DikuSTM
A Transactional Memory Library for C++ and C++0x

Master's Thesis

Jesper Alf Dam
jalf@diku.dk

Department of Computer Science,
University of Copenhagen

March 1, 2010

Abstract

As multicore processors become increasingly common, robust concurrent programming techniques are getting increasingly important. Locking mechanisms such as mutexes and monitors are commonly used, but are hard to use correctly, and do little to prevent common error situations such as race conditions, deadlocks and livelocks. Code written using these techniques also prevent programmers from abstracting away the implementation details of their code, as individually thread-safe pieces of code can not be safely composed into larger thread-safe components.

Software Transactional Memory (STM) is an alternative concurrency model, which uses transactions as the basic synchronization mechanism; any code within a transaction is executed in isolation from other transactions and *commits* its changes atomically, providing semantics very similar to those known from database systems.

In this work, strategies for designing and implementing a STM system in C++, and the upcoming language revision informally known as C++0x, are explored. With an emphasis on correctness and simplicity of use, generic programming techniques are used to design an elegant and general interface for library users, preserving compile-time type safety and minimizing the scope for programmer error, while also avoiding the performance penalty of run-time polymorphism.

We also develop a *double-buffered* deferred update scheme, eliminating many of the problems typically associated with deferred update systems. As all transactional data is allocated *in-place*, we also avoid the extensive dynamic allocations and pointer indirections typically associated with indirection-based systems.

Resumé

Idet flerkerne-processorer bliver mere og mere almindelige, øges nødvendigheden af bedre teknikker til at håndtere samtidighed også. Mutex'er og monitorer benyttes ofte, men er svære at bruge korrekt og gør intet for at forhindre "race conditions", "deadlocks" eller "livelocks". Kode skrevet ved hjælp af disse teknikker besværliggør endvidere abstraktion, idet individuelt trådsikre programstykker ikke nødvendigvis kan sammensættes til større trådsikre programstykker.

En alternativ måde at kontrollere samtidighed er Software Transactional Memory (STM), en model hvori transaktioner benyttes til at synkronisere adgang til delte data. Al kode i en transaktion udføres isoleret fra andre transaktioner, og ændringer i data foretaget i en transaktion synliggøres atomisk for resten af programmet, ligesom det kendes fra databasesystemer.

I dette speciale udvikles og evalueres et STM system implementeret i C++, med brug af ny funktionalitet inddraget fra den kommende revision af sproget, ofte kaldet C++0x. Vi fokuserer på korrekthed og enkelhed i bibliotekets grænseflade, og præsenterer en elegant og generisk grænseflade der forhindrer en række brugerfejl, og ved at benytte generiske programmeringsteknikker undgås desuden de omkostninger der er forbundet med køretids-polymorfi.

Vi beskriver desuden en ny *dobbelt bufferet* opdateringsteknik, der forhindrer de problemer der typisk er forbundet med "deferred update"-baserede STM systemer og tillader potentielt bedre ydeevne end *indirection*-baserede systemer.

CONTENTS

1. INTRODUCTION

1.1 Motivation

Concurrent programming is hard. While specialized programming paradigms such as CSP[1] and languages such as Erlang have been created to allow programs to scale across large numbers of processors with relative ease, these typically require the entire program to be structured radically different from what we are used to, and from what is often considered convenient. These approaches typically eliminate shared data entirely, instead offering message-passing primitives as the sole mechanism for exchanging data between threads or processes.

Functional programming languages also promise to simplify or even automate parallelization of code due to the elimination of side effects, theoretically eliminating any risk of race conditions, but again, the reality is not quite as simple. At the time of writing, I am not aware of a functional language that is able to give us such a level of concurrency "for free".

No matter how promising these approaches may be, they still suffer from one major problem: they cannot be retrofitted onto existing code. "Messy" imperative languages, full of side effects and shared state are the norm, and most existing code is written in these languages. So even if compilers for functional languages could give us automatic parallelization, and no matter how compelling the advantages of strict message-passing paradigms such as CSP, they do not solve the problem that in the most widely used languages, shared data is the norm, and parallelism and multithreading are extremely error-prone and ensuring correctness is nearly impossible.

