The Efficiency of Software Transactional Memory

ERIK HELIN
and HENRY RODRICK

KTH Computer Science
and Communication

Bachelor of Science Thesis
Stockholm, Sweden 2010

The Efficiency of Software Transactional Memory

E R I K   H E L I N
a n d   H E N R Y   R O D R I C K

Bachelor's Thesis in Computer Science (15 ECTS credits)
at the School of Computer Science and Engineering
Royal Institute of Technology year 2010
Supervisor at CSC was Mads Dam
Examiner was Mads Dam

URL: www.csc.kth.se/utbildning/kandidatexjobb/datateknik/2010/
helin_erik_OCH_rodrick_henry_K10003.pdf

Royal Institute of Technology
School of Computer Science and Communication

KTH CSC
100 44 Stockholm

URL: www.kth.se/csc

Abstract

Concurrency problems in computer programs are notoriously hard to solve correctly. This is largely due to the complexity of reasoning about and using semaphores and similar locking mechanisms. In this study, an alternative model for solving concurrency related problems called Software Transactional Memory (STM) is evaluated.

The evaluation is done using the Haskell programming language, which supports both concurrency models. For benchmarking, solutions to two typical concurrency problems have been developed. The performance aspects studied are *running time*, *memory consumption* and *scalability*.

The results show that the two different concurrency models have similar performance characteristics in many cases, an important exception being that STM performs much worse when a lot of work is performed inside a transaction.

The conclusion is that Software Transactional Memory can be an efficient alternative to semaphores if used carefully.

Referat

Effektiviteten hos Software Transactional Memory

Problem i datorprogram relaterade till samtidig exekvering av kod är ökänt svåra att lösa. Detta beror till stor del på svårigheterna med att resonera kring och implementera denna typ av programkod med hjälp av semaforer och liknande låsmekanismer. I denna studie har en alternativ modell för att lösa samtidighetsrelaterade problem kallad Software Transactional Memory (STM) utvärderats.

Utvärderingen har gjorts med hjälp av programspråket Haskell, vilket stöder både STM och semaforer. För benchmark har lösningar till två typiska samtidighetsrelaterade problem utvecklats. Prestanda har sedan mätts med avseende på *körtid*, *minnesåtgång* och *skalbarhet*.

Resultaten visar att de två olika samtidighetsmodellerna har likartade prestandaegenskaper i många fall. Ett viktigt undantag är att STM-lösningarna blir betydligt långsammare än semaforlösningarna då mycket jobb utförs i en transaktion.

Slutsatsen är att ett omsorgsfullt användande av STM kan vara ett effektivt alternativ till semaforer.

Contents

Chapter 1

Introduction

The purpose of this study is to evaluate the performance of Software Transactional Memory (STM) by comparing it to the traditional semaphore based approach to solving concurrency related problems in programming. In order to understand why Software Transactional Memory has evolved as a way of solving such problems, it is important to understand the concepts of concurrency and parallelism.

In this essay, the terms parallelism and concurrency will be used in the following way:

- A *parallel* program utilizes two or more processors to speed up execution, but is in no semantic way different from a non-parallel one, nor is it non-deterministic. All the parallel aspects of the program are added "under the hood", i.e. with the help of a compiler or a run time system.

- A *concurrent* program is a program which solves a given task with the help of concurrency. This means that the program is using one or more threads to solve a problem. These threads may run on more than one processor, but they do not necessarily have to. A concurrent program is non-deterministic, which means that it is not possible to know in advance exactly in which way the program will execute.

One justified question is whether concurrency is needed in programs at all. The answer is that many problems can be expressed in a very natural way with the help of concurrency. For example, GUI applications usually handle input events from the user in separate threads to make the user interface feel more responsive.

The problems with concurrency arise when the different threads need to communicate with each other. A currently executing thread may at any time be preempted (interrupted) to let another thread use the CPU, and usually the programmer can not control this process at all. This means that the programmer has to make sure that the threads can be interrupted without putting the program in an inconsistenst state.

Today, there are mainly two different approaches to how to solve this problem [1]:

- *Message passing* - Two or more threads communicate by sending messages back and forth between each other. These messages may include data and other information which the threads need to access.
- *Shared memory* - Two or more threads communicate by sharing a block of memory. By writing to a shared memory location, a thread can change the state of another thread.

This essay will focus on the second approach, shared memory. Shared memory programming is usually carried out by using some kind of locking mechanism to prevent two threads from using the shared resources at the same time. The basic building blocks in these locks are *semaphores*, first introduced by Edsger Dijkstra [3]. They behave much like ordinary integers, but with three main differences [4]:

i) A *semaphore* can be initialized to any integer value, but once it is initialized, the value of the semaphore can only be changed by incrementing or decrementing it.

ii) A *semaphore* which is decremented to a negative value blocks the thread that modified it.

iii) A *semaphore* which is incremented by a thread unblocks exactly one blocked thread if any threads are blocked. The thread which is being unblocked is chosen in no particular order.

The operations *increment* and *decrement* are often called *signal* and *wait*. A semaphore which is initialized to the value of 1 is often called a binary semaphore. This is because such a semaphore only allows one thread at a time to access the shared memory. Another common name for binary semaphores is ”mutex” (a portmanteau of the words *mut*ual and *ex*clusion).

By using semaphores, one can make sure that the shared memory is accessed by a correct number of threads at the same time. However, as the complexity of a concurrency related problem increases, the problem becomes difficult to solve by using semaphores. Due to the asynchronous nature of concurrent programs, it is hard to convince oneself (and others!) that the semaphores are being used in a correct way.

Fortunately, other techniques have evolved with the promise to make the writing of concurrent applications easier. One of them is called Software Transactional Memory.

Chapter 2

Background

2.1 Software Transactional Memory

This section gives an overview of some important aspects of Software Transactional Memory (STM) and the alternative strategies used today when designing such systems. In addition to giving an overview of the field, a brief understanding of the topics discussed here will be important later when the particular implementation of STM in Haskell is analyzed.

2.1.1 Definition of Software Transactional Memory

In order to define the concept of Software Transactional Memory, we must first define what we mean by a *transaction*. Larus and Rajwar [13] describe transactions as sequences of actions that appear indivisible and instantaneous to the outside observer.

Transactions are at the core of the programming model for database systems and have been so for a long time. In database systems theory, four properties, commonly known as the ACID properties, are used to describe the desired behaviour of transactions: [20]

- *Atomicity* - Either all actions of a transaction succeed or, in case of a failure, any actions performed by a partial transaction will be undone.

- *Consistency* - A transaction executed in isolation preserves the consistency of the database. For example, if money is to be transferred from an account A to another account B, the consistency requirement is that the sum of A and B is unchanged.

- *Isolation* - Concurrent transactions must be executed as if they were run in isolation, that is, one transaction should not see any inconsistent intermediate states created by another concurrently running transaction.

- *Durability* - If a transaction is successful, the changes made to the database must survive any system failure.

A Software Transactional Memory system is based on nondurable transactions (naturally said to have the ACI properties). The durability requirement is not needed in order to define STM transactions, since it is not a property of the generic computer programs that STM is designed for.

2.1.2 Atomic Blocks

One of the key concepts of STM is the atomic block, a programming construct that allows the programmer to group operations to be performed in isolation. Usually, this idea is presented as a programming language keyword, *atomic*, with an accompanying scope holding the code of the transaction:

```
atomic {
    statements...
}
```

Conceptually, the atomic block is a simple form of Hoare's so-called *conditional critical region* (CCR), a classical mechanism for defining concurrent algorithms. Hoare's goal when introducing CCR was to move the responsibility of creating necessary semaphores from the programmer to the compiler, thus reducing the complexity of concurrent code. [6]

Executing code in an atomic block (i.e. a transaction) has two different defined outcomes: Either the entire transaction succeeds and the changes become visible outside the transaction, or the transaction aborts and leaves the program in an unchanged state. A third, undefined outcome, is that the code in the transaction does not terminate.

