CHRISTIAN-ALBRECHTS-UNIVERSITY

MASTER THESIS

Alternative Software Transaction Implementation in Haskell

Author:
Lasse Folger

Supervisor:
Dr. Frank Huch

A thesis submitted in fulfillment of the requirements for the degree of Master of Science

in the

Programming Languages and Compiler Construction
Department of Computer Science

February 16, 2017

Declaration of Authorship

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel, February 16, 2017

Christian-Albrechts-University

Abstract

Faculty of Engineering
Department of Computer Science

Master of Science

Alternative Software Transaction Implementation in Haskell

by Lasse Folger

TODOTODOTODOTODOTODOTODOTODO
The Thesis Abstract is written here (and usually kept to just this page). The page is kept centered vertically so can expand into the blank space above the title too…

Acknowledgements

TODOTODOTODOTODOTODOTODO
The acknowledgments and the people to thank go here, don't forget to include your project advisor…

Contents

List of Figures

List of Tables

List of Abbreviations

LAH	**L**ist **A**bbreviations **H**ere
WSF	**W**hat (it) **S**tands **F**or
STM	**S**oftware **T**ransactional **M**emory
ACID	**A**tomicity **C**onsistency **I**solation **D**urability
TVar	**T**ransactional **Var**iable

Chapter 1

Motivation

Modern computer architecture includes multicore processors. To utilize these multicore system to their full extend, concurrent and parallel programming is needed. By this new challenges arise. One challenge is the logical issue of splitting the problem in smaller problems which can be processed by different threads in parallel. Aditionally there are technical challenges. For example a new scheduler is needed and hardware accesses (Printer, Display, etc.) need to be coordinated. These are challenges the operating system usually handles. The are other challenges the operating system cannot handle, because they are specific for every program.

The most discussed challenge is the synchronization. If a program works with multiple threads, these threads usually communicate. Communications means to exchange data. Even a simple statement like an assignment can cause problem when used in the parallel threads. The problem is that these operations are non atomic operations. Thus (`x = x + 1`) consist of three parts. first reading the old value, second adding `1`, and thrid write the new value. This means two threads in parallel can both read the old value, then both add `1` to the old value, and then write back the new value. The new value is the initial value incremented by `1`, even though two threads executed an increment operation on this value. This non inteded behaviour is called *lost update*. The efforts to avoid non intended behaviour such as this are called synchronization.

Even though multicore processors are new, the research in the field of synchronization has a long history, starting with (Dijkstra, 1965), which introduces the most basic synchronization tool, the semaphore. The semaphore is a abstract datatype which holds an Interger and provides two *atomic* operations, `P` and `V`. If the value of the semaphore is greater than `0`, `P` decrements the semaphore. If the value of the semaphore is `0` the thread that evoked `P` is suspended. When a thread evokes `V` the value of the semaphore is increased and in the case another thread is currently suspended, because it called `P` on the semaphore, that thread is awakened. After the thread is awakened, it tries `P` again.

This seem to be a simple construct, but its capabilities are enormous. It is highly complex to use a semaphore correctly. The main problem of semaphores is the so called deadlock[1]. This means there is a schedule, where no progress of the systen is possible, because all threads are waiting for a semaphore. The term deadlock is not exclusive for semaphores. It is used for all blocking mechanisms. To avoid such deadlocks is very hard even when using one or few semaphores. It is nearly impossible to avoid deadlocks when you trie to compose semaphore based functions.

To avoid the problems of semaphores while maintaining the expressiveness of semaphores the so called software transactions were introduced (Harris, Marlow, et al., 2005). Sofware transactions are inspired by the long known database transactions

[1] In the course of this thesis I will refer to deadlocks as a static propertie rather than a state of a system.

(Gray and Reuter, 1992). Software transactions provide an interface to program with single element buffers. If you are using this interface the underlying implementation ensures the so called *ACI(D)* properties. **A** for atomicity. This means a transactions appears to be processed instantaneous. **C** for consistency. This means that a consistent view of the system is always guaranteed. **I** stands for isolation. This means the programmer does not need to worry about concurrency and every thread can act as if it were the only thread. **D** stand for durability, but is relevant only for data base transactions.

There is a stable implementation for software transactions in Haskell, namely Software Transactional Memory (called STM in the following). STM provides the ACI(D) properties by optimistically executing the transaction. If a conflict is detected, the changes are discarded and the transactions is restarted (a so called rollback). This works, but is not optimal with regards to efficiency and performance. There are two problems. First the conflict detection. Sometimes the implementation detects a conflict and evokes a rollback, even though it is not necessary. The second problem is the rollback mechanism. Regardless of the conflict, always the whole transaction is reexecuted. This includes operations on data that has not changed, thus an unnecessary recomputation. These problems are discussed in detail in Chapter 2. The aim of this thesis is to provide an alternative implementation that avoids these problems while preserving the ACI(D) properties.

