocable mode is guaranteed to commit, and will never abort and revoke its actions [47]. In this way, even if a transaction has irreversible operations the transaction will not be started until it can run in irrevocable mode where there will never be a need to rollback the operations since the transaction will always commit because it is the only transaction running at that time.

The game code did call irreversible operations inside transactions and also the LLVM IR bitcode that was produced with llvm-g++ used unsafe functions that threw exceptions inside transactions. When this code was processed with opt in the optimization step, Tanger could not transform the irreversible operations and unsafe functions because Tanger does not support irrevocability or exceptions inside transactions. This led to compilation errors that did not allow the compilation process to finish all steps and an executable could not be created.

5.5.1 I/O operations inside transactions

One of the things that was causing problems when gradually transactifying the code was that down the call stack of functions inside transactions there where I/O operations such as printf, fprintf and std::cerr which perform actions that cannot be rolled back if the transaction aborts. The game uses printf to print game status information (such as when loading maps, saved games etc.) during VERBOSE mode and prints error messages before exiting the game, and fprintf and std::cerr are used to store information in a log file about actions taken by the game (loading maps, creating new buildings, processing player orders etc.) and also to print some error messages. In overall, these I/O operations are not used in a consistent manner in the game code, possibly because different persons have been working
with different areas of the code. E.g. sometimes error messages are printed directly with `printf`, and sometimes they are stored in the logfile with `fprintf` and some other times they are printed with `std::cerr`. In either case, they caused compilation errors because Tanger does not support irrevocability. Instead of forcing these operations to be removed or moved outside transactions which would have been difficult since they were usually many levels down the call stack and much code would have needed to be rewritten, Tanger provides an alternative solution where functions can be wrapped as `pure` which allows the functions to be called inside transactions.

A pure wrapped function means that it is safe to call the function even from within transactions without any transactional memory instrumentation [12]. Functions are wrapped pure usually if they are side-effect free like e.g. math functions like `sin` or `cos` that only depend on their input. Even though it usually would be inappropriate to reprint messages if a transaction aborts and retries (which can happen if the functions are wrapped with pure), this solution was still chosen because printing to the console is only turned on if the game is run in `VERBOSE` mode and the writing to the logfile is something that is done without the player noticing it, and also because it would not make any difference if a message is rewritten to the logfile. The I/O operations are non-essential for game play and wrapping them with pure would not affect typical game play because the player would not notice any difference.

The original implementation of Tanger provided a few pure wrapper functions from the start, but no one for `printf`, `fprintf` and `std::cerr` so pure wrapper functions needed to be written for these. Pure wrapper functions were defined for `puts`, `printf`, `fputs` and `fprintf` but not `std::cerr`. The reason why pure wrappers were defined for `puts` and `fputs` was because `printf` and `fprintf` called these internally and therefore these could be called inside transactions and Tanger would not have been able to transform these functions. `std::cerr` was not wrapped as pure because of complications wrapping C++ functions as pure and because `fprintf` could be used as a substitute. The pure wrapper functions were complemented with some attributes that Tanger needed to interpret them, and an example of how this was done for `puts` and `printf` is shown in Figure 14.

```c
static int tanger_wrapperpure_puts(__const char *__s) __attribute__((weakref("puts")));
static int tanger_wrapperpure_printf(__const char *__restrict __format, ...) __attribute__((weakref("printf")));
```

Figure 14: Pure wrapper functions for puts and printf.

5.5.2 Memory operations inside transactions

While gradually transactifying the game code, in addition to I/O operations inside transactions, Tanger was having problems with transforming functions `memcpy`, `memset`, `memcmp` and C++ operators `new` and `delete`. Since Tanger already came with predefined pure wrappers for `memcpy`, `memset` and `memcmp` in the file `tanger-stm-std-string.h` it was included directly in `Concurrency.h`. However, Tanger was having more troubles with transforming C++ `new` and `delete` operators which were used more frequently in transactions. Pure wrapper functions could not be written for them because they can throw exceptions in some cases, and Tanger does not support exceptions inside transactions. E.g. if operator `new` fails to allocate the requested memory a `bad_alloc` exception is thrown.
Even though pure wrappers for `new` and `delete` could not be used, Tanger came with predefined pure wrappers for the C `malloc` and `free` dynamic memory allocation functions in the file `tanger-stm-internal.h`, which was also included in `Concurrency.h`. Operators `malloc` and `free` were used to replace `new` and `delete` inside transactions because they do not throw any exceptions, but care had to be taken because `new` and `delete` also calls the constructor and destructor of the objects they allocate memory for, which `malloc` and `free` do not. However, this turned out not to be a problem since the allocation of memory was only done for templates of an `unsigned short` and `int` without special constructors or destructors. After allocating memory locations for them they are all assigned a value before they are read, so unassigned memory will never be read.

### 5.5.3 C++ STL inside transactions

Another problem that was encountered while transactifying the code, was using the C++ Standard Template Library’s containers like `List` and `Vector` inside transactions and using constructors of advanced game classes like e.g. the `Building` class. An example of the problem with STL containers was a list that contained Buildings that were waiting for the map to clear of units and resources before they could be built. Once a `Building`’s requested map area is clear of obstacles, the `Building` is erased from the list with the function `List.erase()`. However, using the `List.erase()` function inside transactions did not compile because Tanger could not transform it. One of the possible reasons could be because the underlying implementation of `List.erase()` uses `delete`, which Tanger cannot transform. However, neither the `erase()` function of STL containers nor other container functions could be used in transactions, so another reason could be that STL containers use precompiled functions in their underlying implementation, and can therefore not be transformed by Tanger.

Since the STL containers could not be used inside transactions, the Boost libraries’ `Boost::Multiindex` container was experimented with. It is similar to STL containers but can contain multiple indices which the container can be sorted and accessed in different ways with. However, there was no difference in using `Boost::Multiindex` from STL `List`, and Tanger could not transform neither when using their respective `erase()` function inside a transaction. The LLVM disassembler tool `llvm-dis` was used on the linked bitcode file (before optimization with `opt`) to disassemble it to a human readable program to try to understand why it could not be transformed. The problems seemed to be somehow related with memory allocators and exceptions that could be thrown from them when allocating and freeing memory. But this is also strange behavior because e.g. STL `List.erase()` should according to several sources e.g. [48; 49] never throw an exception (although it does not have a `nothrow` attribute). Nonetheless, the generated LLVM IR bitcode could somehow throw exceptions inside transactions which Tanger could not transform. Figure 15 shows a simple test program that Tanger was not able to transform because `List.erase()` was used inside a transaction.
Since the generated LLVM IR bitcode could throw exceptions (even though it should not), explicit exception catching was experimented with. Putting the `List.erase()` function in Figure 15 inside a `try-catch` block did not help even when catching all exceptions. Neither catching exceptions with `catch(void*)` nor `catch(...)` stopped LLVM from generating bitcode that allowed exceptions inside transactions. The generated LLVM IR bitcode still had paths that could jump directly out of the transaction. Finally after discussing the issue with Torvald Riegel, one of the main developers of Tanger, a bug report was opened with the simple test program in Figure 15 as bug #81 in Tanger’s bug reporting system\(^\text{15}\).

The default memory allocator of STL containers is the `default allocator template allocator` which is the only predefined allocator in the standard library and which all STL containers use. It has a function `allocate()` which uses the operator `new` to allocate memory and a function `deallocate()` that uses the operator `delete` to free allocated memory. Previously it was discovered that Tanger could not transform `new` and `delete` because e.g. `new` can throw a `bad_alloc` exception if it cannot allocate the requested amount of storage. To investigate whether the allocator of the STL containers was the reason e.g. `List.erase()` (where the allocator uses the `delete` operator) could not be transformed by Tanger, a custom allocator `c_allocator` was written that used the Tanger pure wrapped C functions `malloc` and `free` instead of new and delete for dynamic memory allocation in functions `allocate()` and `deallocate()`. The `c_allocator` was written following the example in chapter 15 of [49], and the program in Figure 15 was modified to that of Figure 16 where the list uses the `c_allocator` instead of the default allocator for dynamic memory allocation. The list in the program of Figure 16 is now initialized using the custom allocator which is responsible for the dynamic memory management, but the problem of LLVM generating bitcode with exceptions inside transactions still persisted, and Tanger was not able to transform the program.

---

\(^{15}\) https://se01.inf.tu-dresden.de/bugzilla/index.cgi
Strange behavior with LLVM and Tanger

While experimenting with the custom allocator and writing template classes a very strange behavior, possibly a bug, was found with LLVM and Tanger when compiling. A simple program with only two lines of code inside a transaction could only be compiled if the lines were placed in one particular order, but if the order of the lines were changed, the program could not be transformed. Consider the program shown in Figure 17 which has a template class Bar defined with only two functions: the constructor and destructor. They both have the throw() specifier which means that the functions are not allowed to throw any exceptions. The only two lines of code that are executed inside a transaction are lines 17 and 18. Line 17 initializes a new Bar object b, and line 18 simply prints text to the console using a Tanger pure wrapper for the printf function. If one of either line is omitted, the remaining line does not cause any problems when compiling, and Tanger can transform the code. However, if both lines are left inside the transaction with the initialization of b before printing with printf as in Figure 17, Tanger cannot transform the code and the program cannot be compiled. The strange thing is now that if line 17 and 18 switch order so that the printing is done before initializing b, then Tanger can transform the code and the program can be compiled.

This is completely strange and unstable behavior that normally should not occur, and the program was presented to Tanger developer Torvald Riegel who also agreed that this was weird and could not explain why this was happening. Perhaps there was something wrong with the Tanger pure wrapper functions, g++ or llvm-g++, but exactly what was causing the problem was not found. In either way, at this point it was finally considered that LLVM and Tanger as a STM compiler was too unstable at that point to continue with because of all the workarounds that had to be done, its inability to support basic C++ and the fact that it was behaving unpredictably and incorrectly.
In the future, the goal of the Tanger project is that it will support exceptions inside transactions so that if LLVM generates IR bitcode that can throw exceptions inside transactions, the code can be transformed by Tanger. To fully support C++, Tanger also will support transactional versions of new and delete, because if it does not then memory allocated inside transactions that later get aborted will leak, and all changes a transaction makes will not be rolled back. Tanger will also support the C++ Standard Template Library so that templates, STL containers and STL functions can be used inside transactions. E.g. since the creation of bug report #81 with the problem of using List.erase() inside a transaction, the bug has been noted by the Tanger developers and is currently being worked on.

