Universität Karlsruhe (TH)

Research University • Founded 1825

System Architecture Group

Department of Computer Science

Software Transactional Memory for the AmbiComp distributed Java Virtual Machine

Clemens Koller

Diploma Thesis

Advisers:	Prof. Dr.-Ing. F. Bellosa
	Dr.rer.nat. T. Fuhrmann

February 14th, 2009

Ich versichere hiermit, die vorgelegte Arbeit in dem gemeldeten Zeitraum ohne fremde Hilfe verfasst und mich keiner anderen als der angegebenen Hilfsmittel und Quellen bedient zu haben.

I hereby declare that this thesis is a work of my own and that only cited sources have been used.

Karlsruhe, den 14. Februar 2009 Clemens Koller

Abstract

The increasing prevalence of multi-core processors and distributed systems makes parallel programming more and more important. But correct synchronization of concurrent processes is difficult because locks, the most common tool to handle concurrency, are flawed and error prone. Software transactional memory promises to revolutionize the development of parallel applications by borrowing the concept of atomic transactions and applying it to code blocks. But until now most research in this area focused on the performance on single systems.

In this thesis, we present a new software transactional memory algorithm that is designed from the ground up with distributed systems in mind, the AmbiComp distributed Java virtual machine in particular. Another novel feature of our proposed solution is that the old values of modified shared objects are not discarded. Instead, these old versions are used to reduce the number of transactions that need to be aborted because of conflicts.

The evaluation of the algorithm with both theoretical—using a script interpreter we especially developed—and practical tests confirm the potential of our approach. Under heavy contention the reduction of aborts is only marginal, but with a high reader to writer ratio more than 50% of the aborts can be avoided as compared to using only the most recent versions.

Contents

Chapter 1

Introduction

The move to multi-core processors and distributed systems in recent years has increased the demand for parallel computing. Only by employing parallelism is it possible to utilize the full power of these systems. But developing parallel programs is more difficult than writing sequential ones because of the need to synchronize the concurrent computations. The most commonly used solution for synchronization are mutual exclusion locks, but they introduce a new class of software bugs, of which race conditions and deadlocks are the most prevalent.

Transactional memory introduces a new way of synchronization which aims to avoid the problems of locks while at the same time simplifying the development of parallel programs. To do this, transactional memory borrows the concept of atomic transaction known from databases and applies it to memory accesses. This moves the burden of correct synchronization from the software developer to the transactional memory system. But so far the research on transactional memory focuses mainly on improving the performance on multi-core and multi-processor systems, distributed systems are often disregarded.

The solution we propose in this thesis is a newly developed software transactional memory algorithm. It was designed from the ground up with distributed systems in mind, especially the AmbiComp distributed virtual java machine (ACVM). The ACVM enables Java applications running on different nodes to transparently share globally accessible objects. The goal of our algorithm is to also transparently synchronize the accesses to those objects by allowing the developer to indicate which code blocks needs to be executed atomically.

Another novel aspect of our approach is that the old value of a shared object is not discarded when it is modified. Instead we keep a number of old versions of each object, which allows us to go back in time when conflicting accesses occur and enables us to reduce the number of unresolvable conflicts that lead to aborts.

In this theses we describe our algorithm and present an implementation of it. We also present a solution for the testing of concurrent code which enables us to execute concurrent code in a specified execution sequence or alternatively in all possible sequences. We then evaluate, using both theoretical and practical tests, whether the usage of old and potentially outdated versions of shared objects can indeed reduce the number of aborts while still providing a correct result.

1.1 Structure of the Thesis

The thesis is structured as following:

- *Chapter 2* provides background information about the problems caused by lock-based synchronization and how transactional memory aims to solve them. We also give an overview of the history of transactional memory and discuss the latest developments in this area.
- *Chapter 3* describes the algorithm we use for our approach to software transactional memory.
- *Chapter 4* covers the design and details of our implementation.
- *Chapter 5* provides information about *stmtest*, an application used to test our solution in a reproducible manner. We also discuss the results obtained by this theoretical evaluation.
- *Chapter 6* presents the practical results we obtained by making our solution available to Java.
- *Chapter 7* is a summary of the thesis and contains directions for further work.

Chapter 2

Background and Related Work

2.1 Background

In recent years there has been a fundamental change in the computer industry. While Moore's Law still holds true and the number of transistors that can fit in the same space is doubled roughly every 18 months, it is no longer possible to increase their clock speeds without overheating or consuming too much power. Instead, many manufacturers are now focusing on multicore architectures to increase performance: multiple processors—the state-of-the-art are up to eight of those so called cores—are combined into a single processor and communicate using shared hardware caches. To take advantage of this architecture it is necessary to exploit parallelism. A single task needs to be divided into several smaller ones which are then solved concurrently and in the end their results are combined to solve the bigger task.

Another change has been the move away from single and big computer systems to ubiquitous computing. We are more and more surrounded by small computers embedded in objects of our daily lives: mobile phones are becoming increasingly versatile and powerful, modern cars and houses are filled with a plethora of computing systems aimed at increasing the security and making the life of their users easier. Their abilities to support and entertain the user are limited though because most of those small systems work on their own. Interconnecting those devices is the obvious next step to increase their productivity and effectiveness by forming a sensor actuator network.

2.1.1 AmbiComp

To program and use these distributed systems a distributed execution environment is needed. The AmbiComp project [1] aims at providing such an environment by using a distributed Java virtual machine, the so called AmbiComp Virtual Machine (ACVM).

Unlike other distributed virtual machines the ACVM does not target the field of cluster computing where heavy computing power is the main focus. Instead it is meant to run on small embedded devices with 8-bit micro-controllers, equipped with a network interface and possibly sensor systems. It focuses on allowing easy and rapid development of software for novel applications of such distributed sensor networks.

To achieve this the ACVM creates the illusion of a *single system image* by using a global shared heap and by allowing threads to migrate between nodes. This allows the programmer to develop software for the sensor network in the same ways as for a single multiprocessor system. The distribution of the shared code and the exchange of data is automatically managed by the interconnected ACVM instances. Apart from an additional transcoding step of the Java bytecode the distribution is thus completely transparent to the programmer.

The core component for the automated distribution are *Global Accessible Objects* [2]. Each of these GAOs represent a block of memory which can be a Java object or array, a code block or an execution context. The addresses of the GAOs are globally unique and independent of the GAO's actual location in the distributed system. Local objects are automatically upgraded to GAOs by the ACVM when references are passed to other instances, allowing other nodes to a access the data they represent. References to static objects are obtained from an oracle using a distributed hash table.

When accessing a GAO its address is automatically resolved using *scalable source routing* SSR [3], an ad-hoc routing protocol. It arranges all nodes into a flat virtual ring topology uncoupled from the actual physical network. Due to this GAOs can be accessed without a change of address or a need to use proxies even when they are moved to another node.

2.1.2 Synchronization/Concurrent Programming

Both scenarios mentioned above—the multicore architectures and the distributed systems—make developing applications harder than it used to be on single processor systems because of the need to synchronize the access to shared data. The most widely used fundamental primitive for this still are mutual exclusive locks. This popularity stems from their seemingly easy programming model and the availability of efficient and scalable implementations. But for systems containing more than just a handful of locks those advantages rapidly fade away and a number of serious problems arises.

To guarantee the correctness of an application the programmer has to ensure that the necessary locks are acquired on each access to a shared object. Especially on larger projects where the locking strategy might change over time can this only be achieved by very careful programming and comprehensive documentation of the needed locks. Failing to do so can cause data loss and lead to *race conditions*, making the application only work if there happens to be no concurrent access because of the execution scheduling. Tracking down bugs like this takes a considerable amount of time and skill.

Another difficulty when using locks is to balance the granularity of the locks against the time spent to acquire and release the locks. A very coarse-grained approach with only one global lock obviously minimizes the locking overhead but will severely limit parallelism. Fine-grained locking solves this problem at the cost of more overhead and a more difficult implementation because of the increased number of locks involved.

When dealing with a bigger number of locks there is also the problem of avoiding *deadlocks* which can occur when threads attempt to acquire the same locks in a different order. When many locks must be acquired, and particularly if the set of locks is not known in advance, the needed deadlock avoidance can be awkward. Locking can also lead to *priority inversion* when a thread with a higher priority is forced to wait for a low-priority thread that is holding exclusive access to a resource it needs. On systems with strict priority scheduling where only the thread with the highest priority is allowed to run this can lead to deadlocks too.

Furthermore, great care has to be taken to ensure that all acquired locks are released again when they are no longer needed. Otherwise the execution will also come to a halt.

To avoid those mistakes, this favors the usage of very simple locking strategies to reduce the number of locks that need to be taken care of. These strategies often are overly pessimistic and serialize the execution of code that is actually conflict free. Thus the parallelism of an application is artificially limited by the complexity caused by locks.

Efficient and scalable implementations also are mainly available on single computer systems where the hardware often provides special atomic instructions (such as test-and-set or compare-and-swap) to support them. On distributed systems the implementation of locks is not as simple, especially in a decentralized system with no central lock manager.

The problems of locks can be summarized as follows:

- Software using many locks is hard to develop and maintain.
- Bugs such as race conditions, deadlocks and priority inversion are hard to track down and solve.
- Their advantages only partly apply to locks in distributed systems.

2.1.3 Transactional Memory

One approach to solve the drawbacks of conventional locking is Transactional Memory, which can be implemented purely in hardware or software, or as a hybrid system. In this thesis we will focus on *Software Transactional Memory (STM)*, an increasingly popular solution to synchronization problems which has been the focus of researchers for several years.

Instead of putting most of the burden of correct synchronization on the programmer by just offering a primitive for mutual exclusion like locking does, STM introduces the concept of *transactions*. A transaction starts when entering an

```
atomic                          atomic
{                               {
  x++;                            if (x != y)
  y++;                              while (true) {}
}                               }
```

Transaction A Transaction B

Figure 2.1: Example of a transaction that might never complete if eager versioning is used.

atomic block and ends when leaving it. The STM runtime ensures that shared data is accessed in a safe manner by guaranteeing that all code inside a transaction either fully executes or not at all. Furthermore, the intermediate states of the data accessed during the transaction is not visible to concurrently running transactions of other threads. In database systems, where transactions are a common way to solve conflicts of concurrent operations, those features are referred to as *atomicity* and *isolation*, respectively. If the atomic block is only isolated from all other atomic blocks the STM provides *weak atomicity* whereas *strong atomicity* guarantees than an atomic block is isolated from all other operations.

Transactions are executed speculatively and all changes to objects due to write operations during a transaction are only tentative. If no conflicts are detected the transaction *commits* and the results of its write operations become permanent. Otherwise the transaction has to be *aborted* and is re-executed from the beginning until it succeeds. When aborted a *rollback* has to be performed and all changes by the transaction have to be undone. To achieve this the STM runtime can use *eager versioning* and perform all changes in-place, logging them to a journal and undoing them during an abort. Using *lazy versioning* all changes are made on copies of the actual objects and discarded during abortion. The implication of this is that isolation can only be guaranteed if no irreversible operations (such as I/O) are allowed inside transactions. This limitation can be overcome by queuing up all those operations in a buffer and performing them outside of the transaction after it has committed. If the language supports it the type system can also be used to prevent the usage of disallowed operations [4].

While eager versioning can provide better performance, especially when the number of rollbacks is low, it weakens the isolation of the transactions during execution. Even though isolation will not be violated—the transaction will be aborted if conflicts are detected when committing—this still can cause problems. Figure 2.1 shows a contrived example: Initially x and y are equal. If Transaction A now eagerly increments x and Transaction B executes before y is also incremented it enters an endless loop.