Chapter 2

Introduction

2.1 Software Transactional Memory

Software Transactional Memory (STM in the following) is a programming language independent synchronization concept. Today STM is available in all common programming languages[1]. To understand the benefits of STM, take a look at the following example:

```
type Account = MVar Int

transfer :: Account -> Account -> Int -> IO ()
transfer src dst am = do
  balSrc <- takeMVar src
  balDst <- takeMVar dst
  putMVar src (balSrc - am)
  putMVar dst (balDst + am)
```

This is a simple implementation of a bank account and an associated transfer function. This implementation uses an `MVar` for synchronization. An `MVar` is a buffer with a capacity of one. This buffer can either be empty or filled. If the MVar is empty, every `takeMVar` operation on this MVar blocks until it is filled. If the MVar is filled, `takeMVar` empties the `MVar` and return the value. `putMVar` is the opposite operation. It fills the MVar with a value, if it is empty and suspends if the MVar is already filled.

This means `transfer` first empties both `Accounts`, then modifies the balances and at last writes back the new balances. At first glance this function seems to work fine, but the following example contains a deadlock:

Thread 1:

```
main = do
  transfer acc1 acc2 50
```

Thread 2:

```
main = do
  transfer acc2 acc1 50
```

The problem is the mutual access of the MVars. If both threads take their `src` at the same time, they will both wait for `dst` [2]. To avoid this deadlock we can rewrite the code:

```
transfer src dst am = do
  srcBal <- takeMVar src
  putMVar src (srcBal - am)
  dstBal <- takeMVar dst
  putMVar dst (dstBal + am)
```

[1]Even though STM is language independent, I will present the STM library in Haskell since this thesis is about STM in Haskell

[2]In fact is `transfer acc1 acc1 50` enough to evoke a deadlock

This indeed solves the problem regarding the deadlock. In return we lose consistency. For a brief moment we see an inconsistent state. Since the amount is allready subtracted from one account, but not yet added to the other account. This inconsistent state is observable by other threads. This is not possible in the first implementation.

We can use STM to avoid these problems. STM provides a single element buffer named `TVar`. A TVar always holds an value and is never empty. TVars are read and written with the functions `readTVar` and `writeTVar`, respectively. In contrast to `putMVar` and `takeMVar`, the TVar operations are not `IO` actions but `STM` action[3]. `STM` is an instance of Monad, hence multiple `STM` actions can be combined using the comfortable do-notation. The following code represents the example from above implemented with TVars instead of MVars:

```
type Account = TVar Int

transfer :: Account -> Account -> Int -> STM ()
transfer src dst am = do
  srcBal <- readTVar src
  dstBal <- readTVar dst
  writeTVar src (srcBal - am)
  writeTVar dst (dstBal + am)
```

Note the type of transfer is no longer an `IO` action, but an `STM` action. Apart from this the code looks similar to the MVar version.

In order to execute a transaction the function `atomically :: STM a -> IO a` is used. Since `readTVar` and `writeTVar` do not lock the TVar, the following example contains no deadlock:

Thread 1:

```
main = do
  atomically $
    transfer acc1 acc2 50
```

Thread 2:

```
main = do
  atomically $
    transfer acc2 acc1 50
```

This is because STM ensures the *ACID* properties. The ACID properties were Introduced in (Gray and Reuter, 1992) for database transactions. These properties were adapted for software transactions later on. In the case of software transactions the ACID properties mean the following:

- *Atomicity*: the transaction executes all operations or none.
- *Consistency*: all modifications of a transaction are committed at the same time. No transition state is observable.
- *Isolation*: no concurrency is observable by a transaction. Each transaction can work as if it is the only transaction.
- *Durability*: ensures the perseverance of the changes. In the case of software transactions this is not necessary.

These properties explain the name `atomically`, because the enclosed code appears to be executed instantaneously without any interactions with other threads. Before we turn over to the implementation of STM, we take a deeper look at the interface of the STM.

[3]If you are wondering when I use `SMT` and when STM. I use `STM` when I refer to the Haskell type constructor and STM when I refer to STM as library

`newTVar :: a -> STM (TVar a)` creates a newTVar. Since a TVar always holds a value, an initial value has to be passed to create a TVar. There is no function like `newEmptyTVar`.