An abort happens if access to a shared memory location conflicts with another transaction or if a deadlock is discovered. If a transaction aborts, the system re-executes it in hope that the problem will not occur again.

2.1.3 Concurrency Control

The fact that conflicting transactions must be aborted and re-executed does not put any specific constraints on how and when the STM system should detect conflicts and trigger the abortion.

As it turns out, this activity, which is called *concurrency control*, is handled by different STMs using either a *pessimistic* or *optimistic* approach. [13]

In a pessimistic STM, conflict detection and conflict resolution (abortion and re-execution) takes place instantaneously after a conflict occurs. This might seem like the most efficient approach since you “throw away” the least possible number of computations in case of a rollback.

However, if there are more than two simultaneous transactions, it's not always clear what the best strategy is. Assume that two transactions A and B both conflict with a transaction C but not with each other. It would suffice to abort C, but it is hard for the pessimistic algorithm to know this since it has no knowledge in advance regarding whether A and B are conflicting or not [22].

Because of such scenarios, the alternative approach, optimistic concurrency control, validates only when it is time to commit.

2.1.4 Update Strategies

One problem when designing an STM system is to decide on how the shared memory should be updated by a transaction. As with concurrency control, there are two main strategies here: *Direct updates* and *deferred updates*. [8]

In an STM system with direct updates, transactions directly modify the shared resources. The original value of the modified object must be stored by the system in order to be able to restore the original state if the transaction aborts. It is also the responsibility of the STM system to provide some kind of concurrency control on memory access.

Deferred updates, on the other hand, are made private to the transaction (a separate record). When the transaction commits, the entire record is written to the shared memory. This strategy is good when there are many aborted transactions, since the shared state is not changed and does not have to be restored. However, there is considerably more work involved in committing the transaction with this strategy, and the commit action must appear atomic to the other transactions.

2.1.5 Further Design Aspects

As discussed in section 2.1.3, the choice of transaction to abort in case of a conflict might have an impact on performance. There is also the possibility that a transaction hardly gets to commit because of repeated conflicts with other transactions. STM systems usually provide some kind of *contention manager* to deal with such issues. [22]

Another thing that one must decide when designing an STM is what it should mean for two transactions to conflict. This could be defined in several ways. One possibility is to say that transactions conflict if they try to access the same word in memory. Another possible definition is that a conflict occurs when transactions access the same object (which of course would be a language specific definition). On which level a conflict is detected is called the *granularity* of an STM system. [25]

A third problem is how to deal with side-effects. Imagine for example that an I/O action takes place inside a transaction (like printing a message on the screen). In general, such actions cannot be undone and must be avoided in transactional code [11]. As we will see, this is less of a problem in the Haskell programming language, since side-effects are well isolated from *pure* (side-effect free) functions.

2.2 Introduction to Haskell

A short review of the Haskell programming language follows below. Readers well familiar with the language may skip to the next section.

2.2.1 Main features

Concurrent Haskell is an extension to the programming language Haskell98 [18]. Haskell has three primary features which will be important when discussing concurrent applications:

- Haskell is a *functional* programming language. Being a functional language means that the program only consists of evaluation of expressions [9].
- Haskell is a *pure* programming language. Being pure means that it is not possible to update the value of a variable, that is, once a variable is initialized, the value of it can not change [9].
- Haskell is a *lazy* programming language. Being lazy means that Haskell only evaluates an expression when it is needed [9]. The reason for Haskell doing so is to avoid recomputation of expressions.

2.2.2 Example program

```haskell
-- Multiplies the two numbers a and b
mult :: Num t => t -> t
mult a b = a * b
```

Code 2.1. A simple Haskell example

Example 2.1 shows a very basic Haskell program multiplying two numbers:

- The first line is just a comment. A one-line comment always starts with `--`.
- The second line is an example of Haskell's typesystem. Haskell has support for true polymorphism, which in this case means that `mult` can take any kind of number as input, and returns a number of the same type.
- The third line specifies the input arguments as well as the body of the function. Note that Haskell doesn't use parentheses around the arguments.

2.2.3 Monads

A question which arises is how Haskell can manage input/output (I/O). After all, I/O can be seen as updating the state of the ”world”, and since Haskell is pure, it can not modify the world outside a function. The answer is called monads. A *monad* can be viewed as a wrapper around the world. The `print` function in Haskell does not lack a return value, instead it returns a monad that contains two things: [1]

- The new world, which essentially is the old world with the evaluation of `print` applied to it.
- The pure result of `print`, which is the string that `print` is supposed to print on the screen.

With the help of this abstraction, Haskell can be a pure language and at the same time perform I/O. Monads can also be used for other purposes than I/O. This will be of great importance when discussing the implementation of STM in Haskell.

2.3 Concurrency In Haskell

The following section discusses the main components of Haskell's concurrency support, and how they are used.

2.3.1 Threads

Haskell has several different kinds of threads, all for different purposes. To understand how concurrency works in Haskell, it is important to have an understanding of the various kinds of threads:

- *Sparks* - A spark is the lightest of all threads in Haskell. In fact, it is not even a thread to begin with, instead it's a potential thread. A spark is created with the `par` function. A spark is turned into a Haskell thread when the thread scheduler in Haskell has the capacity available to spawn new threads.
- *Haskell threads* - A Haskell thread is much lighter than an Operating System (OS) thread in terms of system resources used [15]. The Haskell threads run on a finite-sized private stack but they all share the same heap. Multiple Haskell threads usually run on a single OS thread. Haskell threads are created with the function `forkIO`.
- *Operating System threads* - An OS thread is an abstraction over the operating system's native threads (POSIX or Windows threads). OS threads are used by Haskell in order to support Symmetric Multiprocessor Parallelism (SMP). The typical scenario is to use the same number of OS threads for an application as the number of cores on the CPU. An OS thread is created with the function `forkOS`.

The different threads are controlled by the Haskell Runtime System's (RTS) own scheduler.

[1]This is a bit simplified, for the full explanation, see *Tackling the Awkward Squad* [12].

2.3.2 Semaphores

Haskell supports communication between threads using shared memory. To prevent two or more threads from accessing the same memory at the same time, Haskell introduces the type `MVar`. An `MVar` can be viewed as a binary semaphore[18] with the exception that if there are more than one waiting threads, they are woken up in first in first out (FIFO) order [24]. In contrast to semaphores, `MVar`s can also store a value in addition to its use as a simple lock. The reason for this is due to the functional nature of Haskell. In imperative programs, one often uses a semaphore as a lock around a global variable. This global variable can be seen as the value which is stored inside the `MVar`. `MVar`s support both the *signal* and *wait* operations with the functions `putMVar` and `takeMVar`. These functions can also access and update the data stored inside the `MVar`.

Haskell also has support for ordinary semaphores through the type `QSem`. `QSem` is built on top of `MVar` and works much as a regular semaphore. The only difference between traditional semaphores and `QSem` is that `QSem`, just like `MVar`, wakes the thread which has been asleep for the longest time when *signal* is called [24].

2.3.3 Software Transactional Memory

Much in the same way as `MVar`s are used to share data between threads, `TVar`s are used to share data between transactions in Haskell. Similar to the `MVar`, the `TVar` supports the functions `putTVar` and `readTVar`. The `TVar` holds both the data which is shared, as well as regulates the access to it. This is, once again, very similar to the design of `MVar`. However, the type signature of `readTVar` is

```
readTVar :: TVar a -> STM a
```

while the type signature for `takeMVar` is

```
takeMVar :: MVar a -> IO a
```

The important difference is that `readTVar` returns an `STM` monad, while `takeMVar` returns an `IO` monad. Because of Haskell's type system, this results in three important rules for transactions:

- No STM actions can be performed outside a transaction, since any function using `readTVar` or `putTVar` must be of type `STM a`.
- No side-effects can occur inside a transaction, since any function calling another function with return type `IO` must also be of type `IO`. However, all transaction must be of type `STM`, and therefore can not call any `IO` functions.
- Transactions can be expressed very naturally with the `do` syntax, since `STM` is a monad.