The optimistic execution of transactions results in an increased concurrency: it is no longer necessary for a thread to wait for objects that are locked by other threads. And different threads can concurrently modify disjoint parts of the same

```
boolean MoveToFront(int x)
{
  atomic
  {
    Node prev = head;
    Node current = head.next;
    while (current)
    {
      if (current.value == x)
      {
        prev.next = current.next;
        current.next = head.next;
        head.next = current;
        return true;
      }
      prev = current;
      current = current.next;
    }
    return false;
  }
}
```

Listing 2.1: Example of how to use atomic blocks to modify a linked list.

data structure that normally would be protected by a single lock. In most practical applications the number of conflicts—and thus the number of re-executed transactions—is relatively small compared to the number of successful commits and thus STM should be able to achieve a significant increase of performance over lock-based synchronization. The overhead caused by logging all write operations and checking for conflicts can cause an STM synchronization strategy to perform worse than an optimized fine-grained locking scheme though, but it still maintains the advantage of being much easier to use.

Listing 2.1 illustrates how easy synchronization can be when using STM. The code shown is used to check whether a certain value is in a linked list, and if it is found it gets moved to the head of the list. Trying to make this function thread-safe using locks would either remove the parallelism (by using a single lock) or make it overly complicated and error-prone by using fine-grained read and write locks on each node. With STM all that is needed is the addition of the *atomic* keyword to indicate that the function needs to be executed as a transaction and the runtime ensures correctness.

Besides all those advantages, recent research [5] illustrates that transactional memory systems might not be ready yet for practical usage. While STM solves many of the problems created by locking, it does introduce a variety of new issues:

- The abortion and re-execution of transactions after a rollback causes a non-determinism that complicates the debugging of applications. This mitigates the increased productivity when working with STM.

- The handling of exceptions thrown while executing within a transaction and the propagation of consistent exception information is difficult. Imagine that an out-of-memory exception is raised within one transaction because another transaction has already claimed all available memory. By the time the exception is handled the second transaction might have been rolled back and the reason for the exception is no longer visible. A re-execution of the first transaction could actually be successful now.
- The interaction with legacy code that is not prepared to be used within a transaction is difficult. If it includes locks or other synchronization primitives it can even be impossible to be used in conjunction with STM. Making the locking visible outside of the transactions would violate isolation, but otherwise correctness can no longer be guaranteed.
- Depending on the implementation, STM can cause a big amount of overhead severely affecting performance and making STM unusable. Even the best state-of-the-art STM systems can be over 100 times slower than sequential execution [5]. Approaches to reduce the overhead include using annotations to mark private data within transactions, but this also dilutes the ease of use of STM.

2.2 Related Work

This section provides an overview of already existing STM implementations and shows the progress in researching transactional memory. The selection of STM systems is by no means complete, instead it focuses on implementations providing novel ideas or approaches to solve some of the aforementioned problems.

2.2.1 First STM Systems

While all prior work on transactional memory was built around hardware support Shavit and Touitou [6] introduced the concept of software transactional memory. Their implementation only allows *static transactions*, i.e. transactions that only access a predetermined set of M locations.

Each transaction contains two vectors: old[i] contains the value of location i on the first access to that location, new[i] is the value to be written to the respective location. The data structure containing the M locations also includes a reference to each transaction that is currently accessing one of the locations. A transaction performing a commit gradually tries to acquire ownership of all required locations using a sequence of load-link and store-conditional operations. It then confirms whether all read locations still contain the value that was read, updates all locations with the new value and releases ownership again. If any of these steps fail, the transaction releases all the ownerships it has acquired and helps the transaction that caused it to fail by trying to commit that transaction's new values. As such this mainly is an implementation of a non-blocking M-word compare-and-swap operation used together with optimistic execution.

Evaluation shows that even this simple approach can outperform other means of synchronization, especially on systems with 32 and more processors.

2.2.2 Language Support for Transactions

To overcome the limitations of static transactions Fraser and Harris [7] suggest a new approach for the Java programming language that allows arbitrary access to objects within transactions. This is achieved by not storing the meta-data used to perform the STM operations with their respective objects but instead in a separate data structure. An ownership function is used to map each memory location to one or more ownership records in that data structure. This allows transactions to work with any kind of object and not just a predetermined set of objects, thus greatly improving the usability.

Since this frees the programmer from having to discriminate between objects managed by STM and objects that are not, they propose to add an *atomic* keyword—similar to *synchronized*—to Java. This allows all information and implementation details of the STM system to be hidden, moving the responsibility of correctly using the STM operations from the programmer to the compiler and the Java virtual machine. Together with the ability to enter the atomic block only once a condition is true, this basically is an implementation of Hoare's *conditional critical regions* [8].

This approach greatly simplifies the problem of synchronization because the programmer only has to indicate *which* group of operations he wants to perform in isolation of each other, but not *how* this isolation should be guaranteed.

Harris et al. [4] take this approach one step forward by adding transactions to Concurrent Haskell. They argue that a purely functional language provides the perfect setting for STM because the type system explicitly separates computations which may have side-effects from effect-free ones. Taking advantage of Haskell's monadic type system, it is easy to ensure that the code within transactions cannot perform any irrevocable input/output effects. Another property of functional languages is that most computation takes place in the purely functional world and explicit reads from and writes to mutable objects are relatively rare. And since only these explicit operations need to be covered by STM system, the performance overhead is negligible. Despite being a powerful and interesting approach this is of little practical value though as neither the monadic type system nor the functional execution map naturally to the much more popular imperative languages like Java.

2.2.3 Lock-Based Software Transactional Memory

Instead of trying to make transactions non-blocking to increase concurrency, STM can also be implemented using locks. The idea behind this is to use STM as a way to automatically achieve fine-grained locking without the risk of deadlocks. This approach promises to improve performance significantly by using much simpler algorithms and being much more cache friendly.

An implementation of an *encounter time* lock acquisition STM algorithm is given by Ennals [9]. It uses *revocable two phase locking* for writes: Whenever a transaction tries to perform a write operation to an object it first has to acquire an exclusive lock on that object. This lock is held until the transaction either commits or is aborted. If the lock cannot be acquired because another transaction is already holding it the transaction either waits for the lock to be released (and aborts if a timeout is reached) or aborts the other transaction. Deadlocks cannot occur in this algorithm because all the locks are revocable and thus the hold and wait condition is not fulfilled.

For reads an *optimistic concurrency control* is used: For each read object its version number is also recorded. When committing, those version numbers are compared to the current versions, and if any newer versions exist the transaction is aborted. This avoids the cache contention that would occur if multiple readers were synchronizing their access using read locks.

Alternatively, Dice et al. [10] show that STM can also be implemented using *commit time* locking and a global version clock. When a transaction starts, it records the current value of the version clock as its read time. On each access to an object, both reads and writes, its version is compared to the read time, and if it is greater the transaction is aborted. During commit a lock for each written object is acquired and the version numbers of all read and written objects are re-checked. Finally the global clock is incremented and all written objects are updated with their new value and the new time stamp.

Evaluation shows that these lock-based STM algorithms are significantly faster than the earlier non-blocking approaches and consistently are more than twice as fast.

2.2.4 Distributed Software Transactional Memory

Most of the aforementioned algorithms rely on blocking instructions or centralized components such as version counters or real-time clocks, which makes them unsuitable for distributed systems.

Manassiev et al. [11] suggest an STM like approach called Distributed Multiversioning (DMV). In their system each node maintains its own replica of the shared memory and during execution all operations are only performed on this local copy. During commit a shared memory consistency protocol is used to distribute the changes to the other nodes. To avoid conflicts a unique system wide token needs to be acquired before a commit is performed.

Another approach for a distributed STM system called Distributed Dynamic Software Transactional Memory System (DDSTM) [12] is provided by Kotselidis et al. They combine the non-distributed DSTM2 [13] transaction engine with a remote communication framework based on Java RMI. When a transaction attempts to commit it first tries to resolve local conflicts using the DSTM2 algorithm, then validates itself against all distributed transactions on the cluster and finally commits. The distributed validation is performed by acquiring a global serialization number from a master node and then sending a set containing its accessed objects to all other nodes. The validation with the lowest number then gets committed while all the other transactions are aborted. The new values of the committed transaction are distributed by a master node which forces all other nodes to update their values too and to abort all running transactions that depend on an overwritten object.

Chapter 3

Distributed STM

Building on the AmbiComp platform (cf. sec. 2.1.1) we present a novel approach that applies the idea of software transactional memory to a fully decentralized distributed setting. Our *Distributed STM* algorithm does not require any centralized components and can—in an advanced version—even entirely avoid locking.

3.1 Overview

In the spirit of AmbiComp to make the development of distributed applications as easy as possible our STM algorithm will not be accessible by the user directly. Instead it will only be necessary to mark blocks as atomic and the Java virtual machine will ensure that all accesses to shared objects are done transactional, aborting and re-executing the block automatically in case conflicts are detected. Listing 3.1 shows where the STM algorithm hooks into the virtual machine:

- At the start of an atomic block *tstart* is called to initialize the transaction.
- When reading a shared object *tread* is called and tries to find a non-conflicting version of the object.
- On a write to a shared object *twrite* gets called and the changes are performed on a local copy.
- At the end of the atomic block the transaction attempts to commit by calling *tend*.

```
atomic                // -> Calls tstart
{
  int a = A.intField; // -> Calls tread
  A.intField = 17;    // -> Calls twrite
}                     // -> Calls tend
```

Listing 3.1: Transactional access to the shared object A.

By making the STM algorithm transparent to the programmer it becomes much easier to write correctly synchronized applications. It also enables us to change the details of the underlying implementation without any modifications to the programs utilizing the atomic keyword.

3.2 Data Structures

Distributed STM is an object-based STM algorithm. To ensure correctness it relies heavily on versioning of the shared data and on passing messages between the concurrent users.

3.2.1 Global Accessible Objects (GAOs)

The shared data managed by Distributed STM are AmbiComp's *Global Accessible Objects* (GAOs). Each GAO has a globally unique GAO identifier (GID) which can be resolved by the underlying routing system to send messages to the node managing the GAO. A GAO is a distributed data structure and consists of two parts

- A version history list. This is a linked list of the current and one or more previous versions of the GAO.
- A pending version tree. It contains new versions of the GAO that have been created by transactions but which are not yet committed.

These different versions can be distributed across the nodes and are referenced by their unique GAO version identifier (GVID). Through the predecessor links in the version history list we also induce an ordering relation of the different versions of a certain GAO. The first entry of the version history represents the *head version*, i.e. the most recently committed version of the GAO. Each version contained in the version history list or the pending version tree consists of:

- The GVID of the version itself. This is mainly for convenience as a version can only be accessed if you already know its GVID.
- The stored data, i.e. the data the GAO actually represents. This can for example be the member fields of a Java class.
- A reference to the version's predecessor, i.e. to the version of the GAO which was committed directly before this one.
- The *write set* containing the GVIDs of all GAOs that were written by the transaction that created this version.
- The *read set* which contains the GVIDs of all GAOs the transaction creating this version read before committing.

- A reference counter used to determine by how many transactions this version currently is used.

It is important to notice that the versions are—apart from the reference counter—immutable objects once they are added to a GAO and thus can easily be replicated, e.g. for caching purposes. Only the reference counter needs to be updated consistently. This can be done easily because they are only needed for the pruning of no longer needed versions (cf. 3.5). Thus any delayed updates on the reference counters do not interfere with the STM mechanism at all and can at worst delay the release of memory.

