Rui Cachopo

Software Transactional Memory implementation and benchmarking in Go

Computer Science Tripos – Part II

Churchill College

May 5, 2024

Declaration

I, Rui Cachopo of Churchill College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

I, Rui Cachopo of Churchill College, am content for my dissertation to be made available to the students and staff of the University.

Signed:

Date: May 5, 2024

Acknowledgements

The comparatively small margins of this page cannot possibly contain all acknowledgements due in this dissertation. I give instead a much paler summary, and hope my gratefulness can be shown in some other way.

First of all, I would like to thank Tim Harris for supervising this project. My work would have been much less interesting, and this dissertation much less coherent, without his guidance and support.

I would also like to thank John Fawcett and Matthew Ireland, for making sure I stayed mostly on track for the last three years.

Many thanks are due to my parents, for leading me most of the way to the finish line. To my mother, for all her support and willingness to help. To my father, for letting me stand on his shoulders when producing this work.

I would like to thank AN for always being present and willing to hear about my work, and RM for coercing me to step away from it occasionally. Finally, I would like to thank them both for helping me keep a vestige of sanity.

Proforma

Candidate number:	2399F
Project Title:	Software Transactional Memory implementation and benchmarking in Go
Examination:	Computer Science Tripos – Part II, 2020
Word Count:	11970[1]
Lines of code:	5234[2]
Project Originator:	2399F
Supervisor:	Tim Harris

Original aims of the project

The core goals of the project were to implement a Software Transactional Memory and to make a meaningful evaluation of the system produced. This evaluation involves both quantitative benchmark results and qualitative results related to the usability of the system. The technical deliverables are implementations of the STM and the benchmark produced — as there is no existing realistic benchmark in the language chosen for this project.

Work completed

All the success criteria of the project have been met. I implemented the Go Versioned Software Transactional Memory and demonstrated its correctness and performance by producing an implementation of the STMBench7 benchmark and accompanying tests. Due to the difficulties described in the section below I had much less time than I had hoped to work on extension tasks.

The completion of this work required substantial research into Transactional Memory and Go's concurrency primitives and memory model. Implementation of an application as large as the STMBench7 benchmark also required knowledge and use of software engineering patterns and good practices.

Special difficulties

In November 2019 I had a climbing accident that resulted in a dislocated elbow. This halted project work, and in fact all academic work, for a few weeks until I was able to use a keyboard again.

Starting in March 2020, the COVID-19 pandemic forced me, and indeed the vast majority of students, to work from home for the rest of the academic year. This meant that the last month of work on the project and this dissertation were completed in an environment much less productive than expected.

[1] Calculated using TeXcount: app.uio.no/ifi/texcount

[2] Calculated using Github's "Contributors" feature.

Contents

1 Introduction **1**
- 1.1 Motivation and previous work . . . 1
  - 1.1.1 The problem . . . 1
  - 1.1.2 Possible solutions . . . 2
  - 1.1.3 Transactional memory . . . 3
- 1.2 Golang . . . 4
- 1.3 The STM chosen . . . 4
- 1.4 Evaluation and benchmarking . . . 4

2 Preparation **6**
- 2.1 Starting point . . . 6
- 2.2 Requirements analysis . . . 7
  - 2.2.1 Functional requirements . . . 7
  - 2.2.2 Quality requirements . . . 8
- 2.3 Research: using Go to implement an STM . . . 9
  - 2.3.1 Polymorphism . . . 9
  - 2.3.2 Object semantics . . . 10
  - 2.3.3 Errors and panics . . . 11
  - 2.3.4 Goroutines . . . 12
  - 2.3.5 Thread-local storage . . . 12
  - 2.3.6 Memory model . . . 12
- 2.4 Engineering methodology . . . 13

3 Implementation **14**
- 3.1 STM concepts . . . 14
  - 3.1.1 Conflicts . . . 14
  - 3.1.2 Aborts and retries . . . 15
  - 3.1.3 Lifetime of a transaction . . . 15
  - 3.1.4 Different methods of dealing with conflicts . . . 15
- 3.2 STM interface in Go . . . 16
  - 3.2.1 Typing . . . 16
  - 3.2.2 Transaction representation . . . 18
  - 3.2.3 “Transactional” functions . . . 18
  - 3.2.4 “Atomic” execution . . . 19
- 3.3 GVSTM . . . 19

3.3.1 Versioned data structures . . . 19
3.3.2 Algorithm overview . . . 20
3.3.3 Validation and commit algorithms . . . 21
3.3.4 Read-only transactions . . . 22
3.3.5 Garbage collection of old versioned bodies . . . 23
3.3.6 Extensions . . . 24
3.4 STMBench7 . . . 25
3.4.1 Collection of results . . . 25
3.4.2 Data structures . . . 26
3.4.3 Operations . . . 26
3.4.4 Design patterns . . . 28
3.4.5 Testing . . . 29
3.4.6 Extensions . . . 29
3.5 Repository overview . . . 30

4 Evaluation **32**
4.1 Success criteria . . . 32
4.2 Benchmark results . . . 33
4.2.1 Workloads without structure modifications . . . 33
4.2.2 Introducing structure modifications . . . 34
4.3 Usability . . . 35

5 Conclusion **39**

Bibliography **39**

A Comparison of operation executors **A1**

B STMBench7 example output **B1**

C Benchmarking methodology **C1**

D Project Proposal **D1**

List of Figures

3.1 The versioned box data structure. 20
3.2 The STMBench7 data structure. 27

4.1 Total throughput of a read-only workload. 34
4.2 Total throughput of a read-dominated workload. 35
4.3 Total throughput of a read-write workload. 36
4.4 Total throughput of a read-dominated workload with structure modifications. 37
4.5 Total throughput of a read-write workload with structure modifications. . . 38

List of Listings

1.1 Synchronisation by convention. 2
2.1 Composability. 8
2.2 Interface types. 10
2.3 `error` and `panic`. 11
3.1 The STM interface. 16
3.2 A transactional counter. 17
A.1 The GVSTM operation executor. A1
A.2 The coarse-grained operation executor. A2

Chapter 1

Introduction

This dissertation describes the implementation and benchmarking of a Software Transactional Memory (STM) in Go, a programming language that emerged just as research on STMs dwindled. In this chapter I aim to give a brief historical background of STMs and explain the choices I made when proposing the project. In particular, I explain the choice of STM algorithm, programming language and benchmark to use.

1.1 Motivation and previous work

To have some context on the motivations behind STMs, it makes most sense to first remember the historical trends in processor design. From there, we can see how modern hardware limitations caused a revolution in software and the problems that this caused to programmers. Finally, we can see how STM tackles these problems.

1.1.1 The problem

In the last few decades, and especially in the period of 1986–2003, single-core processor performance grew exponentially at such a rate that hardware became obsolete within one or two years. This was mostly thanks to two "laws" of processor design: Moore's law [**?**] and Dennard scaling [**?**]. Moore's law is the observation that, due to the production of smaller transistors, the number of transistors in a processor doubles every two years. Although the prediction in the 1965 paper was a doubling in transistor count every year, this was revised in 1975 to a doubling every two years. Dennard scaling states that as transistors get smaller and consume less power, their power density stays constant, so the power consumption of the whole processor remains constant.

The combination of these laws impacts processor performance in three main ways:

- Smaller transistors can switch faster, meaning that the clock frequency of the processor can be increased.
- The higher transistor budget allows for more complex hardware design. One way this was exploited was the use of pipelining, which reduces the critical path inside processor and allows the clock frequency of the processor to increase.

- More complex hardware design was also used to build out-of-order, superscalar processors. These processors could exploit Instruction Level Parallelism (ILP) [**?**, Chapter 3] latent in most programs.

This processor performance "free lunch" [**?**] ended in the early 2000s, when Dennard scaling ceased to hold and the low-hanging fruit of processor design had been harvested:

- In post-Dennard scaling, power consumption becomes a limiting factor in the complexity of a single core and on the clock frequency of the processor.
- The performance penalties caused by processor stalls, such as cache misses and mispredicted branches, increases with the length of the processor pipeline. This means deeper pipelines yield diminishing returns.
- The amount of ILP latent in most programs is limited. This means that increasingly complex superscalar hardware will also yield diminishing returns.

As the limits of single-core performance and latent ILP were reached, processor designers shifted their focus to multicore processors, which are capable of exploiting Thread Level Parallelism (TLP) [**?**, Chapter 5]. As opposed to ILP, which can be automatically exploited by the processor in most applications, TLP usually requires the program to be written using multiple threads. This means that the burden is now on programmers to write concurrent applications if they want to reap the benefits of increasing processor performance. Enter the software revolution towards concurrent programming.

1.1.2 Possible solutions

Locking. The model most commonly used in concurrent programs consists of multiple threads, which communicate through access to shared memory. Low level primitives such as locks are used to provide condition synchronisation and mutual exclusion when accessing shared resources. This model is sufficient for simple programs, but it does not scale to large applications. Problems such as deadlock and priority inversion are hard to detect and debug in large programs. The lack of a semantic link between a mutex lock and the resource it protects leads to what Herlihy and Shavit call "synchronisation by convention" [**?**, Chapter 18].

```
1 /* When a locked buffer is visible to the I/O layer BH_Launder
2  * is set. This means before unlocking we must clear BH_Launder,
3  * mb() on alpha and then clear BH_Lock, so no reader can see
4  * BH_Launder set on an unlocked buffer and then risk to
5  * deadlock.
6  * /
```

Listing 1.1: Synchronisation by convention. This Linux kernel comment demonstrates how real-world concurrent systems rapidly become complex.

Monitors and condition variables. Several approaches have been suggested as solutions to the problems present when using locks and shared memory. The most basic of these is the use of slightly higher level primitives, such as monitors and condition variables. Java is a good example of this: the JVM associates one monitor and one condition variable with each object. A `synchronized` code block runs inside a monitor, which can be used to provide mutual exclusion. The condition variables can be used for condition synchronisation using the `wait`, `notify` and `notifyAll` methods.

Message passing. Moving away from threads and shared memory, other concurrency models have been created. The most widespread involve message passing between strongly isolated processes — so called because, unlike threads, they do not share memory. Hoare's Communicating Sequential Processes [**?**] and Milner's Calculus of Communicating Systems [**?**] are two formal models that use this idea. These models are mostly employed by functional programming languages such as Erlang, which allow no shared memory and in which all data is immutable. Concurrent programs in these languages can be simpler to write and debug, but often do not perform as well as programs written in more "traditional" languages such as Java and C++.

Transactional memory. Yet another approach, and the focus of this dissertation, is Transactional Memory (TM) [**?**]. When using transactional memory, a programmer organises their code into transactions, similar to the transactions in the transactional model of databases. It is then up to the underlying TM implementation to provide atomicity and isolation of the transactions executed. That is, the memory operations of each transaction executed appear to occur atomically, and no transaction can see the results of a partial execution of any other transaction.

1.1.3 Transactional memory

Transactional memories provide a very simple interface to programmers: to write a correct concurrent program, place all parallel code that accesses shared data inside transactions, and let the underlying TM ensure the program is correctly synchronised. Transactional memories also provide composability, meaning it is simple to compose existing transactions to achieve more complex behaviour. As an example, consider that we want to dequeue an element from a concurrent queue Q_1 and enqueue it in concurrent queue Q_2, but without any other thread observing that the element is in neither or both queues at the same time. Attempting to solve this problem with locks would be fairly complex, involving either additional locks or exposing the internal locking logic of the concurrent queues. In a TM model, however, the solution is trivial: simply execute both operations in the same transaction. This is discussed in more depth in Section 2.2.1.