5.6 Transactifying with the Intel C++ STM compiler

Since the first option of using LLVM and Tanger as the STM compiler turned out to be too complicated and unstable, and gcc-tm could not be used since there was no ready implementation to experiment with, the only choice left was the Intel C++ STM compiler. The Intel C++ STM compiler had been used previously in the Atomic Quake project [12], so there was a good chance that it would also be able to compile the Globulation2 project. However, the Intel C++ STM compiler is an experimental project under development and there are no current plans of a commercial release, so there were no guarantees that it would actually work. The version used in this project is the Intel® C++ STM Compiler, Prototype Edition 3.0\textsuperscript{16} which supports both irrevocability and transactional memory for C++ features such as exception handling, templates and I/O, and is not to be confused with the regular Intel C++ Compiler which does not support STM.

At that point there was no available implementation of the Intel C++ STM compiler and SwissTM or any other STM library except Intel’s own STM library. However, a release of the Intel® TM ABI specification 1.0 had been made which defines an application binary interface between assembler level code (IA32 and Intel(R)64), and the transactional memory library. When the decision to switch compiler from LLVM and Tanger to the Intel C++ STM compiler was made, a project was simultaneously started at the Distributed Programming Laboratory at EPFL by Aleksandar Dragojevic which aimed to change the underlying Intel STM library of the Intel C++ STM compiler to SwissTM using the Intel ABI.

### 5.6.1 Using the Intel C++ STM compiler

The installation of the Intel C++ STM compiler was much simpler than with LLVM and Tanger since everything needed is installed as a single unit. Prior to invoking the compiler, however, some environment variables need to be set to enable the Intel compiler commands. A C++ program is compiled with the command `icpc` and a C program with `icc`, but to compile a STM program that uses transactions they both require the `-Qtm_enabled` compile flag to turn on the macro that enables the STM mode of the compiler.

The Intel C++ STM compiler uses the keyword `__tm_atomic { statements; }` to mark a region of code to execute as a transaction, where all the statements within the scope of the curly brackets will be executed as a single atomic transaction. Figure 18 shows an example of how this is done, and it is very similar to the transactional code in the example of Figure 1, where a transaction is written as `atomic { statements; }`.

```
__tm_atomic {
    //Start of the transaction
    Statements;
} //End of the transaction
```

Figure 18: Intel STM transaction example.

In addition to `__tm_atomic`, the Intel C++ STM compiler supports several other TM keywords. The `__tm_waiver` keyword is used inside a transaction to annotate a region of code as non-transactional but without making the runtime switch to irrevocable mode. This means that memory accesses are not monitored by the STM runtime for conflict detection and rollback, and operations in the scope of `__tm_waiver` behave as if they were executed outside transactions.

The `tm_callable` keyword is used to annotate a function that can be called from inside a transaction, and the compiler will generate an instrumented transactional clone of the function which is used when it is called inside transactions. The normal non-transactional and uninstrumented version of the function is used when the function is called outside transactions. A `tm_callable` annotated function is allowed to use irrevocable operations and call legacy code. All function annotations are written as `__declspec(annotation)` in Windows syntax and as `__attribute__((annotation))` in Linux syntax, as shown in the example of Figure 19.
The `tm_safe` keyword is similar to `tm_callable` and is also used to annotate a function that can be called inside a transaction, and it also has an instrumented transactional clone that is used inside transactions while the normal uninstrumented function is used outside transactions. However, `tm_safe` annotated functions are not allowed to call irrevocable operations, legacy code or `tm_callable` annotated functions, and can only contain operations that can be executed atomically.

The `tm_pure` keyword annotates functions that do not need any TM instrumentation, and are safe to execute as they are inside and outside transactions. The compiler will not create a transactional clone of the function, and it is up to the programmer to make sure the function can safely be executed inside transactions. The functions annotated with `tm_pure` must only access local data, and can be used for math functions like e.g. sin or cos which only depend on their input.

The `tm_unknown` keyword is used on functions whose appropriate TM attribute cannot be determined by the programmer, in which case the compiler performs an analysis to determine the appropriate execution mode. If the function definition can be seen by the compiler, it will promote the function to either a `tm_callable`, `tm_safe` or `tm_pure` function and recompile the function. If not, the compiler generates a request for a dynamic switch to irrevocable mode, and the TM runtime will, in order to guarantee safe execution, switch to serial irrevocable mode when a transaction with such a function is executed.

There are strict rules and restrictions between statements and functions annotated with the Intel STM keywords stating what other annotated functions or statements they can call or get called by (for a complete reference see the users’ guide [50] included in the installation package of the Intel C++ STM compiler). The compiler aides in upholding these rules by generating compile errors whenever a rule is broken. E.g. if a `tm_safe` function calls a `tm_callable` function the compiler will generate an error with the rule that has been broken, in this case that a `tm_callable` function cannot be called by a `tm_safe` function, and the name of the function that broke the rule.

If a function is not annotated it will by default be treated as a `tm_unknown` annotated function, and the transaction executing the unannotated function may or may not switch to irrevocable mode depending on the compiler analysis performed during compilation. If unannotated functions are used inside transactions the compiler will warn that the unannotated functions are not transaction ready. In this way the programmer receives a lot of help from the compiler in finding functions inside transactional code that are not annotated. The warnings and errors from the compiler can be used to track unannotated functions down the call stack of an annotated function being called inside a transaction. Each time a new function used inside a transaction is annotated and the program is recompiled, if there are new warnings or new errors then they are the result of incorrectly annotated or unannotated functions being called by the newly annotated function. Since unannotated functions are treated as `tm_unknown` functions by default, in order to lower the chances of the STM runtime switching

```c++
class Foo {
public:
    Foo();
    ~Foo();
    __attribute__((tm_callable)) void readSharedMem(void){}
};
```

Figure 19: Example of how to annotate a function as `tm_callable` in Linux syntax. Only the declaration of the class Foo is given here.
to irrevocable mode and to increase parallelism, it is therefore important to correctly annotate as many functions as possible that are called inside transactions to prevent the STM runtime to switch to irrevocable mode unnecessarily.

5.6.2 Compiling with the Intel C++ STM compiler

When the Intel C++ STM compiler had been installed, the first thing that was done was to compile the simple test programs that could not be compiled by LLVM and Tanger. The strange and unpredictable behavior of Figure 17 which was discovered when compiling with LLVM and Tanger did not manifest itself when compiling with the Intel C++ STM compiler. The execution order of the initialization of the Bar object b and the call to printf was interchangeable. It did not matter what order the lines were executed in, the program still compiled. The program in Figure 15 where the STL’s List.erase() function was used inside a transaction could not be compiled with LLVM and Tanger, but the Intel C++ STM compiler could also compile that program.

The Intel C++ STM compiler did not have any problems to compile the C++ features that LLVM and Tanger struggled with, so now the question was whether it could compile the Globulation2 project or not. The construction environment in the SConstruct file of the Globulation2 project had to be modified to use the new compiler. The construction environment needed knowledge of the path to the Intel C++ STM compiler and the CXX and LINK variables of the construction environment had to be changed to use icpc as the compiler and linker, as well as adding the -Qtm_enabled compiler flag to the compilation commands to enable the STM mode of the compiler.

When building the original sequential version of Globulation2 however, it turned out that the Intel C++ STM compiler could not compile the Globulation2 project. There were three files that could not be compiled, namely, YOGServerBannedIPListManager.cpp, YOGPlayerStoredInfo.cpp and YOGServerGameLog.cpp which are related to the Ysagoon Online Game server and used for internet game play. They all produced the same error internal error: backend signals, which is a generic message not specific to any particular issue. The error simply means that the compiler suffered an unexpected error during compilation, and there are many different conditions which can produce such an error, so the reason why it occurred could not be found. However, the code that was producing these errors could be identified, and in each file there were two different methods that were producing the errors, one to read data from a stream and one to write data to a stream. The data read and written to the stream is YOG specific information and includes Player status information, banned IP’s and game log information. These files are only used for internet game play when using the YOG server and as such are not important for the project since the goal was to test the simulation part of the game using as much automation as possible. The automation can be achieved by having a single human player play a non-network game against multiple AI opponents. Therefore the code in these methods was commented out since the methods were not going to be used in tests anyway, and the original source code of Globulation2 could now be built and the game played without any more problems. In fact, the game could be compiled with the -g option which produces debugging info and with the -O3 option which enables all optimizations, something that could not be used when compiling with LLVM and Tanger.
5.6.3 Annotating with the Intel STM keywords

The parallel mutex lock version of Globulation2 could also be compiled with the Intel C++ STM compiler, and using simple test transactions in the game code worked as well. When starting to transactify the game code by replacing mutex locks with transactions the compiler started to produce many warnings about non transaction ready functions that were called inside the transactions. The functions produced these warnings because they were unannotated, and by default the compiler treats unannotated functions as \texttt{tm\_unknown}. To increase parallelism and reduce the chance of the STM runtime switching to serial irrevocable mode because of the unannotated functions, the functions were annotated with the appropriate Intel STM keywords. To determine the appropriate annotation for a function, the functions and the nested functions they called had to be thoroughly inspected to determine the level of instrumentation they needed. Inspecting the call stack of functions inside transactions revealed that the majority of functions that were called inside transactions belonged to classes \texttt{Building}, \texttt{Game}, \texttt{Map}, \texttt{Team} and \texttt{Unit}. The grand total of annotated functions was in the end 234, of which 106 functions were annotated as \texttt{tm\_callable}, 103 as \texttt{tm\_safe}, 22 as \texttt{tm\_pure} and three as \texttt{tm\_unknown}. Furthermore, 137 code segments were annotated with \texttt{__tm\_waiver} so that they would be executed non-transactionally and would not be instrumented or monitored by the STM runtime. These code segments were mainly print methods that either printed error messages or wrote to the game’s log file. As explained earlier when wrapping these types of print function calls with Tanger pure wrappers, they are not important for the game play and are therefore considered reasonably “safe” to execute without any conflict detection by the STM. The functions annotated with \texttt{tm\_pure} were either simple functions that only used local data, or read data that would never change once set like e.g. the width and height of the map used in the game or the number of teams in a game.