3.2.2 Transactions

To manage all the data of an atomic block that currently is executing a *transaction* object is used. Since atomic blocks are executed entirely on one node these objects can be local and do not need to be shared. A transaction contains three entries which all are updated during execution:

- A container with copies of all objects modified in the atomic block so far. All write operations are performed on these copies.
- The write set containing new GAO versions for all GAOs which have been written to in the atomic block so far. While the block is executing these versions are not yet part of their respective GAO.
- A read set containing the GVIDs of all GAO versions that have been read by the transaction.

3.3 Operations

For the four main operations of Distributed STM—tstart, tread, twrite and tend—pseudo-code is given in Listing 3.2 as an overview. In the following we will illustrate each operation in detail to show how correctness is guaranteed by ensuring isolation and atomicity.

3.3.1 Starting a transaction (tstart)

The operation tstart is called at the beginning of an atomic block to start a new transaction. A new transaction record is created with an empty read and write set and no local copies. Optionally the execution context of the current thread can be cloned so that in the case of a rollback all changes to objects on the stack can be undone. Alternatively, all changes to objects on the stack can be recorded in the local copies container and written back to the stack after a successful commit.

```
tstart:
  Create a new transaction record
  Clone the current execution context

tread(GAO):
  If read_set.Contains(GAO)
    return local_copies[GAO]

  GVID version = GetHeadVersion(GAO);
  while (version)
    if (Check(version))
      local_copies[GAO] = GetGao(version)
      read_set[GAO] = version
      return local_copies[GAO]
    else
      version = version.predecessor
  Abort()

Check(version):
  check_set = transitive closure of
                version's read and write sets
  foreach (GAO g, GVID v) in read_set
    if check_set[g] is newer than v
      return false

  return true

twrite(GAO, value):
  If not read_set.Contains(GAO)
    tread(GAO)
  If not write_set.Contains(GAO)
    write_set[GAO] = CreateNewVersion(GAO)

  local_copies[GAO] = Value

tend:
  acquire(global_lock)
  foreach (GAO g, GVID v) in the write_set
    if not v.predecessor == GetHeadVersion(g)
      Abort()
      release(global_lock)
      return

  foreach (GAO g, GVID v) in the write_set
    v.read_set = read_set
    v.write_set = write_set
    v.value = local_copies[g]
    add v as new head version to the version list of g

  release(global_lock)
```

Listing 3.2: Pseudocode of the locking STM Algorithm

```
atomic  // Transaction 1        atomic  // Transaction 3
{                               {
  y = A;                          B = A + 1;
  x = C;                        }
  D = x - y;
}

atomic  // Transaction 2        atomic  // Transaction 4
{                               {
  A = 1;                          C = B + 1;
}                               }
```

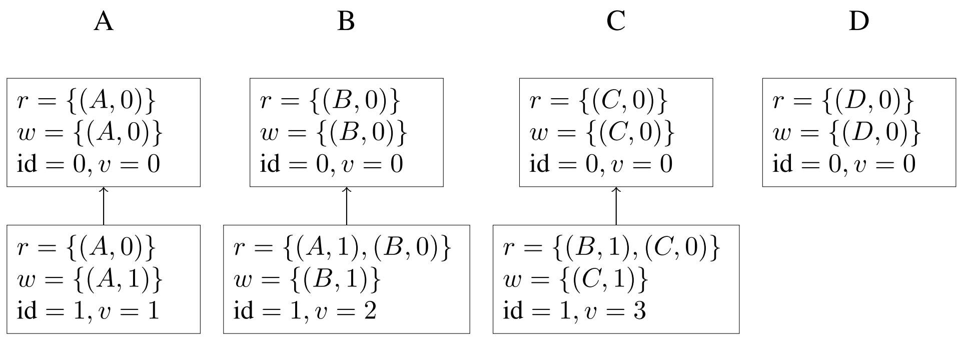

Figure 3.1: Transactions and GAOs used to illustrate the creation of the check set. Capital letters represent GAOs, x and y are local variables.

3.3.2 Reading a shared object (tread)

When a GAO is read the tread operation is used to find a suitable, that is non-conflicting, version of the GAO. If the GAO is already in the transaction's read set then it has been read before by this transaction. In this case the already existing local copy which was created by the previous read can be used. Otherwise a version V from the GAO's version history list has to be fetched. In general it should pick the head version because it represents the most up-to-date data of the GAO, but as we'll see this is neither necessary nor always possible.

Once a version has been fetched the transaction needs to check if the version is suitable or if it is conflicting with one of the versions of the other GAOs that were read before. To do this the *check set* of V is constructed. It is the transitive closure of read and write set of V and is created by recursively unifying the read and write sets of V and the versions in those sets. If there exist dependencies to different versions in the check set, only the most recent version is used. This check set represents all read and write dependencies of V.

For an example see figure 3.1. Lets assume that all four GAOs have an initial version with id $= 0$, the value $v = 0$ and their read set r and write set w only contains themselves. If transaction 1 now reads A, then gets interrupted and the other three transactions completely execute and commit we end up with the version history lists as seen in the figure. When transaction 1 resumes execution it fetches the head version of C—which is version 1 in this case—and constructs the check set c_1 by unifying the check sets of all GAOs it has in its read or write set. So it is $c_1 = c_0 \cup c_1 \cup b_1 = \{(B, 1), (C, 1)\} \cup a_1 = \{(A, 1), (B, 1), (C, 1)\}$.

After the check set of the version to be read is created we can use it to check for conflicts. If the transaction's read set contains a version that is older than a version in the check set, it can not use this version or it would violate atomicity. Coming back to our example, this means that transaction 1 must not read version 1 of C (called $(C, 1)$) because it has already read $(A, 0)$ but $(C, 1)$ depends on $(A, 1)$. Instead of aborting the transaction now and potentially discarding all calculations the transaction has performed so far, we can try to find another suitable version. This is done by following the predecessor links in the version history list to find a non-conflicting version. If no such version can be found, or a limit l for the allowed number of backstepping steps is reached, the transaction is forced to abort and re-execute. While using these older versions reduces the number of conflicts caused by read operations, it can increase the number of commit conflicts, so the best value for l depends on the characteristics of the transactions. In our example $(C, 0)$ would be suitable and transaction 1 could commit even though it has read old and not up-to-date data. This does not cause a conflict though because we only guarantee that atomic blocks do not influence each other, we do not impose a sequential order of execution.

A special case can occur when no suitable version can be found because this is the very first read performed on the GAO and no version at all exists so far. In this situation, a new version initialized with the default value of its data is created and added to both the transaction's read and write set.

Please also note that the version IDs in our example are only for illustration and in practice are not strictly increasing integers. Instead, we use the partial order of the versions given by their relative position in the GAO's version history list to compare versions. The older version can be reached by following the predecessor links of the newer version whereas the newer version cannot. If both versions can't be reached from the other one then they can not belong to the same GAO.

After a suitable version has been found, it is added to the transaction's read set and a local copy of it is created. The data of the local copy is then returned to the caller of tread so execution can continue.

3.3.3 Writing a shared object (twrite)

With most of the work of avoiding conflicts done when reading GAOs, writing them becomes comparatively easy. If the GAO is in the transaction's write set, it has been written to before within this transaction and we only need to update its local copy. Otherwise, if the GAO to be written is not in the read set of the transaction yet, a tread is performed on it to check if a suitable version of it exists. After checking if the GAO is still not in the write set (this can happen if the tread created a new version), a new version of the GAO is created and added to the write set. Then the value of the local copy is updated and execution of the transaction resumes.

When creating a new version, we can take advantage of the fact that the GVIDs do not need to strictly increase but instead can be any unique number. This removes the need to coordinate the creation of a new GVID with other nodes to ensure correct ordering. Because of this, the creation of a new version can be made a local operation and the uniqueness can be ensured by using a reasonable big random number.

3.3.4 Committing a transaction (tend)

At the end of an atomic block when a transaction attempts to commit its changes the tend operation is called. If the write set of the transaction is empty then it can commit immediately as it does not modify the global state and any conflicts caused by reading GAOs would have been noticed during tread. Otherwise, each new GAO version in the transaction's write set is assigned its new value from its local copy, and it's read and write set are set to the transaction's sets.

The locking version of Distributed STM then acquires a global lock to perform to following operations atomically. At first we need to check if the direct predecessor of every new version in the write set is still the head version of the respective GAO. If this is not true for any version then the transaction needs to be aborted to avoid a write conflict, which would overwrite the results of another transaction. If all versions pass the check, they get added as the new head versions of their respective GAO and the global lock can be released.

Performing this phase atomically is necessary to deal with concurrent commits of transactions. Due to the need of acquiring a global lock, the commits thus get serialized and are processed in a first in, first out order.

3.4 Distributed Consensus Protocol

To avoid the locking and its drawbacks during commits a *Distributed Consensus Protocol* can be used instead to handle coincidental commits. The protocol we suggest here is a randomized consensus protocol because this enables us to arbitrarily prefer some transactions over others. For example, long transactions with many complex and resource intensive operations could be given a higher weight than short transactions which can be re-executed much more easily. In the following explanation we assume equal weights for all transactions for simplicity reasons.

When the consensus protocol is used a committing transaction adds each of its new versions in the write set to the pending version tree of the respective GAO. While doing this the transaction checks if the new version's predecessor is still the head version. If this is not the case the transaction needs to be aborted.

When such a new pending version is added to a GAO that already has $n - 1$

other pending versions, it gets assigned the probability n^{-1}. The probabilities of the existing pending versions are adjusted accordingly by multiplying them with $\frac{n-1}{n}$. Using these probabilities the GAO then selects one pending version. This positive vote, as well as the $n-1$ negative votes for the pending versions that were not chosen, are shared with the other GAOs of their respective write sets.

Therefore a GAO receives m_i votes for each of its pending versions i when its respective write set consists of m_i GAOs, $m_i - 1$ from the other GAOs and its own vote. Using these votes, the new probabilities p_i for each pending version are calculated. Let m_i^+ be the number of positive votes for the pending version i. Then it is $p_i = \frac{m_i^+}{m_i} / \sum_{0 \le j \le n} \frac{m_j^+}{m_j}$. Using these new probabilities, the next round of the protocol begins and a new pending version is chosen again.

These iterations continue until one pending version has either received m_i positive or negative votes. If it is $m_i^+ = m_i$ then the winning pending version i is moved to the version history list as the new head version and all other pending versions are discarded. Otherwise the losing version is discarded and removed from the pending version tree, and the probabilities of the remaining pending versions are scaled accordingly. The voting then continues until no more pending versions are left. Progress of the protocol is guaranteed because p_i can never be zero. This means that eventually a transaction will successfully commit and no livelock occurs.

Another advantage of the consensus protocol compared to the global lock approach is that it allows transactions with disjoint write sets to commit concurrently without interfering with each other. This possible increase in parallelism comes at the price of the overhead of the protocol though.

3.5 Garbage Collection

With the algorithm explained so far, there is no limit to the number of entries in the version history list of a GAO. It contains all the versions of the GAO since it was created. While this means that transactions can avoid read conflicts by just going back in the history far enough, it poses a number of problems.

Foremost, reading an old version makes the transaction depend on that version. And the older the version is, the higher the chance that this will lead to a conflict at commit time. Greatly increasing the chance of commit time conflicts to remove the chance of read conflicts is not a reasonable thing to do though, because if a transaction has to be aborted it should do so as early as possible. This way the least number of superfluous operations are performed before the re-execution of the transaction.

Secondly, a long version history list increases the effort needed to create the transitive closures to check for conflicts during a read. This is because each version also depends on its predecessor and thus the dependencies of the predecessor also have to be included into the closure.