Transactional memories can be implemented in hardware or software. Hardware Transactional Memories (HTM) [**?**] are usually implemented on top of the cache coherence protocol of a processor, and are offered as an Instruction Set Architecture (ISA) extension. Because they operate at such a low level, HTMs are often very efficient, but the size of transactions is limited by the size of cache available and the implementation cost

is extremely high. Software Transactional Memories (STM) [?] are much more flexible, but implementing them efficiently can be a challenge. Efficiency concerns with regards to STMs are so great that they have been called a "research toy" [?].

1.2 Golang

In this project I aim to implement an STM and evaluate its performance. I have chosen to do this in Go[1] because it is a language that emerged after the wave of research on STMs in the early 2000s, and so there is no exploration of how the language's features can be exploited to increase the applicability of STM approaches. Additionally, Go is designed with concurrency as a primary focus. In fact, one of the primary uses of Go has been to build server applications, and there is a growing environment of tools and libraries to support this. Examples are Docker and Kubernetes implementations, the etcd key-value store system[2], the Elasticsearch-Logstash-Kibana stack for logging[3] and Prometheus for metric tracking[4]. One of my goals when choosing to use Go for this project was to eventually contribute to this environment, and make it easy for software engineers to build server applications using STM.

1.3 The STM chosen

Although there has been substantial research on the topic of transactional memory, and many STMs have been implemented, there are few applications that actually use them. From a software engineering perspective, there are many possible reasons for this, such as adoption costs and little to no support of commonly used libraries. As one of my goals in choosing my project was to one day develop an STM that could be adopted in industry, it made sense to base my project on an existing STM with a real-world application. Such an STM is the Java Versioned Software Transactional Memory (JVSTM)[5], which has been used in the FenixEDU project [?]. Because my STM uses the same version box–based algorithm [?] but is implemented in Go instead of Java, I have decided to name it Go Versioned Software Transactional Memory (GVSTM).

1.4 Evaluation and benchmarking

Performance is a crucial factor in the evaluation of STMs. Although it has been shown that STMs can largely outperform sequential code in multicore computers [?], it falls to each STM implementation to show that it is sufficiently efficient to be adopted. With that goal in mind, it is important for the success of this project to demonstrate how the

[1] golang.org
[2] etcd.io
[3] elastic.co/what-is/elk-stack
[4] prometheus.io
[5] web.ist.utl.pt/joao.cachopo/jvstm/

STM implemented performs when compared to alternatives, even if it does not outperform them.

With that in mind, I have chosen to include in the evaluation of my STM the results of the STMBench7 benchmark, which can be regarded as a stress test of STMs [**?**]. STMBench7 aims to be a realistic benchmark, meaning that instead of performing a long series of simple transactions, such as reading and writing a handful of variables, it performs elaborate operations on complex data structures. A description of the benchmark and its data structures and operations can be found in Section 3.4.

As of the start of this project, there is no Go implementation of STMBench7. This means that a considerable part of the work to be completed, both in terms of lines of code written and time investment, will consist of adapting the existing Java version to be implemented in Go. Because of the complexity and size of the benchmark, my proposal is to implement a subset of the original 45 operations. Completing the implementation of the benchmark is an extension goal of the project.

Chapter 2

Preparation

I begin this chapter with an overview of my relevant experience and work completed before the start of the project (Section 2.1). I then describe the requirements of an STM, from both a functional perspective and one of producing a system that can be adopted in a software engineering environment (Section 2.2). I also give an overview of the research conducted to become familiar with the features of Go that had a significant impact on the design decisions of this project (Section 2.3. I conclude the chapter by describing my engineering methodology, as well as details of the systems used (Section 2.4).

2.1 Starting point

Here I summarise knowledge relevant to the project that I acquired outside of the Computer Science Tripos. Within the Tripos, the courses most relevant to this project are Part IB Concurrent and Distributed Systems, which covers the basic shared memory synchronisation mechanisms, and the Multicore Semantics and Programming Unit of Assessment, which covers memory models and more advanced synchronisation mechanisms such as Transactional Memory.

Experience with Go programming. During my internship after Part IB of the Tripos I learned Go and used it to build server applications using the microservice architecture. My experience with Go here was mostly using both internal and external libraries, and did not involve using Go's concurrency primitives explicitly.

Knowledge of the STM algorithm. I was first introduced to the concept of Transactional Memory in the last concurrency lecture of the Part IB Concurrent and Distributed Systems course, which touched on a few topics. I was intrigued, so I spent some time reading about transactional memory, and about the JVSTM in particular. This gave me a good general understanding of the algorithm used, but I did not look into implementation specifics.

Prototype built for project proposal. Before proposing this project to my supervisor I created a prototype of the GVSTM, to verify that the task I was setting out to do was

feasible within the context of a Part II project. This prototype was just over one hundred lines of code and lacked essential features such as garbage collection of old versions — by far the most complex part of the STM algorithm. As discussed in Section 3.3.5, this meant that a program using the GVSTM would quickly run out of memory.

Access to JVSTM and STMBench7 codebases. The JVSTM codebase is freely available in its GitHub repository[1]. The STMBench7 benchmark is sadly no longer available at its homepage[2], but it is available in the JVSTM benchmarks repository[3].

2.2 Requirements analysis

The requirements of a transactional memory can be divided into functional and quality requirements. Functional requirements must be met for a TM to be classified as such and to provide correct synchronisation. Quality requirements should be met for a TM to be a viable alternative to other synchronisation mechanisms and to be adopted by software engineers. In this project I will make sure that the GVSTM meets the functional requirements. The quality requirements will be use as an aspect of evaluation of the GVSTM, but are not part of the success criteria of the project.

2.2.1 Functional requirements

ACID properties. Any transactional memory must provide Atomicity, Consistency and Isolation according to the ACID properties of the database transactional model. Atomicity guarantees that each transaction either commits or aborts fully — no transaction may conclude with only some of its memory operations successfully executed. Consistency guarantees that the result of executing a transaction on a consistent view of the program will result in a consistent view of the program. Isolation can be provided to varying degrees. In the context of the ACID properties, isolation guarantees that no committed transaction may have seen the results of a partial execution of another transaction. Durability concerns the resilience of committed transactions to system failures, often by using non-volatile storage. Because TM operates only in main memory, durability is not a concern today, though this may change in the future.

Opacity. In practice, modern transactional memories are expected to make stronger guarantees than those provided by the ACID properties. Opacity [**?**] is widely regarded as the correctness property to be achieved by TMs, and is stronger than properties imported from the area of database concurrency control such as serialisability and strict serialisability [**?**]. In summary, opacity strengthens the isolation condition so that no transaction, including aborted and in-flight transactions, can access inconsistent states. The GVSTM provides opacity by only allowing each transaction to access the state consistent with its read timestamp — see Section 3.3.1.

[1] github.com/inesc-id-esw/jvstm
[2] dcl.epfl.ch/transactions/wiki/doku.php?id=stmbench7
[3] github.com/inesc-id-esw/jvstm-benchmarks/tree/master/stmbench7

Serialisation order. Although not sufficient, the property of strict serialisability gives rise to a useful construct: the serialisation order of transactions. This is the sequential order in which transactions appear to occur. It is common to say, for example, that transaction A is serialised after transaction B.

Reading and writing transactional variables. In the general case, a transactional memory could provide the properties above to many kinds of operations, such as I/O and, in the case of Go, channel communication. Providing such a general solution from scratch, in addition to implementing a benchmark to meaningfully evaluate it, would be far beyond the scope of an undergraduate project. As such, I decided to focus on implementing an STM that provides the properties above only to reads and writes of transactional variables. Section 3.3.6 briefly discusses how the GVSTM could be extended to handle arbitrary operations transactionally. There is, however, a case to be made against using different concurrency models such as TM and channel communication in the same program.

Composability. One of the most appealing features of STM when compared to other synchronisation mechanisms is the ease of composition of transactions. As shown in Listing 2.1, given transactional methods to add and remove an element from a queue, a transactional method to move an element from one queue to another is trivial, as is the creation of an atomic version of said method.

```
1  func transactionalEnqueue(queue Q, element E) {...}
2
3  func transactionalDequeue(queue Q) E {...}
4
5  func transactionalMove(queueOne, queueTwo Q) {
6      element := transactionalDequeue(queueOne)
7      transactionalEnqueue(queueTwo, element)
8  }
9
10 func atomicMove(queueOne, queueTwo Q) {
11     Atomic(func() {
12         transactionalMove(queueOne, queueTwo)
13     })
14 }
```

Listing 2.1: Composability. The pseudocode presented here uses syntax similar to that of Go, but does not obey the syntax of transactional methods defined later in this dissertation.

2.2.2 Quality requirements

Performance. One of the principal concerns of STM designers is performance. Depending on the algorithm used, it may be necessary to keep large read- and write-sets or perform complex conflict detection operations, which may introduce significant overheads

(see Section 3.1.4). For software engineers to choose to adopt an STM it must demonstrate that its performance is at least comparable to that of other solutions, in addition to simplifying the programming model used. Performance may have different meanings in different applications:

- Some applications may require an upper bound on the latency of any operation.
- Some applications may prefer to minimise the latency of read operations at the cost of increasing the latency of write operations.
- Some applications may try to ensure maximal throughput, while giving no guarantees on the performance of individual operations.

Of the three examples here, the first is likely to occur in real-time applications, such as high-frequency trading or security-critical applications. Transactional memory is unlikely to be a sensible solution in these cases. The latter two examples are more likely to occur in server applications, in which STM is much more promising. The GVSTM in particular is heavily read-optimised, and guarantees low-latency read-only operations regardless of the presence of write-operations.

Usability. As discussed in Section 1.1.1, synchronisation using locks is difficult for a number of reasons. Programming using an STM should not only make it easier to write correct concurrent programs, but also be easy and intuitive to use and shorten iteration times for software engineers. Short of surveying a large number of software engineers with experience programming using multiple concurrency models, an evaluation of usability will be somewhat subjective.

2.3 Research: using Go to implement an STM

Part of the research necessary to complete this project was to become familiar with features of Go that are essential to the implementation of an STM and of a benchmark that is itself a significant software project. In this section I summarise these features, focusing on where they differ from Java's, the language originally used to implement the JVSTM and STMBench7. I will include small code examples, and I recommend that those unfamiliar with Go read this section. More detail on these topics can be found in Go's specification[4] and memory model.[5]

2.3.1 Polymorphism

Go, at the time of writing (version 1.14), does not offer generic programming, but supports polymorphism in the form of interface types. Interface types declare a set of methods — an interface. Any type that implements all methods of an interface is said to implement the corresponding interface type. See Listing 2.2 for an example.

[4] golang.org/ref/spec
[5] golang.org/ref/mem

A particularly interesting interface type is the empty interface. Because its method set is empty, all types implement it. This makes it useful for implementing libraries that must support operations on arbitrary types, because it allows the creation of a single method that supports all types instead of having distinct methods for each type. Compared to generic programming this has the disadvantage of requiring users to explicitly cast values to the particular types they expect said values to have. This is cumbersome but, sadly, the best solution available in Go at the moment.

```
1  type bird interface {
2      eat()
3  }
4
5  type duck interface {
6      walk()
7      quack() string
8  }
9
10 type mallard struct {...}
11
12 func (m mallard) eat() {...}
13
14 func (m mallard) walk() {...}
15
16 func (m mallard) quack() string {...}
17
18 type barnacle struct {...}
19
20 func (b barnacle) eat() {...}
21
22 func (b barnacle) honk() string {...}
```

Listing 2.2: Interface types. Type `mallard` implements `duck` and `bird`. Type `barnacle` implements `bird` but not `duck`. Both concrete types implement the empty interface `interface{}`.