Besides functions to create and access TVars, there are functions to alter the control flow. `retry :: STM a` is a generic STM action that indicates a failure, thus whenever a transaction engages a `retry` it restarts. The transaction is **not** restarted immediately. The transaction restarts, if at least one of the TVars it has read is modified. If the transaction would restart immediately (and no TVar has changed), the transaction would run into the same `retry` again.

With `orElse :: STM a → STM a → STM a` you are able to express alternatives. `orElse` executes the first transaction and ignores the second transaction, if the first transaction is successful. If the first transaction fails (retries), the second transaction is executed instead.

Note that it is not possible to execute `IO` action within a transaction, which means that not side effects can occur. Furthermore this means restarting a transaction will never lead to the re-execution of irreversible operations. The reason is that the computations of transactions are done within the `STM` Monad. In other words the type system of Haskell forces us to write correct transactions.

For single threaded programming, abstraction and composability are key features. These features allow us to combine smaller pieces of code into more complex pieces of code. These feature are not available for lock based concurrent programming. Compose correct lock based concurrent functions most likely lead to deadlocks or inconsistencies. Consider the following example:

```
withdraw :: Account -> Int -> IO()
withdraw acc am = do
  bal <- takeMVar acc
  putMVar acc (bal - am)

deposit :: Account -> Int -> IO()
deposit acc am = withdraw acc (-am)
```

These are functions to withdraw and deposit money from and on an account. The natural way to implement `transfer` is:

```
transfer :: Account -> Account -> Int -> IO()
transfer src dst am = do
  withdraw src am
  deposit dst am
```

We reuse the functions that are already defined instead of coding everything from the scratch. In our example this is equivalent to the solution suggested above to eliminate the deadlock. This implementation is free of deadlocks, but it lacks consistency. Thus building complex concurrent operations can not take advantage of abstraction and composability. We always need to code everything from the scratch. This is error prone in compared to the step wise combination of smaller operations into bigger operations.

STM allows us to use this important programming paradigm for concurrent programming. Thus the following example provides deadlock freedom as well as consistency.

```
withdraw :: Account -> Int -> STM()
withdraw acc am = do
  bal <- readTVar acc
```

```
  writeTVar acc (bal - am)

deposit :: Account -> Int -> STM()
deposit acc am = withdraw acc (-am)

transfer :: Account -> Account -> Int -> STM()
transfer src dst am = do
  withdraw src am
  deposit dst am
```

We can combine arbitrary transactions to more complex transactions while preserving the ACI properties. This greatly benefits the readability of the code. In addition it increases the efficiency of the development process, because we are able to reuse code that we already found to be correct. This was also one of the main motivations of paper (Harris, Marlow, et al., 2005) which forms the foundation of STM in Haskell.

2.2 Implementation

In this we explore the current implementation of STM in Haskell, more specific in GHC. For a detailed description of the implementation refer to `https://ghc.haskell.org/trac/ghc/wiki/Commentary/Rts/STM`.

Even though the current implementation uses a low level C-library, we retain an abstract view on the implementation, since the technical details are not important for the course of this thesis. An abstract view on the implementation is presented to understand how the ACI properties are ensured.

The execution of a transaction (a call of `atomically`) is split in two phases. First the computation phase and second the commit phase.

2.2.1 Computation Phase

Each transaction holds a log for the TVars it has accessed. The log contains four elements per entry. These are:

- tvar
- expectedValue
- newValue
- versionNumber

The `versionNumber` is only used to prevent a very subtle bug and thus not considered in this thesis. The log is extenden and modified by the transactional operations `writeTVar` and `readTVar`. `newTVar` on the other hand creates the new TVar directly. Whenever `readTVar` is called the associated TVar is lookep up in the log. If it is present, the `newValue` is returned. If it is not present, a new entry for the log is created. While `tvar` is the passed TVar, `newValue` and `expectedValue` are the actual value of the TVar and `versionNumber` is the actual verison number. This is the one of the two times in the computation phase when the transaction accesses the actual mutable data structures. After the entry is created and added to the log, the actual value is returned.

A call of `writeTVar` also looks up the associated TVar in the log. If it is present, the field `newValue` is set to the value passed to writeTVar. If it is not present, a new entry is created. The `tvar` is the passed TVar and the `newValue` is the passed value and `expectedValue` is the actual value of that TVar. This is the other time the actual mutable data structures are accessed in the computation phase.