During the process of annotating function declarations with Intel STM keywords, there were some functions being called inside transactions with inaccessible code that produced compiler warnings of them not being transaction ready. The functions included e.g. STL container functions like \texttt{begin()} and \texttt{push\_back()}, operators like e.g. \texttt{operator\*} and \texttt{operator++} for some complex classes as well as accessing third party code like e.g. Boost shared pointers. These were functions that could not be annotated since their code was not available, and therefore had to be left unannotated which normally meant that they would be treated as \texttt{tm\_unknown} by the compiler. However, the new 3.0 version of the Intel C++ STM compiler is supposed to have support for transactional C++ including the STL libraries, making it different from the 2.0 version used in the Atomic Quake project [12]. Even though a warning is generated by the 3.0 compiler for these functions, if the function definitions are seen before they are called inside a transaction, a transactional version of the function should be generated which would prevent the STM runtime to switch to serial irrevocable mode because of that function. It would not have been wise to put these function calls inside \texttt{__tm\_waiver} blocks since then they would not be instrumented at all and would be unsafe to call because e.g. some STL containers are accessed and updated often inside transactions and uninstrumented manipulation of the containers could lead to inconsistent memory.

To test whether the STL \texttt{List} container (the most used container in the game) used specialized STM versions of its functions, a simple test program was written where multiple threads conflictingly read and write data in a shared list. This program was run while using Intel’s STM statistics gathering mechanism. To enable the gathering of transactional statistics, the environment variable \texttt{ITM\_STATISTICS} has to be set to either \texttt{Simple} or \texttt{Verbose} (default is \texttt{None}). In \texttt{Simple} mode statistics about the number of executed transactions, the number of transitions to serial irrevocable mode and the number of times a transaction is retried due to
conflicts are gathered. In addition to these statistics, in Verbose mode the number of bytes read and written in transactions are also recorded. The statistics are written to a file called `itm.log` and include the total measurements for each transaction and thread, as well as for the whole program. In the generated statistics file of the test program there were records of retries but no records of transitions to serial irrevocable mode. If the statistics mechanism was correct, this meant that there had been conflicting transactions where some transactions had to abort and retry, but that the compiler could use specialized STM versions of e.g. `List.erase()` since the STM runtime did not have to switch to serial irrevocable mode to execute transactions that used the `List.erase()` function.

The reason why three functions were annotated as `tm_unknown` was because otherwise they produced the generic `internal error: backend signals` error. For some strange reason they could not be annotated `tm_callable`, that is, the compiler could not generate transactional versions of the functions, even though a `tm_callable` annotated function is allowed to execute irrevocable code and legacy code. The three functions were the constructors of the `Unit` and the `Building` classes, as well as the `handleActivity()` in the `Unit` class which decides what a unit should do (e.g. to heal itself, eat food, convert to an enemy team etc.). Another strange thing was that if all code in the `handleActivity()` function was enclosed in a transaction (has the effect of nesting a transaction), then the function could be annotated as `tm_callable`. However, to keep the code as simple as possible for testing purposes, the nested transaction option was not chosen and the function was instead annotated as `tm_unknown`.

All the mutex locks that protected critical sections in the game could be replaced by transactions except where they were used for the four thread barriers in the game. In total 23 of the 27 mutex lock protected critical sections were changed to transactions, and the remaining four mutex lock protected sections were only used to guard entry to the barriers. The barriers are implemented with a pthread condition variable and when threads wait at the barrier they call the `pthreads_cond_wait()` which makes the thread wait for a broadcast signal on the condition variable and also unlocks the mutex lock. The entry to the barrier could only be protected with a pthread mutex lock because using transactions generated the generic `internal error: backend signals` compile error.

### 5.7 Tests and experiments

At the time when the STM version of Globulation2 was finished, there still was not any working version of the Intel C++ STM compiler that used SwissTM as the underlying STM library, although work on it had started. Therefore the tests and experiments performed were done with a STM version of Globulation2 that used Intel’s own STM library. This Section describes the different tests and experiments that were constructed and performed.

#### 5.7.1 The Test Cases

When choosing test cases, or more specifically, the game scenarios to use in the experiments with the different Globulation2 versions, they should be such that they could be reproduced multiple times and represent the majority of scenarios that could happen when playing the game. In the end, four saved games with different characteristics were chosen as the test suite that would best represent a majority of the game but still be reasonably fast to execute. The saved games were called `G2_enemy_1min.game`, `G2_enemy_1sec.game`, `Playground_7enemies_7min.game`, and `strange2_enemy_5min.game` after the map they
are played on, the number of enemy AI Teams in the game and how long time they approximately take to finish when playing with the original sequential version of Globulation2 with graphics turned on. All saved games have enemy AI players that battle the human Player, and saved games can all be played completely automatically without any human interaction to reach the end of the game where either the human Player is defeated and loses the game or defeats all other opponents and wins the game. The reason is because the human Player is paired up with an AI in the same Team, and together they battle the other AI Teams. This means that the ally AI will in effect be the only one controlling the human Player’s Team.

The two test cases with the G2 map are saved games right at the end of a game where the enemy Player has been severely weakened to only a few remaining units and buildings and is just about to be defeated. The other two test cases on the strange2 and the Playground maps are saved games right at the start of a game. Together the four test cases constitute scenarios at the start, middle and end of playing a game, and all three maps have somewhat different topologies and starting points making them strategically different. Figure 20 shows the topologies of the maps G2, Playground and strange2 which were used in the test cases.

The Playground test case is a scenario where seven Teams fight each other on a big and open map with limited resources. The Playground map favors quick and early attacks since there are few obstacles and not so much resources to develop your base with, at least not in the early phases with so many opponents competing for the same resources. The game starts with only a few units and a swarm (base building) for each Team.

The strange2 test case uses a big and very open map for only two Teams where each Team starts with many swarms and over 100 units. Only a thin wall of resources blocks the path between the two Teams and a hole in the resource wall is quickly produced, which has the effect of almost immediately starting a massive war between both Teams. There is no time for base building on this map.

The G2 test scenario started out with three enemy Teams in the beginning with all Teams having a few units and a swarm. But the test scenario has been played and saved at the point where two of the enemy Teams have been defeated and now only one enemy Team is left in a weakened state. The G2 map topology is such that all four Teams are cut off from each other in the beginning and almost completely surrounded by an impenetrable wall of stone, and there are many resources that need to be cleared from the map until Teams can attack each other. The clearing of resources is time consuming on this map which gives time to build big armies and bases before the initial enemy contact.

Although the ideal would be to have completely deterministic test cases, this is very difficult to achieve in a game that incorporates randomness, even if all game play is
completely automatic. For all test cases except the Playground scenario, the outcome of a finished game has always been the same, but a few rare times in the Playground scenario the human Team has won even though most of the time one of the enemy Teams win. Also the length of the time it takes to complete the Playground and strange2 test cases can vary sometimes.

5.7.2 Statistics with the Intel statistics mechanism

When the game had been fully transactified and all 234 functions annotated, the statistics mechanism was tried on the three different game versions: the sequential, the mutex lock and the STM version. The sequential and mutex lock versions could use the statistics mechanism, albeit with no useful information since they do not have any transactions. However, the STM version could not always use the statistics mechanism and would produce a segmentation fault non-deterministically. Using Verbose or Simple mode for the statistics gathering did not make any difference. When statistics were successfully written to the itm.log file for the STM version, there were never any retries or transitions to irrevocable mode which was somewhat unexpected. The test case used to gather statistics from was the G2_enemy_lmin.game saved game with close to 200 units and more than 100 buildings (the longer test cases would produce segmentation faults), so transaction conflicts seemed reasonable.

The conditions for when the segmentation fault occurred (non-deterministically) when gathering transaction statistics with the STM version was when multiple threads were used and the game was compiled with -O3 optimizations, but since the segmentation faults occurred non-deterministically this was not certain. The segmentation fault was successfully caught with the gdb debugger, and the problem was traced down to the Intel libintlc.so.5 and libitmdyn.so libraries. The problem seemed to originate from the implementation of the Intel C++ STM compiler or Intel STM, and not any programming error.

While testing the Intel statistics mechanism on simpler programs than the Globulation2 game, some strange statistics were gathered by Intel’s statistics mechanism. Consider the somewhat contrived, but simple, test program in Figure 21.

```c
#include <iostream>
int main(int argc, const char** argv)
{
    int txs = 0;
    __tm_atomic {
        __tm_waiver { txs++; }
        printf("Hello\n");
    }
    printf("Txs: %d\n",txs);
}
```

Figure 21: Strange statistics with the Intel statistics mechanism. Running the program generates statistics that claim there have been two transactions.

In the test program there is only one transaction, and there is no multithreading, so it should reasonably only be executed once. The transaction increments a transaction counter

---

17 The gdb debugger lets a programmer see what's going on “inside” a program while it executes. [http://www.gnu.org/software/gdb/](http://www.gnu.org/software/gdb/)
(without instrumentation), and prints a generic text. When the transaction finishes, the
transaction counter is printed and the output is as expected 1. But the Intel statistics in the
\texttt{itm.log} claim that there were two transactions, but no retries or transitions to irrevocable
mode. When compiling, the compiler warns that the \texttt{printf} that prints “Hello” is not
transaction ready, but if this would make the STM runtime switch to irrevocable mode then
this should be recorded by the Intel statistics mechanism, which it is not. And in any case, the
transaction should only be executed once. If it would be the case that when the transaction is
about to execute the \texttt{printf}, it aborts and restarts in irrevocable mode and thus executes the
transaction twice, then there still should be a record of a retry being made, but there is not. It
is uncertain why the Intel statistics mechanism is faulty, but switches to irrevocable mode do
not seem to be reported correctly and the statistics mechanism seems somehow broken.

\section*{5.7.3 \hspace{0.1cm} Timing the simulation part}

In order to compare the performance and scalability of the three versions of Globulation2,
some sort of timing had to be introduced to them. Since all the parallel work is done inside the
\texttt{Game.syncStep()} which represents the simulation part of the main loop, it was considered
the best place to measure time in. All other parts of the main loop are executed sequentially
and it would not make much sense timing those parts since they would all be executed
identically in all three versions. The \texttt{Game.syncStep()} method is called in each iteration of
the main loop and it is the only part of the main loop where threads are used for parallel
computation in the two parallel versions of Globulation2. The other parts of the main loop
handle GUI input, network communication, Player orders and GUI updates, and are therefore
not interesting to measure. The approach of measuring time spent in the \texttt{Game.syncStep()} method
is a coarse-grained way of measuring performance.