2.3.2 Object semantics

Function arguments in Go are passed by value, including method receivers. This means that in the example in Listing 2.2, each time a method of the `mallard` type is called, a copy of the struct is made. If the method changes some fields of the struct the value held by the parent is unchanged. In large software projects, and indeed ones ported from an object oriented language, it is useful to have the concept of an object, in which updates on an object made by one part of the program can be seen by other parts.

To get such semantics in Go, it suffices to change the receiver type of methods to a pointer type. This means that instead of type `mallard` implementing interface type

`duck`, pointer type `*mallard` implements interface type `duck`. This is, in practice, more common in Go, and can be seen for example with the implementations of the `sync.Locker` interface type, which is the interface implemented by locks.

2.3.3 Errors and panics

For the sake of brevity, I will assume the reader is comfortable with Java's exceptions, and with the distinction between checked and unchecked exceptions. Because exceptions can alter the flow of execution, they raise some questions with STM implementations:

- If a transaction causes an exception, should it commit or abort?
- If the former, will the transaction maintain the consistency property mentioned in Section 2.2.1?
- If the latter, will the condition that caused the exception still hold and be reproducible?
- Should a transaction be able to raise an exception due to seeing an inconsistent state — if the STM does not provide opacity?

Go's analogous to checked exceptions are values of type `error` — ordinary values that can be manipulated, passed as arguments and returned by functions. Due to Go's support of multiple return values, functions that may generate an error `error` typically return it as the last return value. It is expected that the caller of a function that returns an `error` will check that error is `nil`, similarly to the handling of exceptions in Java.

Go's analogous to unchecked exceptions are calls of the `panic` function. A call of `panic` shifts execution up the call stack until the program exits, reporting the error condition, or the `recover` function is used to stop the panicking sequence. Calls of `panic` should only be used when unexpected errors occur — it is unidiomatic to use calls of `panic` for control flow. Listing 2.3 shows an example of the use of `error` and `panic`.

```
1 func main() {
2     _, err := os.Create("/file/path")
3     if err != nil {
4         panic(err)
5     }
6     // Calculate results and write to file
7 }
```

Listing 2.3: `error` and `panic`. In this case failure to create the file is sufficiently unexpected that we want the program to terminate.

2.3.4 Goroutines

Go supports goroutines, which are essentially light-weight threads. Goroutines are inspired by coroutines, but are subtly different, particularly because they imply concurrency and coroutines do not. I assume that the reader is familiar with threads, so instead of explaining goroutines from scratch I will describe the main aspects in which they differ from threads and that are relevant to this project:

- The only memory associated with a goroutine is its dynamically sized call stack. This is a lot cheaper than threads in Java, which have a large, fixed-size stack and associated objects holding state such as Thread ID and thread-local storage — see Section 2.3.5.
- Goroutines are anonymous. Go does not have a representation of a running goroutine analogous to Java's `Thread` objects, and it is not possible in Go to stop or resume one goroutine from another.
- Goroutines are multiplexed onto OS threads by the Go runtime. Because goroutines are so cheap to create and run, it is often practical to have very large numbers of goroutines running concurrently, even if the number of available processors and OS threads is small.

2.3.5 Thread-local storage

Java supports thread-local storage. As the name suggests, this is storage that holds variables that differ between threads. Because in an STM each transaction is running in its own thread, thread-local storage can be used to keep any bookkeeping necessary by the transaction. As an example, the JVSTM uses Java's `ThreadLocal`'s to keep the transaction objects necessary to validate and commit read-write transactions — see Section 3.3.2.

As mentioned in Section 2.3.4, goroutines are much more light-weight than Java's threads, and do not support thread-local storage. When I started this project, I spent some time researching ways to implement thread-locals. Thread-locals can be implemented by using a thread ID, or in this case goroutine ID, as an index in a map of ID to the corresponding thread-local state. However, as described in Section 2.3.4, goroutines are anonymous, and there is not an appropriate way of determining the ID of a goroutine — the only ways are inefficient, "hacky" and generally frowned upon[6].

2.3.6 Memory model

The memory model of a language defines which values can be seen when multiple threads access shared memory concurrently. As work by Peter Sewell and others has illustrated, defining memory models is difficult in the presence of optimising compilers and complex hardware. The "Multicore Semantics and Programming" unit of assessment[7] is a good introduction to this topic.

[6] A brief, entertaining overview of this can be found at blog.sgmansfield.com/2015/12/goroutine-ids

[7] cl.cam.ac.uk/teaching/1920/Multicore

Go's memory model, similarly to Java's, defines a happens-before relationship between synchronisation operations. In Go, the synchronisation operations are fewer: they mostly concern lock acquisition, the creation of goroutines and channel communication. The memory model lacks guarantees on shared memory accesses analogous to the guarantees provided by Java's `volatile` and `final` variables, on which the JVSTM relies to provide synchronisation of its lock-free components. That being said, the documentation of the `sync/atomic` package seems to assume that its operations have similar semantics to those of Java's `volatile` variables.

2.4 Engineering methodology

Version control and backup policy. I kept the source code of my project in a Git repository publicly available on my GitHub profile. The LaTeX and accompanying source files of this dissertation are in a private Git repository. In addition to the repositories hosted on GitHub and the clones on my local machine, I kept a backup of the contents of my local machine in an external disk, which I updated weekly.

Test-driven development. One of the most important lessons learnt during my internships is the value of test-driven development in software projects. However, because this project involved designing an interface that was subject to change, it made more sense to complete the implementation of the STM before writing tests for it. During the implementation of STMBench7, the data structure invariant tests were written as soon as possible to detect bugs early.

Tools. I used the GoLand IDE to write, test and benchmark my code, and the `gofmt` tool[8] to format it according to common style guidelines.

Versions and licenses. I wrote my project using Go version 1.12. My implementation of the GVSTM is based on the description of the JVSTM algorithm [**?**]. My implementation of STMBench7 is based on the Java source code of the benchmark, version 1.2, licensed under BSD-3-Clause[9].

[8] golang.org/cmd/gofmt
[9] opensource.org/licenses/BSD-3-Clause

Chapter 3

Implementation

I begin this chapter with an overview of some concepts that are central to the understanding of STMs regardless of the language and algorithm used (Section 3.1). I then describe the steps taken to design a general approach to STM implementation in Go (Section 3.2), and how this is based on the preliminary research summarised in Section 2.3. The main implementation tasks of the project, the GVSTM and STMBench7, are discussed in sections 3.3 and 3.4, respectively. At the end of the chapter, I include an overview of the repository for the interested reader to use as reference (Section 3.5).

3.1 STM concepts

The concepts discussed here are applicable to STMs in general and will be used in later sections when describing the implementation of the GVSTM.

3.1.1 Conflicts

Two concurrent transactions are said to conflict semantically if the result of their execution is not the same as the result of some sequential execution of the two transactions. This implies that the operations of one of the transactions affect the outcome of the other.

Because in this project I focus only on operations on transactional variables, this can occur only if one transaction writes to a transactional variable from which the other transaction reads — a Read-After-Write conflict on a transactional variable. This, in fact, is an overly conservative approximation to conflict detection — it detects physical conflict, not semantic conflict. The difference between the two can be shown with a simple example: consider two transactions, T and U, that access the same shared counter initially holding the value 0. Transaction T checks if the value of the counter is less than 100 and, if yes, performs some computation. Transaction U increments the value of the counter from 0 to 1. If T and U are executed concurrently there may be physical conflict, because T reads from a transactional variable to which U writes, but there is no semantic conflict because the value of the counter is still less than 100, and so the outcome of T does not change.

3.1.2 Aborts and retries

In the GVSTM, the only reason for a transaction to abort or retry — discard its progress so far and restart its computation — is if it conflicts with another transaction. This is necessary to maintain the correctness properties of the STM. This behaviour should not be exposed to users — the STM should automatically retry when it detects a conflict.

Some STMs allow users to explicitly abort or retry transactions. Because STMBench7 does not use these operations I do not have a meaningful way of evaluating them, and so I decided not to add them to the GVSTM implementation. That being said, my design makes the addition of these features simple, as described in Section 3.3.6.

3.1.3 Lifetime of a transaction

The lifetime of a transaction can be divided into four main stages:

Start The transaction is created, along with its arguments and any bookkeeping necessary.

Execution The transaction performs its computation, possibly including reads and writes to transactional variables.

Validation A check is made to guarantee that the transaction does not cause a conflict. If the transaction fails the check it typically retries. If the transaction passes the check it commits.

Commit The transaction is now successfully completed. The work to be done here depends on the algorithm used — this is explained in Section 3.1.4.

3.1.4 Different methods of dealing with conflicts

STMs can be classified based on when validation is performed and when writes to transactional variables are installed:

Deferred update. The writes to transactional variables are installed only at commit-time. That is, when a transaction writes to a transactional variable it does not change the value stored in the transactional variable immediately. Instead, it stores the written value in a shadow-copy private to the transaction. Then, if the transaction terminates successfully, it installs the values stored in its shadow-copies in the transactional variables, at which point the new values become globally visible. This increases the amount of work done at commit-time, but makes aborting a transaction very simple: the new values are private so they can simply be discarded.

Direct update. Writes to a transactional variable immediately install the new value in the globally accessible transactional variable. At commit-time there is no work necessary to install the new values, but aborting is more complex. Because the new values are already installed, it is necessary to roll-back the changes made. When one transaction

aborts, all other transactions that have read the results of a write made by the aborting transaction must also abort. This causes cascading aborts, which waste large amounts of computation.

Lazy conflict detection. Validation is done only after the execution stage of the transaction has completed, immediately before the commit stage. This has the potential to waste a lot of computation, because transactions already destined to abort will run all remaining computation. On the other hand, the overhead of individual reads of transactional variables can be much lower when compared to eager conflict detection.

Eager conflict detection. Validation is done at every read of a transactional variable. This often implies that a direct update scheme is used. If, for example, one transaction reads a transactional variable to which a concurrently executing transaction has already written, the reading transaction automatically aborts. This has the potential of unnecessarily aborting transactions: in the previous example, aborting is necessary only if the writing transaction commits successfully.

3.2 STM interface in Go

In this section I present the design for a general STM interface in Go, based on the features of the language described in Section 2.3. Listing 3.1 shows this interface and Listing 3.2 shows an example of a transactional counter implemented using it.

```
1 type TVar interface {}
2
3 type Transaction interface {
4     Load(tVar TVar) interface{}
5     Store(tVar TVar, value interface{})
6 }
```

Listing 3.1: The STM interface. This represents the minimum functionality that an STM should be able to provide in Go.

3.2.1 Typing

An STM should be able to synchronise access to variables of arbitrary types. As mentioned in Section 2.3.1, Go does not support generic programming, but supports polymorphism in the form of interfaces. Because all types must be supported, the functions to store to or load from a transactional variable must operate with values of the empty interface type.

Because the load of a transactional variable returns an `interface{}` type, the user must cast the returned value to the expected type. This prevents static type checking of values in transactional variables, and hence increases the burden on the programmer to

```
1  type TransactionalCounter struct {
2      value TVar
3  }
4
5  func NewTransactionalCounter() *TransactionalCounter {
6      return &TransactionalCounter{ value: newTVar(0) }
7  }
8
9  func (c *TransactionalCounter) Increment(tx Transaction) {
10     tx.Store(c.value, tx.Load(c.value))
11 }
12
13 func (c *TransactionalCounter) Get(tx Transaction) int {
14     return tx.Load(c.value).(int)
15 }
16
17 func (c *TransactionalCounter) AtomicIncrement() {
18     Atomic(func(tx Transaction) {
19         tx.Store(c.value, tx.Load(c.value))
20     })
21 }
22
23 func (c *TransactionalCounter) AtomicGet() (value int) {
24     Atomic(func(tx Transaction) {
25         value = tx.Load(c.value).(int)
26     })
27     return
28 }
```

Listing 3.2: A transactional counter. This example shows both transactional functions and atomic versions of those functions.

guarantee that values loaded from transactional variables have the correct type. Sadly, this is presently the only solution that supports all types.