This log fulfills two purposes. One purpose of the log is the use in the commit phase which is described in 2.2.2. The other is the interaction between `readTVar` and `writeTVar`. The `readTVar` operations are able to see the results of preceding `writeTVar` operations in the log. Without the log `writeTVar` would need to access the actual TVar. This on the other hand would imply that other transactions would be able to see inconsistent intermediate states of the system; a violation of the ACI properties. It may seem unnecessary to read a TVar that the transaction itself wrote before. The transaction should know what it writes and thus does not need to access such TVars. Nevertheless there are two reasons to allow it. The current implementation allow the user to combine all transactional actions in an arbitrary manner and the library ensures (at compile time) that it works correctly. To restrict the user to only read TVars he has not yet written, no longer allows the library to give this kind of guarantee at compile time; this contradicts the design concept of Haskell. The second reason is one of the core motivations of STM in Haskell: composability. With the restriction it is not possible to combine arbitrary correct STM functions to new more complex STM functions.

Take a look at the following example:

```
transaction = do
  a <- readTVar t1
  b <- readTVar t2
  writeTVar t1 b
  writeTVar t2 a
```

This code would lead to the following log:

```
log  = {(t1,a,b),(t2,b,a)}
```

The log contains two entries, because the transaction accessed two TVars. The first entry contains the information that `t1` hold the value `a` when it was first read and the value `b` is the new value of it. `t2` hold the value `b` and the new value is `a`. Before we will examine the commit phase, we will look at the other operations of the STM interface.

`newTVar` creates a new TVar and initializes this TVar. Afterwards this TVar can be used like already existing TVars. Even if the transaction is rolled back, the new created TVars are not deleted explicitly. This work is done by the garbage collector, since the TVars are not further referenced, if the transaction that created them is rolled back.

`retry` aborts the computation and returns a results that indicates a failure. This result may be intercepted by `orElse` or is passed to `atomically` directly.

If `atomically` receives an result that indicates an failure, it aborts the transactions. Aborting a transaction means to discard the log. Since no observable operations are performed in the computation phase, nothing has to be undone. As soon as at least one of the TVars in the log has changed, the transaction is restarted. If the transaction is restarted immediately and no TVar has changed the transaction would reach the same `retry` again. These changes can be checked by comparing the `expectedValue` in the log with the actual value in the TVar. To avoid busy waiting the thread do not repeatedly check if the value has changed. The TVar has a queue for wait waiting threads. Each time a transaction successfully commits and writes a TVar it also checks if there is someone waiting in this queue. The committing thread then notifies all waiting threads.

`orElse` on the other hand reacts differently on the the result that indicates a failure. The implementation works with nested transactions, but to explain this in detail would go beyond the scope of this thesis. Nested transactions are not able to publish their writes on their own. When a nested transaction successfully commits(we see in the next section what this means), its log is integrated in the log of the surrounding transaction. Integrated means the logs are merged and in the case one entry is in both log the entry of the outer transaction is discarded. If the nested transaction fails, because `retry` occurred and it is the first transaction of `orElse`, the log of the inner transaction is integrated in the log of the surrounding transaction, but the `newValue` fields of the inner log are ignored. If the nested transaction fails to validate the outermost transaction is rolled back.

In conclusion the interface functions of STM are processed in the computation phase as follows:

- `writeTVar`: Look up TVar in log. If present update `newValue`. If not present read actual TVar and create new entry.
- `readTVar`: Look up TVar in log. If present return `newValue`. If not present read actual TVar and create new entry.
- `newTVar`: Create and initialize a new TVar.
- `retry`: Return a result that indicates a failure.
- `orElse`: Create a nested Transaction and reacts on the return value of that transaction.

2.2.2 Commit Phase

After the log is calculated and no further STM actions need to be processed, the commit phase starts. At first the transaction validates if the values in its log are still correct by *validating* its log. Validation denotes the process to check if the expectedValues are equal to the actual values in the TVars. In other words for each entry in the log the transaction reads the actual TVar and compares the value with the `expectedValue` in the log. If at least one of these values does not match, the transaction is considered *invalid*. If the validation returns the transaction is instantaneously rolled back, by discarding the log and restart it computation. If all values match the the transaction is considered *valid*. If the validation returns valid, each entry in the log is processed.

If `expectedValue` differs from `newValue` the associated TVar is locked. The transaction has acquired all locks it needs, it validates again. This seems a bit wasteful in terms of resources, but locking the TVars is considered an expensive operation and thus the implementation tries to avoid this when ever possible. This process reduces the chance that the transaction acquires all locks and then finds out it is invalid and consequently a unnecessary locking of TVars. If the validation fails at this point, the transaction is rolled back after the locks has been released. If the transaction has acquired all locks and is valid the transaction is is ready to publish its changes. This means iterating on the log and update the actual TVars where `expectedValue` and `newValue` differ and simultaneously releasing the locks.