Storing all versions since the creation of a GAO apparently also is a waste of resources as these versions and the value they represent needs to be stored somewhere, even if no transaction can ever commit after reading one of those versions.

Because of these reasons the version history list should be kept short. To do this without removing versions that are still used by an active transaction or that might be needed in the future to avoid a read conflict, a reference counter is used. Each time a transaction adds a GAO version to its read set it increments that version's reference counter. After a commit or when a transaction is aborted, the counters in all read versions are decremented.

These reference counters are then used to determine when a version will no longer be needed because no transaction will ever go back far enough in the version history list to read it. When all the predecessors of the versions in the check set of any GAO version v have a reference count of zero then these predecessors can be deleted. This is because v will never cause a read conflict under these conditions, so the older versions are no longer needed.

Chapter 4

Implementation

For an initial evaluation of our STM algorithm we decided to not directly integrate it into the AmbiComp Java virtual machine. Instead, we developed it as a C/C++ library which can be included into any kind of application, including the ACVM. To keep the implementation lightweight and versatile, it does not contain any code dealing with the distribution of objects. The underlying application is expected to provide the necessary means to transparently read from and write to distributed objects. This greatly simplifies the implementation since this allows us to focus on the algorithm without having to deal with communication protocols. For our purposes all data structures reside in the same address space and can be accessed directly.

4.1 Design

Our implementations consists of four main classes: *cStmManager* and *cTransaction* provide an interface for the user whereas *cGaoManager* and *cGaoVersion* are managing the GAOs and their different versions. Figure 4.1 shows the interaction between these classes using a very simple transaction that only reads and writes one GAO and then has to be aborted during commit.

At the beginning of an atomic block the application calls the cStmManager to create a new cTransaction object. This transaction is then used to access GAOs. On a read the transaction performs the necessary steps as explained in Section 3.3.2. It uses the cGaoManager to get a version of the GAO and checks if there are any conflicts with other GAOs that already have been read before. If conflicts are found, the next older version in the version history is used until there either are no conflicts and the GAO can be returned, or there are no versions left and the transaction has to be aborted. When a GAO is written to the transaction uses the cGaoManager to create a new cGaoVersion for it. At the end of the atomic block the cStmManager is used to finish the transaction by trying to perform a commit. In our example we assume that the commit fails because of another concurrent commit and the application is notified about this by the return value.

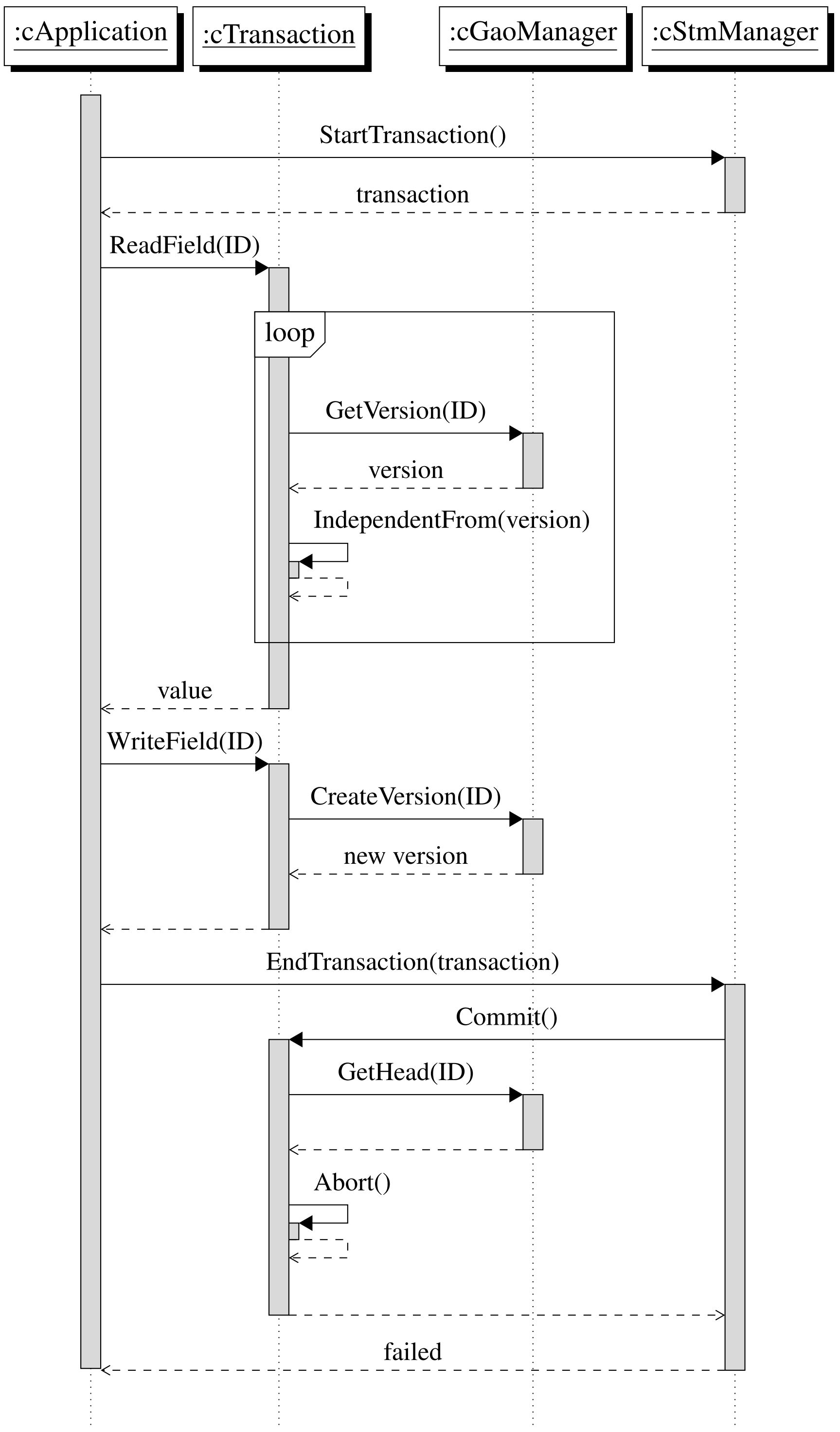

Figure 4.1: Interaction between objects during a simple transaction.

In the following each of the classes will be explained in more detail with a focus on its main features and characteristics.

4.1.1 cGaoVersion

The cGaoVersion class does not contain any logic and is only used as data storage. Each instance of it represents one version of a GAO as described int Section 3.2.1.

```
struct cGaoVersion
{
  cMap<GaoId, GaoVersionId> mReadSet;
  cMap<GaoId, GaoVersionId> mWriteSet;
  cLocalVariable            mValue;
  GaoVersionId              mOlderVersion;
  GaoVersionId              mThisVersion;
  unsigned int              mRefCounter;
};
```

Listing 4.1: Fields of cGaoVersion.

The read and write sets contain the respective dependencies of a version. They are implemented as associative containers which use the identifier of the GAO (GID) as key and the identifier of its version as value. Since the ACVM is running on embedded hardware where the C++ Standard Template Library (STL) is not available or causes to much overhead, we created the class *cMap* which provides a map similar to the one found in the STL. It is implemented using a red-black tree to provide fast $O(log(n))$ look up because this is the most used operation. The slower insert operation compared to a primitive implementation based on an unsorted linked list is insignificant since the contents of the read and write set do not change over time.

The value of a GAO version is of type cLocalVariable which is a typedef that allows our implementation to work with any kind of data. The only requirement is that cLocalVariable must provide a copy constructor which performs a deep copy of all references that do not point to objects managed by STM. This is necessary since our STM algorithm needs to make copies of the objects it manages in several situations. These copies need to be independent, i.e. a change to a copy may not modify any other copy of the same object or the original object, to guarantee the isolation between different transactions.

The link to the version's predecessor which is used to form the version history list is stored as the GVID of the next older version. For convenience reasons the version also contains its own GVID, but it does not include the GID of the GAO it belongs to. This is not needed because a version is characterized to belong to a certain GAO by being part of its version history list.

4.1.2 cStmManager

The class cStmManager is provided as an interface to the application for creating and finishing transactions. It is implemented as a singleton and coordinates the allocation and creation of cTransaction objects.

```
class cGaoVersion
{
public:
  cTransaction* StartTransaction();
  bool EndTransaction(cTransaction** rTransaction);
};
```

Listing 4.2: Interface of cStmVersion.

The `StartTransaction()` method returns a pointer to a newly created cTransaction instance which subsequently can be used to read and write GAOs. The returned object is valid until any of its operations fail and cause the transaction to abort—in which case the transaction frees itself—or until `EndTransaction()` is called. When `EndTransaction()` is called by the application the cStmManager calls the `Commit()` method of the transaction to attempt a commit. If the commit fails because of a conflict the transaction frees itself, otherwise the cStmManager deallocates the transaction. In either case the pointer to the transaction object is set to `NULL` to ensure that the application does not attempt to work with a transaction that is no longer valid.

4.1.3 cTransaction

As mentioned above, the class cTransaction manages the read and write accesses to GAOs. To ensure that all transactions are created through the cStmManager singleton the class has a private constructor to prevent the user from allocating new transactions on his own. The `ReadField()` and `WriteField()` methods are used to read and write GAOs identified by their GID by implementing the tread and twrite operations, respectively. If these methods fail because of an unresolvable conflict, they call `Abort()` to free all objects allocated by the transaction so far and indicate the failure to the application with their return value.

`ReadField()` differs from tread though when it comes to checking for conflicts using the transitive closure. Whereas tread first constructs the complete transitive closure and then checks it for conflicts, `ReadField()` already detects conflicts during the recursive creation of the closure by calling `IndependentFrom()` for each version added to it. This allows us to find potential conflicts early on and prevents us from wasting time and resources on creating a possibly huge transitive closure that already causes a conflict with its first versions. To further speed up this operation the results of `IndependentFrom()` are cached in a map so each version has only to be checked once during a `ReadField()` call. The cache also allows us to avoid infinite recursion caused by circular dependencies by setting the cached entry of the version to be checked to independent. This causes the recursion to end as soon as a mutual dependency of two versions is detected.

```
class cTransaction
{
public:
  bool ReadField(const GaoId& rGaoId,
                 cLocalVariable& rValue);
  bool WriteField(const GaoId& rGaoId,
                  cLocalVariable& rValue);
private:
  void Abort();
  bool Commit();
  bool IndependentFrom(const GaoVersionId& rNewVersion);

  cMap<GaoId, cGaoVersion*>  mReadSet;
  cMap<GaoId, cGaoVersion*>  mWriteSet;
  cMap<GaoVersionId, bool>   mCheckedVersions;
};
```

Listing 4.3: Interface and fields of cTransaction.

As mentioned in Section 3.3.2 it can happen that `ReadField()` is used to access a GAO which does not exist yet, for example the very first access to a static field of a class. In those cases the value passed to `ReadField()` is used as the initial value of the GAO.

Contrary to the read and write sets in a cGaoVersion the sets here do not map a GID to its respective GVID but instead directly point to the cGaoVersion object. For the read set this is done for performance reasons to avoid the additional overhead caused by resolving the indirection. When the GAOs are actually distributed, this overhead can be significant as it could involve network communication on each read to a GAO that is already in the read set. In those cases the read set actually functions as a GAO cache. While these performance reasons apply to the entries of the write set too there is another reason the GVID is not used in them: The versions in the write set contain newly created versions by the transaction which are not yet made visible to other transactions as they have not been committed yet. This means that for purposes of the STM system those versions do not exist yet and thus there also is no mapping yet which resolves their GVID. We can also take advantage of this when a transaction is aborted using the `Abort()` method. Since all the versions in the write set are only visible to the transaction which created them these versions can easily be discarded without having to perform reference counting.