To measure time spent in the \texttt{Game.syncStep()} method of the three Globulation2
versions, a timing mechanism was introduced to the \texttt{Engine.run()} method in the \texttt{Engine}
class. A first timestamp is recorded right before \texttt{Game.syncStep()} is called, and a second
timestamp is recorded immediately after \texttt{Game.syncStep()} returns. The function used to
record the timestamps is the Unix function \texttt{gettimeofday()}\footnote{Described in the POSIX 1003.1-2001 standard.} found in the header file
\texttt{<sys/time.h>}. It obtains the current time in seconds and microseconds since a certain time
epoch and stores them in a \texttt{timeval struct} found in the header file \texttt{<time.h>}. To get the
number of microseconds spent in the \texttt{Game.syncStep()} method, the difference between the
second and first timestamp is calculated. The resolution of the system clock is unspecified and
the exact number of microseconds might not be entirely correct, but individual microseconds
are not that important and by making a large amount of measurements this minor detail does
not affect the end results in any significant way.

When the game is run with the timing mechanism, a file with the time measurements of
\texttt{Game.syncStep()} for each thread count, each test case and each game version is created.
The first row written to the statistics file is the time when the time measurements were started, i.e.
the first time when the main loop was executed. The second row is the sequence of time
measurements collected during the execution of the game together with a loop counter that
counts in which iteration of the main loop the time measurement was taken. The third and
final row contains the time when the time measurements were finished, i.e. the time when a
game is either lost or won. All game versions were modified to automatically exit when a
game has been won or lost and the final timestamp has been written to the statistics file.
Automatically exiting without requiring user input increases automaticity and makes it easier to perform multiple tests in a batch.

An example of what the output in a statistics file looks like is shown in Figure 22. All time measurements performed during a run of a test case are written to the statistics file. For the shorter G2_1sec test case around 14 time measurements of the simulation part are performed before the game ends, and with the longer test cases strange2 and Playground the number of measurements are between 10,000 and 30,000.

![Figure 22: Partial output of a statistics file with time measurements of the simulation part.](image)

### 5.7.4 Analyzing time measurements

Having statistics files with time measurements of the simulation part does not in itself say much about the performance of the different versions on the different test cases, especially when there are tens of thousands of time measurements in each file. Therefore a Java program `StatsAnalyzer` was created that analyzes the output generated to a statistics file, and writes the result to an "Analysis file". The `StatsAnalyzer` calculates the maximum, minimum, median, arithmetic mean and standard deviation in microseconds of the times spent in each iteration of the simulation part, before writing the results of the analysis to the Analysis file.

Since all time measurements from the execution of a test case are written to the statistics file, different iterations of the simulation part can take different amounts of time to finish since there might be a different amount of units or buildings to process for the different iterations. Furthermore, the game is non-deterministic due to the use of a random seed. Therefore the standard deviation is calculated to measure result variance because different executions of the game might behave differently. An example of what the output from the `StatsAnalyzer` program looks like is shown in Figure 23.

![Figure 23: Analysis file created by the StatsAnalyzer program. Times are in microseconds.](image)
5.7.5 Automating experiments

In order to achieve full automation when running experiments with the test cases, it had to be possible to start and play the game without any human interaction. Somehow the graphic rendering of the game had to be found so that it could be turned off. Otherwise the game would need some human interaction to manually load a test case from the games main menu each time a new experiment is run. Running experiments without graphics rendering would also make the experiments finish faster since no time would be spent waiting for the screen updates before running the next step of the simulation part in the main loop. Moreover, if the graphics could be turned off with a command or configuration option it would make it easier to switch between the no graphics version and the version with graphics.

Luckily, Globulation2 was already designed to be able to run without graphics using the –nox command when running the game from the console, probably in order to make it easier for the developers to run tests. Starting a game using the –nox command runs the game without using the X server, and the game is executed completely without drawing any window. To run a game with the –nox command, it is also necessary to specify three other input. The first has to be the name of a saved game file, and e.g. the name of one of the four test cases can be given. The second is a number specifying the number of steps or iterations to run the main loop for, and choosing zero steps will make the game run until the end. The final input is also a number and specifies the number of times to run the chosen saved game. An example of how a game would be run from the console can e.g. be:

user@server:~$ ./glob2-sequential -nox G2_1enemy_1sec.game 0 1

To be able to run all tests completely automatically with one command from the console and to ease testing, a script was written that varied the desired game configurations. It controls which game version to use in the tests: the sequential, parallel mutex lock or parallel STM version, how many threads to run the game with, whether to run the game with or without graphics, which test case to load and run, how many times to repeat the experiment, and whether to analyze the generated statistics file or not using the StatsAnalyzer. Running the script will automatically load and play a game, and the timing mechanism will measure times spent in the simulation part and write them to file, and finally when a game is either lost or won the game will exit.

5.7.6 Producing graphs

To make the Analysis files produced by the StatsAnalyzer more human readable and to summarize the results from multiple Analysis files, a Java program GnuPlotDataCreator was written that summarized the Analysis files and generated .data files that were readable by the graph plotting program gnuplot. Gnuplot is a command line driven multi-platform plotting program for data and functions [51].

The GnuPlotDataCreator reads all Analysis files in a directory and calculates the average mean time from all Analysis files of the same experiment. Taking the average mean from different runs of the same test case, .data files for plotting the average mean time spent in the simulation part for each of the four test cases can be produced. Furthermore, a .data file for a fifth graph averaging the mean times from the four test cases can also be produced to create a "total average" graph of the mean time spent in the simulation part. For the total average graph, the mean times from each test case were weighted with the number of measurements that were taken in the test case. The weighting was done in order to prevent short test cases (the two G2 test cases) with extreme time values to skew the graph since the short test cases
might not be representable for a normally played game from start to finish of Globulation2
(like Playground and strange2).

## 5.8 Summary

The computer game Globulation2 was parallelized to a mutex and STM version, and compiled with the Intel C++ STM compiler. An attempt to use LLVM and Tanger as the STM compiler was made, but since they were too unstable and could not compile the game, the Intel C++ STM compiler was chosen instead. It supported C++ features like C++ STL and irrevocable operations inside transactions which was need by Globulation2. A timing mechanism was introduced to the game so that the iterations of the simulation part of the main loop could be timed. Programs that analyzed the time measurements were also written.
6 Results

This Section starts by describing the issues that were faced when selecting the multi-core computer used for the experiments, and continues with describing the way experiments were conducted. Finally the obtained results and graphs are explained and discussed.

6.1 The multi-core computer used in experiments

Once the test suite was ready to be used to measure performance and scalability of the different game versions, the desire was to find a multi-core computer with as many cores as possible that could run the experiments. The more cores the computer has the more the benefits of parallelism are increased in the multi-threaded versions of the game. The computer with most cores that was made available for the project by the Distributed Programming Laboratory at EPFL was an 8-core computer with four Dual-Core AMD Opteron(tm) 8216 2.4 GHz processors with 64 bit architecture.

The workstation that was used to compile the different versions of Globulation2 with the Intel C++STM compiler was a single core Intel(R) Pentium(R) 4 CPU 3.00GHz processor with 32-bit architecture. Naturally, the Intel C++ STM compiler that was used produced IA-32 bit code for x86 (both Intel and AMD) processors. However, since the compiler version that was used on the workstation produced IA-32 bit code, the three Globulation2 versions would need to be recompiled to guarantee compatibility with the AMD64 instruction set used by the AMD processors of the 8-core machine. Intel does also provide a 64-bit version of the STM compiler that is compatible with the Intel64 instruction set, but unfortunately it does not support the AMD64 instruction set (the compiler could not be installed on the 8-core machine). The incompatibility is possibly intentional in order to deny the usage of the Intel compiler with Intel’s main competitor AMD.

Since the game could not be compiled to use the AMD64-bit instruction set, another way to run the game on the 8-core machine is to statically link all the libraries that the game uses, including e.g. C++ standard libraries, and then compile it to a 32-bit executable. 32-bit code can be executed on an AMD64 processor, but a 32-bit executable cannot dynamically link in libraries since normally there are no compiled 32-bit versions of the libraries available on a 64-bit machine.

The Intel C++ STM compiler provides a few different linker options for static linking, and these were experimented with in many different combinations in order to run Globulation2 on the 8-core machine. The -static-intel option statically links only the Intel provided libraries, the -static-libgcc option links the libgcc library statically and the -Bstatic option specifies that the following libraries are linked statically. The -Bstatic option was used e.g. together with the -m32 option (which generates code for a 32 bit environment) and the -pthread option (which provides multithreading support via the POSIX thread library). To link all libraries statically and prevent linking with shared libraries the Intel compiler also provides the -static option. To use this option, all intel libraries, C++ libraries like libstdc++, libgcc etc. and third party libraries like libpthread need to be available e.g. by adding them to the compile directory and adding the path of the directory to the library path LD_LIBRARY_PATH. However, compiling with the -static option produced a warning related to dynamic linking in the Intel library libitm.a, and running the game on the 8-core machine did not work because the Intel shared library libitmdyn.so could not be found (even though
it was in the correct path). Using different combinations of all the available static linking options it was possible to run a test program with transactions on the 8-core machine but not Globulation2, probably because static libraries of all required libraries did not exist or could not be created.

As an alternative approach to statically linking libraries to Globulation2, two tools to “statify” an executable were used on Globulation2: statifier and Ermine. The statifier tool creates one single executable file from dynamically linked executables and all the libraries they require [52]. In this way the single executable can be copied to different machines and executed without also requiring all the needed libraries, which is useful e.g. if static linking does not work or the source code for precompiled libraries is not available. The Ermine tool packs a GNU/Linux application together with any needed shared libraries and data files into a single executable so that it can be copied to and executed on any other GNU/LINUX machine without further modifications [53]. One of its highlighted features is that an executable statified on an x86 machine can be executed on x86_64 machine without 32-bit compatibility libraries. Unfortunately both statifying tools could only create an executable version of Globulation2 that produced a segmentation fault when executed on the 8-core machine.

Since there was so much trouble using the 8-core AMD64 machine, finally another machine with fewer cores also available at the Distributed Programming Laboratory at EPFL was chosen for the experiments. However, an account was received from the Center for Parallel Computers (PDC) at KTH to use the 32-core machine Key with 32 1.6 GHz cores of IA64 (Intel) type. But it turned out that the IA64 (or Itanium) instruction set of Key was different from the Intel64 instruction set which is the only 64-bit instruction set that the Intel C++ STM compiler supports, so Key could not be used for the experiments even though both the compiler and machine were from the same company. Finally, a quad-core (4-core) machine with an Intel(R) Core(TM)2 Quad CPU Q6600 @ 2.40GHz was used for the experiments.