If the validation returns invalid it means at least one `expectedValue` is no longer correct. To roll back is essential to retain the ACI properties. The failed validation indicates that transaction has read an outdated value and possibly worked with this value. Take a look at the following example:

```
transaction = do
  a <- readTVar t1
  writeTVar t1 (a+1)
```

If this transaction is processed by two transactions in parallel. Both would read the initial value of `t1`, say `1`. So both would note in their log `(t1,1,1)`. After the `writeTVar` the log of both transactions would look contain `(t1,1,2)`. After that both transactions try to commit. Assume one transaction commits before the other transaction tries. [4] Then the transaction would find its log to be valid and lock `t1`. After that its log is still valid and so it modifies `t1` and releases the log. Then the second transaction tries to commit. Since the actual value hast changed to `2` it does no longer match the `expectedValue` and the transaction is rolled back. If the transaction would not be rolled back at this point and commit instead. The transaction would write `2` to the TVar (that already contains `2`). In the end this would means the value of `t1` is `2` after both transactions have finished. This is certainly not the intended behaviour after incrementing the TVar that holds `1` twice. This is the well known *lost update* problem.

By rolling back the second transaction it reads `t1` once more. The log contains `(t1,2,2)` after the `readTVar` operation and `(t1,2,3)` after the `writeTVar` operation. Then the transaction validates, locks, validates and finally publishes it modifications. In the end the value of `t1` is `3`; just as intended.

2.2.3 Notes on the Implementation

Larus and Rajwar describe in their book(Larus and Rajwar, 2007, Chapter 2) different design options to be done when implementing a (Software) Transactional Memory. While most of these options effect only the performance of a system, some also effect the semantics of the system. We will discuss in this section the design options that are important for this thesis [5].

Deferred and Direct Updates

The way a STM system modifies the underlying data structures can either be *deferred* or *direct*. Direct updating systems are writing the actual objects when a write operation is called. In the case of Haskell this would mean, every time `writeTVar` is called. Deferred updating systems on the other hand buffer the write operations to commit them later on. Haskell STM is a deferred updating system, since the values are buffered in the writeSet before they are committed. This design options does not effect the semantics of the system. While a direct system loses performance, when a transaction is rolled back, because the initial values need to be restored, a deferred systems contains an overhead due to the need to log values and looking them up. Neither mechanism is better than the other in general; it depends on the application that STM is used in. (Harris, Plesko, et al., 2006) compares a deferred and a direct system. They show that the performance of a direct update system is significantly higher than that of a deferred system, when reads outnumber writes by far.

Early and Late Conflict Detection

A STM system needs to detect conflicts in order to ensure the ACI properties. This can be done as soon as the conflict occurs or later before the transaction commits. If

[4] For simplicity we assume that no other transaction is running besides the two we are looking at.

[5] The names used in the following part are taken from (Larus and Rajwar, 2007, Chapter 2)

the system uses a late conflict detection, transactions may work on an inconsistent state. This may lead to loops or exceptions. So this design decision is relevant for the semantics. Haskell STM uses a late conflict detection. By validating the log before comitting the transaction a possible conflict is detected. This implies the transaction may work on an inconsistent state until it attempts to commit. This means the transaction may run into an infinite loop, because it saw an inconsistent state. To avoid this problem, additional validations are performed each time the executing thread yields. Exceptions raised by the transaction are handled like a `retry`. If the log is valid, the transcation waits until at least one TVar changed. If it is invalid the transaction is restarted immediately. In conclusion, the user of STM can not observe that the transaction worked on an inconsistent state.

Synchronization

The last important property of a STM system is the way it synchronizes transactions. In order to validate correctly the systems needs to make sure the validation result does not depend on race conditions and is correct until the commit is completed. This means either concurrent transactions are delayed or their commit does not change the each others validity. In Haskell the first approach is taken. When a transaction commits, the TVars in the log that are updated are locked, thus other transactions that may conflict are not able to commit at the same time. In order to avoid a deadlock, all locks are released and the transaction is rolled back when it tries to aquire the lock for a locked TVar. In the worst case this leads to the roll back of both transactions, however the chances are narrow. Rolling back the transaction seems to be harsh instead of waiting until the other transaction finishes and then trying to commit, but if two transactions try to lock the same TVar, both transactions try to write this TVar. This means at least one of the transactions is rolled back, since the TVar is logged with the old value. Thus the first transaction to commit would modify a value in the log of the other transcation.

2.3 Problems

In this section we turn over to the problems in the current implementation. These problems can be examined independently. The first problem is about *when* a transaction is rolled back and the second problem is about *how* a transaction is rolled back.