The last point means that a transaction can be expressed as:

```
transaction = do expression 1
                 expression 2
                 ...
                 expression n
```

Since an important part of almost all programs is to communicate with a user via I/O, it must be possible to use the data returned from a transaction. This problem is solved with the function `atomically`, which has the type:

```
atomically :: STM a -> IO a
```

Haskell's runtime system uses deferred updates by writing to a record instead of updating a variable directly [7]. Optimistic concurrency control is used when validating transactions, and each time a transaction is validated, it locks the validator to ensure atomicity. Instead of locking the entire validator, there is also an option to just lock the current `TVar` being validated [7].

2.4 Literature review

One of the earliest traces of Software Transactional Memory is a paper by Lomet published in 1977 [14]. Even though the programming language construct he proposes has a different name, it is essentially the same idea as STM. Lomet's goal was also the same as that of contemporary STM researchers – to move the responsibility of concurrency control from the programmer to the system.

The STM-like system proposed in Lomet's paper could be described as being rather aggressive since it has both pessimistic concurrency control and direct updates. No practical implementation of Lomet's system was ever created.

The term *Software Transactional Memory* first appeared in a paper by Shavit and Touitou in 1995 [19], and the system described in the paper is commonly known as the first actual example of transactional memory implemented in software (hardware based solutions were proposed earlier).

Just like Lomet's STM, Shavit and Touitou's system is pessimistic and does direct updates. One of the major drawbacks with their design is that the transactions are static, that is, the resources being guarded by a transaction must be explicitly stated.

Another major contribution, at least as far as STM in Haskell goes, is Keir Fraser's thesis *Practical lock-freedom* from 2004 [5]. The thesis proposes the implementation of STM that Haskell's variant is based on.

Fraser presents STM as a lock-free abstraction for concurrent programming and relates it to other lock-free techniques. Specifically, he presents a generalization of the atomic *compare-&-swap*-instruction found in many modern processor architectures. His multi-word compare-&-swap shares many properties with STM, since both techniques allows for several instructions to be executed atomically.

The first paper about STM in Haskell was written by Simon Peyton-Jones et al [7]. This paper discusses how STM can be implemented in Haskell, and explains the major problems and their respective solutions. The STM implementation in the paper lacks extensive testing, which is something the authors themselves mention at the end.

This was picked up by Tim Harris et al [17] in a paper which discusses how Haskell STM applications can be profiled. The paper presents detailed statistics of several profiled applications. The profiling part is very thorough, but the paper only compares different STM applications to each other. This makes it harder to fully understand the performance of STM compared to other concurrency techniques. The benchmarks were run at a time when the implementation of STM wasn't as mature as it is today, and it would be interesting to see these tests being run with the latest version of GHC.

An article was later written by Don Jones Jr, Simon Marlow and Satnam Sing [10], discussing in great detail the aspects of profiling parallel applications in Haskell. An implementation of a flexible profiling framework in Haskell is presented together with a GUI application called ThreadScope. ThreadScope visualizes the tracing of a program, and several use cases are discussed. Unfortunately, ThreadScope does not yet support tracing of STM transactions.

In 2009, Tim Harris et al followed up their own paper by further examining at which level the profiling of STM applications should be done [21]. In this paper, they turn their focus to the individual transactions instead of the applications. This results in much more fine-grained statistics, which reveals more of the performance characteristics of an STM application.

Simon Marlow et al [23] wrote in 2009 a paper which compares different implementations of singly-linked lists in Haskell. The implementations differ in which concurrency technique they are making use of. The compared techniques are `IORef`, `MVar` and STM. The paper offers a detailed description of the different implementations, as well as a comparison of the running time and scalability of the different solutions. The tests are run using version 6.10.1 of GHC. Our study is based upon this work, but will focus on different kinds of concurrency problems and we will only compare the `MVar` and STM techniques. However, we will further extend the measurements to also include how much memory the different concurrency techniques are using.

Chapter 3

Method

3.1 Method

To measure the efficiency of STM, two different solutions have been developed for two different kinds of concurrency related problems. The first kind of solution uses semaphores for concurrency control, while the second kind uses STM.

Two different kinds of problems have been chosen for testing the performance in different situations. The first problem is a traditional producer-consumer problem, while the second problem is more of a synchronization problem. Both problems are described in detail in section 3.2.

The two different solutions for each problem have been benchmarked using the tools described in section 3.1.1. The following aspects of the solutions are benchmarked:

- *Running time* - How long time a solution runs on a given input.
- *Memory usage* - How much memory a solution uses on a given input.
- *Scalability* - How the running time for a solution is affected by running it on a CPU with one to four cores.

In order to empirically determine the efficiency of STM, the benchmarks for the different solutions have been compared to see if any variant outperforms the other.

3.1.1 Tools

To benchmark the solutions, two different tools have been used:

- *Criterion* - Criterion[16] has been used for reliable runtime statistics.
- *GHC Runtime System* - The GHC Runtime System (RTS) has been used for statistics about memory consumption with the help of the garbage collector.

3.2 Problems

The problems are taken from *The Little Book of Semaphores* [4] and the semaphore based solutions follow the ones in the book. These solutions are well tested, which minimizes the risk for bugs in our implementations.

In all the solutions, the threads need to perform some work. Since the solutions only are used for benchmarking, the work consists of solving the *n-queens problem* [2] for different sizes of n.

The full source code for the solutions can be found in appendix A.

3.2.1 The Producer-Consumer Problem

The first concurrency problem in this study is the classical *Producer-Consumer* problem. The main reason for studying this problem is that it often appears in real world applications, not just as puzzles in textbooks.

In the Producer-Consumer problem, two types of threads exist, all sharing the same queue. The first kind of thread, the *producer*, generates data of some kind and pushes it to the queue. The second kind of thread, the *consumer*, pops data from the queue and processes it.

An event driven system could be implemented using this technique, where multiple threads (the producers) could fire events (produce data) to be handled by one or more event handlers (consumer threads).

Implementation using semaphores

Among the possible solutions to this problem, a standard solution exists using three semaphores. First of all, a mutex is used to serialize access to the queue. The two other semaphores could be called *items* and *spaces* (these are the names used in *The Little Book of Semaphores* [4]) and are used for bound checking.

The *items* and *spaces* semaphores could conceptually be though of as representing the number of items in the queue and the number of empty slots, respectively [1]. Hence, *items* is initially set to zero and *spaces* is set to the capacity (maximum size) of the queue.

When a producer is ready to push an item on the queue, it will first call *wait* on the *spaces* semaphore. If the queue is full, *spaces* will be zero and the producer blocks. Similarly, consumers will wait on the *items* semaphore and block if no items exist to consume.

The *items* semaphore is signalled by the producer when it has pushed a new item to the queue and the *spaces* semaphore is signalled by the consumer when an item is popped.

A possible danger when implementing this solution is that locks are taken in the wrong order. Suppose that the producer thread first calls *wait* on the queue-access

[1] In practice, this model is false. For example, a producer will decrement the *spaces* semaphore to zero *before* filling the last slot in the queue, and other threads could get to run in-between.

mutex and then on the *spaces* semaphore. If the queue is full, the producer will block, holding both the *spaces* and the *mutex* locks. Now, if a consumer is to pop an item off the queue, it must first acquire the mutex lock, but since this lock is held by a producer, the consumer will also block and the system deadlocks.

Implementation using STM

When implementing a Producer-Consumer solution in STM, it is possible to think of the problem in a sequential manner. Instead of using awkward bound checking semaphores, the length property of the queue can be used as long as the bound checking takes place inside an atomic block.

For example, a producer thread that is ready to push an item will enter an atomic block and acquire a reference to the queue from a transactional variable (`TVar`, as described in 2.3.3). The producer will then check the length of the queue and compare it to a variable specifying the capacity of the queue. If there is space left in the queue, the buffer will be updated and written back to the `TVar`, otherwise the STM function `retry` will be called.