3.2.2 Transaction representation

All operations on transactional variables should be associated with an existing transaction. This means that any part of a program that attempts to perform an operation on a transactional variable must have access to the object representing the transaction with which the operation is associated. In the JVSTM, for example, this is achieved by storing that object in thread-local storage. Loads and stores of a transactional variable then access this thread-local storage to make any bookkeeping necessary. However, as mentioned is Section 2.3.5, Go does not support thread-local storage.

Without thread-local storage, all the variables accessible inside a function must be either global variables or arguments passed to the function. The bookkeeping data of transactions should not be globally accessible, because that would allow users to accidentally violate the isolation guarantees provided by the STM. Not only that, it would be extremely difficult for the load and store operations of a transactional variable to determine with which transaction the operation should be associated. This means that, to be able to perform an operation on a transactional variable, functions must receive a transaction object as an argument, and pass it to the functions that perform the transactional variable operations.

The inner structure of transaction objects is likely to be specific to each STM implementation and only code internal to the STM implementation should be able to access it. For these reasons, users of an STM should not be able to create or modify transaction objects.

3.2.3 "Transactional" functions

The fact that functions operating on transactional variables must receive a transaction object as an argument leads to the existence of a "contagious" property of functions. This property applies to all functions that must receive a transaction object as an argument and can be defined recursively:

1. A function is transactional if it operates on a transactional variable.
2. A function is transactional if it calls a transactional function without surrounding said call in a call to `Atomic`.

Defining this property has some subtle advantages. Firstly, the inclusion of a transaction in the parameters of a function clearly marks the function as transactional. For this reason, I adopted the convention of always passing the transaction as the first argument of any transactional function, to make it more visible that the function is transactional. Secondly, because creation of transactions is restricted to the STM implementation, a user cannot perform operations on transactional variables without having a valid transaction created by the STM code.

3.2.4 "Atomic" execution

In this dissertation, and indeed most of the literature on STMs, it is usual to say that transactions occur "atomically". This is somewhat of a misnomer, because the term is not used to entail only the atomicity property described in Section 2.2.1. Instead, atomic execution is taken to mean "execution according to all the correctness properties claimed by the STM in question". In the case of the GVSTM, and hence throughout this dissertation, that property is opacity, as also described in Section 2.2.1.

A user of an STM should be able to define and atomically run a transaction. According to the definition of transactional functions introduced in Section 3.2.3, it makes sense to represent a transaction to be executed as a transactional function. Due to Go's treatment of functions as values, atomic execution can be achieved by passing a transactional function as an argument to some function `Atomic`, which handles any necessary bookkeeping.

I did not include this function in the STM interface because the characteristics of STMs can somewhat vary. In particular, the signature required by the `Atomic` function will depend on the bookkeeping necessary: some STMs may, for example, require the `Atomic` function to take as an argument whether the transaction to be executed is read-only or read-write. An example use of this function is shown in Listing 3.2.

3.3 GVSTM

In this section I describe the implementation of the GVSTM. Most aspects of the implementation are similar to those of the JVSTM, but I also describe the design work necessary to adapt the algorithm to Go's features.

3.3.1 Versioned data structures

The JVSTM uses multi-version data structures, called versioned boxes, to implement transactional variables [**?**]. A versioned box contains a history of the values installed in the transactional variable. That is, a sequence of version-value pairs, where the version is the logical timestamp at which the value was installed in the transactional variable. This logical timestamp is the serialisation order number of the transaction that stored the corresponding value.

Semantics. The key property of versioned boxes is that their operations semantically appear to occur at arbitrary logical timestamps:

- A load of a versioned box is made with an associated read-timestamp and returns the most recent value that was not installed after the given timestamp.
- New values are installed in a versioned box with a given commit-timestamp, representing writes of the given values to the transactional variable at the given timestamp.

Implementation. Versioned boxes are implemented using a linked list of version-value pairs ordered in reverse version number order (newest first). The elements in the list are called versioned bodies. A version box is simply a wrapper for this linked list. A graphical representation of this structure is shown in Figure 3.1.

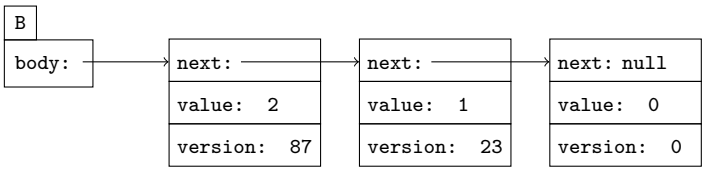

Figure 3.1: The versioned box data structure. A load with version number 0 to 22 will yield value 0. A load with version number 23 to 86 will yield value 1. A load with version number larger than 86 will yield value 2.

A read of a versioned box traverses the list of bodies until it finds a version no greater than the timestamp given and returns the value associated with that timestamp. The installation of a value in a versioned box adds a new body to the list of bodies — in the GVSTM the installation of a value in a versioned box is guaranteed to be made with a version number greater than all version numbers already present in the box, so this only requires adding the new body to the head of the list.

3.3.2 Algorithm overview

According to the classification of STMs described in Section 3.1.4, the GVSTM performs lazy validation and deferred updates. This means that validation and installation of new values are done only at commit-time. To do this, the GVSTM uses transaction objects that store the bookkeeping necessary and are created when the transaction starts. Assume for now that all transactions perform both reads and writes of transactional variables.

Transaction objects. Each transaction object maintains a read-timestamp, a read-set, and a write-set:

Read-timestamp
: The read-timestamp is used to read from versioned boxes during the execution of the transaction. When the transaction is created, the read-timestamp is read from a global logical clock, which counts all the transactions that have committed. This means that the read-timestamp of every executing transaction is the serialisation order number of the latest transaction to have committed when the transaction started.

Read-set
: The read-set contains all the versioned boxes read by the transaction. When the transaction object is created, this set is empty.

Write-set
: The write-set of the transaction contains the shadow-copies of transactional variables

— it is a map from versioned boxes to the latest values the transaction has written to them. When the transaction object is created, the map is empty.

The transaction objects are used when reading and writing to versioned boxes — a read adds the box to the read-set; a write of a value adds the box-value pair to the write-set. At commit-time all the entries in the write-set are installed in the versioned boxes.

3.3.3 Validation and commit algorithms

Here I describe the validation and commit algorithms used by the GVSTM, and how the commit algorithm was adapted from that of the JVSTM to reduce the commit-time bottleneck.

Commit lock. Validating a transaction concurrently with the installation of changes on transactional variables by another transaction is extremely complex. For this reason, the original implementation of the JVSTM [**?**] uses a global commit lock to serialise the validation and commit process: when a transaction terminates it acquires the lock, validates and commits, then releases the lock. Later work extended the JVSTM to have a lock-free commit algorithm [**?**]. The implementation of this extension would make for an interesting Part II project on its own right, so I decided to implement the lock-based commit algorithm.

Commit algorithm. The commit algorithm of the GVSTM simply installs all the written values in the corresponding versioned boxes. At the start of the commit, a commit timestamp is calculated by adding one to the commit-timestamp of the latest committed transaction. The commit-timestamps of transactions dictate their serialisation order. Each value installed in a versioned box is installed with this commit-timestamp as its version number.

Commit-time bottleneck reduction. The commit algorithm is run sequentially, whilst transaction execution is run in parallel. According to Amdahl's law [**?**], this makes the commit algorithm the sequential bottleneck of the GVSTM, especially as the number of transactions run in parallel increases. It is hence desirable to shorten this bottleneck as much as possible.

As mentioned in Section 3.3.1, the installation of a value in a versioned box consists of adding a new body to the head of the list of bodies. The JVSTM declares the fields of a body `final` and fills them in immediately before adding the body to the linked list. Java's memory model then guarantees that the initialisation of the fields of a body occurs before the addition of the body to the list. This prevents a race condition between a committing transaction and a reading transaction, which would cause the reading transaction to see an uninitialised body.

Go does not have a construct with memory model semantics equivalent to Java's `final` variables, so the GVSTM must use a synchronising memory operation when adding a body to a list of bodies. This also means that the bodies no longer have to be created at commit

time. Instead, they are created and added to the write-set of a transaction when it first writes to a transactional variable. This shifts the overheads of allocating memory for the versioned bodies from the sequential commit-time algorithm to the parallel execution of concurrent transactions, reducing the length of the bottleneck of the algorithm.

Validation algorithm. The validation algorithm of the GVSTM determines if the validating transaction can be serialised directly after the last committed transaction. For this to be possible, the transaction must not conflict with any committed transaction. As described in Section 3.1.1, a physical conflict occurs when a read and a write of a transactional variable occur in concurrent transactions. We can then analyse the two possible cases:

Read by a committed transaction, write by the validating transaction.
This does not cause a conflict — the write has not been installed yet, so the reading transaction must have read an earlier value. This is consistent with the serialisation order of commits: the committed transaction is serialised before the validating transaction.

Write by a committed transaction, read by the validating transaction.
This may cause a physical conflict: because the validating transaction is serialised after the committed transaction, it should have read a value at least as recent as the committed transaction. However, because the validating transaction reads using its read-timestamp, it will only have read a value at least as recent as the committed transaction if the commit occurred no later than the read-timestamp of the validating transaction. If this is not the case, the validating transaction has violated strict serialisability by reading a stale value, and hence must abort.

The validation algorithm then only needs to check that, in every versioned box in the read-set of the validating transaction, there is no value installed later than the read-timestamp of the validating transaction. If there is, the transaction must abort. If there is not, the transaction may commit without causing conflict.

3.3.4 Read-only transactions

The validation algorithm is necessary to ensure that transactions appear to occur atomically. In particular, the validation algorithm of the GVSTM checks if the transaction can appear to occur atomically at commit time. It is not necessary in the general case that transactions appear to occur at commit time — but introducing this restriction keeps the validation and commit algorithms simple.

An important observation is that, due to the use of versioned boxes, all reads by a transaction appear to occur at the same time — the read-timestamp issued at the start of the transaction. A transaction that only performs reads of versioned boxes — a read-only transaction — is then by construction atomic, and can always be serialised at its read-timestamp. Because read-only transactions are always serialisable, there is no need for them to validate. Because read-only transactions do not write to transactional variables,

the commit algorithm does nothing. As such, read-only transactions are allowed to bypass the validation and commit stages entirely while preserving the correctness of the STM. Because of this, the GVSTM handles read-only transactions differently:

- The transaction object does not contain a read-set, because validation is unnecessary. This also removes the overhead of adding the versioned box to the read-set when a transactional variable is first read, so individual reads of versioned boxes are also more efficient.
- The transaction object does not hold a write-set, because the transaction makes no write operations.
- Read-only transactions cannot abort, because they are serialisable by construction.
- Read-only transactions do not need to wait to acquire the commit lock, so given otherwise equal conditions their performance is independent of contention in the commit lock. This gives guarantees on the latency of read-only transactions regardless of the presence of inter-conflicting read-write transactions.

Always start as a read-only transaction. It is not desirable to require a user to distinguish read-only transactions. One reason for this is that it is not always possible to determine ahead of time if a transaction will be read-only. It may be, for example that a transaction will perform a write of a transactional variable if some condition holds, but this condition cannot be determined before the transaction starts. In these cases, the user would have to default to declaring the transaction as a read-write transaction and potentially miss out on the increased performance of read-only transactions. This approach is also more error prone: a mislabelling of a transaction may violate the invariants of the STM. For these reasons, the GVSTM attempts to execute all given transactions as read-only transactions. If the transaction attempts to perform a write operation it aborts and retries as a read-write transaction.