2.3.1 Unnecessary Rollback

Remember the STM implementation of `transfer` and its example use given in 2.1:

```haskell
transaction1 = do
  atomically $
    transfer acc1 acc2 50
```

```haskell
transcation2 = do
  atomically $
    transfer acc2 acc1 50
```

The implmentation is correct, but not verry efficient in this case. Take a look at the inlined functions to understand the problem:

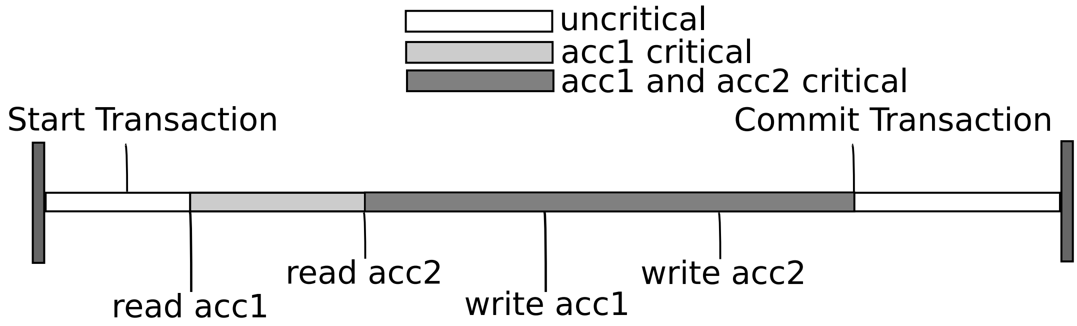

FIGURE 2.1: Time when the update of `acc1` or `acc2` causes a rollback.

```
transaction1 = do
  a1 <- readTVar acc1
  a2 <- readTVar acc2
  writeTVar acc1 (a1 - 50)
  writeTVar acc2 (a2 + 50)
```

```
transaction2 = do
  a1 <- readTVar acc2
  a2 <- readTVar acc1
  writeTVar acc2 (a1 - 50)
  writeTVar acc1 (a2 + 50)
```

Due to the scheduler the threads can run in a sequential order. This case may occur, but is not desirable. It means there is no performce improvement by executing this on multiple cores/processors. Thus the efforts to use multiple threads are futile in the first place. This is not a problem specific to STM, but to all synchronization mechanisms. If the resulting multi threaded program is not scheduled in a way that it is executed parallel, these mechanisms are a performance deterioration rather than a performance improvement. Since we cannot access the scheduler, we ignore this case.

The second case is that these transactions are run in parallel. This should be the better case, because the implementation has a chance to improve the performance. Sadly this is not the case. To understand why, we need to take a close look at the execution. Let us assume both threads execute their computation phase at the same time. This means both read the initial values of `acc1` and `acc2` and add these information to their log. Furthermore add both transactions entries for `writeTVar acc1` and `writeTVar acc2` to their log. Then both transactions try to commit, thus try to lock the TVars. It is possible both transactions are rolled back at this point. Lets assume `transaction1` acquires the locks for `acc1` and `acc2`. Since no TVars were modified after `transaction1` read them, it validates and commits. If `transaction2` tries to access the TVars before `transaction1` has finished committing, it is rolled back. Thus it is possible for `transaction2` to read the old value once again. If `transaction2` is descheduled for the time `transaction1` commits, it is rolled back afterwards, because the values of `acc1` and `acc2` have were changed by `transaction1`. In conclusion no performance improvement was achieved. The most efficient execution is if both transactions are executed in a sequential order. As mentioned before, this not desirable for multithreaded programs.

This leads to two questions:

- When is it needed to roll back a transaction?
- How can we avoid or at least decrease rollbacks?

A transaction needs to be rolled back if it is operating on data that is not a snapshot of the current memory. In other words if a value has changed after the transaction

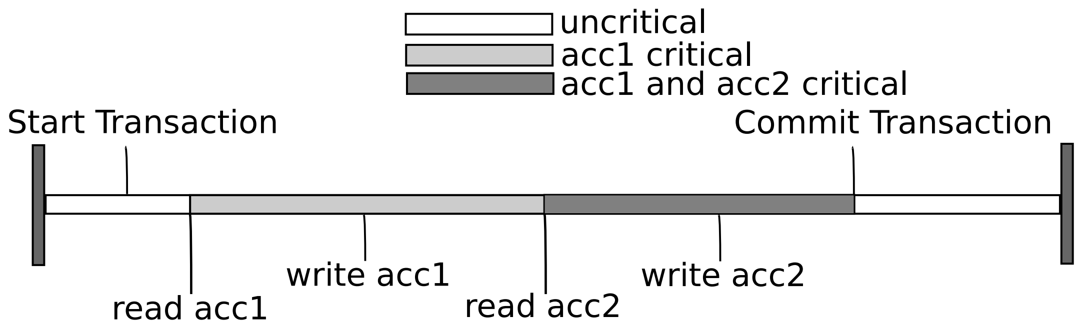

FIGURE 2.2: Effect of rearranging code with regards to the time `acc1` and `acc2` are critical for `transaction1`.