The transaction can also abort and retry if inconsistencies are discovered at commit time. Suppose that two producers P_1 and P_2 try to push to the queue concurrently. Further, assume that there is one empty slot in the queue. If P_1 and P_2 enter the atomic block at the same time, they will both discover that there is still room in the queue and will attempt to push an item. Now, if for example P_1 commits first, P_1s transaction will be successful and the number of empty slots in the queue will be set to zero. When P_2 tries to commit, the STM system will discover that the *actual* state of the queue differs from the state of the queue at the beginning of P_2s transaction. P_2s transaction must therefore abort and retry, hoping for better luck next time.

The consumers can be implemented in a similar way. The only big differences are that `retry` is called if the queue is empty and that a value – the item popped from the queue – is returned from the transaction.

It could be argued that this solution is suboptimal since the entire queue is wrapped in a single `TVar`, which possibly introduces false conflicts. For instance, assume that there are 10 filled slots in a queue with capacity 100. Now, if one consumer and one producer begin their transactions at the same time and the producer commits first, the consumer will abort on commit. Since the queue was not extremely short and the threads modified different ends, there is (in theory) no real conflict between the two operations.

However, we chose the single TVar approach since it results in a simpler solution. Also, we believe that this solution allows at least as much concurrency as the semaphore solution, since it locks the entire queue on update as well.

3.2.2 The Bathroom problem

The second concurrency problem is about a unisex bathroom. At a big company, both men and women share the same bathroom. There are however two restrictions put on the employees on how they may use the bathroom. The first restriction states that men and women are not allowed to be in the bathroom at the same time. Second, only three persons are allowed to be in the bathroom at any given point.

Implementation using semaphores

The solution to this problem follows the solution in *The Little Book of Semaphores* [4] at page 177 very closely. Two semaphores, *numMale* and *numFemale*, are used for keeping track of the number of males as well as females currently in the bathroom.

To synchronize how males and females may enter the bathroom, a so-called lightswitch is used. A *lightswitch* is a synchronization mechanism which conditionally locks a mutex. The idea corresponds closely to how a lightswitch in a room is used. The first person (thread) which enters a room (critical section) turns on the lightswitch (locks the mutex). More people (threads) may enter or leave the room (critical section) as long as the last person (thread) leaving the room turns off the lightswitch (releases the mutex). The solution uses one lightswitch for the males and one lightswitch for the females, both locking a mutex *empty* which tells if the bathroom is empty or not.

For a ”male” thread to enter the bathroom, it needs to lock the mutex *empty* with the help of the male lightswitch. The thread then calls `wait` on the semaphore *numMale*. After performing a calculation, it calls *signal* on the *numMale* semaphore. Finally it unlocks the mutex *empty* through the male lightswitch. The algorithm works similarly for ”female” threads.

The only problem with the solution above is if a male thread enters the bathroom first. Then, as long as male threads keep entering the bathroom, no female thread may enter. Therefore, the female threads will possibly never be allowed to enter, even though they entered the queue to the bathroom before the male threads. To solve this, a mutex named *turnstile* is used. The *turnstile* mutex is locked before using a lightswitch to lock *empty*. As soon as the lightswitch acquires the lock on *empty*, the turnstile is signalled. Due to this, threads of different gender enter as they arrive to the bathroom.

Implementation using STM

The STM solution consists of several transactions. The transactions are designed to perform tasks similar to the semaphores in the solution above. All the transactions are laid out sequentially in a `do` block. Hence, if a thread has to retry one of the transactions, it won’t be able to immediately go on and perform the following transactions.

The semaphore *turnstile* is modelled as a `TVar Bool`. The logic of locking the turnstile is modelled as a transaction which checks the value of the `TVar`. If the value is `True`, i.e. it's locked by some thread, the transaction calls `retry`.

The next transaction performs the same logic as a lightswitch combined with the *numMale/numFemale* mutex. A `TVar Gender` called *occupier* is used, where `Gender` is either `Male`, `Female` or `None`. *occupier* monitors which gender is currently in the bathroom. The transaction checks if *occupier* is of the same gender as the thread. If the gender is the same or `None` (the room is empty), the transaction will continue. Otherwise, the transaction is retried. A `TVar Int` called *count*, holding the number of people currently in the bathroom, is then read. If *count* is larger than three, the transaction will call `retry`. Otherwise, the transaction increases *count* by one and succeed.

After the gender check follows a transaction that will simply write `True` to the `TVar` representing the *turnstile*. This indicates that the thread has passed the gender checking transaction and that another thread may complete the *turnstile* transaction.

The thread then performs a computation, which as mentioned earlier consists of solving the *n-queens* problem [2].

The last transaction decreases *count* by one. If *count* is equal to zero, the transaction will also set *occupier* to `None`. This indicates that the bathroom is empty and that threads of any kind may enter.

Chapter 4

Results

4.1 Results

All the running time and memory tests were run using GHC 6.12.1 on a computer equipped with an Intel Core 2 Duo processor at 2× 2.2 GHz and 4 GB of RAM. The operating system used was the 32-bits version of Ubuntu 9.10. The tests were run 3 times with the average value used as the result.

The scalability tests were run on a computer equipped with a Intel Core i5 CPU at 4 × 2.67 GHz. The compiler and RTS used was GHC 6.12.1 and the operating system was the 32-bit version of Ubuntu 9.10. The computer was equipped with 4 GB of RAM.

In both problems, only the running time was considered when benchmarking the scalability.

4.1.1 Bathroom

For all the benchmarks of the solutions to the Bathroom problem, the input in table 4.1 was used. The sequences describe in which order people of different gender arrive to the bathroom.

Length	**Sequence**
8	M FF MM FFF
15	M FF MM FFF MMM FFFF
24	M FF MM FFF MMM FFFF MMMM FFFFF
35	M FF MM FFF MMM FFFF MMMM FFFFF MMMMM FFFFFF

Table 4.1. *M* stands for male, *F* for female

In all the figures, *n* stands for the input to the n-queens problem. The n-queens computation was used as a simulation for a toilet visit.

Running time

For the running time test, the application `Fast` is included as well. `Fast` spawns n threads which calculate the n-queens problem for a given input. The threads do not have any restrictions on them, i.e. maximum concurrency is achieved.

The results can be seen in figure 4.1.

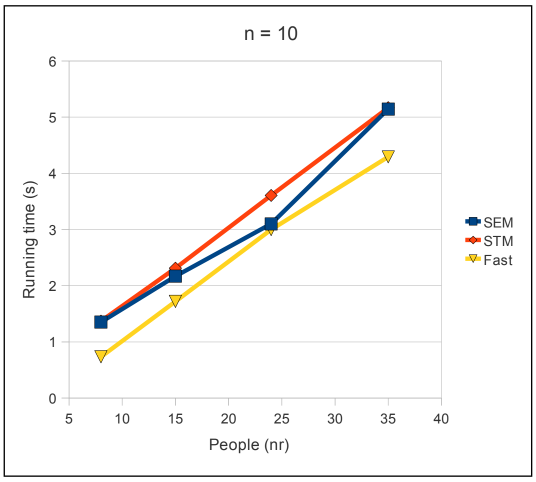

Figure 4.1. Running times for input sequences in table 4.1

Memory consumption

The results can be seen in figure 4.2.

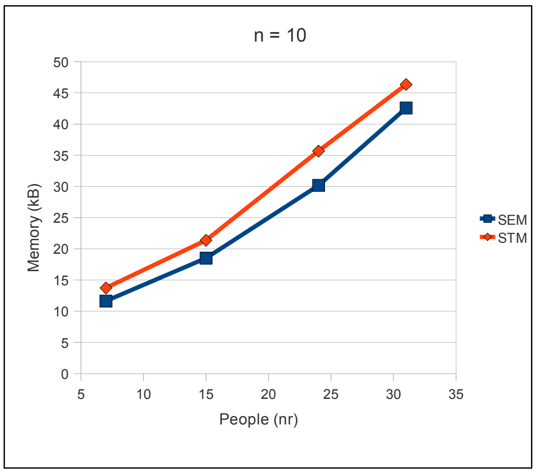

Figure 4.2. Memory consumption for input sequences in table 4.1

Scalability

Two different versions of the problem were considered for the scalability benchmark. The first version was the original solution, only allowing three threads in the bathroom at the same time. The second version allowed the same number of threads in the bathroom as the number of cores that was used. For example, when running on one core, one thread was allowed, on two cores, two threads were allowed and so forth.