6.2 The experiments

When a compatible multi-core computer had been found, the tests were run completely automatically in the following manner. Experiments were conducted using the four test cases G2enemy1min.game, G2enemy1sec.game, Playground7enemies7min.game, strange2enemy5min.game, measuring for each iteration of the main loop the times spent in the simulation part. The test cases were all run without graphics using the –nox command and with unlimited steps, i.e. until the game execution finished by itself because the player either won or lost a game. The number of threads were varied for the test cases up to the number of cores of the computer in order to measure scalability (beyond should reduce the benefits of parallelism). The sequential version of Globulation2 was only run with one thread because it does not create any threads, but for the parallel mutex lock and STM version the number of threads was steadily increased up to the number of cores of the computer (to four threads). Each execution of a test case was repeated three times for each thread count in order to have more accurate data and measure variance, since different game executions might produce somewhat different time measurements.

6.3 Experiment results

The first experiments of measuring the times spent in the simulation part of the game’s main loop were finally repeated 10 times in order to have more reliable results. For each test
case and thread count, an average mean time from all 10 repetitions of the test case was calculated. Figure 25 shows the total average time of all four test cases and Figure 24 shows the average time for each test case individually.

![Figure 25: Total average of all four test cases, with 10 repetitions of each test case.](image)

![Figure 24: Average mean time for the four test cases, with 10 repetitions of each test case.](image)

The four test cases in Figure 24 have quite different graphs and suggest that the type of game that is played, and thus the number of Players, units and buildings, affect the measured times. Test cases strange2 and Playground are much more dominant in the total average graph since they are played for a much longer time and thus have many more time measurements than the G2 test cases. Playground has many Players that all fight each other, while strange2 starts out with only two Players but with many units and buildings in the game. G2 is a test
case right at the end of a game with the remaining enemy Player severely weakened down to only a few of buildings and units. The test cases are also somewhat non-deterministic even though they always start with the same random seed, and at least for the Playground test case, it could on some rare occasions run for a very long time (up to 8 hours for the mutex version with three or more threads) before it completes. Usually though, it completes within three to four minutes, and the measured times are similar for both short running and long running executions and are not affected by how long time the test case needs in order to complete.

Based on the initial test results of the individual test cases in Figure 24 and that of the total average graph of Figure 25, there are two major issues with the performance of the STM version that are apparent in all graphs. The STM version performs much worse than the mutex version and it scales poorly when the level of parallelism is increased, i.e. when increasing the number of threads. It is clear that the STM version is much more sensitive to the number of threads than the mutex version. In overall the mutex version does not scale well either but it is much less sensitive to an increase in threads, and the total performance degradation between one to four threads is small (652 ms) compared to the STM version (1423 ms) in the total average graph of Figure 25.

The use of STM came with an overhead that was significantly larger than the overhead for using mutex locks when compared to the sequential version with one thread. The bad performance of the STM cannot be due to transaction aborts since only one thread is used and therefore there cannot be any conflicts. Since the granularity of mutex lock protected code sections and transactions is the same, the reason why the STM version performs worse than the mutex version with one thread must be because the overhead of handling the start and end of transactions, STM read and writes and handling the STM runtime in general is much larger than the overhead of handling locks in the mutex version. Also the overhead of handling threads and thread barriers is most probably one of the reasons for both the mutex version and the STM version performing worse than the sequential version.

Since the STM overhead was very large, the granularity of transactions in the STM version was likely affecting the scalability negatively, because the potential for speed-up when parallelism was increased did not outweigh the slow-down experienced due to the overhead of handling the transactions. More coarse-grained transactions could increase the performance in this case, although this cannot be concluded for general situations. Another likely reason for bad scalability is many switches to serial irrevocable mode, where transactions are serialized, not allowing for any parallelism. Furthermore, increasing the number of threads increases the chances of transaction conflicts to occur and the number of aborts to increase, which could also affect the scaling somewhat negatively.

Even though the Intel statistics mechanism was not working correctly (see Section 5.7.2), it could occasionally be used for the two short G2 test cases without receiving a segmentation fault and the statistics showed that there were no retries at all for the G2 test cases. However, the statistics mechanism was not entirely reliable and it was not certain if the statistics mechanism reported retries correctly. Also when using the Intel statistics mechanism the collected data was skewed by more than doubling the measured times spent in the simulation part. The overhead of gathering STM statistics skewed the time measurements, and therefore the graphs for the G2 test cases were not based on measurements where statistics were gathered during the execution of the test cases.

The bad scaling of the STM version could also be related to the reason why the mutex version did not scale well, and finding the cause for the mutex version not scaling well might help in improving the STM version. One reason for this could be that the threads have little work to do and spend a relatively long time waiting in barriers, so that the overhead of
handling threads outweighs the benefits of increased parallelism. However, load balancing is probably not a big issue since work is divided equally to threads in loops (see Section 5.3.2), so that all threads get approximately the same amount of work to perform. Perhaps the scalability could be improved if dynamic work assignment using e.g. a “queue of tasks” could be used.

As a conclusion, the STM version performed worse than the mutex version and did not scale well when the number of threads was increased, probably due to transaction overhead and the transaction granularity that was used, as well as switches to irrevocable mode and perhaps also transaction retries. The next Section discusses additional experiments that were conducted, and different ways to improve the performance and scalability.
7  Analysis

First some additional experiments are described that were conducted to better understand
the reasons for the bad performance of the STM version. Improvements to the performance
and scalability were identified, and some were implemented. To use SwissTM was one of the
possible improvements, and the issues that were faced when trying to use it are described.
Finally, suggestions for further improvements that can be made are given.

7.1  Additional experiments

The first results were bad, showing that the STM version did not perform well or scale
well. Therefore some additional experiments were conducted to better understand the reason
for the bad performance and to come up with ideas that could improve the performance.

7.1.1  The Intel C++ STM compiler vs g++

To see how much impact the compiler had on the results, the test suite described in Section
6.2 was performed, but with the experiments with the mutex version repeated with the g++
compiler as well to compare the performance of the two compilers. The results are shown in
Figure 26 which shows the total average mean time of all four test cases.

Comparing g++ with the Intel C++ STM compiler for the mutex version shows that the
compilers produce almost equally efficient executables. The Intel version scales slightly
worse beyond three threads, but with so few threads and because of the similarity in graphs,
whether or not the trend would continue beyond four threads is hard to say with certainty.
What can be said is that the difference between the compilers is not significant.

7.1.2  Measuring the STM overhead

Both the mutex version and STM version of the game had been shown not to scale well on
a 4-core machine, specially the STM version. To further see how bad they would scale if run
on a single core machine, and to see the difference in performance degradation between the
mutex version and the STM version, the test suite described in Section 6.2 was repeated on the single core Intel(R) Pentium(R) 4 CPU 3.00GHz machine that was used to compile the different game versions. The total average mean time of all four test cases is shown in Figure 27.

![Figure 27](image)

**Figure 27:** Total average of all four test cases on a single core machine.

Up to three threads the mutex version scales similarly to when run on the 4-core machine and has mean times close to when running on the 4-core machine, but after four threads it scales very bad. In overall the STM version scales very bad too. Why the STM version but not the mutex version performs so much worse on a single core machine than a multi-core machine is uncertain, but perhaps the overhead of handling STM increased as a result of less powerful hardware. The cache size of the processor in the 4-core machine is 4096 KB while the cache size of the processor in the single core machine is 1024 KB, and the reduced performance might be because handling the STM requires much memory which increases cache misses when there is less memory available. Even with one thread, the performance of the STM version degrades significantly compared to when run on the 4-core machine.

To further test the performance degradation, tests with up to 8 threads were performed on the 4-core machine. Only one run of each test case was performed instead of the usual three runs, and the results are shown in Figure 28 as the total average mean time of all four test cases.

![Figure 28](image)

**Figure 28:** Total average of all four test cases with up to 8 threads on 4-core machine.
Even though only one run of each test case was made, generally the STM version scales very bad beyond three threads. Much STM overhead seems to be added for each thread count, possibly because switches to irrevocable mode limit parallelism or perhaps because many conflicts occur and transactions have to abort. The usefulness of STM seems to be increasingly overshadowed by the overhead of handling an increased amount of threads. The results so far have indicated that there is a threshold at three threads after which the performance degrades more strongly for both the mutex and the STM version, and increasing the number of threads beyond three threads comes with higher performance penalties.

7.1.3 Median times vs mean times

When looking at the time measurements in the statistics files of the experiments, it seemed like there were some occasional extreme values measured that were much longer than the average time measurements, especially for the STM version, and they seemed to appear somewhat randomly. E.g. in Figure 29 we see the Analysis file created by the StatsAnalyzer program for a statistics file from a run of the strange2 test case with four threads using the STM version of Globulation2. The longest time measured was approximately 83 ms but the mean and median times were much shorter, approximately 7 ms and 5 ms respectively. The occasional extreme values also affected the standard deviation making it quite high with respect to the average mean time (93.5 %).

<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Analyzed 7134 time measurements (iterations of simulation phase).</td>
</tr>
<tr>
<td>Total execution time was 170 seconds.</td>
</tr>
<tr>
<td>Max: 83423 μs</td>
</tr>
<tr>
<td>Min: 1214 μs</td>
</tr>
<tr>
<td>Arithmetic mean: 6883.390244 μs</td>
</tr>
<tr>
<td>Median: 5299 μs</td>
</tr>
<tr>
<td>Standard Deviation: 6436.841506056398 μs</td>
</tr>
<tr>
<td>Standard Deviation in percent of arithmetic mean (sd/mean): 93.5 %</td>
</tr>
</tbody>
</table>

Figure 29: Analysis file with evidence of extreme values. Times are given in microseconds.

The occasional extreme values had the effect of increasing the mean time more than the median, and to see whether the median times scaled better than the mean times when increasing the number of threads, graphs were produced for the average median times as well. Figure 30 shows the average median and mean times for the normal test suite described in Section 6.2.
The results showed that the graph of the median times followed the graph of the mean times and also scaled bad, and that the median times where much shorter than the mean times for the STM version. The difference between median times and mean times was much larger for the STM version than the mutex and sequential version of the game, which had very similar mean and median times. The occasional extreme values were identified as being a small reason why the STM version scales poorly, but they are not the only reason. Why they were more profound in the STM version is not clear, but perhaps they occurred because of the overhead of handling STM e.g. if the STM runtime switched to irrevocable mode or there were many transaction aborts.