In most common languages, primitives such as mutexes are used to serialize execution of certain code paths. Locking a mutex ensures that all other threads attempting to gain access to the same mutex are blocked until the thread currently owning the lock releases it. However, such a locking mechanism only ensures serialized access to *code*, when it was really the *data* that we wanted to protect. Ultimately, what matters is that only one thread at a time attempts to modify a piece of data, and that no reads occur while a write is ongoing. But many different code paths may attempt to access the same data, and so with this type of locking mechanisms, we have to protect every one of these code paths, which is tedious and error-prone. A further problem is the risk of deadlocks: if threads acquire the same locks in a different order, we create a potential deadlock — each thread may own a lock that another thread needs, so each one gets blocked waiting for other threads to release their locks, which never happens because they too are blocked and waiting.

Many variations and refinements of these concepts exist, which I have not described in detail. Monitors, semaphores, barriers and reader-writer locks are available in various languages or threading APIs, but suffer from many of the same flaws; they serialize access to code, rather than data. As a consequence, software written using these primitives is not *composable*: two individually thread-safe pieces of code cannot be safely composed into one larger thread-safe operation, as the following example shows. Assuming the following pseudocode class

[1] Communicating Sequential Processes — a theory of concurrency developed by C.A.R. Hoare in 1978, in which message-passing "channels" are used as the sole synchronization and communication primitive between concurrent processes

representing a bank account,

```
class Account {
  synchronized Withdraw(amount);
  synchronized Deposit(amount);
}
```

where the member methods for withdrawing and depositing money are both thread-safe (as indicated by the Java-like `synchronized` keyword), there is no way to implement a general *money transfer* function in terms of them. A transfer of money between two accounts consists of first withdrawing money from one account, and then depositing it on the other — but with the catch that the operation as a whole must be atomic: other threads in the system may never see the intermediate inconsistent state where money has been withdrawn from one account, but not yet deposited on the other. Such an atomic "transfer" function cannot be written given the existing interface. We have no way to prevent other threads from accessing the account after the money has been withdrawn from the source account and before they are deposited on the destination account. Instead, we would have to open up the class and add a `Transfer(Account dest)` method — but this is only postponing the problem. We still have no way to deal with more complex transactions, such as "if possible, take 100 euros from account A, otherwise take whatever is on account A, and the remaining amount from account B, and deposit the full amount on account C", or "take 20 euros from each of A, B and C, and deposit everything on account D".

Another option would be to break the abstraction that the class represents a bank account, and add explicit `LockAccount()` and `UnlockAccount()` methods, which would expose implementation details and introduce potential errors as the user may forget to unlock an account after use.

Ideally, we would like some kind of high-level "atomic" primitive, able to ensure that all modifications that occur within a block of code marked as "atomic" is carried out atomically. Then the above transfer function could simply be implemented as follows:

```
void Transfer(srcAccount, destAccount, amount) {
  atomic {
    src.Withdraw(amount);
    dest.Deposit(amount);
  }
}
```

Rather than protecting individual code paths, this would have the effect of protecting the data that is manipulated: the state of each account is updated atomically, so that no matter how other threads access the same accounts, the system will be in a consistent state.

Of course, defining such an `atomic`[2] primitive is easier said than done. It is not obvious how the system should even *know* which data modifications occur within an atomic block, much less ensure atomic updates of the relevant data. But theoretically, such a primitive could solve many of the problems with concurrency, without requiring programmers to rewrite their code in a completely different programming paradigm. As shown above, such an atomic block, if it can be implemented, fits naturally into imperative languages such as Java and C++.

[2] It has been pointed out that the term "atomic" is somewhat misleading [11] — the properties we are interested in are a subset of the ACID properties known from databases: namely, atomicity, consistency and isolation, or ACI. But the key property of our `atomic` block is really isolation, rather than atomicity — the modifications made inside the block must be carried out in isolation, so changes made within the block cannot be seen by other threads, and changes made *elsewhere* in the system are not seen inside the "atomic" block. However, the name "atomic" is more common in the literature.