Write in a read-only transaction. Evidently, the behaviour just described should not be exposed to the user. When a read-only transaction attempts to write, control should return to the `Atomic` function automatically. This is achieved by using a call of `panic`.

It is interesting to note that write-only transactions also never fail validation, because a set of writes can always be serialised after any set of transactions. In practice, however, write-only transactions are very uncommon, so not worth optimising.

3.3.5 Garbage collection of old versioned bodies

As the reader may have noted, writing to versioned boxes increases the size of the linked list they store. This means that a program using the GVSTM will slowly run out of memory by filling it with old versioned bodies. This of course is not acceptable, so an algorithm to reclaim old bodies must be employed. This is by far the most complex aspect of the GVSTM implementation — the original description of the lock-free algorithm for

the JVSTM is six pages long [**?**, Section 4.4.7]. However, due to the page limit of this dissertation I will have to limit my description to a short overview, and I will not be discussing all the race conditions that must be avoided by an implementation of the algorithm. The interested reader can examine my implementation of the GVSTM or read the original description of the algorithm cited above.

Active transaction records. An active transaction record holds the commit-timestamp of a committed transaction and the set of versioned bodies created by the corresponding transaction. The record also holds a counter of currently executing transactions whose read-timestamp is equal to the timestamp of the record — that is, all transactions that are semantically reading at the same point in time as the transaction corresponding to the record committed. When a transaction starts, it increments the counter of the record corresponding to its read-timestamp. When a transaction terminates — either through a commit or an abort/retry — it decrements the counter.

Linked list of records. The GVSTM keeps a linked list of transaction records in reverse timestamp order. When a transaction commits, it must create a new active transaction record and add it to the head of the list. Read-only transactions do not need to do this, but they must increment and decrement counters appropriately.

Cleaning bodies. When all executing transactions have started after a particular timestamp, the bodies of the corresponding record can be cleaned. That is, the bodies stored in the record can be freed to be reclaimed by the OS. Both Java and Go are garbage collected, so in both cases this means simply removing the bodies from the corresponding lists — the runtime garbage collector takes care of the rest. In practice, determining that all executing transactions have started after a certain timestamp while avoiding race conditions is rather complex — again, the interested reader is welcome to read the original description of the algorithm.

3.3.6 Extensions

Here I discuss some extensions that could be made to the GVSTM. I did not implement these extensions partly because of the time restrictions on the project, and partly because STMBench7 does not leverage any of them, so I would not be able to evaluate them meaningfully. Nevertheless, these features increase the expressiveness of an STM and could be implemented in future work.

Abort and retry semantics. As mentioned in Section 3.3.4, a call to `panic` is used to abort a read-transaction and retry its execution as a read-write transaction. In this case, the call to `panic` takes as an argument a value denoting an attempted write in a read-only transaction. Using the same principle, the `panic` function could be used with other values to denote an explicit abort or retry by the user. The GVSTM would then distinguish between these values and abort or retry as required. This is a very simple addition: the only modifications required are a change in the atomic function to handle

these values and the addition of functions `Abort` and `Retry` exposed to the user, which are wrappers for the calls of `panic`. When preparing my project I had planned to implement this extension, but ultimately decided against it due to the lack of its use in STMBench7.

Different nesting semantics. The GVSTM presently only supports flat nesting. It could be an interesting Part II, or indeed Part III, project to implement one or more other nesting schemes [**?**].

Side effects as monads. The GVSTM only handles accesses to transactional variables. If a transaction creates other types of side effects, such as printing to a file or channel communication, those effects are not transactional. One approach to changing this would be to represent these effects as monads, in the form of functions kept by the transaction object. On commit, each of these monads is executed. This solution only handles cases in which the transaction is producing output. Much more careful thinking is required in the case where the transaction consumes some input — it could require, for example, the creation of transactional wrappers for channels and I/O streams, which could be made available as a library and could increase the appeal of the GVSTM to software engineers.

3.4 STMBench7

In this section I describe my implementation of STMBench7, with a focus on its software engineering aspects.

3.4.1 Collection of results

In my project proposal (Appendix D) I set out to produce an implementation of STMBench7 in the form of files to be used by Go's `test` tool. However, after examining the Java source code of the benchmark, I noted that it produced more detailed statistics than Go's `test` tool for this particular application. As such, I amended the format of my STMBench7 implementation to be a single Go executable, which more closely resembles the design of the Java implementation.

To make it easier to collect the results of a large number of benchmark runs, I made the benchmark output the generated statistics to a file as well as the console. Each run of the benchmark creates a new file, whose name is the time at which the benchmark was run and which contains information about the parameters used.

To automate the collection of results of multiple runs of the benchmark I made my implementation accept the parameters of the benchmark as command line flags and created a shell script to run the benchmark with a variety of parameters. This should not be done inside the Go executable to guarantee that runs of the benchmark are independent — no need to garbage collect the results of previous runs, for example.

3.4.2 Data structures

The data structure used by the benchmark represents the data structure that would be used by a Computer Aided Design application. As mentioned in Section 2.3.2, object semantics are often useful, and this is the case throughout STMBench7. In the rest of this section I will refer to elements of the STMBench7 implementation as objects, even though Go is not object oriented, to mean that the elements are represented using the interface pattern described in Section 2.3.2. All the data structure interfaces described here need to be implemented using each synchronisation mechanism.

Design library. The design library is a set of composite parts. Each composite part has one document and a graph of atomic parts. The atomic parts are connected by connection objects. The composite part also has a pointer to the root of the graph.

Module. A module has one manual and a tree of assemblies. Assemblies at the leaves of the tree are called base assemblies. There is a many-to-many relation between base assemblies and composite parts. All other assemblies are complex assemblies.

All objects in the data structure — the design objects — also contain references to their parent, so for example it is possible to get the parent composite part of any given atomic part, and the root complex assembly of any assembly. It is worth noting that in STMBench7 only one module is used. A visual representation of the data structure is shown in Figure 3.2.

Backend data structures. The benchmark data structures are implemented using backend data structures. To increase the fairness between synchronisation mechanisms I kept these implementations as similar as possible. This guarantees that any differences in performance are due to the synchronisation mechanism and not to the backend data structure implementation. Unfortunately the implementation used is naive and not very inefficient. It is worth drawing attention to two of the backend data structures:

ID pool An ID pool is a finite set of unique identifiers for a particular type of object. There is an ID pool for documents, one composite parts, etc. When an object is created it gets an ID from the corresponding pool, if any are available. When an object is destroyed it returns its ID to the pool.

Index An index is a map of some key type to all the objects of a particular type. Most indexes have IDs taken from an ID pool as keys, but there is also an index of documents indexed by their titles and index of composite parts indexed by build date. When an object is created or destroyed it is added or removed, respectively, from the corresponding index or indices.

3.4.3 Operations

STMBench7 supports four types of operations on its data structure, which are summarised below. It is important to note that operations can fail due to operation logic. This is not

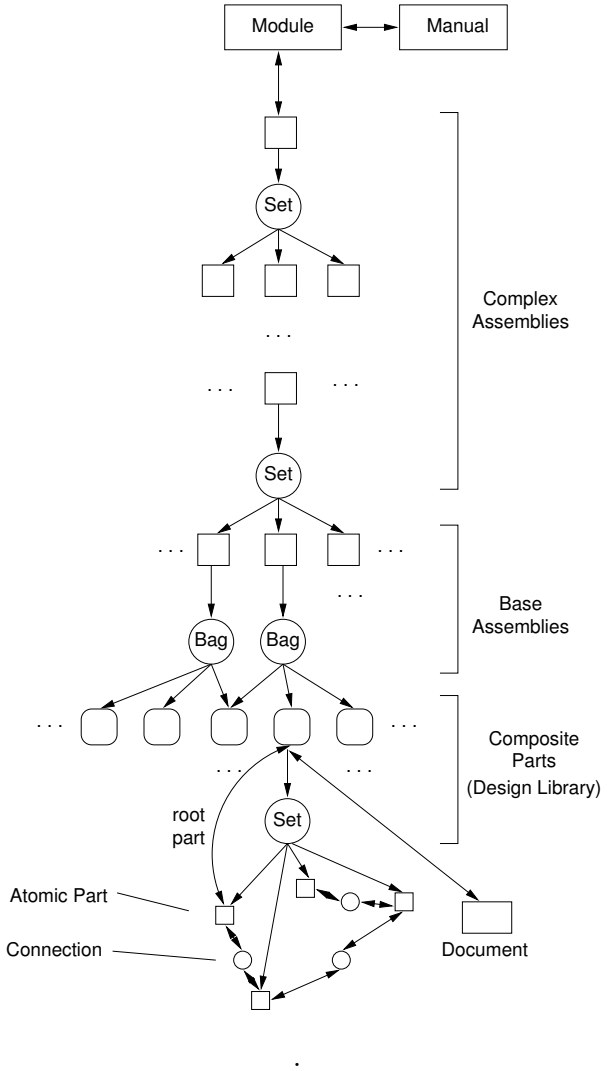

Figure 3.2: The STMBench7 data structure. This diagram is copied from the original STMBench7 paper [**?**]

due to transactions aborting or conflicting, but for example that ID pools are empty or that a random traversal reached a base assembly with no composite parts. This means that when looking at benchmark results it makes sense to look at the total throughput and not only at the throughput of successful operations.

Long traversals. Long traversals traverse very large numbers of objects. Most long traversals traverse the whole tree of assemblies, and some also traverse the graph of atomic parts of all composite parts. Most long traversals have a read-only and a read-write variant. Long read-write traversals only perform write operations on the attributes of atomic parts or documents — they do not modify structure.

Short traversals. Short traversals reach much smaller numbers of objects — instead of traversing the whole tree of assemblies, most short traversals take a random path from the root assembly to a composite part or from an atomic part to the root assembly. Most

short traversals have read-only and read-write variants, and the write operations are only on attributes of design objects.

Short operations. Short operations do not traverse the data structure at all. Instead, they use the indexes to find all objects of a type or find a random object. Again, most short operations have read-only and read-write variants, and the write operations are only on attributes of design objects.

Structure modification operations. Structure modifications make major changes to the data structure. They add and remove assemblies and composite parts and create and remove links between them. This conflicts with all traversals, so the inclusion of structure modifications in the benchmark workload severely impacts performance, as shown in Section 4.2.1.

3.4.4 Design patterns

The Java implementation of STMBench7 makes extensive use of object oriented design patterns [**?**] to make it easy to extend the benchmark with additional synchronisation approaches, such as STMs. One of the challenges of porting the benchmark to Go was understanding these patterns and how they can be adapted to a non-object oriented language, and it was a good exercise of software engineering skills. The most prominent patterns are described below.

Command. Operation executors are an implementation of the command pattern. Operation executors handle any top-level synchronisation necessary before executing an operation. In the case of the coarse-grained locking approach this consists of acquiring the global read-write lock, in a mode dependent of whether the operation to be executed is read-only or read-write. The operation executor for the medium- grained locking approach is much more complex: a large number of locks must be acquired in a specific order, according to which operation is being executed. Writing this executor is rather error prone, as the slightest mistake is likely to cause deadlock or race conditions. The GVSTM operation executor, on the other hand, simply wraps the operation to be executed in a call to the `Atomic` function.

Factory method and abstract factory. STMBench7 provides interfaces for all its benchmark and backend data structures, which are used throughout the benchmark. Each synchronisation mechanism provides its own implementations of these interfaces, along with the corresponding factory methods and abstract factories. This layer of abstraction makes it simple to, for example, change the implementation of the backend data structures without touching the rest of the codebase.