read this value. When a transactions reads a TVar, this TVar becomes *critical* for the transaction. Critical means a modifications of that TVar causes the transaction to roll back. Figure 2.1 visualizes when the TVars `acc1` and `acc2` are critical for `transaction1`. When `readTVar acc1` is executed the values becomes critical and stays critical until the transaction commits. If any other transaction commits a modification to `acc1` or `acc2`, while `acc1` and `acc2` are critical for `transaction1`, `transaction1` needs to be rolled back.

This insight brings an intuitive way to deal with this problem. If we minimize the time the TVars are critical for a transaction we reduce the chance that this transaction is rolled back. If we rearrange the operations of `transfer`, we are able reduce the time `dst` is critical. Note that we can rearrange the operations to a certain degree without changing the semantics of the resulting code due to the ACI properties.

```
transfer src dst am = do
  srcBal <- readTVar src
  writeTVar src (srcBal - am)
  dstBal <- readTVar dst
  writeTVar dst (dstBal - am)
```

With this implementation of transfer the time in that both TVars are critical is reduced. Figure 2.2 shows the effects of this for `transaction1`. The second TVar, namely `acc2`, is shorter critical than in the initial implementation. The time `acc1` is critical has not changed at all. Nevertheless it shows that delaying the execution of `readTVar` can reduce the time values are critical and by this the chance the transaction is rolled back. Our aim is to delay the execution of `readTVar` as far as possible to avoid reduce to time that a TVar is critical for the transaction. We have already seen one option to achieve this; rearrange the operations of a transaction. This would require a kind of preprocessing in the compiling process, for example a source to source code transformation. The aim of this thesis is to provide an pure Haskell library that can be used like one. I do not want to implement an extension to the compiler nor do I want to provide a source to source code transformer. The only other option is to alter implementation of `readTVar` and `writeTVar` without changing the *external* semantics of STM. External semantics are the semantics the user can observe and which effect the user written code.

Transfer basically consists of two parts. Decreasing the balance of the source acound and increasing the balance of the destination account. The actual values of `src` and `dst` are not important for these transactions. If we delay the evaluation of readTVar to the commit phase, we avoid the aforementioned **unnecessary** rollback, because no transaction would read a value, that is overwritten by another transaction afterwards.

If we refer to our example:

Thread 1:

```
transaction1 = do
  a1 <- readTVar acc1
  a2 <- readTVar acc2
  writeTVar acc1 (a1 - 50)
  writeTVar acc2 (a2 + 50)
```

Thread 2:

```
transaction2 = do
  a1 <- readTVar acc2
  a2 <- readTVar acc1
  writeTVar acc2 (a1 - 50)
  writeTVar acc1 (a2 + 50)
```

If we change the semantics of `readTVar` by delaying the evaluation, the following happens. Both transactions will execute the computation phase simultaneously. This means for `transaction1` adding `(acc1,(a1 - 50))` and `(acc2,(a2 + 50))` to their writeSet (this is analog for `transaction2`). At the first glance this seems to be incorrect since the value of `a1` and `a2` are not yet present. For Haskell this is quite common. Haskells is a non-strict language, which means passing unevaluated expressions is normal.

After the computation phase the commit phase follows. The first step is to lock the read TVars in order to perform the validation. Since both transactions used the same TVars, they will commit successively instead of parallel.

Assume `transaction1` gets the locks first and trys to validate[6]. Since the readSet is empty the validation is unnecessary; the empty readSet is always valid. Before `transaction1` is able to commit its writeSet it needs to evaluate `a1` and `a2`. By doing this the `readTVar` operations are evaluated. Hence the TVar and its version number is added to the readSet and the actual value is return. Since the readSet is already validated at this point it is superfluous.

After `transaction1` finished and released the locks, `transaction2` aquires these locks and validates. The readSet of `transaction2` is also empty, thus the validation succeeds. Then `transaction2` evaluates the `readTVar` operations and commits its writeSet before releasing the locks.

Both transactions run parallel as far as possible and did not roll back. In Chapter 3 are the limitations of this idea presented and the challenges that arise when implementing it. Furthermore are solutions to these challenges introduced.