1.2 Enter transactional memory

Transactional memory, as first proposed by Herlihy and Moss [8] offers to define just such a programming model. They proposed a way to introduce an abstraction similar to database transactions into general purpose programming languages. A database transaction guarantees four properties known as the ACID properties; a transaction occurs *atomically*, so the rest of the system will never see a partially completed transaction. They preserve *consistency*, and they execute in *isolation*, as if they were the only process executing in the system. The *durability* property used in database transactions is irrelevant in the context of transactional memory, as all data in main memory is transient and we do not normally expect programs to be able to survive hardware malfunctions or sudden shutdowns.

However, the *ACI* properties provide a very nice abstraction under which to coordinate concurrent reads and writes of shared data in a multithreaded system.

These *transactional* properties were originally envisioned as built into the hardware: the processor itself should have a separate cache into which intermediate modifications during a transaction could be written, and when a transaction commits, this buffer would be flushed, atomically updating main memory. Thus, the system memory itself would be given transactional behavior. Such systems are commonly called HTM, for Hardware Transactional Memory.

There were some notable shortcomings of this approach:

- it would require special hardware support, meaning it cannot be used on the billions of computers already in use across the world, from mainframes to servers, PCs or mobile phones,
- the hardware must necessarily impose some limitations in terms of the size of the transaction-local buffer. If a transaction attempts to modify more data than will fit in the buffer, how can this be handled safely and without violating the transactional properties?

To solve this, hybrid approaches have been proposed which exploit such hardware support, but are *also* able to fall back to a software implementation. If the transaction-local hardware buffer overflows, an interrupt could be fired, invoking software handlers executing the remaining part of the transaction using buffers dynamically allocated in main memory, and using more conventional synchronization mechanism to ensure the ACI properties.

A more radical solution is to abandon the idea of specialized hardware entirely, and implement transactional memory in software only (STM, or Software Transactional Memory), as first proposed by Shavit and Touitou [13].

Despite the advantages offered by a transactional memory, the idea has not yet caught on outside the research community. This is largely because no production-quality implementation exists. Many different approaches and implementation techniques are still being explored, and there is still no consensus yet as to what the desired semantics of such a system should be, or what operations would be necessary for real-world use.

1.3 Goal of this thesis

The overarching goal of this thesis is to bring STM systems one step closer to real-world use: a STM system should be developed with the goal of being as usable in real-world code as possible. To achieve this, it must not incur an unacceptable performance penalty, and the syntax and semantics should be well-defined and intuitive enough to enable programmers to understand and reason about the concurrency in their code.

In the following chapters, I will describe and implement such a STM system in C++. The C++ language is chosen for the following reasons:

- C++ is a widely used and supported language on almost every platform. A STM library for use in C++ will benefit a wide range of programmers,
- The accepted way to improve or extend the C++ language is through new libraries — language extensions are generally only considered once existing libraries have shown that a feature would be beneficial, and if implementing the feature as a language extension rather than a library would provide significant benefits. C++ is also expressive enough that many complex new concepts and features can be implemented as libraries without requiring modification of the compiler or the underlying runtime library, which is a common way to extend other languages or platforms, such as Java or .NET. It is possible in C++ to define a reasonably type-safe, performant and general STM library as a library. In most other languages, such a system would require at least some language extensions, and indeed most STM research on these platforms have focused on extending the language with new primitives and semantics,
- C++ has historically evolved in a very "bottom up" manner: rather than new features being added by the C++ standards committee to benefit C++ programmers as a whole, language improvements have often been added by the C++ developer community through libraries such as the Standard Template Library (STL) [16] or the Boost libraries [2], pioneering techniques such as generic programming and template metaprogramming. As explained by Stroustrup [17], library extensions are preferred to language extensions, and many language extensions are intended as support for library developers, rather than for users of the language. This tradition of external libraries influencing the course of the C++ language makes the development of a library-only STM system in C++ a tempting next step,
- the next revision of the C++ language is due to be finalized within the next year or two, and partial compiler support is already common. This new revision, commonly known as *C++0x*, enables several new features which may aid in the implementation of a STM system, either by enabling better performance or a cleaner and more concise syntax for users of the system.