The `Commit()` method only assigns each version in the write set the transaction's read and write set converted to the right format. It then passes a commit message containing its write set to the cGaoManager by calling its `Commit()` method which then performs the actual commit as described further below.

4.1.4 cGaoManager

The cGaoManager is a singleton class that manages access to the GAOs and performs commits of transactions. It provides the `GetHeadversion()` method which is used by cTransaction's read to obtain the GVID of the most recent version of the specified GAO. Additionally it can also resolve a GVID to its associated cGaoVersion instance using `GetVersion()`. This is done by maintaining two maps which map a GID to a GVID and a GVID to a cGaoVersion, respectively.

```
class cGaoManager
{
public:
  GaoVersionId GetHeadVersion(const GaoId& rGaoId);
  cGaoVersion* GetVersion(const GaoVersionId& rVersionId);
  GaoRelation  GetGaoRelation(const GaoVersionId& a,
                              const GaoVersionId& b);
  void Commit(cCpMessage& rMsg);
};
```

Listing 4.4: Interface of cGaoManager.

The cGaoManager also offers a method to determine the relation between two GAO versions. The four possible relations—equal, newer, older and undetermined—are characterized by the position of each version in the other's version history list. If version a can be reached by following the predecessor links of version b then b is newer than a and vice versa. If neither version can be reached by following the predecessor links of the other version the relation is undetermined. This can only happen when the two versions do not belong to the same GAO. Finally the versions are equal if they both have the same GVID.

The second part of the cGaoManager handles the commit of transactions and implements the consensus protocol. To allow concurrent commits the protocol is running in a separate thread which waits for commit messages to arrive. Once a message is received the thread adds the new versions as pending versions as described in section 3.4. It then performs the steps of the consensus protocol while handling new commit messages after every step. When a transaction successfully commits or needs to be aborted it gets notified by this thread and takes the appropriate action.

Chapter 5

Testing

When using the Distributed STM algorithm in a multithreaded application, the non-determinism inherent in these applications makes it impossible to test and evaluate the algorithm and its implementation in a reproducible manner. With only the scheduler deciding about which threads are executing at a given time there is the possibility that many aspects of the implementation do not get executed at all. It is also difficult to see how adjustments to the implementation affect the characteristics of the algorithm when the execution order of the operations can change with every test pass.

5.1 STM Test

To overcome these problems we developed a single-threaded application called *stmtest* which allows us to execute simple transactions in a given execution sequence. During execution it collects a number of statistics which allow us to evaluate the algorithm. It also checks if all conflicts are detected and resolved and thus if the correct and expected results are obtained. Because the execution order is given we can easily test cases which are very unlikely—but not impossible—to occur in a real multithreaded application, for example that there is a thread switch after every operation.

With the algorithm implemented in C++, stmtest was developed in that language too. At its core, stmtest is an interpreter for *stm scripts* (see section 5.2) written using the Boost Spirit Parser Framework [14]. With its heavy use of C++ templates and operator overloading it enabled us to directly inline a slightly modified Extended Backus Normal Form [15] (EBNF) description of stm script's grammar into the source code. The resulting parser gets generated by the C++ compiler during compilation which removes the need for external tools and grammar description files that other parser generators for C/C++ require. This allowed for rapid development as it permitted us to mix the code freely with the interpreter of the parsed script and the creation of the statistics.

5.2 STM Scripts

An example of an stm script can be seen in Listing 5.1. It includes six atomic blocks which represent six different transactions. They are named implicitly by the order they occur in the script file, the first atomic block being transaction one. The transactions one to three are used to create a read conflict on the variable A, the last two cause a commit conflict on C. As shown in the listing it is possible to use C++ style comments to document the script.

In stm scripts all variable identifiers consist only of a single letter. Capital letters stand for GAOs which are shared between all transactions and which are accessed using the Distributed STM algorithm. The variable name used in the script is also used as the unique identifier for the GAO. Small case letters represent variables which are local to the atomic block, that is they are not accessible in another block. These local variables are initialized with the value 0 at the start of an atomic block. All variables are signed integers and the operations allowed on them are assignments and the four basic arithmetic operations (addition, subtraction, multiplication and division).

Each non-empty line in an stm script represents one atomic instruction. The numbers given at the start of a line with an operation and at the end of an atomic block are sequence numbers. They indicate in which execution order each of the operations should be performed. For the given example this means that the first atomic block is completely executed (operations one to four), then the first instruction of the third atomic block (operation five) and so on. Note that the start of an atomic block does not have a sequence number because it does not matter when the transaction is started as long as it is started before any other operation is performed.

When an stm script is interpreted, all its instructions are put in an execution queue ordered by their sequence number, starting with the lowest. Then the first instruction in the queue is removed and executed till there are no more instructions left in the queue. If an instruction causes the transaction it belongs to to be aborted because of an unresolvable conflict then all other remaining operations of the respective transaction are also removed from the queue. For the re-execution of the aborted transaction all its operations then are appended to the end of the execution queue. The implication of this is that each transaction will be aborted one time at most, because the execution of all aborted transactions is serialized since their instructions are not interleaved.

5.3 Schedule Generator

While the ability to execute interleaved transactions in a specified execution order makes testing the algorithm less difficult, the effort needed to actually come up with the right scheduling sequences remains. There also is the aforementioned problem that aborted transactions are always re-executed without interleaving them with the remaining other transactions. Theoretically the stm script format could be extended

```
/* Create initial versions
 * This is needed so we get a read conflict on A
 * and not a commit conflict */
atomic{
  1: A = 0;
  2: B = 0;
  3: D = 0;
}:4

/* The next 2 transactions are interleaved
   to get a read conflict on A */
atomic{
  6: A = 1;
  7: B = 1;
}:8

atomic{
   5: x = A;
   9: y = B;
  10: C = x + y;
}:11

// Interleaved increment
atomic{
  12: D = D + 1;
}:14

atomic{
  13: D = D + 1;
}:15
```

Listing 5.1: stm script with transactions causing a read and a commit conflict.

to also include sequence numbers for the re-executed operations. But this would make writing stm scripts overly complicated—especially when transactions need to be aborted several times—and thus contradicts the idea of having a tool that eases testing.

To cope with these problems we enhanced stmtest with a schedule generator which can execute a given script in all possible execution sequences. This full coverage allows us to confirm that the algorithm always provides the correct results, independent of how the transactions are interleaved. Additionally this can be used to see whether adjustments to the algorithm only affect specific sequences or have a more general effect.

The generation of the schedules is done by performing a modified depth search over the tree of operation sequences (see figure 5.1 for an example). The operations of each transaction in a script are added to separate operation queues and the

```
atomic  // Transaction 1        atomic  // Transaction 2
{                               {
  1: A = 0;                       4: A = 1;
  2: B = 0;                       5: B = 1;
}:3                             }:6
```

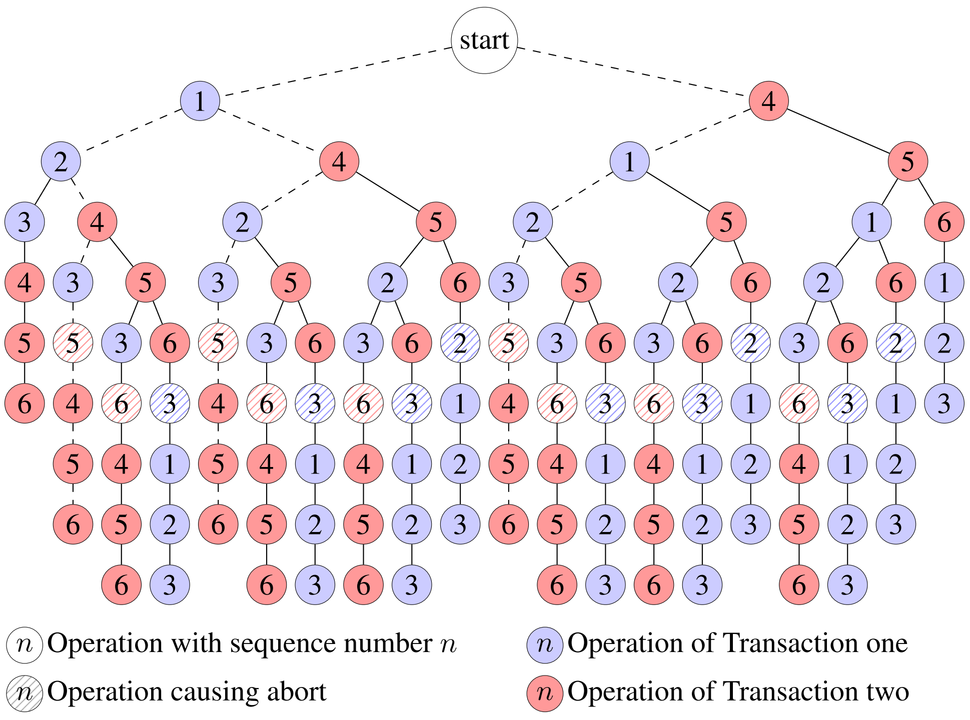
start
Operation with sequence number n
Operation of Transaction one
Operation causing abort
Operation of Transaction two

Figure 5.1: Illustration of the schedule generator.

children for each node are determined by taking the first operation of each queue. When a transaction is aborted all its operations simply are added to its queue again and thus get interleaved with the other remaining transactions. Instead of making a copy of the state of the STM algorithm and script interpreter at every node of the tree though we execute each path from the root of the tree to a leaf separately. This allows for a much simpler implementation because making a copy of the state would result in the need to copy a big number of memory locations and to update all the references to them. Thanks to avoiding the copying and because of the usually short length of an execution path this also does not result in a performance disadvantage.

Executing each path separately also keeps memory consumption low as we only need a list to store the path that was taken during the last execution and for each node in the path its next sibling. After execution the first path of the example in figure 5.1 this list would be $((1, 2), (1, 2), (1, 2), (2, -), (2, -), (2, -))$ indicating that the first three operations were taken from transaction one and that the next option was transaction two. The last three operations were taken from transaction two and there were no other transactions left that could have been executed instead. To generate the next execution path all entries at the end of the list with no next sibling are removed and the last entry is changed to the sibling. In our example this would be $((1, 2), (1, 2), (2, 1)$. The transactions are then executed according to this new path and the end of the path execution continues by taking an operation from the transaction with the lowest number that still has operations remaining. This is repeated till all nodes in the execution path list have no next sibling and thus when all paths in the tree have been covered.

5.4 Optimizations

One problem that can be seen in figure 5.1 is that even a very trivial stm script has a relatively big execution tree and 20 different execution paths. This is because the number of possible execution paths grows exponentially even when the additional execution path caused by aborts are not factored in. A script with 4 trivial transactions (see Listing 5.2) already has over one million different execution paths and executing them all takes over one minute on an up-to-date system. To make stmtest usable on less trivial scripts we thus needed to reduce its run time with several optimizations.

5.4.1 Equivalence Classes

To reduce the number of execution paths we can take advantage of the fact that a commit is the only operation that actually changes the state of the STM system. All other operations—starting a transaction or reading and writing a shared object—only change the state of the transaction performing the operation. This means that the order of the same read and write operations of different transactions between two consecutive commits (or a commit and the root of the tree) has no impact on the algorithm at all. All these permutations form an equivalence class and we only need to execute one path of each class to still obtain a full coverage of all paths.