In both versions, the input used was sequences of length 35. The corresponding input sequence can be seen in table 4.1.

The results are shown in figure 4.3 and 4.4.

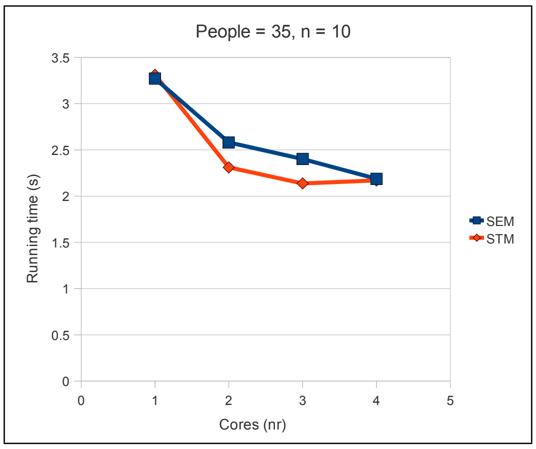

Figure 4.3. Scalability when three threads are allowed in the bathroom

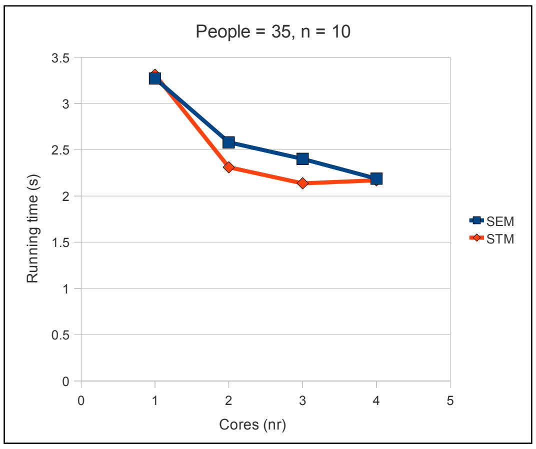

Figure 4.4. Scalability when the same number of threads as cores are allowed

4.1.2 Producer-Consumer

Three different versions of the Producer-Consumer problem were benchmarked. In the first version, no extra work is done in the consumer threads. The second version computes the n-queens problem for a 9×9 chess board in the consumer threads. This computation is performed outside the critical section, i.e. when the threads haven't locked any shared resource. In the last version, the consumer threads also computes the n-queens problem for a 9×9 chessboard. However, the computation now takes place inside the critical section.

In order to be able to benchmark the Producer-Consumer problem, the problem was slightly modified. The original version of the problem does not specify when the program should terminate. In the benchmarked version, a producer produces a finite amount of items. A producer is done once it has produced all its items. Additionally, the benchmarked solution puts a restriction on how many items a consumer is allowed to consume. A consumer is done once it has consumed all its item. A consumer is also done if there are no items in the queue, nor any producers left. When all producers and consumers are done, the program terminates.

In all the diagrams, the variables used are:

- *Buffer* - the size of the buffer.
- *Producer* - the number of producer threads.
- *n* - the input to the n-queens problem for the producer threads.
- *loops* - the number of items produced by a producer as well as consumed by a consumer.

Running time

The results can be seen in figure 4.5, figure 4.6 and figure 4.7.

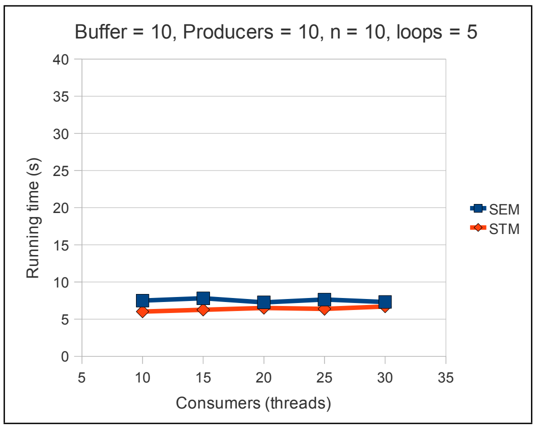

Figure 4.5. No work in consumer threads

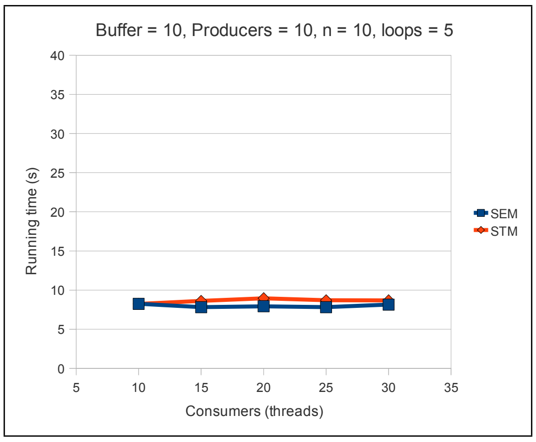

Figure 4.6. Work outside critical section in consumer threads

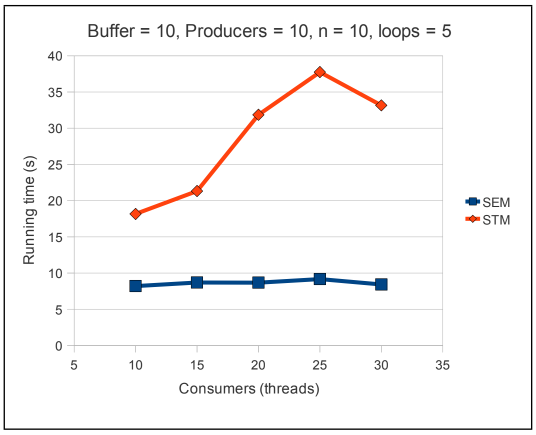

Figure 4.7. Work inside critical section in consumer threads

Memory consumption

The results can be seen in figure 4.8, figure 4.9 and figure 4.10.

Scalability

The results can be seen in figure 4.11, figure 4.12 and figure 4.13.