However, I have no illusions that my implementation will be the last word on the subject. Many different implemention strategies and approaches are still being tried, and even between existing implementations, accurate and realistic performance measurements are difficult, making it hard to determine which is "best" from a performance point of view. Instead, this report will explore one specific implementation strategy to provide the desired semantics with acceptable performance.

A general and reusable *interface* for STM systems is also presented, so that if and when a better STM system is created, it can reuse the interface designed here, enabling users of the STM system to swap out the STM implementation without having to rewrite their own code. A uniform interface would also enable standard test suites to be applied to different STM implementations, allowing the relative performance characteristics of different STM systems to be studied and compared. A general and reusable *interface* for STM systems is also presented, so that if and when a better STM system is created, it can reuse the interface designed here, enabling users of the STM system to swap out the STM implementation without having to rewrite their own code. A uniform interface would also enable standard test suites to be applied to different STM implementations, allowing the relative performance characteristics of different STM systems to be studied and compared.

To achieve these goals, I will first analyze the requirements we as programmers and *users* of the STM system place upon it. It must define a suitable interface and semantics that

make the system as easy, intuitive and useful as possible — it must fit well into the language conventions, which in the case of C++ largely means that the system must work well with the STL and with generic programming.

We must then consider how the system can actually be implemented in C++. We must decide on the strategy used to implement the STM semantics: what is the mechanism that keeps transactions from seeing uncommitted modifications made in other transactions? Performance is a consideration as well: the system does not need to outperform traditional lock-based multithreaded code, but it must perform "well enough" to be considered a usable alternative.

Based on this analysis, the DikuSTM system is described and implemented. Finally, the system is evaluated, partly by testing that its performance is acceptable, and partly by demonstrating that the syntax and semantics are such that the library can be easily used in a broad range of real-world cases.

1.4 Source code availability

Source code for the DikuSTM library is available at either `http://jalf.dk/thesis/dikustm.zip` or at `http://www.diku.dk/forskning/performance-engineering/Jesper-Dam`.

The author can be contacted at `jalf.diku.dk` or `mail@jalf.dk`.

2. FUNDAMENTALS

Before getting into the detailed requirements and design considerations of our STM library, we must first establish some definitions and assumptions. In Section 2.1, we will detail the terms and concepts with which the reader is assumed to be familiar, as well as the environment in which the STM system is intended to work — including both the requirements placed on the underlying hardware, and the restrictions and considerations that must be kept in mind due to our choice of programming language.

An introduction to STM is provided in Section 2.2.1. We will establish some basic terms and definitions, and describe a hypothetical and syntactically simplified STM system, to illustrate what the basic programming model looks like and to introduce the basic operations that an STM system must provide.

The purpose of this chapter is not to perform a complete analysis of the problem domain, or attempt to solve all the problems encountered, but simply to establish a basic understanding of the domain: what does transactional memory programming "look like", and what assumptions must be made to make it possible to implement such a system?

2.1 Prerequisites

In the following, the prerequisites for the rest of the report are described: the assumptions made about the hardware on which the STM system is executing and some key characteristics of C++ that the STM system must respect.

The reader is assumed to be familiar with concurrency terms such as deadlocks, livelocks and race conditions, as well as common synchronization primitives such as mutexes. A basic knowledge of transactions in the context of databases is also assumed: in particular, the ACID properties of transactions, and commit or rollback of a transaction.

Finally, the reader is assumed to be familiar with C++ [9]. A few features from the upcoming revision C++0x [10] are used as well, but prior familiarity with these is not essential, as they will be explained as necessary in this text.

2.1.1 Hardware

When programming concurrent programs in a shared memory environment, we cannot normally assume that writes from one thread are immediately visible to other threads, or even that reads/writes occur in the order in which they were listed in the source code. Both the compiler and the processor may reorder instructions to improve performance. So to safely exchange data between threads, we assume the presence of a *memory barrier* primitive — that is, an operation that reads and writes may never cross. When a memory barrier is encountered, *all* reads and writes previously issued must complete before execution continues, and likewise, no read or write originally placed after a barrier may be moved up before it. These barriers are typically implemented as special hardware instructions, and exposed as intrinsic functions in the compiler.