The dashed execution paths in figure 5.1 all belong to the same equivalence class and illustrate the idea: For the rest of the execution path it does not matter in which order the operations 1, 2 and 4 are executed between the start and operation 3. The class also gets assigned a weight based on the number of permutations and thus the number of paths the class contains, so in our example it would be 3. This weight is used to correctly calculate the collected statistics.

Because of the way the schedules are generated only minor adjustments are needed to only execute one path per equivalence class. We order the children of a node by the number of the transaction they belong to by always taking the next operation from the first transaction that still has operations available. This way we always execute the first element—the one where the relevant operations are in ascending order—of each class before any other path of the same class . All other paths then can be skipped by never executing a read or write operation of a transaction with a lower number after one of a transaction with a higher number because we know that this execution path has already been covered.

Figure 5.2 shows the first path of each of the six equivalence classes and the remaining execution tree for our example. Also shown is the weight of each class.

Especially for scripts with many transactions can this optimization greatly improve performance. For the script in Listing 5.2 the number of executed paths is reduced to 33120 which is less than 3% of the total number of execution paths. Thanks to this the runtime is down to 1.7 seconds which is a significant improvement over the 1 minute it took when executing all paths. This makes stmtest usable for the evaluation of less trivial scripts.

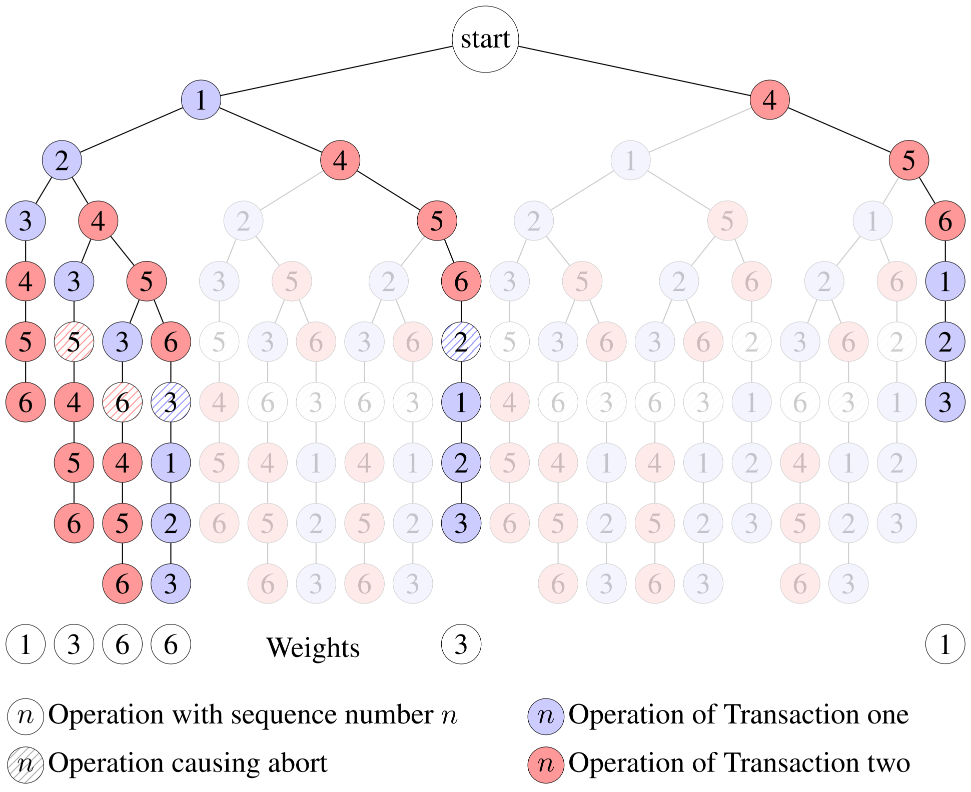
start
Weights
1
3
6
6
3
1
Operation with sequence number n
Operation causing abort
Operation of Transaction one
Operation of Transaction two

Figure 5.2: Remaining execution tree after taking advantage of the equivalence classes.

5.4.2 Memory Pools

Further inspection of the performance of stmtest revealed that the main bottleneck besides the number of execution paths is not the algorithm itself. It actually is the huge number of dynamic memory allocations and deallocations that are performed by the many data structures used to implement the algorithm. Introducing mem-

```
// Create initial version of D
atomic{
  1: D = 0;
}: 2

/* Interleaved increment */
atomic{
  3: D = D + 1;
}: 6

atomic{
  4: D = D + 1;
}: 7

atomic{
  5: D = D + 1;
}: 8
```

Listing 5.2: stm script with 1033272 different execution paths.

ory pools for the most frequently used containers allowed us to further improve performance by almost 500%.

5.5 Evaluation and Discussion

One of the main concepts of our STM algorithm is that it should theoretically be able to avoid conflicts and the resulting aborts by using older versions. Using stmtest we evaluated if this goal is actually achieved. Our first test was the simple script shown in Listing 5.3. For this test the GAOs A and B were initialized with the value 0 before execution started. This assured that there are only read conflicts during execution and no write or commit conflicts.

The results of executing all possible execution paths of this script are shown

```
atomic{
  5: A = 1;
  6: B = 1;
}:7

atomic{
  4: x = A;
  8: y = B;
  9: C = x + y;
}:10
```

Listing 5.3: stm script with two transactions causing a read conflict on A.

old versions		failed			$(A, B, C) =$	
limit	used	reads	commits	aborts	(1,1,0)	(1,1,2)
∞	1	0	0	0	97.14%	2.86%
none	0	3	0	3	88.57%	11.43%

Table 5.1: Statistics of script 5.3 when executed with different version history list lengths. The last two columns describe the distribution of the values of (A, B, C) after execution.

in Table 5.1. When there is no limit on the length of the version history list the script is executed without any conflicts. This is achieved by going at most one version back to avoid an abort. If we remove the ability to access older versions by setting the version history list's length to zero—and thus only store the most recent version—we obtain different results. In this case we get three unavoidable failed reads which lead to aborts. This confirms that our approach to reduce the number of aborts by keeping old versions available indeed works.

There is a price to pay for this advantage though: Using old versions also influences the resulting values of the GAOs after execution. Both $C = 0$ and $C = 2$ are valid results since we do not make any guarantees of the execution sequence of transactions. But it is indisputable that $C = 2$ is the more natural and desirable result as it is based on the most recent data. $C = 1$, however, is the result of using outdated data. The cost of avoiding aborts thus is the increased probability of obtaining values that are not up-to-date.

This test also shows that our STM algorithm alone is not enough to fulfill all synchronization needs. If a transaction wants to ensure that it only works with the most recent versions it also needs other synchronization objects like *barriers*. It is

```
atomic{
  1: A = 0;
  2: B = 0;
}:3

atomic{
  5: A = 1;
  6: B = 1;
}:7

atomic{
  4: x = A;
  8: y = B;
  9: C = x + y;
}:10
```

Listing 5.4: stm script with three transactions causing both read and write conflicts.

old versions		failed			
limit	used	reads	writes	commits	aborts
∞	2	10920 (1.88%)	42644 (3.24%)	280046 (37.09%)	333610
one	1	10920 (1.88%)	42644 (3.24%)	280046 (37.09%)	333610
none	0	22314 (3.70%)	49930 (3.79%)	272760 (36.48%)	345004

Table 5.2: Statistics of script 5.4 when executed with different version history list lengths.

possible to build a barrier using Distributed STM though, for example by using a GAO as counter and busy waiting until it has reached the right value.

For our second test we used a more complicated script (see Listing 5.4) with read, write and commit conflicts. We also did not initialize the GAOs this time, so they are created on the first access. As can be seen in Table 5.2 we once again manage to reduce the number of aborts by using older versions. The table also shows two other interesting results though. First, it is clearly visible that the usage of old versions reduces the number of failed reads and writes at the cost of more failed commits. When the transactions are long or computationally demanding these delayed aborts can actually result in a decreased performance even though the total number of aborts is reduced. Second, more older versions do not necessarily result in less aborts as can be seen by the equal results for version history list lengths of one and two. These older versions do get used, but they do not further help to avoid conflicts.

Chapter 6

Java Integration

One drawback of the evaluation using stmtest is that the grammar of the stm scripts only allows us to write very simple transactions. This means the results of the tests in the previous chapter are of a rather synthetical nature. To determine if the found characteristics of Distributed STM also hold true in more practical situations we decided to make the implementation available to Java code. We did this by exposing the functionality of our C++ library in a Java package using Java Native Interface (JNI).

6.1 DistributedSTM Package

The DistributedSTM package contains the interfaces and classes that are required to allow a Java application to perform transactions. At its core is the wrapper class Transaction (see Listing 6.1) which provides the interface from Java Code to our library. Each Transaction instance is linked to a cTransaction object by storing the address of the native instance. The wrapper itself is lightweight since each of its methods just needs to call the respective method of the underlying cTransaction object.

Contrary to stm scripts where GAOs are simply integers, the DistributedSTM package works with references. This allows us to manage the access to any kind of Java object. The only requirement is that the object needs to have a public `clone()` method because we need to be able to make deep copies as explained in section 4.1.1. This is ensured by requiring that each object read or written by a transaction implements the Clone interface. Writing this implementation is easy by using the implementation of the Object base class, which makes preparing any class to be used as GAO a matter of seconds.

To further simplify the usage of Distributed STM and the integration into existing code, the user also does not have to deal with GIDs. We automatically generate the identifier for each object by using the `hashCode()` method provided by Object. Since the Java virtual machine uses the memory address of an object as its hash, the hash is unique while the application executes and therefore fulfills our

```
public class Transaction
{
  public Transaction()
  {
    nativeCreate();
  }

  public <T extends Clone> T readGao(T gao)
    throws StmException
  {
    return (T) nativeRead(gao, gao.hashCode());
  }

  public <T extends Clone> T writeGao(T gao)
    throws StmException
  {
    return (T) nativeWrite(gao, gao.hashCode());
  }

  public void commit() throws StmException
  {
    nativeCommit();
  }

  private long nativePointer;
}
```

Listing 6.1: The wrapper class Transaction used to access the STM implementation from Java.

requirements for a GID. Sharing an object with another thread to access it concurrently thus is done by simply passing its reference, just like it would be done if the object was no GAO. It is just necessary that all read and write operations on the fields of the object are performed using the methods of Transaction. This ensures that the operations do not alter the original object but instead work on the versions managed by our STM algorithm. The fields of the initial GAO object actually never get written to, this object is only used to provide the hash and as initial value for the first GAO version.

To work around the limitation that Java does not provide a way to roll back already executed code we use exceptions. Whenever a method of a transaction fails because of an unresolvable conflict an exception is thrown to abort the transaction. The application is responsible for catching the exception and re-executing the transaction. Also care has to be taken that no objects which are not GAOs are modified within a transaction as there is no way to undo the changes to them. An example of how this can be done is shown in listing 6.2. The transactional code is simply put into a loop which is repeated until no exception is thrown and thus the

```
Foo gao = new Foo();

boolean done = true;
do
{
  done = true;
  try
  {
    Transaction t = new Transaction();
    t.writeGao(gao).x = 1;
    t.commit();
  }
  catch(StmException e)
  {
    done = false;
  }
}while(!done);
```

Listing 6.2: Exemplary usage of the DistributedSTM package to write to a GAO.

transaction was committed successfully. As this loop is the same for every transaction it would be possible to automatically create it. For example the user could mark all his transactions with the atomic keyword and a simple text replacement before compilation could add the required code.