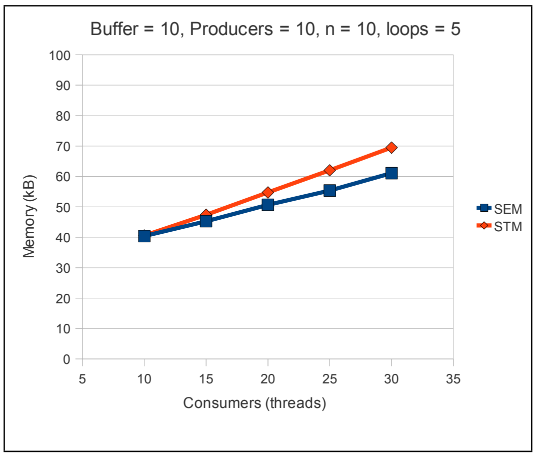

Figure 4.8. No work in consumer threads

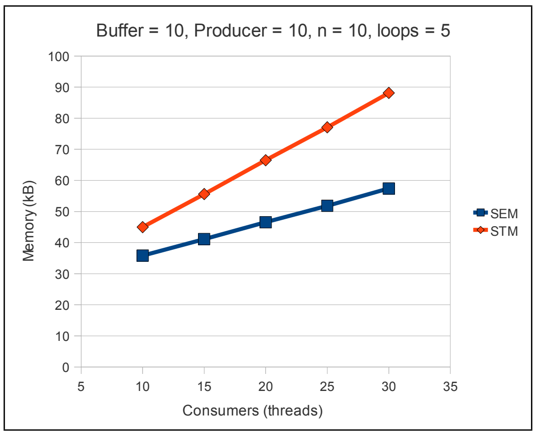

Figure 4.9. Work outside critical section in consumer threads

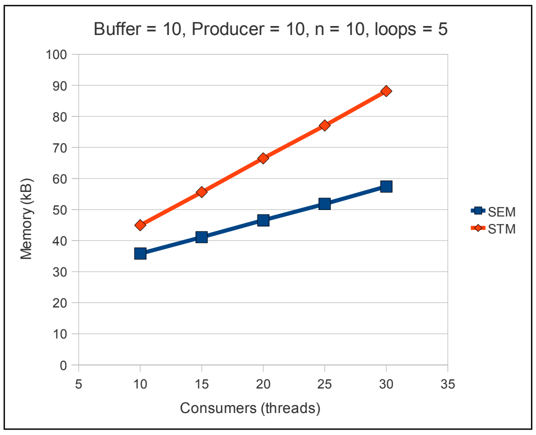

Figure 4.10. Work inside critical section in consumer threads

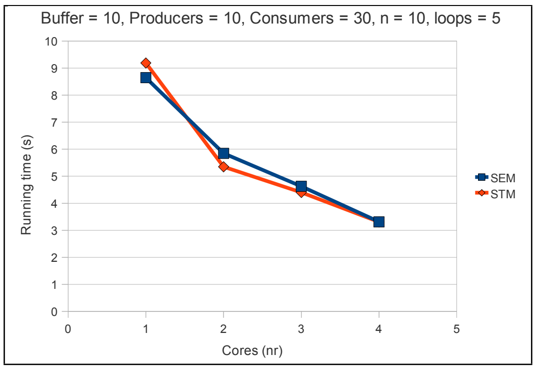

Figure 4.11. No work in consumer

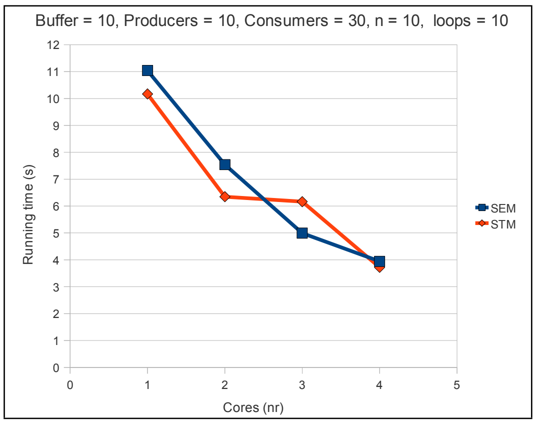

Figure 4.12. Work outside the critical section in consumer threads

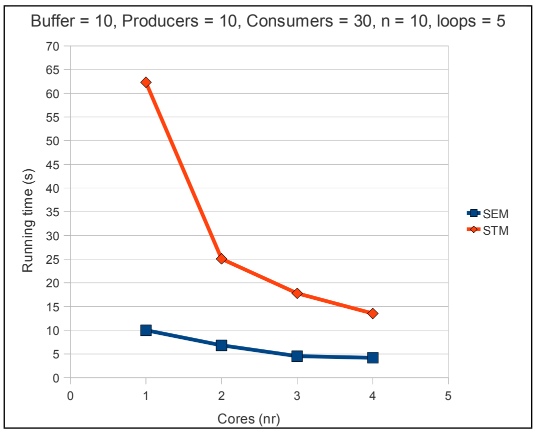

Figure 4.13. Work inside the critical section in consumer threads

4.2 Discussion

To begin with, we would like to point out that the problems used in this evaluation are by no means a complete representation of all kinds of concurrency problems. Hence, the results, discussion and conclusion should be read with this in mind.

We would also like to point out that there are several aspects of concurrency that we haven't benchmarked and which relate more to software engineering. Among these are how long time it takes to implement the different solutions, how easy they are to maintain as well as how easy it is to compose different modules.

In the results, figure 4.7 clearly shows that the STM solution performs worse than the semaphore solution. This is most certainly due to Haskell's use of optimistic concurrency control and deferred updates in the implementation of STM. Each transaction will begin by altering the buffer, which is followed by a computation. When the transaction tries to validate, it is likely that another transaction has finished earlier and updated the state of the buffer. In this case, the first transaction will not validate and therefore has to implicitly call `retry`, forcing the computation in the transaction to be redone. This causes the STM solution to waste time when performing the same computation over and over again.

This can be further verified by looking at figure 4.13. The more cores the STM solutions gets, the more transactions it performs in parallel. Therefore, the running time will decrease, since the additional unnecessary computations will be distributed over many cores. Perhaps, this problem will be marginalized in the future, since CPUs are likely to get more and more cores.

It is also noticeable that the running time of the semaphore solution almost didn't increase between the tests in figure 4.6 and figure 4.7. One might expect the running time to be worse when the computation takes place while locking of the buffer. The reason for this is that the computations performed in the critical section are small and that the semaphore based solutions never redo any work.

In all the tests, the memory consumed by the solutions increased linearly with the input. This was true for both the STM and the semaphore solutions, which can be seen in all the diagrams regarding memory consumption. The memory consumption of the STM solutions seem to grow a little bit faster than the semaphore solutions, but only by a constant factor.

Our experiments imply that the STM based solutions scale at least as well as the semaphore based solutions as long as the work done in the critial sections is minimal. This is not a big surprise, since less work in the critical sections is likely to cause fewer conflicts and also results in less overhead when transactions abort and retry.

However, the tests do not tell us much about how well the two approaches work as the size of the application increases. Our test programs are small and have fine lock granularity, which leads to low locking overhead. In general, it is hard or even impossible to maintain such fine lock granularity in a larger system – it is not uncommon to lock entire objects or parts of the system (for example using monitors) just to be on the safe side.

Since STM code is not subject to deadlocks and similar issues, it is much easier to keep the transactions small than it is to keep the lock granularity fine. So even though STM performs worse than semaphores when the work done in the critical section increases, it is possible that the overhead of using STM is more or less negligible in a larger system.

4.3 Conclusion

Not all concurrency problems are naturally modelled as transactions. Our STM based solution to the Bathroom Problem is a good example of this. The solution looks very much like the semaphore based solution but with the semaphores replaced by small transactions checking some condition. This leads us to think that programmers using STM still need to understand the fundamental problems of concurrency to some extent, and that it is important to understand in which contexts STM is suitable. This also includes understanding the performance issues when a transaction performs a lot of work.

However, we feel that it is more straightforward to model the solutions with transactions. It is also significantly easier to reason about the correctness and we believe that this is one of the big advantages of STM.

Given the results of our study, we think that the efficiency of STM is in many cases sufficiently good for it to be used as an alternative to the traditional concurrency model using sempahores.

Bibliography

[1] Concurrent programming, March 2010. `http://en.wikipedia.org/wiki/Concurrent_computing`.

[2] Eight queens puzzle, April 2010. `http://en.wikipedia.org/wiki/Eight_queens_puzzle`.

[3] Edsger W. Dijkstra. Cooperating sequential processes. pages 65–138, 2002.

[4] Allen B. Downey. *The Little Book of Semaphores*. Green Tea Press, `http://greenteapress.com/index.html`, 2008.

[5] K.A. Fraser. Practical lock-freedom, ph.d. thesis, university of cambridge. *Technical Report, No. 579, February 2004*.

[6] Tim Harris and Keir Fraser. Language support for lightweight transactions. *SIGPLAN Not.*, 38(11):388–402, 2003. ISSN 0362-1340.

[7] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable Memory Transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, New York, NY, USA, 2005. ACM. ISBN 1-59593-080-9.

[8] C. Herzeel, P. Costanza, and T. D'Hondt. Reusable building blocks for software transactional memory. In *Second European Lisp Symposium (ELS'09)*, 2009.

[9] Paul Hudak. Conception, evolution, and application of functional programming languages. *ACM Comput. Surv.*, 21(3):359–411, 1989. ISSN 0360-0300.