Given such a primitive, we can work around the uncertainties of the memory model under

which our code is executing. We also assume the ability to execute certain operations atomically: reading and writing of native word-sized memory fields are assumed to be atomic, which is the case on most hardware. A more specialized requirement for my implementation is that incrementing and decrementing 16-bit integers must also be possible to do atomically. The x86 architecture defines such atomic increment/decrement operations for all integer sizes, but on platforms without this operation, the operation can either be emulated using the widely supported *compare and swap* operation, or the 16-bit fields can be widened to a size on which increment/decrement can be performed atomically.

2.1.2 The C++ language

The C++ language leaves many operations *undefined*. When undefined behavior is invoked, we can make no assumptions about the state of the system. Where many high-level languages offer a completely well-defined environment, in which every action is handled in a well-defined way (for example, bounds-checking on array accesses at run-time to catch and prevent out of bounds memory accesses ensures that even if such an operation is attempted, the system is left in a valid state). The C++ standard simply leaves the result of such operations undefined, and since the consequences of the action are not known, they cannot be recovered from safely. Segmentation faults can typically be caught through some mechanism provided by the operating system, but this merely keeps the application from terminating — it does not guarantee that the application is in the same state as before the error occurred.

This means that we must guarantee that transactions never encounter an inconsistent application state: if they do this, they could enter execution paths that should be logically impossible, thus perhaps entering an infinite loop, or performing out-of-bounds memory accesses, both of which would be impossible to recover from safely in C++.

2.2 Transactional Memory

A transaction in the context of transactional memory is inspired by database transactions, although with some important differences. As with database transactions, a Transactional Memory (TM) transaction consists of a sequence of operations to be executed in isolation, and *committed* atomically. It is possible that a conflict may occur during the transaction itself or in the commit phase. In that case, the transaction is *rolled back*, and all its speculative changes reverted.

It is possible that, after a transaction t_0 has accessed an object x, but before t_0 has committed, another transaction t_1 commits a modification to x. In this case, t_0's view of the application state is no longer valid when it attempts to commit, and the transaction must be rolled back. Another scenario may be that the same transactions access two objects, x_0 and x_1: first t_0 reads x_0, then t_1 commits a modification to both x_0 and x_1, and finally, t_0 attempts to access x_1. In doing so, t_0 encounters an inconsistency: the version of x_0 it saw earlier, and the version of x_1 it is about to read have *never* coexisted. If the transaction is allowed to proceed, it will see an *inconsistent* view of the application. Since the programmer may have written t_0 to assume some kind of consistency between x_0 and x_1, proceeding with the read could violate some programmer-defined invariant, causing unexpected behavior. This means that a transaction must validate each access of shared variables, and roll back immediately if an inconsistency is detected. A final cause for rollbacks may be that the transaction cannot acquire exclusive locks of the objects to be modified during the commit phase. Perhaps another transaction is in the process of committing changes to the same objects, so again, the transaction must roll back and release its locks.

In general, if the system rolls a transaction back due to a detected conflict or inconsistency, it will automatically retry the transaction immediately. The programmer is not notified that

the transaction failed to commit, and can simply assume that *when* the transaction returns, it will have committed successfully. By contrast, a programmer-initiated *abort* will roll back the transaction, and then instead of retrying, return control to the caller.

In order for transactions to execute in isolation, they typically rely on some mechanism for creating private copies of shared data, so that their speculative changes can be made to one instance of the object, while another preserves the original value so the transactions changes can be reverted in the case of a rollback. To avoid ambiguity, I will describe the "original" or "non-speculative" version of an object as the *canonical* one, while uncommitted modified versions are termed either transaction-local or private objects.

The set of objects accessed by a transaction is known as its *read set*. The subset of this which is modified by the transaction is known as its *write set*. In some TM systems, all code is implicitly executed in a transaction and so all types of data must support transactional accesses, but in most STM implementations, only a subset of types support transaction semantics and can be safely accessed by multiple transactions. These must typically define some additional metadata for the TM system to use, and are said to be *transactional* or we may simply call them *shared* objects or types.