7.1.4 Fine-grained timing

The simulation part of the main loop of Globulation2 had four different classes that used threads for parallel computation: the Game, Map, Team and TeamStat classes. To further track down what in the simulation part was causing the parallel versions of Globulation2 to scale poorly, individual timing mechanisms were added to the parallel thread sections of the mutex and STM versions of the game. The timing mechanism is started in each of the four classes right before threads are created and their start routine is called, and stopped right after sequential computation is started again when all threads have returned and joined with the master thread. The fine-grained timing essentially times the parallel computation in each of the classes during each iteration of the main loop, and the output is written to a separate file for each of the four parallelized classes.

The execution of the test cases took much longer time after the fine-grained timing and writing to file was introduced in each thread section, e.g. test case strange2 which usually takes around four to fives minutes to complete now took between 10-30 minutes. Since it now would have taken much longer time to run the test suite of Section 6.2, experiments were only run for one short and one long test case: the G2-1min and strange2 test cases. Figure 31 shows the measured times in the four different parallel thread sections for the G2-1min test case, called Game, Map, Team and TeamStat after the class they were implemented in.
The G2-1min test case might not be representative of a normally played game due to its short execution time, but what can be seen for the test case is that the majority of time in the parallel sections is spent in the Team thread section, and that all thread sections generally scale bad. The Team thread section is the parallel section which has most significance for the overall measured time spent in the simulation part because it has the longest time measurements, and though all thread sections generally scale bad it is the Team thread section that matters the most. This becomes clearer when comparing the graph of the Team’s thread section with the total average for all test cases in Figure 25, which are very similar for the STM version of the game.

There is one exception however, the STM version’s Game thread section scales very well between one to two threads while the mutex version’s Game thread section scales poorly. But in each thread section the STM version comes with an extra overhead for the STM which makes the STM version always take significantly longer time than the mutex version even when the STM version scales better than the mutex version in the Game thread section.

The coarse-grained timing described in Section 5.7.3 that timed the simulation part was simultaneously used with the fine-grained timing, and the surprising results are shown in Figure 32.
The surprising observation is that with the added overhead of writing the fine-grained times to file make both the STM version and mutex version scale very well, even though the overhead is done completely sequentially outside the parallely computed sections. With four threads the STM version even outperforms the mutex version. However, the times spent in the simulation part increased roughly 100 times when the fine-grained timing was used and the good scaling should not be considered more than an interesting observation since in overall the game only slowed down when in fact it was speedup that was sought after.

The strange2 test case is a much longer test case played right from the start of the game and represents a more normally played game than the G2-1min test case. It is also the most dominant of all the thread sections in the graph of the total average mean time spent in the simulation part of Figure 25. The results of the fine-grained timing performed for all the four thread sections of the strange2 test case are shown in Figure 33.
For the strange2 test case only one run for each thread count and game version was made because the execution time took so much longer time when the fine-grained timing had been added. The variance was not considered to be that important and approximate results were considered enough to compare them to the results of the fine-grained timing of the G2-1min test case.

The results show that the thread sections Game, Map and TeamStat are somewhat similar in the way they scale for the STM and mutex versions, but Team is not. Most time is spent in the Team thread section as in the G2-1min test case, and the Team thread section actually scales well between one and two threads but then scales very poorly when increasing the number of threads further. In contrast to the strange2 test case, the G2-1min test case’s Game thread section scaled well but the Team thread section scaled poorly. As a conclusion, most speedup could be gained if the times in the Team thread section could be improved somehow.

As with the G2-min test case, times spent in the simulation part were measured simultaneously with the fine-grained timing and the results are shown in Figure 34. Similarly to the total average mean time in the simulation part measured for the G2-1min test case, the results were somewhat surprising. Both the STM and mutex version’s graphs fluctuate with the STM version even performing better than the mutex version for two threads. But as with the total average mean time graph of the G2-1min test case, it should not be considered more than a curiosity since only one run of the experiment was made for the strange2 test case and because the measured times increased significantly after introducing fine-grained timing, although not as much as with the G2-1min test case.
Now it was known that of all the parallel sections it was in the Team thread section where the majority of time was spent in, and that it was the part that scaled worst since it had the greatest difference in time between the STM version and mutex version.

### 7.1.5 Transaction retries

Many transaction conflicts that led to many transactions aborting and retrying might have also affected the scalability of the STM version negatively. The retries could be one of the reasons why the STM version’s mean times always were longer than the mutex version, and the bad scaling could be because the more threads that were used the more transaction conflicts occurred. Since the Intel statistics mechanism did not work properly (see Section 5.7.2), a custom counting mechanism was added to the game that counted the number of times transactions were restarted. In the retry counting mechanism, a flag is set when a transaction starts for the first time, and is reset only when the transaction commits, not when it aborts.

To add the retry counting mechanism the macros for the start and end of a transaction in the header file `Concurrency.h` were augmented. The macro for the start of a transaction was augmented with the code in Figure 35 directly following the code which starts a new transaction (see Section 5.6.1).

```c
__tm_waiver {
  txs[id]++;
  if(!txactive[id]) {
    txactive[id] = true;
  } else {
    txretries[id]++;
  }
}
```

**Figure 35:** Counting transaction retries.

The `__tm_waiver` keyword is used to run the code non-transactionally without instrumentation and guarantees that it will not induce any transaction conflicts by itself. The number of transactions a thread has started is saved in the `txs` array and is updated in the beginning of each transaction. The `txactive` array stores the transaction state of individual
threads. If it is false it is the first time the transaction is started and the transaction state is set to true, indicating that the transaction has now become active and will not become inactive until the transaction commits. If it is true it means that it was previously set to true by the transaction which aborted and now is retried, and so the txretries array which stores the number of transaction retries for each thread is incremented.

The macro for the end of a transaction was augmented with the code \texttt{txactive[id] = false;} directly after the transaction commits, so that the transaction does not become inactive when a transaction retries but only after it commits.

Output is written to file for each of the four thread sections Game, Map, Team and TeamStat only when a retry is detected. In each iteration of the simulation part, after the parallel computing in the thread sections has completed it is checked whether new retries have occurred since the last iteration of the simulation part. If no new retries have been detected, no output is rewritten. If one of the thread sections detect that new retries have occurred then the number of iterations of the simulation part, the number of transactions and the number of retries so far as well as the current retries/transactions quota are written to a file specific for that thread section.

The whole test suite of Section 6.2 was not run since the added writing of transactions retries increased the execution time. Only one of the long test cases strange2 was run once, varying the number of threads on the 4-core machine. For the four thread sections, there were only retries in three of them: Game, Team and TeamStat. The following numbers are from one successful run of the strange2 test case using four threads (note that there might have been more transactions but these were not recorded since the transaction and iteration count are only written to file when a retry is detected).

- Team had 18,777,172 transactions in 14,492 iterations, with 29,807 of the transactions being retries, i.e. about 0.16% of the transactions.
- Game had 3,458 transactions in 5,675 iterations, with 66 of the transactions being retries, i.e. about 1.9%
- TeamStat had 1,223,316 transactions in 8,576 iterations, with 751 of the transactions being retries, i.e. about 0.06%.
- Map had no transaction retries.

The number of retries is many for the Team thread section but the retry/transaction ratio is low, while for the Game thread section the number of retries is less but the ratio is higher. For the TeamStat thread section both the number of retries and the percentage is low. When running with one thread there were no retries as expected since there is only one thread executing transactions, so no transaction conflicts can occur. With two threads there were no retries for TeamStat or Map, and four retries in 2,594 transactions in Game and 7,615 retries in 18,998,620 transactions in Team. With three threads there were no retries for TeamStat or Map, and 33 retries in 4,381 transactions in Game and 23,294 retries in 16,995,270 transactions in Team. Even though the number of retries steadily increases with the thread count, the overall number of retries is not high compared to the number of transactions and is probably not the main reason for degraded performance.

Another possible reason for the bad performance of the STM version could be due to switches to irrevocable mode. It is uncertain if some of the retries could be because transactions in non-irrevocable mode have to abort and retry because some other transaction caused a dynamic switch of the STM runtime to irrevocable mode. Since the Intel’s statistics mechanism could not be used properly, there was no way of checking if the STM runtime
actually did switch to irrevocable mode or not. There is no interface to the STM runtime available to the programmer.

7.2 Improvements

Some of the improvements to the performance of the STM version that were identified were also implemented. These were to remove extreme values to better understand their impact and to write a more coarse-grained STM version.

7.2.1 Extreme values

What role the occasional extreme values that were detected when measuring times spent in the simulation part played in the poor scaling of the game was investigated next. The extreme values made the mean times for the STM version go up, and thus also the STM version to scale worse.

The standard deviation is calculated as a measurement of the result variance since different executions of the game might behave differently. If the variance of data is small, this is good since the results are more reliable. Many of the experiments (e.g. the test suite in Section 6.2) were repeated several times for each scenario to see the variance. An example of a run of the test suite that shows the standard deviation for the total average mean times is shown below. The standard deviation/mean time ratio for the sequential version is quite high, higher than for the mutex version, but the STM version’s ratio is also very high. In one of the test cases the ratio standard deviation/mean was about 0.98 for the STM.

Sequential: thrsds:1 mean:1105.48 sd:666.66 sd/mean:0.60 max:8802 mean/max:0.12
Mutex: thrsds:1 mean:2069.00 sd:791.91 sd/mean:0.38 max:14341 mean/max:0.14
Mutex: thrsds:2 mean:2271.38 sd:757.47 sd/mean:0.33 max:18400 mean/max:0.12
Mutex: thrsds:3 mean:2276.87 sd:649.55 sd/mean:0.28 max:17583 mean/max:0.12
Mutex: thrsds:4 mean:2720.84 sd:820.26 sd/mean:0.30 max:16760 mean/max:0.16
STM: thrsds:1 mean:4123.58 sd:1831.98 sd/mean:0.44 max:25804 mean/max:0.15
STM: thrsds:2 mean:4241.95 sd:1946.05 sd/mean:0.45 max:61559 mean/max:0.06
STM: thrsds:3 mean:4681.78 sd:2350.78 sd/mean:0.50 max:78358 mean/max:0.059
STM: thrsds:4 mean:5546.86 sd:4286.99 sd/mean:0.77 max:77965 mean/max:0.071

From the collected data it can be seen that the maximum recorded value is very high compared to the average mean, especially for the STM version. The observed occasional extreme values like the maximum value affect the mean value as well as the way the STM version scales since the extreme values get larger the more threads that are used.