Builder. Builders are used to create and destroy the benchmark data structures while respecting the invariants of the benchmark. They encapsulate the addition and removal of objects from the corresponding indexes as well as the creation of other objects associated

with the object being created, such as the document of a composite part and the children assemblies of a complex assembly.

These patterns make the addition of new synchronisation approaches to the benchmark simple: the new synchronisation approach must only provide implementations of the data structures, an operation executor and add its abstract factories to the main file. There is no need to change any of the benchmark logic, such as its operations and builders.

3.4.5 Testing

As with any software engineering project, testing is important to guarantee that my implementations of both the GVSTM and STMBench7 are correct. The original benchmark implementation included both invariant tests on its data structures and opacity tests. Here I explain my reasoning about what tests should be included in my implementation.

Invariant tests. The invariant tests of STMBench7 check that both the benchmark and backend data structures are in a consistent state. I implemented these tests as soon as I had implemented the factories and builders of the data structures, which proved to be useful: the tests helped me find a couple of bugs I had overlooked, and allowed me to fix them before beginning to implement the operations of the benchmark, which would have caused much more complex errors. I kept running these tests as I implemented the operations, which made it much more efficient for me to debug my code.

Opacity tests. The opacity tests of STMBench7 check that the execution of the benchmark by an STM satisfies the opacity property described in Section 2.2.1. The GVSTM algorithm is the same as that of the JVSTM, whose opacity property has already been shown. Furthermore, bugs in the GVSTM implementation would cause large errors in any application using it. These errors would then cause the invariant tests to flag the error, so the opacity tests give me no new information.

Not only that, the opacity tests rely on retracing a concurrent execution serially, including the random short traversals. The Java implementation relies heavily on thread-local random state to achieve this. However, as discussed in Section 2.3.5, Go does not support thread-local storage, so implementing these tests would require a significantly different approach to the one used in Java — a very substantial engineering effort for no tangible benefit to this project.

3.4.6 Extensions

Here I discuss some extensions that could be made to my STMBench7 implementation. Most of them focus on making the benchmark results more representative of real-world applications.

Implementing all 45 operations. In my project proposal I set out to implement only a subset of the original 45 operations of STMBench7, and listed implementing all the operations as a possible extension. In my project I achieved this extension goal.

Although implementing the benchmark as a whole was a much more ambitious goal than I had expected, the addition of the remaining operations by itself was reasonably simple thanks to the modular architecture of the benchmark.

Efficient backend data structures. As mentioned in Section 3.4.2, the implementations of the backend data structures using locking approaches and STM are similar to increase the fairness of the benchmark. The implementation in question is very inefficient, because the data structure must be copied on write to prevent concurrent access. A very valuable extension would be to provide more efficient implementations of the backend data structures. In the case of locking this would only involve removing the copying operations. In the case of the GVSTM this would involve the creation of complex versioned data structures such as sets and maps. The JVSTM provides this as part of a library. The data structures are based on persistent data structures [**?**]. The creation of such a library to be distributed with the GVSTM would also make it much more appealing to software engineers.

Containerising. One of the strongest appeals of STMs is that they scale much better than other approaches to large numbers of cores. During my project I collected results by running the benchmark on my laptop only, but it would be interesting to run the benchmark on server class machines. To make this easier, it would be useful to create an STMBench7 container to facilitate deployment on a cloud computing platform.

Rewrite the benchmark to be fairer. The method to create and register a complex assembly is unfair to STMs: if the ID pool for either complex or base assemblies is exhausted the method undoes all its work. This involves a lot of operations on backend data structures. Some STM implementations, such as the GVSTM extended with abort and retry semantics (Section 3.3.6), could simply abort or retry when they detect such an error, which would be much more efficient.

3.5 Repository overview

One of my goals when proposing my project was to use it as a step towards producing an open source library providing STM functionality in Go. In order to evaluate my STM implementation, my project also had to include the implementation of a benchmark for STMs. Because I wanted this benchmark to be usable by other STM implementations, I also needed to define an interface for STMs in Go. These three "modules" [**?**, Section 1.1] of the project should not necessarily be part of the same repository — a software engineer may want to use only the GVSTM, or care only about the interface to produce their own STM implementation. For this reason, the three modules should not necessarily be kept together in the long run. In the scope of this project, however, it is much easier to handle code submission and dependency management in a single repository, so that is how I have structured my project. The repository holding the source code of my project has one directory for each of the modules. Their contents are summarised below.

STM interface. The STM interface defined in this project is reproduced fully in Listing 3.1.

GVSTM. The implementation of the GVSTM is reasonably short — less than 300 lines of code. That being said, the algorithm being implemented is rather complex, especially in avoiding race conditions in the garbage collection of old bodies. A good understanding of the algorithm is necessary to understand the implementation of the transaction objects and the active transaction records. The implementation of versioned boxes and versioned bodies is reasonably well contained.

STMBench7. The STMBench7 benchmark is by far the largest part of this project in terms of lines of code — approximately 5000. The structure of the benchmark is described in Section 3.4, and the organisation of source files is made to resemble that of the Java implementation. Future versions of my implementation may change this. Although I wrote all code submitted on my own, my STMBench7 implementation is mostly a translation of the Java version, with modifications to use the features of Go and the STM interface I defined.

Chapter 4

Evaluation

In this chapter I recall the success criteria of the project and describe how they were achieved (Section 4.1). I give quantitative results of benchmarking the GVSTM with STMBench7 (Section 4.2) and a qualitative comparison of using locking approaches and the GVSTM to implement STMBench7 (Section 4.3).

4.1 Success criteria

All the success criteria of the project were met, as well as the extension goal of completing the implementation of STMBench7. In this dissertation I also discuss extension goals for which design work was made, but whose implementation would entail too large a time investment to be included in this project. Below I summarise the goals set out in my project proposal (Appendix D) and specify how they were met.

To have a working implementation of the GVSTM.

- Transactions appear to occur atomically and in isolation from the rest of the program, as demonstrated by the fact that the invariant tests of STMBench7 complete successfully. The GVSTM also provides the stronger property of opacity, as described in Section 2.2.1.
- Transactional functions are composable, as described in Section 3.2.3.
- The invariant tests of STMBench7 show that the GVSTM implementation is correct, as discussed in Section 3.4.5.

To have a partial implementation of STMBench7. This goal has been exceeded, in that I have completed the STMBench7 implementation.

- All the operations of STMBench7 have been implemented, and results collected from a variety of workloads. These results are summarised in Section 4.2.
- As discussed in Section 3.4.1, the original aim of structuring the benchmark as test files supported by Go's `test` tool was amended upon further inspection. Instead, the benchmark is a standalone executable file that produces more detailed statistics for STM evaluation.

To make a meaningful evaluation of the GVSTM.

- In Section 4.2 I examine the performance of my GVSTM implementation compared to that of coarse- and medium-grained locking approaches.
- In Section 4.3 I draw on my personal experience of programming using the GVSTM and using locks to compare the usability of each approach.

4.2 Benchmark results

In this section I summarise the results obtained by running STMBench7 with the GVSTM and locking approaches. A detailed evaluation of the GVSTM would include metrics such as the maximum latency of operations — particularly read-only operations — as well as CPU utilisation and the memory overhead of storing multiple versions. However, due to the space restrictions on this dissertation, I will focus only on the total throughput, in operations per second, of the GVSTM and locking approaches using different workloads and number of goroutines. Appendix B contains detailed results of one run of STMBench7. Appendix C describes my benchmarking methodology.

4.2.1 Workloads without structure modifications

In this section I show the results of benchmarking using three workloads:

Read only workload 100% read-only operations

Read-dominated workload 90% read-only operations

Read-write workload 60% read-only operations

Long traversals, including long read-write traversals, are enabled; structure modifications are disabled. The ratios of each kind of operation are the default values defined by STMBench7. Due to the inefficient implementations of the backend data structures (Section 3.4.2) I reduced the number of composite parts to 50, to make the initialisation of the benchmark less time consuming.

Figures 4.1, 4.2 and 4.3 show the throughput of the three synchronisation approaches in each of the workloads. Note that, because of the benchmarking methodology used (Appendix C) the error bars are barely visible in most cases.

As shown in Figure 4.1, the throughput of all synchronisation approaches in a read-only workload scales with the number of cores used, and minimal additional throughput is achieved by using simultaneous multithreading (SMT). Note that in pure read-only applications no synchronisation is necessary.

Figure 4.2 represents the best use-case of STMs, and of the GVSTM in particular. The read-dominated workload keeps the rate of conflicts low, so the GVSTM scales almost as well as the medium-grained locking approach, despite being much more simple and general-purpose. The use of SMT slightly degrades performance, because sharing a core causes individual transactions to have a higher latency, and hence a higher chance to conflict.

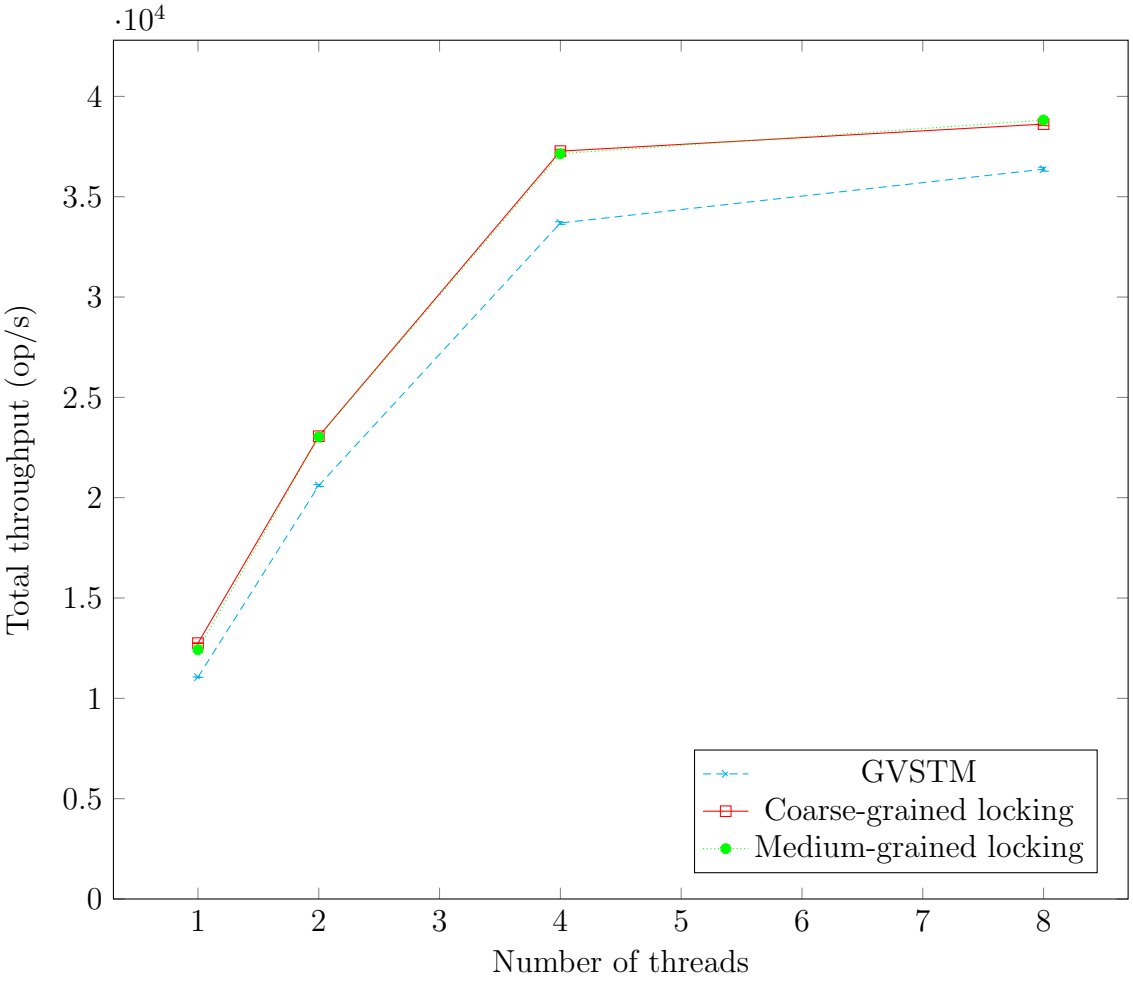

Figure 4.1: Total throughput of a read-only workload.