[10] Don Jones, Jr., Simon Marlow, and Satnam Singh. Parallel performance tuning for Haskell. In *Haskell '09: Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, pages 81–92, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-508-6.

[11] Simon P. Jones. *Beautiful Concurrency*. O'Reilly Media, Inc., 2007. ISBN 0596510047. URL `http://research.microsoft.com/Users/simonpj/papers/stm/index.htm`.

[12] Simon Peyton Jones. Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In *Engineering theories of software construction*, pages 47–96. IOS Press, 2001. ISBN 1-58603-1724.

[13] Jim Larus and Ravi Rajwar. *Transactional Memory (Synthesis Lectures on Computer Architecture)*. Morgan & Claypool Publishers, 2007. ISBN 1598291246.

[14] D. B. Lomet. Process structuring, synchronization, and recovery using atomic actions. *SIGPLAN Not.*, 12(3):128–137, 1977. ISSN 0362-1340.

[15] Simon Marlow, Simon Peyton Jones, and Satnam Singh. Runtime support for multicore Haskell. *SIGPLAN Not.*, 44(9):65–78, 2009. ISSN 0362-1340.

[16] Bryan O'Sullivan. Criterion, a new benchmarking library for haskell, September 2009. `http://www.serpentine.com/blog/2009/09/29/criterion-a-new-benchmarking-library-for-haskell/`.

[17] Cristian Perfumo, Nehir Sönmez, Srdjan Stipic, Osman Unsal, Adrián Cristal, Tim Harris, and Mateo Valero. The limits of software transactional memory (STM): dissecting Haskell STM applications on a many-core environment. In *CF '08: Proceedings of the 5th conference on Computing frontiers*, pages 67–78, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-077-7.

[18] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 295–308, New York, NY, USA, 1996. ACM. ISBN 0-89791-769-3.

[19] Nir Shavit and Dan Touitou. Software transactional memory. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213, New York, NY, USA, 1995. ACM. ISBN 0-89791-710-3.

[20] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database Systems Concepts*. McGraw-Hill Science/Engineering/Math, fourth edition, 2001. ISBN 0072554819.

[21] Nehir Sonmez, Tim Harris, Adrian Cristal, Osman S. Unsal, and Mateo Valero. Taking the heat off transactions: Dynamic selection of pessimistic concurrency control. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–10, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-3751-1.

[22] M. F. Spear, V. J. Marathe, W. N. Scherer III, and M. L. Scott. Conflict detection and validation strategies for software transactional memory. In *DISC*, 2006.

[23] Martin Sulzmann, Edmund S.L. Lam, and Simon Marlow. Comparing the performance of concurrent linked-list implementations in Haskell. *SIGPLAN Not.*, 44(5):11–20, 2009. ISSN 0362-1340.

[24] `libraries@haskell.org`. Control.Concurrent.MVar, March 2010. `http://www.haskell.org/ghc/docs/latest/html/libraries/base/Control-Concurrent-MVar.html`.

[25] X. Wang, Z. Ji, C. Fu, and M. Hu. A review of transactional memory in multicore processors. *Information Technology Journal 9 (1): 192-200*, 2010.

Appendix A

Source Code

Here is the relevant source code for the two problems. The complete source code can be found at `http://bitbucket.org/helino/bachelor-thesis`.

A.1 Producer-Consumer problem

A.1.1 Semaphore

```haskell
module ProdCons(test) where

import qualified Data.Sequence as S
import Control.Concurrent
import Control.Concurrent.MVar
import Control.Concurrent.QSem

import Producer
import Consumer
import System(getArgs)

test :: [Int] -> IO()
test (bufSize:prods:cons:queens:loops:[]) =
    do
        spaces <- newQSem bufSize
        items <- newMVar 0
        mutex <- newQSem 1
        buffer <- newMVar (S.empty)
        prodAvail <- newMVar (prods)
        pvars <- spawnProducers prods spaces items mutex buffer
            queens loops prodAvail []
        cvars <- spawnConsumers cons spaces items mutex buffer
            prodAvail []
        joinThreads (pvars ++ cvars)


joinThreads :: [(MVar ())] -> IO()
joinThreads [] = return ()
joinThreads (fst:rest) =
    do
        takeMVar fst
        joinThreads rest

spawnProducers :: Int -> QSem -> MVar Int -> QSem -> MVar (S.Seq
    Int) -> Int -> Int -> MVar Int -> [MVar ()] -> IO [MVar ()]
spawnProducers 0 _ _ _ _ _ _ _ _ list = return list
spawnProducers n spaces items mutex buffer queens loops prodAvail
    list =
    do
        joiner <- newEmptyMVar  :: IO (MVar ())
        forkIO $ produce spaces items mutex buffer queens loops
            joiner prodAvail
        spawnProducers (n-1) spaces items mutex buffer queens loops
            prodAvail (joiner:list)

spawnConsumers :: Int -> QSem -> MVar Int -> QSem -> MVar (S.Seq
    Int) -> MVar Int -> [MVar ()] -> IO [MVar ()]
spawnConsumers 0 _ _ _ _ _ _ list = return list
spawnConsumers n spaces items mutex buffer prodAvail list =
    do
        joiner <- newEmptyMVar  :: IO (MVar ())
        forkIO $ consume spaces items mutex buffer joiner prodAvail
        spawnConsumers (n-1) spaces items mutex buffer prodAvail (
            joiner:list)
```

```haskell
module Producer (produce) where

import Control.Concurrent.MVar
import Control.Concurrent.QSem
import qualified Data.Sequence as S
import Queens

produce :: QSem -> MVar Int -> QSem -> MVar (S.Seq Int)-> Int ->
    Int -> MVar () -> MVar Int -> IO()
produce _ _ _ _ _ _ 0 joiner prodAvail =
    do
        left <- takeMVar prodAvail
        putMVar prodAvail (left -1)
        putMVar joiner ()
        return ()
produce spaces items mutex buffer queens loops joiner prodAvail =
    do
        (putBuffer items spaces mutex buffer) $! (nsoln queens)
        produce spaces items mutex buffer queens (loops-1) joiner
            prodAvail

putBuffer :: MVar Int -> QSem -> QSem -> MVar (S.Seq Int) -> Int ->
    IO ()
putBuffer items spaces mutex buffer num =
    do
```

```
        waitQSem spaces
        waitQSem mutex

        seq <- takeMVar buffer
        putMVar buffer (seq S.|> num)

        signalQSem mutex
        signalQSem spaces
        numItems <- takeMVar items
        putMVar items (numItems+1)
```

```
module Consumer (consume) where

import Control.Concurrent.MVar
import Control.Concurrent.QSem
import qualified Data.Sequence as S
import Queens

consume :: QSem -> MVar Int -> QSem -> MVar (S.Seq Int)-> MVar ()
    -> MVar Int -> IO()
consume spaces items mutex buffer joiner prodAvail =
    do
        numItems <- takeMVar items
        if numItems == 0
            then do numLeft <- takeMVar prodAvail
                    putMVar items numItems
                    putMVar prodAvail numLeft

                    if numLeft == 0
                        then finish joiner
                        else consume spaces items mutex buffer
                            joiner prodAvail
            else do
                    putMVar items (numItems-1)
                    waitQSem mutex

                    seq <- takeMVar buffer
                    let num = S.index seq 0
                    putMVar buffer (S.drop 1 seq)
                    signalQSem mutex
                    signalQSem spaces

                    consume spaces items mutex buffer joiner
                        prodAvail

release :: QSem -> MVar Int -> QSem -> MVar (S.Seq Int)-> MVar ()
    -> MVar Int -> Int -> IO()
release spaces items mutex buffer joiner prodAvail n =
    do
        signalQSem mutex
        signalQSem spaces

        consume spaces items mutex buffer joiner prodAvail

finish :: MVar () -> IO()
finish joiner =
    do
        putMVar joiner ()
        return ()
```