To better understand how much the extreme values affect the performance, the StatsAnalyzer program was updated to filter out the extreme values. First the program calculates the average mean value \( m \) as before, but then it goes through all time measurements again and only keeps the time measurements if they are low enough. Different thresholds were used where a time measurement \( t \) was kept if \( m/t > \text{threshold} T \), where \( T = \{0.0, 0.15, 0.25, 0.5, 0.75\} \).

The test suite of Section 6.2 was run and the results of filtering out extreme values for all five thresholds are shown in Figure 36, where the normal unfiltered version has threshold 0 and all time measurements are kept. For the other thresholds, the majority of the time
measurements are kept and only a few extreme measurements are filtered out. E.g. with the highest threshold 0.75 the STM version with four threads keeps 16.141 out of 18.144 time measurements i.e. 89%, the mutex version with four threads keeps 30.815 out of 36.375 time measurements i.e. 85% and the sequential version keeps 14.016 out of 17.721 time measurements i.e .79%.

![Graph showing time measurements across different thresholds and thread counts.](image)

Figure 36: Filtering out long time measurements with different thresholds. Filtered measurements of the average mean time for all test cases is shown.

By looking at the resulting graphs the conclusion is that the extreme values do increase the mean value of the times measured and affect performance negatively, but they are not causing the bad scaling because the bad scaling remained even after filtering out extreme values. Figure 37 shows the comparison between unfiltered times where threshold 0.0 was used and filtered values with the highest threshold 0.75.
Since STM overhead had been identified as a potential reason for bad performance and scaling of the STM version, a more coarse-grained version of the STM version was written. The coarse-grained STM version used more coarse-grained transactions, encompassing more of the parallel computation in each transaction than the more fine-grained transactions that were used in the first STM version, from now on referred to as the fine-grained STM version.

In total the number of transactions was reduced from the initial 23 finer-grained transactions to seven coarser-grained transactions. In the Team class the number of transactions was reduced from 16 to two, and in class TeamStat from five transactions to three. The goal was to encompass as much of the parallel code as possible in as few transactions as possible to make the most coarse-grained STM version that was possible. However, to make one big transaction in each of the classes was not possible because of barriers and the need to synchronize threads due to code dependencies. The entry to the barriers was protected with pthread mutex locks and the barriers could not be encompassed by transactions. In the Game class the number of transactions was reduced from three to one, and the Map class only had one transaction which was left as it was. Figure 38 shows the results from running the test suite of Section 6.2 with the coarse-grained STM version and comparing it to the fine-grained STM version for the total average mean time for all test cases.

Figure 37: Comparison between 0.0 and 0.75 thresholds for filtering measured times in the simulation part.

7.2.2 Coarse-grained STM version

In total the number of transactions was reduced from the initial 23 finer-grained transactions to seven coarser-grained transactions. In the Team class the number of transactions was reduced from 16 to two, and in class TeamStat from five transactions to three. The goal was to encompass as much of the parallel code as possible in as few transactions as possible to make the most coarse-grained STM version that was possible. However, to make one big transaction in each of the classes was not possible because of barriers and the need to synchronize threads due to code dependencies. The entry to the barriers was protected with pthread mutex locks and the barriers could not be encompassed by transactions. In the Game class the number of transactions was reduced from three to one, and the Map class only had one transaction which was left as it was. Figure 38 shows the results from running the test suite of Section 6.2 with the coarse-grained STM version and comparing it to the fine-grained STM version for the total average mean time for all test cases.

Figure 38: Comparison between coarse-grained locking and fine-grained locking.
The coarse-grained STM performed much better than the original fine-grained STM, but was still far from competing with the mutex lock version (the original mutex version with finer-grained locking). Times spent in the simulation part are reduced with the coarse-grained STM version and the coarse-grained STM version also scales better (but still not good), probably because there is less transaction overhead with the start and end of a transaction which might be one of the reasons for bad scaling.

To compare the coarse-grained and fine-grained STM versions further, the number of retries was counted for one of the long test cases strange2 which was run once with 4 threads on the 4-core machine. For the four thread sections, there were only retries in two of them: Team and TeamStat. The Game thread section did not have any retries at all after reducing the number of transactions from three to one, and the Map thread section did not have retries as in the fine-grained STM version. The following numbers are from one successful run of the strange2 test case using four threads (note that there might have been more transactions but these were not recorded since the transaction and iteration count are only written to file when a retry is detected).

- Team had 123,271 transactions in 15,368 iterations, with 327 of the transactions being retries, i.e. about 0.27% of the transactions.
- TeamStat had 65,297 transactions in 15,236 iterations, with 545 of the transactions being retries, i.e. about 0.83%

The number of retries and (recorded) transactions for the Team thread section were dramatically reduced with the coarse-grained STM version compared to the fine-grained, but the retries/transaction ratio was very similar. The TeamStat thread section had much less (recorded) transactions with the coarse-grained STM version than with the fine-grained, but the number of retries was only somewhat decreased which is the reason why the retry/transaction ratio increased much.

It is hard to say whether the number of retries is responsible for causing the bad performance of the STM version, and that reducing the number of retries is the panacea. It could also be possible that the number of retries is a result of bad performance caused by something else since the number of retries is not high compared to the number of transactions. E.g. many transaction aborts due to many switches to irrevocable mode. A working Intel statistics mechanism would have helped in answering this question.

### 7.3 SwissTM with the Intel C++ STM compiler

At the end of the project, after all tests and improvements had been carried out, an experimental version of the Intel C++ STM compiler that used SwissTM as the underlying STM library had been finished. The experimental version of the Intel compiler did not have all features fully implemented and all support was not complete, e.g. the statistics mechanism was not working at all. The modified compiler was tried on the simple test program shown in Figure 39.
Unfortunately even the simple test program could not be compiled using the Intel compiler with SwissTM as the underlying STM library, not to mention the STM version of Globulation2. Compiling generated a compile error about an undefined reference to an anonymously named transactional function. The conclusion made by the developers of the experimental SwissTM compatible Intel compiler was that there was something in Intel’s STM library that they did not have. It seemed as if the Intel STM developers had added support for STL in their STM library, but without documenting it. Finally the SwissTM STM library could not be used to evaluate the difference between a STM version of Globulation2 that used SwissTM and a mutex lock version of Globulation2.

7.4 Suggested improvements

This Section briefly discusses some ideas and suggestions that could further increase the performance of the STM version and make it scale better when increasing the number of threads.

Transaction granularity was shown to be important for the performance of the STM version of Globulation2, where the coarse-grained STM version outperformed the fine-grained version. If the number of transactions could be reduced even further than seven, then the performance could maybe also be improved. This would require re-thinking how threads synchronize and how to handle code dependencies.

Work distribution amongst threads was handled by dividing work evenly amongst threads in for loops to be able to quickly parallelize the code and to make as few changes to the original code as possible. Another approach to handling the work distribution is to implement a “thread pool” where idle threads take tasks from e.g. a task queue or heap with work that needs to be done. This would ensure that threads would only work when they actually have something to do and work partitioning would be efficient in the sense that a thread that finishes a task faster than a slower thread can start working with a new task and do more work without having to wait for the slower thread to finish its task. This approach would however require a considerable amount of work in order to restructure the code in the simulation part of the main loop to tasks, and it is uncertain if it would make either the mutex version or the STM version perform better.

The parallel computation in the game was divided in four different classes that all created threads. The initialization and decommissioning of threads is time consuming, and it could be one of the reasons why both parallel versions of the game perform worse than the sequential

```c++
#include <list>
#include "tm.h" //SwissTM

int main(int argc, const char** argv)
{
    std::list<int> aList(4, 9);
    __tm_atomic {
        std::list<int>::iterator it = aList.begin();
        aList.erase(it);
    }
    __tm_waiver printf("Foo Works!\n");
    return 0;
}
```

Figure 39: Simple program that could not be compiled with SwissTM and the Intel C++ STM compiler.
version when using only one thread. To restructure the code so that threads are created in only one thread section could bring the performance of the parallel versions closer to that of the sequential version.

It was never determined exactly what parts of the code were causing the transaction retries, and if these could have been identified the code might have been possible to change so that fewer retries would occur and the performance of the STM would improve. One possible reason for some of the transaction retries could be the way in which lists were used in the game. Many of the buildings and units in Globulation2 are stored in lists and to access a single specific element in the list, the whole list might need to be protected by a transaction in order to safely iterate through the list to find the element. If the lists were instead changed to e.g. maps, then when accessing specific elements in the map the whole data structure might not need to be transactionally protected for as long time because there would not be a need to iterate through all elements of the map. If transactionally protected lists give rise to many transaction aborts, then using a map could decrease the number of aborts and increase performance.

Instead of using a compiler to transactify the code, manual instrumentation could have been used to transactify the code to try to make the STM version more efficient. This is however, a very tedious and time consuming approach for such a large application, and it is not certain whether or not it would have been possible to do. It is also uncertain whether it actually would have improved performance or not.

Changing the STM library from Intel STM to SwissTM could also have produced better results since SwissTM has shown to be good in comparison to other STMs. The Intel STM had already been used in Atomic Quake project and their results had shown that implementations using Intel STM did not perform or scale well compared to a mutex lock version. Even though the compiler version used in this project was a newer version than the one used in the Atomic Quake project, the overall results of this project were no different than those of the Atomic Quake project. If in the future SwissTM and the Intel C++ STM compiler combination works better, than compiling with SwissTM could produce STM game versions that perform better.
8 Summary and Conclusion

In this degree project the use of STM was studied in a multi-player computer game. The non-concurrent open source computer game Globulation2 was modified to one concurrent version that used mutual exclusion locks to protect shared data, and to two concurrent versions that used STM and transactions to protect shared data on different transaction granularity levels. An evaluation of current existing STM compilers was made to see how a program should be transactified in order to be compiled by the STM compilers, and whether the STM compilers could be used to compile the different game versions. LLVM and Tanger was too unstable and lacked support for some C++ features that were needed by the game so it could not be used as the STM compiler. The gcc-tm could not either be used as the STM compiler because there was no implementation available. The only one that could be used to compile the game was the Intel C++ STM compiler, and the STM game versions were finally transactified using the Intel STM. An experimental version of the Intel C++ STM compiler using SwissTM as the underlying STM library was also tried out, but it could not be used since the implementation was not fully functional.