Figure 4.3 shows the result of benchmarking a read-write workload. The coarse-grained approach achieves the same throughput regardless of the number of goroutines used, as the global lock serialises operations. The medium-grained approach is able to extract some throughput by using more goroutines, but the improvement is minimal compared to the read-dominated approach. The GVSTM is able to extract some throughput by using two goroutines instead of one, but increasing the number of goroutines used degrades performance. This is because the rate of conflicts increases, causing more transactions to abort and waste computation. In absolute terms, the performance of the GVSTM is significantly lower than that of both locking approaches, because the overheads of read-write transactions in the GVSTM are substantial.

4.2.2 Introducing structure modifications

Figures 4.4 and 4.5 show the results of benchmarking the read-dominated and read-write workloads with structure modifications enabled.

In my STMBench7 implementation the inclusion of structure modifications decreases single-threaded throughput by about one order of magnitude — note the scale on the vertical axes. This is likely due to the inefficient implementations of the backend data

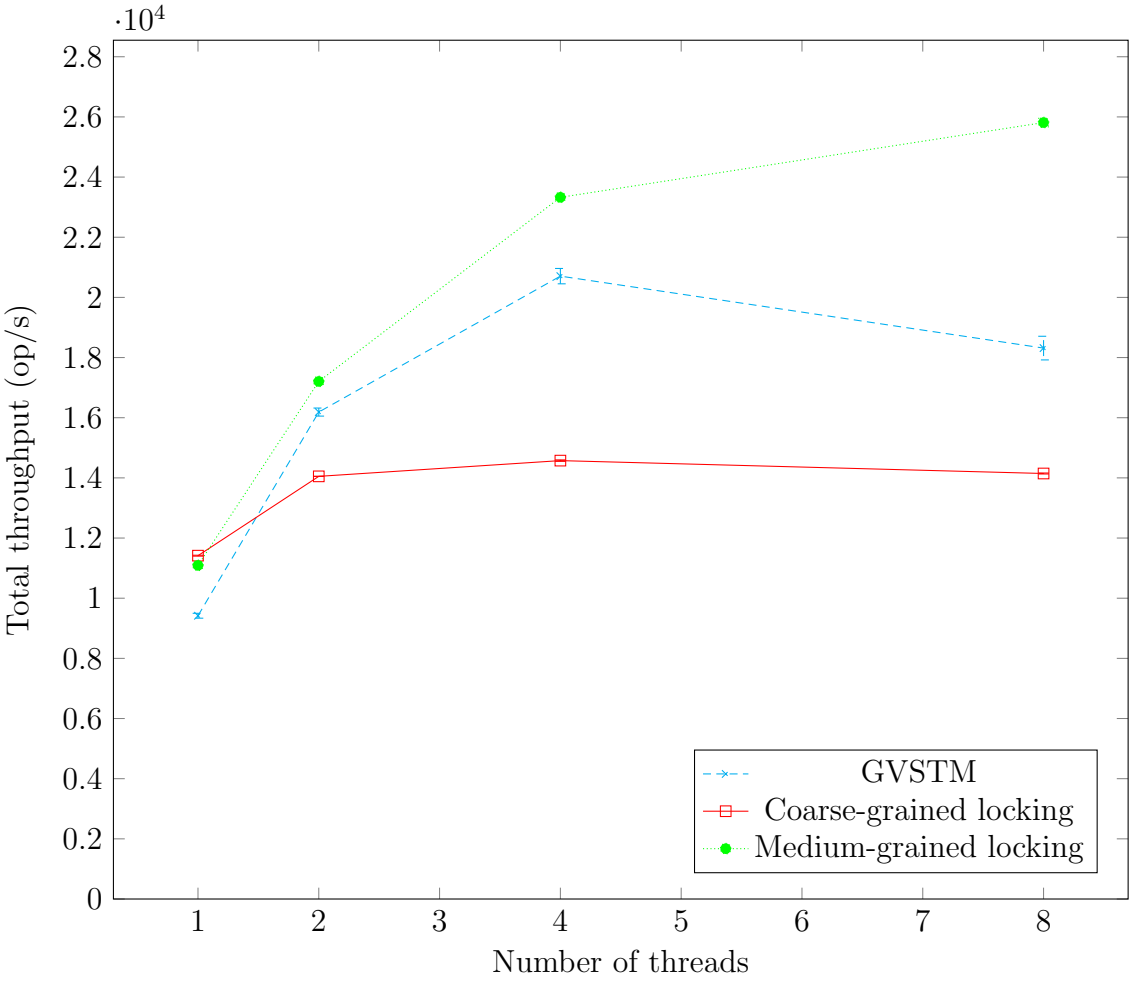

Figure 4.2: Total throughput of a read-dominated workload.

structures, which make structure modifications very expensive operations.

As discussed in Section 3.4.3, structure modifications conflict with most other read-write operations. This means that as more goroutines are used by the GVSTM the rate of conflicts increases, forcing transactions to abort and waste computation. This causes the performance of the GVSTM to rapidly degrade as more goroutines are used, even in a workload with only 10% of read-write transactions.

The execution of structure modifications requires the acquisition of a global write-lock in both locking approaches, essentially forcing sequential execution. This makes it difficult to significantly exceed single-threaded performance.

4.3 Usability

One of the motivations behind STMs is their usability compared to lock-based synchronisation. Because a usability survey of a large number of programmers is beyond the scope of this project, in this section I will be drawing from my own experience implementing STMBench7.

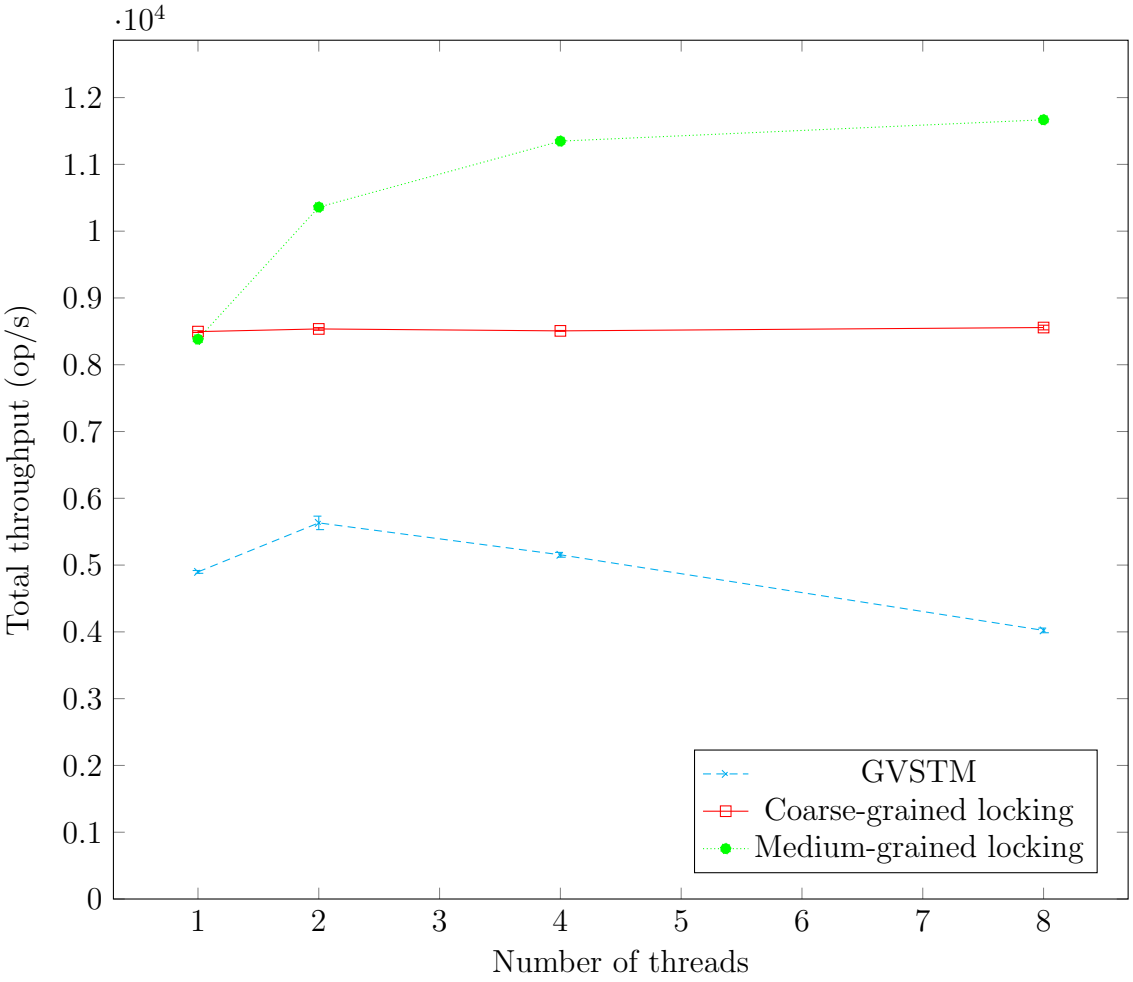

Figure 4.3: Total throughput of a read-write workload.

Correctness. The GVSTM makes it much simpler to correctly synchronise programs, as demonstrated by the operation executors of STMBench7. Implementing the medium-grained executor caused several bugs, some of which were caused because Java supports reentrant locks and Go does not. Appendix A makes a comparison between the operation executors.

Code readability. Reads and writes of versioned boxes are more verbose than reads and writes of normal variables, which can decrease the readability of the code. On the other hand, having every mutable variable wrapped in a versioned box is a useful indicator of the purpose of a variable, reducing the risk of incorrect use.