2.3.2 Unnecessary Recomputations

While the first problem dealed with then question *when* transactions should be rolled back, the second problem investigates the question *how* transactions are rolled back. Take a look at the following example:

```
transaction = do
  a <- readTVar t1
  writeTVar t1 (f a)
  b <- readTVar t2
  writeTVar t2 (g b)
```

This transaction contains two independent statements. The first two lines of the transacton form the first statement. This is independent of the last two lines. Independent means their side effects or results do not influence each other. While the

[6]You could argue that evaluating `readTVar` operation is necessary before validating, but this would not change the validity of the transaction, since the TVars are locked and can not be modified by other transactions at that point.

first line influces the second line, it does not influence the last two lines and vice versa.

If the transaction is now executed it computes its writeSet and readSet first. Then it locks the TVars and validates. The validation fails if either of the TVar has changed after it was read by the transaction. If the validation fails the transcation is rolled back. Which means the readSet and writeSet are discarded, regardless which TVar was the reason for the failed validation.

For example if `t1` was modified after the transaction read it, the transaction is rolled back and both statements are executed again. This includes the read and write of `t2` although the value of `t2` did not change. Hence the exact same code with same inputs and the same (relevant) environment is executed twice. If we are able to invalidate parts of transactions instead of transactions as a whole, we can save time when rolling back a transaction. In the previous example we can use the information that only `t1` changed. If we roll back this transaction, we onky need to execute the first two actions, because the last two actions do not depend on `t1`.

This concludes the overview on the problems of the current STM implementation. We will now turn over to the solution for this problems.

Chapter 3

Concept

I will now explain how I engaged the previously stated problem.

3.1 Unnecessary Rollbacks

Remember the idea given in 2.3.1. I suggested to delay the evaluation of readTVar operations to the commit phase rather than doing them directly in the computation phase. **This part could be the first point of the fix chapter** While the idea would work for this example, the idea would not work for the following example:

```
limitedTransfer src dst am = do
  srcBal <- readTVar src
  if a1 < am
    then return ()
    else do dstBal <- readTVar dst
            writeTVar src (srcBal - am)
            writeTVar dst (dstBal + am)
```

If we use this function the result of `readTVar src` is needed in the computation phase and therefore the evaluation cannot be delayed to the commit phase. The value is needed to decide on the condition of the if expression. To be exact the value is needed to determine the control flow.

This leads to the question whether there is a way to determine if the result of a `readTVar` effects the control flow or not. The current implementation is we are not able to do this. The problem is the bind operator: »= `:: STM a -> (a -> STM b) -> STM b`[1]. This operator allows us to extract the result of an STM action (for example a `readTVar`) from the STM context. This means the STM library loses any possibility to observe this value. Thus the library it not able to decide if the value is used to alter the control flow.

If the library handles a value that is **not** used for branch conditions as if it were used for branch conditions, it may loses performance, but preserves the correct the semantics. If the library on the other hand handles a value that is used for branch conditions as if it were not, the library would not perform unnecessary rollbacks, but may violate the ACID properties. Thus the only way to ensure the correctness of the implementation is to handle all values as control flow critical values.

[1]Remember that the do notation used so far is syntactic sugar for »= and »

Appendix A

Frequently Asked Questions

A.1 How do I change the colors of links?

The color of links can be changed to your liking using:
`\hypersetup{urlcolor=red}`, or
`\hypersetup{citecolor=green}`, or
`\hypersetup{allcolor=blue}`.
If you want to completely hide the links, you can use:
`\hypersetup{allcolors=.}`, or even better:
`\hypersetup{hidelinks}`.
If you want to have obvious links in the PDF but not the printed text, use:
`\hypersetup{colorlinks=false}`.

Bibliography

Dijkstra, Edsger Wybe (1965). “Cooperating Sequential Processes, Technical Report EWD-123”. In:

Gray, Jim and Andreas Reuter (1992). *Transaction processing: concepts and techniques*. Elsevier.

Harris, Tim, Simon Marlow, et al. (2005). “Composable Memory Transactions”. In: *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP ’05. Chicago, IL, USA: ACM, pp. 48–60. ISBN: 1-59593-080-9. DOI: 10.1145/1065944.1065952. URL: http://doi.acm.org/10.1145/1065944.1065952.

Harris, Tim, Mark Plesko, et al. (2006). “Optimizing memory transactions”. In: *ACM SIGPLAN Notices* 41.6, pp. 14–25.

Larus, James R and Ravi Rajwar (2007). *Transactional memory*. Vol. 1. 1. Morgan & Claypool Publishers.