6.2 Evaluation and Discussion

Synchronizing the operations of a red-black tree has become one of the standard tests for STM. The reason for this is that the rebalancing of the tree after elements get inserted or removed causes many other elements to be modified as well. This results in many conflicts between concurrent accesses to the tree and puts a lot of stress on the STM system. Red-black trees also serve as a good example to show how much easier synchronization becomes when using STM. Making the operations atomic can be done in a very short time, but implementing fine-grained locking for such a complicated data structure is a big challenge.

For our evaluation we used the Java implementation of a left-leaning red-black tree [16]. Making it thread-safe using Distributed STM was straightforward and only involved three simple steps. First we wrapped all public methods in the loop mentioned above to enable the abort of transactions. Second, we transformed all read and write operations of objects within the tree to use the respective transactional method. And finally we implemented the Clone interface for the objects managed by STM. So even with this manual usage of STM—as compared to STM that is integrated into the language—it was an uncomplicated task to prepare the tree for concurrent usage by multiple threads. This is mainly because we did not

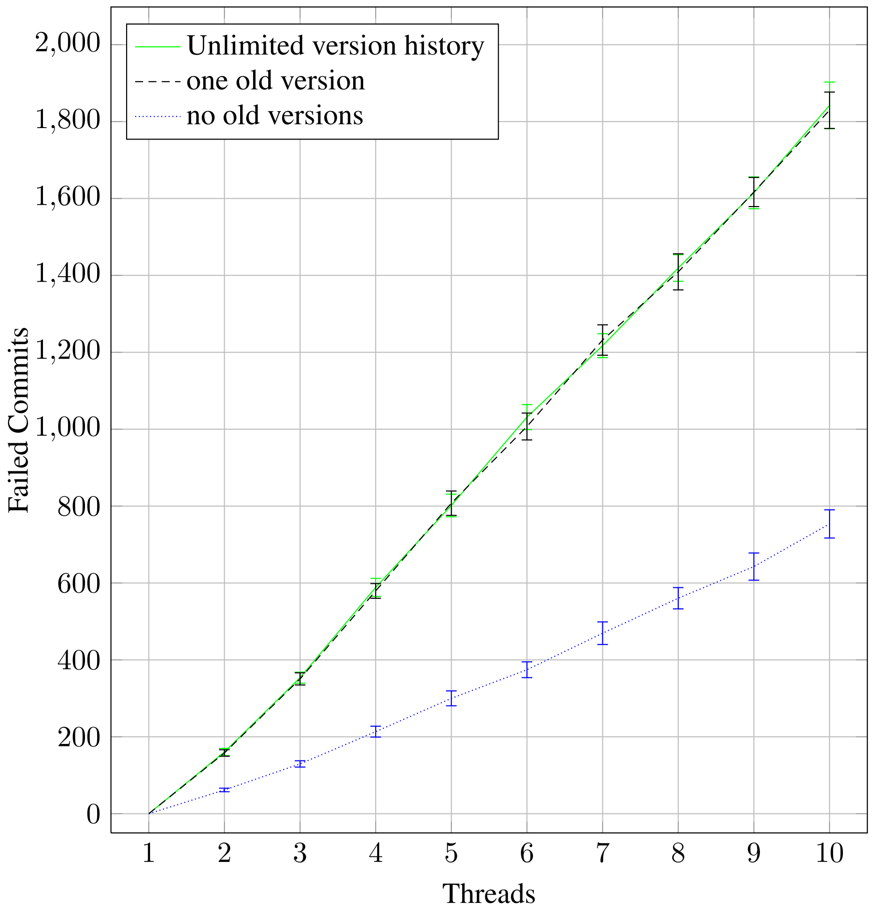

Figure 6.1: Number of failed commits in relation to number of threads and version history length for scenario one.

have to worry about all the problems, in particular deadlocks, which usually make correct synchronization so difficult.

To simulate realistic scenarios for a practical usage of Distributed STM we performed two tests. Scenario one represents a worst-case consisting of 50% read and 50% write operations, which causes a lot of contention. The more practical scenario two changes this distribution to 90% readers and only 10% writers. In both cases there is a total of 600 operations which are performed by a different number of threads. The write operations are evenly split into insertions and deletions. The tree initially contains 100 entries. We also performed the tests on differently sized trees and with a bigger number of operations, but this did not lead to different results.

All our measurements were done under Ubuntu 8.10 running on a 3.6 GHz Intel Core 2 E8400 dual-core processor. Since a dual-core processor obviously does not scale well beyond two concurrent threads, we did not examine the absolute

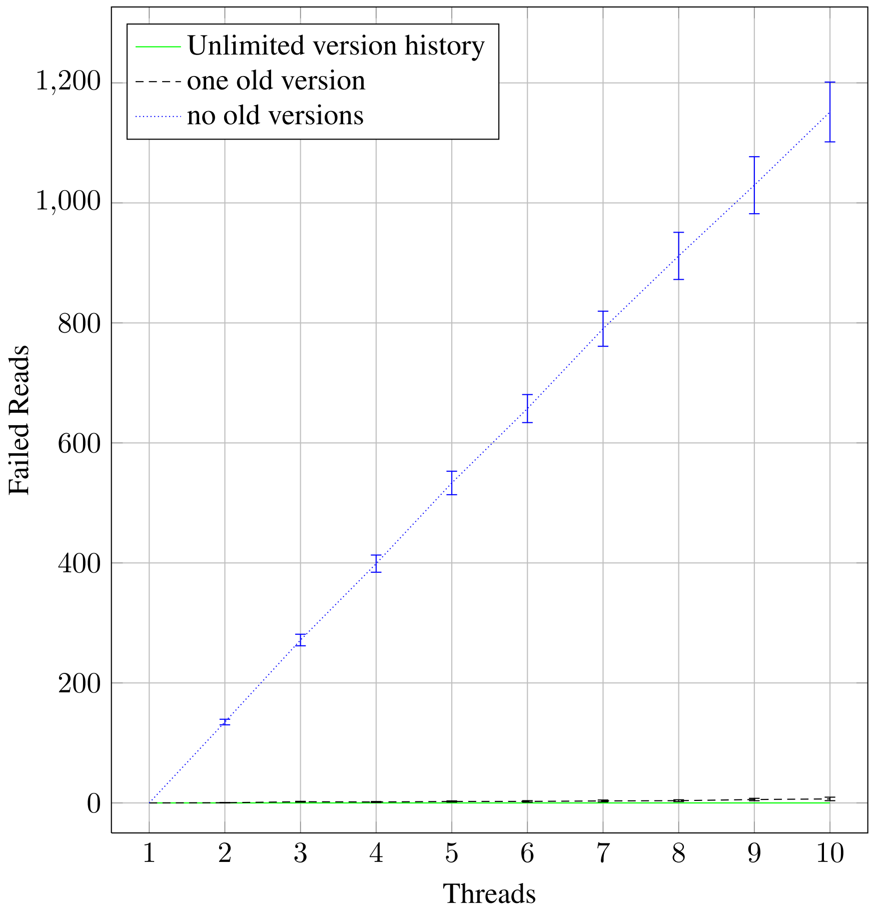

Figure 6.2: Number of failed reads in relation to number of threads and version history length for scenario one.

performance. Instead we looked at the number and the cause of the aborts to see how well our algorithm scales with the number of threads. Additionally, we wanted to see if the results about the version history length that we obtained in section 5.5 also apply to these scenarios.

All results presented in this chapter are the mean average of 50 measurements, and in the figures we also included the standard derivation. This is necessary because the results are highly dependent on three indeterministic factors: the scheduling of the threads by the operating system's scheduler, the type of operation a thread performs and the randomized consensus protocol. Because of the way the LLRB tree is implemented each GAO is read at least once before it is written to. This means that the write operations can never fail and thus do not need to be taken into account here.

Figures 6.1 shows the number of failed commits for scenario one. As expected from the results so far the usage of old versions considerably increases the chance

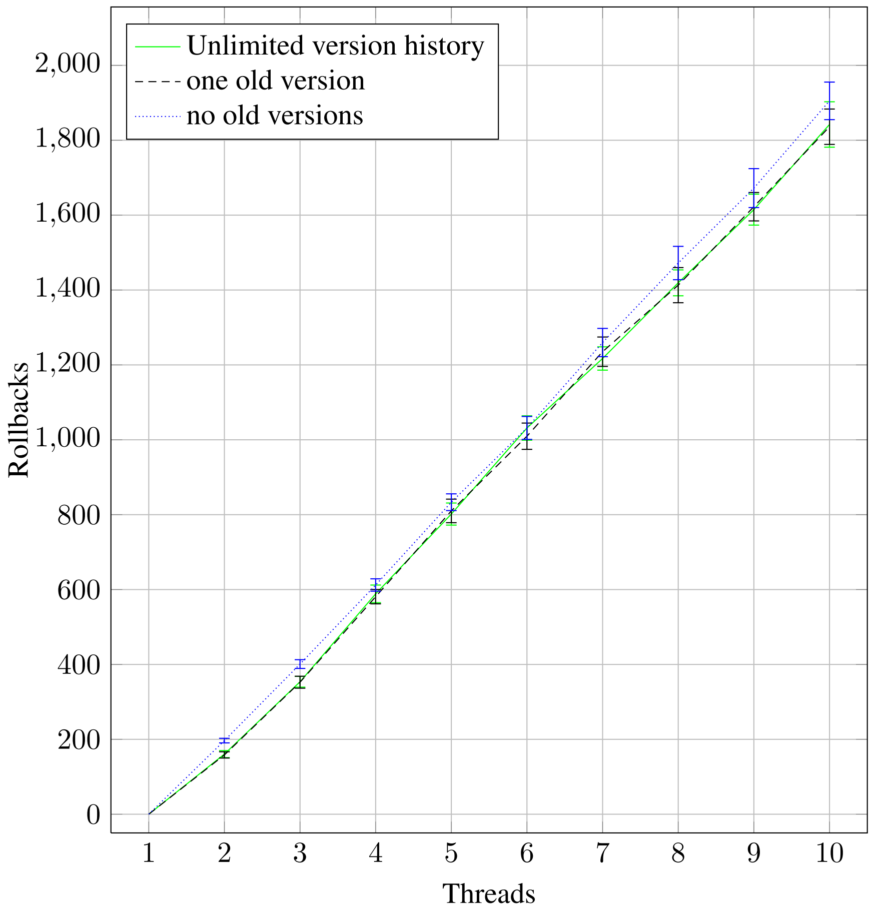

Figure 6.3: Number of rollbacks in relation to number of threads and version history length for scenario one.

that a commit fails. With ten threads and when only using the most recent versions, around 55% of the commits are not successful. When old versions are available this increases to over 75%. This means that less than one quarter of the completed transaction manages to commit. The advantages of using old versions can be seen in figure 6.2 though. Independent from the number of threads can we almost completely avoid failed reads. If old versions are not used around 45% of the transactions needs to be aborted because of a failed read. What really matters though is the total number of rollbacks, which in this case is the sum of the failed reads and failed commits. This number is shown in figure 6.3. As expected, Distributed STM can reduce the number of rollbacks by using the old versions in the version history list. But the margin compared to not using old versions at all is very small. Considering that failed reads are preferred to failed commits, because they result in an earlier abort, the best result in this scenario actually is achieved by only using the most recent version. Not using old versions also removes the overhead

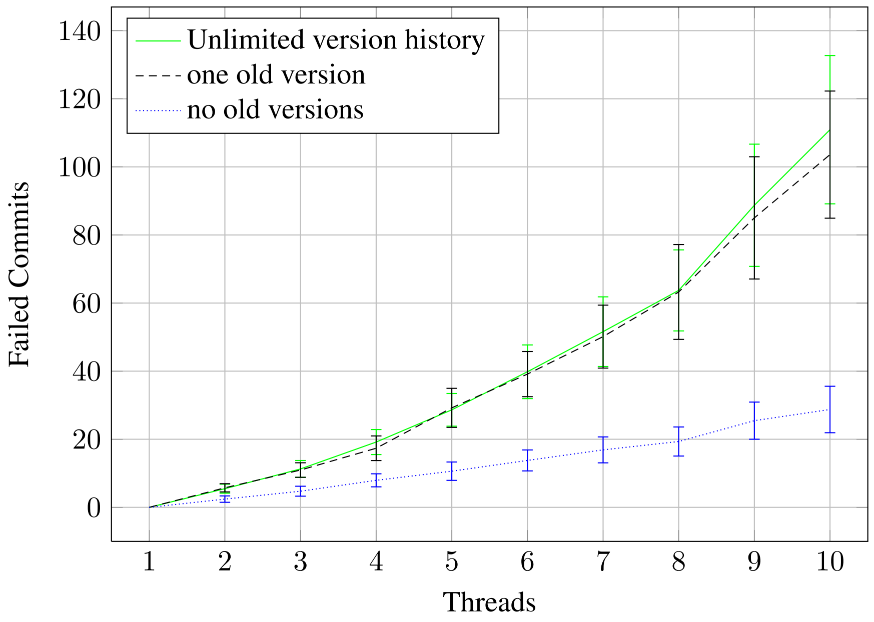

Figure 6.4: Number of failed commits in relation to number of threads and version history length for scenario two.

that would otherwise be necessary to maintain them.