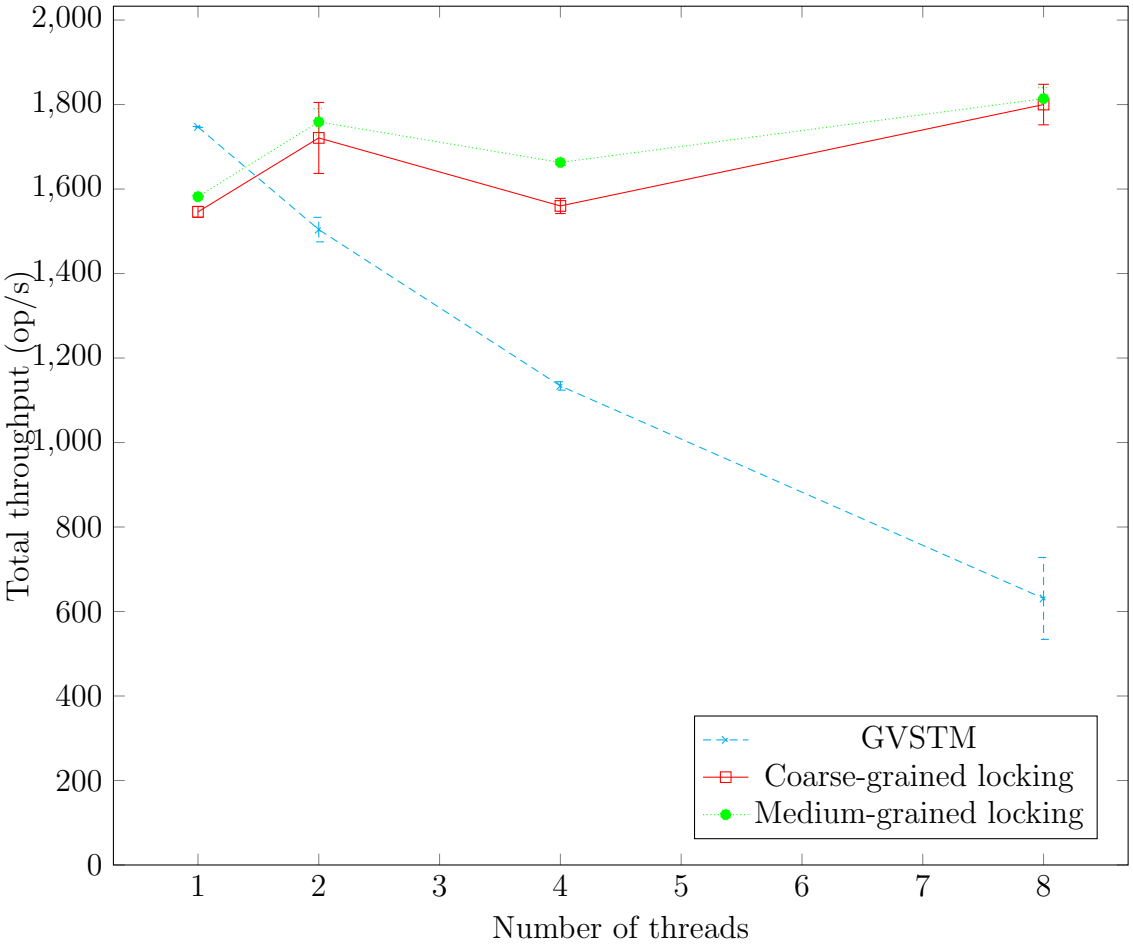

Figure 4.4: Total throughput of a read-dominated workload with structure modifications.

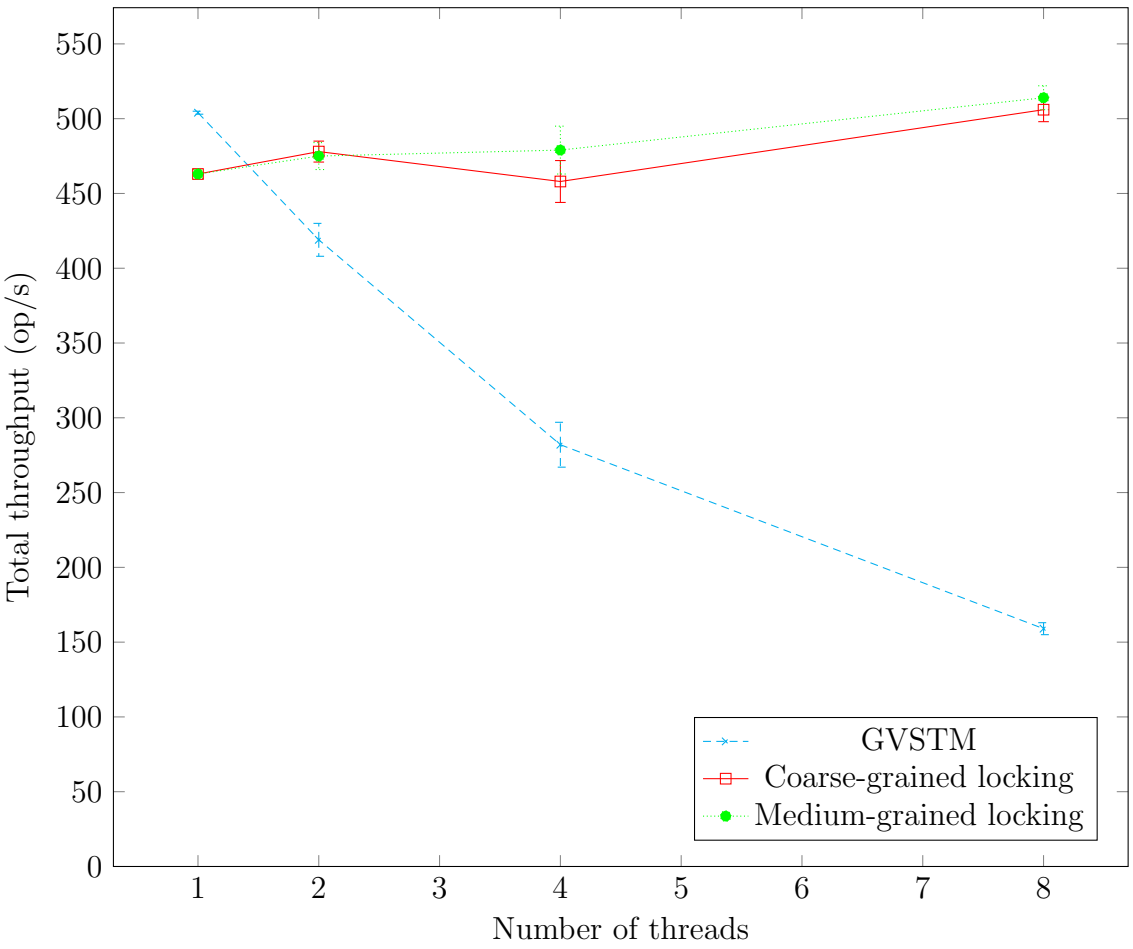

Figure 4.5: Total throughput of a read-write workload with structure modifications.

Chapter 5

Conclusion

This dissertation describes three major contributions to the area of STM implementation in Go:

- The definition of an STM interface in Go.
- The implementation of the GVSTM, a multi-version STM.
- The implementation of STMBench7, a stress test of STM implementations.

Personal reflection. This project was largely successful, and a good first step towards the use of STMs in Go. During the project I explored convoluted features of the Go programming language and its memory model, implemented an STM, including its complex lock-free garbage collection algorithm and gained some experience with system performance measurement.

I would have liked to spend more time adding features to the GVSTM, but unfortunately implementing STMBench7 from scratch was very time consuming. Future projects may benefit from having this work done ahead of time.

Future work. In the near future I will aim to turn the products of this project into open source software, with the goal of encouraging the use of STMs in Go. Because all the code produced is my own, I will have to decide on an appropriate licens under which to release the project. The steps necessary for the project to be released are the following:

- Add to the GVSTM implementations of versioned data structures, such as lists and sets.
- Extend the STM interface to also represent transactions that are able to explicitly retry and abort, and add this functionality to the GVSTM.
- When the steps above are completed, the STMBench7 implementation can be changed to use more sensible backend data structures and possibly exploit abort and retry semantics.

Future Part II projects may find inspiration from the other extensions described in Sections 3.3.6 and 3.4.6.

Appendix A

Comparison of operation executors

Listing A.1 shows the GVSTM operation executor in its entirety. The GVSTM executor simply wraps the operation to be executed in a call to the `Atomic` function.

Listing A.2 shows the coarse-grained locking executor. It determines whether to read-acquire or write-acquire the global read-write mutex based on built-in information about the operations.

The medium-grained locking executor is about 180 lines of code and extremely complex — each operation must acquire some subset of the locks of the structure in a particular order, and bookkeeping must be kept to know which locks are held by each executor.

Implementing the locking executors is error prone: mislabelling an operation as read-only, for example, will violate the invariants of the structure; acquiring medium-grained locks in a different order may cause deadlock; adding an operation to the medium-grained implementation may require a substantial refactor of the executor.

```
1  type gvstmOperationExecutor struct {
2      operation Operation
3  }
4
5  func CreateGVSTMOperationExecutor(operation Operation)
       OperationExecutor {
6      return &gvstmOperationExecutor{operation}
7  }
8
9  func (e *gvstmOperationExecutor) Execute() (result int, err
       error) {
10     gvstm.Atomic(func(tx Transaction) {
11         result, err = e.operation.Perform(tx)
12     })
13     return
14 }
```

Listing A.1: The GVSTM operation executor.

```
1  var (
2      globalWriteLock = &sync.RWMutex{}
3      globalReadLock  = globalWriteLock.RLocker()
4  )
5
6  type cgOperationExecutor struct {
7      operation Operation
8      lock      sync.Locker
9  }
10
11 func CreateCGOperationExecutor(op Operation) OperationExecutor {
12     var lock sync.Locker
13     switch op.ID().OpType {
14     case OperationRO, TraversalRO, ShortTraversalRO:
15         lock = globalReadLock
16     default:
17         lock = globalWriteLock
18     }
19     return &cgOperationExecutor{op, lock}
20 }
21
22 func (e *cgOperationExecutor) Execute() (result int, err error) {
23     e.lock.Lock()
24     defer e.lock.Unlock()
25     return e.operation.Perform(nil)
26 }
```

Listing A.2: The coarse-grained operation executor.

Appendix B

STMBench7 example output

Shown below is the output of STMBench7 for one benchmark run. Detailed statistics of per-operation success rates and times to completion have been omitted for brevity.

```
1  ------------------------------------------------
2  Benchmark parameters
3  ------------------------------------------------
4
5  Number of threads: 4
6  Length: 2m0s
7  Read percentage: 90
8  Synchronisation method: Coarse grained locking
9  Long traversals enabled: true
10 Long read-write traversals enabled: true
11 Structure modification enabled: false
12
13 Operation ratios [%]:
14              TraversalRW:   0.56
15              TraversalRO:   5.00
16         ShortTraversalRW:   4.44
17         ShortTraversalRO:  40.00
18              OperationRW:   5.00
19              OperationRO:  45.00
20   StructureModification:   0.00
21
22 Checking invariants (final data structure):
23 Invariant check completed successfully!
24
25 ------------------------------------------------
26 Summary results
27 ------------------------------------------------
28
29 TraversalRW:
30   successful: 9746
31   maxTTC: 13
```

```
32    failed: 0
33    total: 9746
34  TraversalRO:
35    successful: 87552
36    maxTTC: 16
37    failed: 0
38    total: 87552
39  ShortTraversalRW:
40    successful: 19411
41    maxTTC: 15
42    failed: 58262
43    total: 77673
44  ShortTraversalRO:
45    successful: 348620
46    maxTTC: 14
47    failed: 350100
48    total: 698720
49  OperationRW:
50    successful: 85938
51    maxTTC: 15
52    failed: 1858
53    total: 87796
54  OperationRO:
55    successful: 773664
56    maxTTC: 14
57    failed: 14046
58    total: 787710
59  StructureModification:
60    successful: 0
61    maxTTC: 0
62    failed: 0
63    total: 0
64
65  Total sample error: 33.33% (0.12% including failed)
66  Total throughput: 11040.70 op/s (14576.12 op/s including failed)
67  Elapsed time: 120.00
```

Appendix C

Benchmarking methodology

All results in this dissertation were produced following the same methodology:

- Close all applications and restart my computer, to minimise the number of processes running and competing for resources.
- For each parameter combination (synchronisation mechanism, number of goroutines and workload) run the benchmark five times, discard the results with maximum and minimum throughput to avoid outliers and use the remaining three sets of results.
- Run the benchmark for two minutes each time, to minimise the effects of startup costs. Brief experiments showed that this duration was sufficient to produce stable results.
- Check that the ratio of failed operations does not vary significantly between runs, to avoid skewed results.

It is worth pointing out that discarding the results with minimum and maximum throughput reduced the standard deviation of the samples significantly, making error bars barely visible in most cases. A large number of samples contained at least one outlier, so I decided to take this approach in order to treat all samples consistently.

Appendix D

Project Proposal

The body of my project proposal follows in the next page. The cover page was removed to avoid personal identifiers.

The rest of this page is intentionally left blank.