A.1.2 STM

```
module ProdCons(test) where

import GHC.Conc
import qualified Data.Sequence as S
import System(getArgs)
import Queens

test :: [Int] -> IO()
test (bufSize:prods:cons:queens:loops:[]) =
    do
        buffer <- newTVarIO (S.empty :: S.Seq Int)
        prodsAvail <- newTVarIO prods
        ptvs <- spawnProducers prods buffer queens bufSize loops
            prodsAvail []
        ctvs <- spawnConsumers cons buffer prodsAvail []
        joinThreads (ptvs ++ ctvs)

joinThreads :: [TVar Int] -> IO ()
joinThreads [] = return ()
joinThreads (h:t) =
    do
        atomically $ do val <- readTVar h
                        if val == 1
                            then retry
                            else return ()
        joinThreads t

spawnProducers :: Int -> TVar (S.Seq Int) -> Int -> Int -> Int ->
    TVar Int -> [TVar Int]-> IO [TVar Int]
spawnProducers 0 _ _ _ _ _ list = return list
spawnProducers n buffer queens bufSize loops prods list =
    do
        sync <- newTVarIO (1 :: Int)
        forkIO(produce buffer queens bufSize loops prods sync)
        spawnProducers (n-1) buffer queens bufSize loops prods (
            sync:list)

spawnConsumers :: Int -> TVar (S.Seq Int) -> TVar Int -> [TVar Int]
     -> IO [TVar Int]
spawnConsumers 0 _ _ list = return list
spawnConsumers n buffer prods list =
    do
        sync <- newTVarIO (1 :: Int)
        forkIO(consume buffer prods sync)
        spawnConsumers (n-1) buffer prods (sync:list)

consume :: TVar (S.Seq Int) -> TVar Int -> TVar Int -> IO()
consume buffer prods sync =
    do
        n <- atomically $ do seq <- readTVar buffer
                             if S.length seq == 0
                                then do left <- readTVar prods
                                        if left == 0
                                           then return (-1)
                                           else retry
                                else do let num = S.index seq 0
                                        writeTVar buffer (S.drop 1 seq)
                                        return num
        if n == -1
            then finish sync
            else do consume buffer prods sync

finish :: TVar Int -> IO ()
finish sync =
    do
        atomically $ writeTVar sync 0
        return ()

produce :: TVar (S.Seq Int) -> Int -> Int -> Int -> TVar Int -> TVar Int -> IO()
produce _ _ _ 0 prods sync =
    do
        atomically $ do avail <- readTVar prods
                        writeTVar prods (avail-1)
                        writeTVar sync 0
        return ()
produce buffer queens bufSize loops prods sync =
    do
        (putBuffer buffer bufSize) $! (nsoln queens)
        produce buffer queens bufSize (loops-1) prods sync

putBuffer :: TVar (S.Seq Int) -> Int -> Int -> IO()
putBuffer buffer bufSize num =
    do
        atomically $ do seq <- readTVar buffer
                        if S.length seq >= bufSize
                            then retry
                            else writeTVar buffer (seq S.|> num)
```

A.2 Bathroom problem

A.2.1 Semaphore

```haskell
module Bathroom(test) where

import Control.Concurrent.QSem
import Control.Concurrent
import Lightswitch
import System.Random
import Queens

data Gender = Male | Female
    deriving (Eq, Show)

test :: [Char] -> Int -> IO()
test glist queens =
    do
        space <- newQSem 3
        empty <- newQSem 1
        io <- newQSem 1
        turnstile <- newQSem 1

        maleLocks <- newSwitchlocks
        femLocks <- newSwitchlocks

        mvs <- spawnThreads (c2g glist []) queens space io empty
            turnstile maleLocks femLocks []
        joinThreads mvs

c2g :: [Char] -> [Gender] -> [Gender]
c2g [] list = reverse list
c2g ('M':rest) list = c2g rest (Male:list)
c2g ('m':rest) list = c2g rest (Male:list)
c2g ('F':rest) list = c2g rest (Female:list)
c2g ('f':rest) list = c2g rest (Female:list)
c2g (h:t) list = c2g t list

joinThreads :: [MVar ()] -> IO()
joinThreads [] = return ()
joinThreads (h:t) =
    do
        takeMVar h
        joinThreads t

spawnThreads :: [Gender] -> Int -> QSem -> QSem -> QSem -> QSem ->
    Switchlocks -> Switchlocks -> [MVar ()] -> IO [MVar ()]
spawnThreads [] _ _ _ _ _ _ _ _ list = return list
spawnThreads (h:t) queens space io empty turnstile maleLocks
    femLocks list =
    do
        sync <- newEmptyMVar :: IO (MVar ())
        if h == Male
```

```
            then forkIO $ calc Male space io empty turnstile maleLocks queens sync
            else forkIO $ calc Female space io empty turnstile femLocks queens sync

        spawnThreads t queens space io empty turnstile maleLocks femLocks (sync:list)

calc :: Gender -> QSem -> QSem -> QSem -> QSem -> Switchlocks -> Int -> MVar () -> IO()
calc gender space io empty turnstile switchlocks queens sync =
    do
        waitQSem turnstile
        lock switchlocks empty
        signalQSem turnstile
        waitQSem space

        handleResult $! (nsoln queens)
    where
        handleResult = \num -> do signalQSem space
                                  unlock switchlocks empty
                                  putMVar sync ()
```

A.2.2 STM

```
module Bathroom where

import Control.Concurrent
import GHC.Conc
import System.Random
import Queens

data Gender = Male | Female | None
    deriving (Show, Eq)

test :: [Char] -> Int -> IO()
test glist queens =
    do occupier <- newTVarIO (None :: Gender)
       visitors <- newTVarIO (0 :: Int)
       turnstile <- newTVarIO(False :: Bool)
       tvs <- spawnThreads (c2g glist []) queens occupier visitors turnstile []
       joinThreads tvs

c2g :: [Char] -> [Gender] -> [Gender]
c2g [] list = reverse list
c2g ('M':rest) list = c2g rest (Male:list)
c2g ('m':rest) list = c2g rest (Male:list)
c2g ('F':rest) list = c2g rest (Female:list)
c2g ('f':rest) list = c2g rest (Female:list)
c2g (h:t) list = c2g t list
```

```haskell

joinThreads :: [TVar Int] -> IO ()
joinThreads [] = return ()
joinThreads (h:t) =
    do
        atomically $ do val <- readTVar h
                        if val == 1
                            then retry
                            else return ()
        joinThreads t

spawnThreads :: [Gender] -> Int -> TVar Gender -> TVar Int -> TVar Bool -> [TVar Int] -> IO [TVar Int]
spawnThreads [] _ _ _ _ list = return list
spawnThreads (h:t) queens occupier visitors turnstile list =
    do
        sync <- newTVarIO (1 :: Int)

        if h == Female
            then forkIO $ visit Female occupier visitors turnstile queens sync
            else forkIO $ visit Male occupier visitors turnstile queens sync

        spawnThreads t queens occupier visitors turnstile (sync:list)

visit :: Gender -> TVar Gender -> TVar Int -> TVar Bool -> Int -> TVar Int -> IO()
visit g occ visitors turnstile queens sync =
    do
        atomically $ do queue <- readTVar turnstile
                        if queue
                            then retry
                            else writeTVar turnstile True
        atomically $ lock g occ visitors
        atomically $ writeTVar turnstile False
        handleResult $! (nsoln queens)
    where
        handleResult = \num -> do atomically $ do unlock occ visitors
                                                  writeTVar sync 0
lock :: Gender -> TVar Gender -> TVar Int -> STM ()
lock g occ visitors =
    do
        curr <- readTVar occ
        if (g == curr) || (curr == None)
            then do count <- readTVar visitors
                    if(count < 3)
                        then do writeTVar visitors (count+1)
                                if curr == None
                                    then writeTVar occ g
                                    else return ()
                        else retry
```

```
                else retry

unlock :: TVar Gender -> TVar Int -> STM ()
unlock occ visitors =
    do
        curr <- readTVar visitors
        writeTVar visitors (curr-1)
        if curr == 1
            then writeTVar occ None
            else return ()
```