It is also worth noting that there is almost no difference in the results between an unlimited version history list length and only keeping one older version. The additional overhead from keeping a big number of old versions certainly does not outweigh the minor reduction of rollbacks.

All three figures clearly show that Distributed STM scales linearly with the number of threads. Even though this scenario causes many conflicts because of the big number of concurrent writers, the number of transactions per threads increases slower than the number of threads. This means that we are actually improving performance, although with ten threads we only achieve a speedup of around 2.5 compared to one thread when looking at the number of transactions each thread needs to execute.

This changes when we look at scenario two. Thanks to the lower number of writers the speedup here is almost nine. Now we can also take big advantage of using the old versions: When only using the most recent versions, more than twice the number of transactions needs to be aborted. In this scenario the best result is actually achieved by keeping exactly one old version, but the difference to an unlimited version history list is small. This difference most likely is caused by the big variance of the results caused by the nondeterminism of the test. When considering the overhead caused by keeping more then one old version available though, it is clear that it is best to limit the version history to just one old version, at least for the synchronization of red-black trees.

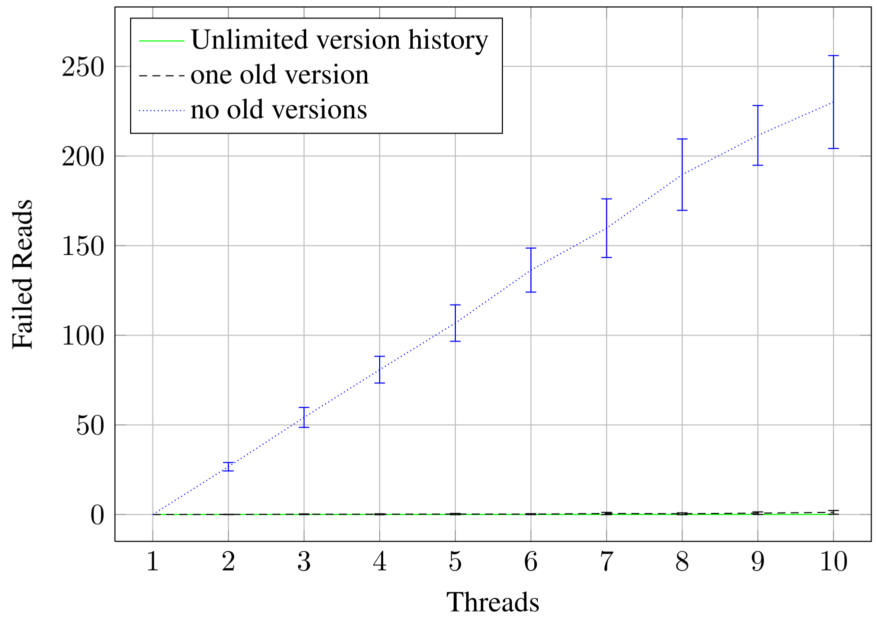

Figure 6.5: Number of failed reads in relation to number of threads and version history length for scenario two.

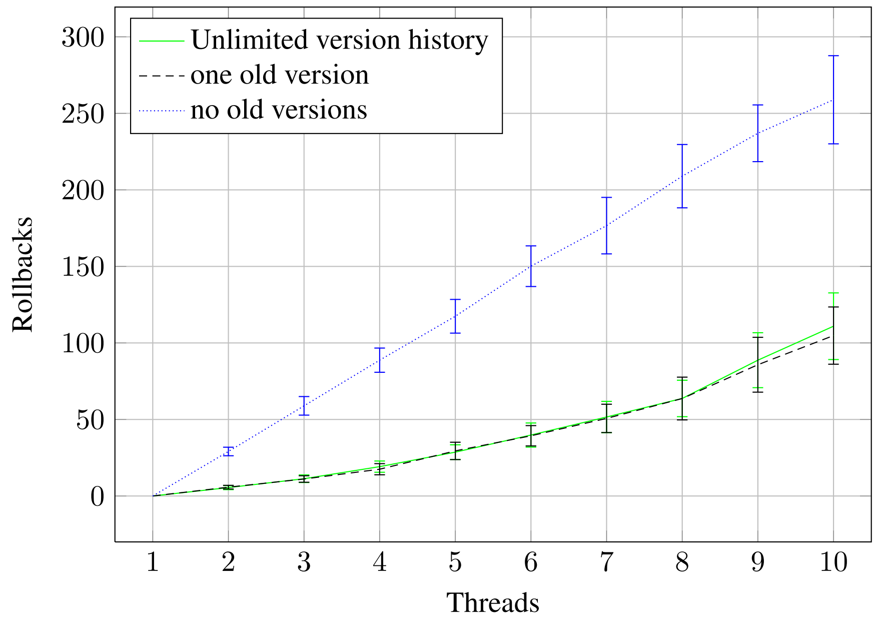

Figure 6.6: Number of rollbacks in relation to number of threads and version history length for scenario two.

Chapter 7

Conclusion

Writing concurrent code is becoming more and more important to fully utilize the power of both modern multi-core processors and distributed systems. Correct synchronization of the concurrent accesses to shared objects by using locks is difficult and error-prone though. Software transactional memory is an attempt to solve this problem, but most STM implementations so far focus on non-distributed systems.

In this thesis, we have developed and presented a novel approach with two key features that separate it from other STM systems. From the beginning, the design of our algorithm and implementation was done with distributed systems, in particular the AmbiComp virtual machine, in mind. Additionally, we keep old versions of shared objects available in a version history list to reduce the number of read conflicts between transactions.

Writing a simple script language and interpreter, which removed the nondeterminism caused by a normal scheduler, allowed us to evaluate Distributed STM in a reproducible manner. We were able to confirm that the old versions can indeed reduce the number of unresolvable conflicts between concurrent transactions in these theoretical tests. The price for this improvement is that the obtained values of the shared objects are in fact correct, but more likely to be not up-to-date. Thanks to the ability to run all possible execution paths of these scripts, we also validated that the algorithm indeed does provide correct results, even in situations that are very unlikely—but not impossible—to occur when using a real scheduler.

To see how our approach performs in a more practical scenario we made it accessible to Java applications and used it to synchronize a red-black tree. Our tests revealed that the usage of old versions provides the best results when there is only little contention. In those cases the number of aborts can be reduced to less than 45%. However, when there are many concurrent writers the effect is less visible and only around 5% of the rollbacks can be avoided. In both cases the number of transactions per thread increases slower than the number of threads though, showing that Distributed STM scales well. One unexpected result was that there is almost no difference in the results between an unlimited version history list and using only one old version. The reason for this probably is that the internal data structures of a red-black tree are modified after almost every operation, which rapidly makes old versions so outdated that they cannot be used to resolve conflicts.

7.1 Future Work

To investigate if the number of old versions available really has more influence on the results than shown here, further tests, especially practical ones, are needed. It would be best to replace the look-based synchronization in an existing multi-threaded application to see how the algorithm performs in a real environment.

There also is a possible modification to the algorithm which might further reduce the number of rollbacks by allowing a transaction to read pending versions. This modification greatly increases the complexity though because it requires a transaction to check if the pending versions it has read have yet become valid versions. If a transaction commits before it is decided whether those pending versions become new head versions or not, it would also be necessary to block the transaction and notify it later. Still, this change could possibly further improve the performance of Distributed STM.

The last task that remains is to actually integrate our solution into the AmbiComp virtual machine. This would make it available to a bigger number of applications and enable the evaluation of the characteristics of the algorithm on a distributed system. Especially the influence of the communication latency between the nodes needs to be investigated.

Bibliography

[1] Ambicomp project. URL: http://www.ambicomp.org.

[2] Bjoern Saballus, Johannes Eickhold, and Thomas Fuhrmann. Global accessible objects (gaos) in the ambicomp distributed java virtual machine. In *SENSORCOMM '08: Proceedings of the 2008 Second International Conference on Sensor Technologies and Applications*, pages 543–548, Washington, DC, USA, 2008. IEEE Computer Society.

[3] Thomas Fuhrmann. Scalable routing for networked sensors and actuators. In *Proceedings of the Second Annual IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks*, pages 240–251, September 2005.

[4] Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable memory transactions. *Commun. ACM*, 51(8):91–100, 2008.

[5] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software transactional memory: why is it only a research toy? *Commun. ACM*, 51(11):40–46, 2008.

[6] 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.

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

[8] C. A. R. Hoare. Towards a theory of parallel programming. pages 61–71, 1972.

[9] Robert Ennals. Efficient software transactional memory. Technical Report IRC-TR-05-051, Intel Research Cambridge Tech Report, Jan 2005.

[10] Dave Dice, Ori Shalev, and Nir Shavit. Transactional Locking II. In *In Proc. of the 20th Intl. Symp. on Distributed Computing*, 2006.

[11] Kaloian Manassiev, Madalin Mihailescu, and Cristiana Amza. Exploiting distributed version concurrency in a transactional memory cluster. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 198–208, New York, NY, USA, 2006. ACM.

[12] Christos Kotselidis, Mohammad Ansari, Kim Jarvis, Mikel Luján, Chris Kirkham, and Ian Watson. Investigating software transactional memory on clusters. In *IWJPDC '08: 10th International Workshop on Java and Components for Parallelism, Distribution and Concurrency*. IEEE Computer Society Press, April 2008.

[13] Maurice Herlihy, Mark Moir, and Victor Luchangco. A flexible framework for implementing software transactional memory. In *Proceedings of the 21th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 253–262, oct 2006.

[14] Joel de Guzman. Boost Spirit Parser Generator Framework, 2008. http://spirit.sourceforge.net/.

[15] ISO/IEC. ISO/IEC 14977 : 1996(E) - Extended BNF, dec 1996. Available at http://standards.iso.org/ittf/PubliclyAvailableStandards/s026153_ISO_IEC_14977_1996(E).zip.

[16] Robert Sedgewick. Left-leaning Red-Black Trees, Sep 2008. Available at http://www.cs.princeton.edu/~rs/talks/LLRB/.