8.1 Summary of Results

Performance data was gathered from the different game versions by timing each iteration of the simulation part of the game’s main loop, and tests showed bad performance for the STM version and that it did not scale well when increasing the number of threads. Moreover, the results showed that both the mutex version and the STM version performed worse than the sequential version when running with one thread in part due to overhead of handling threads and thread barriers. The STM version performed much worse than the mutex version due to the extra overhead of handling transactions and the STM runtime. When increasing the number of threads the mutex version did not scale well, but the STM version scaled much worse. The number of transaction retries was measured for different parallel sections and was not high when comparing the number of transactions to the number of retries. Between 0.05 to 2 % of all transactions were retries depending on which parallel section was measured.

Since the STM overhead was large, transaction granularity was identified as a reason for affecting performance and scalability negatively, and a coarser-grained STM version of the game was written which reduced the original number of transactions in the code from 23 smaller ones to seven larger ones. The coarser-grained version both performed significantly better and scaled better than the finer-grained version, but still worse than the mutex version. The ratio retries/transactions of the coarser-grained version was similar to the finer-grained version, but the coarser-grained version had less retries in total. The overhead of handling the start and end of transactions was one of the major reasons for bad scaling, which made the coarse-grained STM version perform better than the fine-grained STM version. Although in this case coarser transaction granularity improved performance, it cannot be concluded for general cases that using coarser grained transactions improve performance because they also limit parallelism.
Another likely reason for bad performance and scalability was due to the STM switching to irrevocable mode, but the Intel statistics mechanism for monitoring switches to irrevocable mode was not working properly and it could not be used to measure the number of switches to irrevocable mode. The overhead for handling transaction retries was also identified as a possible reason for affecting the scalability negatively, but was probably not the significant factor.

8.2 Lessons learned

Software Transactional Memory is a new and promising concurrent programming technique that recently has been an active area of research. STM promises much in being easy to use and avoiding typical problems with writing concurrent programs, and it delivers on most of its promises. To put a piece of code inside a transaction knowing that the code will not deadlock is both reassuring and easy, but my experience working with this project has showed me that STM comes with some other issues.

Although when using transactions there was no need to worry about how concurrent accesses to the same data might interfere with each other, using transactions was in some areas still complex. If code that is not parallel is to be transactified, threads have to be introduced and work partitioned amongst threads which for large applications is non-trivial. Code dependencies have to be identified and respected and parallelism has to be introduced to code that actually does make sense to parallelize. If on the other hand the code is already concurrent, or a concurrent application is written from the beginning with STM in mind, the use of STM (or any other parallel programming mechanism like e.g. locks) becomes more natural because these issues can be addressed already in the design phase of the application.

Comparing the programmability between STM and using locks, it is easier to use STM, even though with locks one faces the same design issues as with STM when parallelizing code. The code has to be investigated to understand what it does, how it behaves and what data it accesses. Shared data has to be identified and the amount of code to protect from concurrent accesses needs to be decided, including when to release ownership of locks. Code dependencies have to be found and upheld. Independent tasks can be computed in parallel, but the order of dependent code has to be maintained not to have incorrect or erroneous code.

However, there are some additional design issues that have to be addressed with locks but not with STM, which makes STM more straightforward to use. When locks are used, the number of locks to use has to be decided, and this introduces an additional dimension of complexity compared to STM, since STM has only one type of transaction. In this project only one global lock was used in the mutex version of the game to keep the implementation as similar to the STM version as possible. If few locks are used there will be less lock overhead, but can lead to worse performance with multiple concurrent threads because there is more lock contention. Conversely, having many locks decreases lock contention and increase the potential for parallelism and speed-up, but it also increases lock overhead. One of the challenges with using locks is to find the optimum number of locks to use and the optimum balance between fine-grained and coarse-grained locking that maximizes performance.

Using many locks also increases the risk for deadlocks if threads attempt to lock the same set of objects in different order, a problem which cannot occur when using STM because there are no locks and only one type of transaction. When using locks priority inversion can also occur, but not with STM because it is the job of the STMs contention manager to decide what transaction is allowed to complete. Convoying can happen with locks, but is also avoided with
STM since there are no locks and transactions proceed without requiring resources held by other threads.

The conclusion is that implementing parallelism with locks and STM is very similar, but STM is less error prone, requires less synchronization efforts and is easier to use. The biggest design choice with STM is how many transactions to use, and the results of this project show that fewer and larger transactions gave better performance, although this cannot be seen as a general fact since coarser transaction granularity also reduces parallelism which can decrease performance. Furthermore, from this project it was also learned that the advantages of STM come with the price of overhead handling the STM, which reduces performance compared to using locks. Locks have been optimized and used for a much longer time than STM, and today most hardware has been more optimized to handle locks than TM. If hardware were instead produced to optimize TM, hybrid TM implementations combining hardware with STM might be much more competitive performance wise with locks. Looking at STM in this way, the comparison with locks might seem less fair since STM is still an active research area and I believe it has yet to show its true potential.

Working with this project it became clear why STM still has not become widely used commercially. STM is still heavily researched and most implementations are experimental and are not commercially developed. All of the STM tools that were used in this project, the STM libraries and the STM compilers, are all active research projects. They are compatible with only a limited number of other technologies, are buggy or do not support basic functionalities that most programmers take for granted and expect to use if they were to program applications using STM.

Even though writing transactional code might be easy, it is a whole other thing to replace regular memory operations with transactional memory operations. Instrumentation can be done manually by the programmer without requiring sophisticated compiler tools, but this can be very tedious if working with large applications, and honestly is something that a programmer should not need to do. Therefore STM support from compilers is very important for STM to have a widespread success. Unfortunately today’s available compilers, like LLVM with Tanger and the Intel C++ STM compiler, are not quite there yet.

LLVM and Tanger still have much to do before they can be used as freely as the Intel C++ STM compiler. Tanger needs to support irrevocable actions and C++ language features better, something which the Tanger developers are currently working with. Dynamic memory allocation and deallocation with new and delete is needed, handling exceptions inside transactions, printing and other irreversible operations are important to deal with, and C++ STL support is much needed.

The Intel C++ STM compiler was relatively easy to use and had good support for many C++ language features, but it was not working completely correct. Some cases which the compiler should have been able to handle produced strange compile errors, and an important feature such as the Intel statistics mechanism was not working correctly, although it is not needed to compile and execute programs. The STM annotations provided by the Intel C++ STM compiler introduces an additional design issue not inherent e.g. with the LLVM and Tanger. The STM annotations provide a way for the programmer to optimize code, but they also introduce more complexity if the feature is to be used correctly. When annotating e.g. a function with a STM keyword, to determine the correct annotation the programmer must know exactly what other STM annotated functions are being called and what data is being accessed by the function, and whether the accessed data is local for the function or not. Fortunately the compiler provides much help in resolving these issues with compile warnings.
The problem with the Intel C++ STM compiler was rather that the performance was not good, and if STM cannot compete with locks performance wise in commercial applications, than its use will be limited outside the research community. More over there needs to be some unity between researchers of what the best characteristics of a STM system should be so that standards can be made and the many compatibility issues that exist today can be solved. If STM cannot be used in the application one is developing because the tools to integrate the STM are not compatible with either the STM or the application, then what use is there for the STM? Although STM was studied exclusively in this project with tools using the C or C++ programming language, other programming languages like Java and Haskell have also been used before to study STMs. What role the programming language has for the performance of STMs was not studied in this project, but in order for STM to have a more wide-spread success it should not be limited to any one programming language. If these issues can be solved the potential for STM seems enormous. Especially with the constant improvements of parallel computation capabilities in today’s multi-core computers, the demand for programmer friendly ways to parallelize code will surely increase.

In this project the obtained results of the STM version performing worse than the mutex version and not scaling well when increasing the number of threads, were somewhat expected since similar results had been obtained in the similar Atomic Quake project. The hope was that the STM version would have performed better than the mutex version, or at least scaled well. The reason for bad performance was given different possibilities such as switches to irrevocable mode, transaction overhead for the start and end of transactions which favored coarse transaction granularity, and to some extent too many transactions retries. If SwissTM could have been used it could have lead to better performance of the STM version. In either case the question whether SwissTM could be used with other existing technologies in real world applications without requiring manual instrumentation by the programmer was answered, and unfortunately existing technologies did not yet allow SwissTM to be used so freely. If many of the compatibility issues that were come upon during the course of this project were to be solved it would be interesting to see whether the same results would still be obtained, or if the STM version of Globulation2 finally would perform or scale well.

8.3 Future work

A recommendation for similar future projects where a sequential game is to be parallelized is to concentrate the parallelization to the simulation part of the game’s main loop, since that is most probably where the majority of computation is done. If a programmer is to choose between using locks or STM to protect concurrent accesses to shared data, there are several reasons which favor using locks. If good performance is important, which it is for many games, locking is better to use than the STM systems that were evaluated in this project including LLVM and Tanger, and the Intel C++ STM compiler with the Intel STM or SwissTM. Using locks can also be more reliable since there is no need to worry about compatibility issues between STM compilers and STM libraries or whether they can compile the application. However, if these issues are not so important and the programmer is choosing between using multiple locks and STM, then STM can be easier to use since it avoids problems associated with locks like e.g. deadlocking and synchronizing access to shared data is much easier than with locks.
The different game versions developed in this project can be used in future benchmarking, and the calls to the underlying STM library are easily changed in the STM version of the game to benchmark other STMs. For example, if SwissTM can be made fully functional with a STM compiler then it is possible to benchmark SwissTM using the code from this project and compare it to the mutex and sequential version of Globulation2.
9 References


[45]. Cohen, A. GCC for Transactional Memory. [Online] [Cited: July 31, 2009.]
http://www.hipeac.net/node/2419.


http://www.intel.com. 318253-001US.

[51]. Gnuplot Homepage. Gnuplot. [Online] [Cited: September 14, 2009.]
http://www.gnuplot.info/.

[52]. Reznic, Valery. Statifier Main page. [Online] [Cited: September 15, 2009.]


http://hg.globulation2.org/glob2-new/?f=983c6bea60a3;file=doc/main%20engine%20loop.txt;style=raw.