2.2.1 The STM programming model

Before trying to define my own STM system, it is worth illustrating what we would like STM programming to be like. A basic STM system only has to support one single type of operation: It must allow us to mark a block of code as a transaction, ensuring transactional execution of the code. Some implementations consider this to be two separate operations, "begin transaction" and "commit transaction", but at a conceptual level, we simply wish to mark a block of code as transactional. The transaction should commit implicitly when leaving the block.

In pseudocode, such a transactional block could use syntax such as this:

```
void foo(list, length) {
  atomic {
    list.append(x); // append a new element to a list
    ++length; // adjust the length
  }
}
```

In this simple example, we place an `atomic` block inside the function — anything that occurs within this block must be committed atomically when we leave the block. In our case, the transaction simply consists of appending some new element to a list, and then updating a separate variable describing the length of the list. For the sake of simplicity, assume that function arguments are passed by reference, so that the same list may be visible to other threads.

The `atomic` block is dynamically scoped, so that functions called from within the block are also considered part of the atomic block. If function calls temporarily left the atomic block, transactions would no longer be composable.

Implicit in the above example is the *opening* of shared data — when a transaction attempts to access shared data, it must maintain certain metadata to preserve consistency and isolation, and to keep track of which objects must be updated when the transaction commits. To do this, each shared object on which the transaction operates must be opened for reading and/or writing. So a more explicit version of the above sample could look like this instead:

```
void foo(list, length) {
  atomic {
    txlist = list.open();
    txlist.append(x); // append a new element to a list
    txlength = length.open();
    ++txlength; // adjust the length
  }
}
```

Further, for optimization purposes, we may wish to distinguish between opening an object for reading only, and opening it for modification, so the general `open()` function may be replaced by `open_r()` and `open_rw()`.

The *commit* operation is implicit, in that whenever control leaves the `atomic` block, all modifications are committed. An implementation could expose an additional explicit `commit()` function

2.3 Side effects

Since the system may roll back and retry a transaction any number of times, side effects cannot be safely expressed inside a transaction — the side effect would be executed each time the transaction attempts to run, rather than occurring only *once* as part of the atomic *commit* operation. In general, STM systems commonly prohibit side effects entirely, although in principle, side effects could be allowed as long as they are reversible, and we are willing to relax the transactional properties of the system slightly. For example, imagine a transaction that inserts a new node into a linked list. That operation in itself is safe and side-effect free, but the node itself must be created on the heap, dynamically allocated with `new`, which is a side effect, and cannot be safely repeated if the transaction retries.

A solution could be to enable a mechanism for the programmer to specify additional actions to perform on a rollback, so that the side effect can be manually rolled back. In this case, the user could allocate the new node immediately when it is needed during the transaction, and at the same time specify that if the transaction is rolled back, this allocation must be freed. This still violates our atomicity and isolation requirements, as the side effect becomes visible immediately when performed, rather than when the containing transaction is committed — but since this only occurs when the programmer explicitly wishes it, that may be a worthwhile trade-off. If the programmer does not explicitly request otherwise, the ACI properties are maintained.

2.4 Extended operations

In 2006, Harris et al. [6] presented an STM implementation in Haskell which introduced two new operations: *retry* and *orElse*. While not essential in a STM system, these operations provide some useful semantics which simplify a number of tasks, so they should at least be considered when implementing an STM system.

2.4.1 retry

The `retry` operation can be considered the TM equivalent of *condition variables*. The user tests whether a condition holds, and if not, invokes `retry`, to restart the transaction when conditions have changed so that the test may pass. A naive implementation could simply retry the transaction immediately, although this would be extremely inefficient, equivalent

to polling, as the transaction constantly executes and re-executes until the condition being tested succeeds. A more useful implementation will instead record the set of objects opened by the transaction at the point when *retry* is invoked, and only restart the transaction when one of these is modified. This gives us an efficient method for blocking a transaction until some condition holds true. A simple pseudocode example could look as follows:

```
void foo(list) {
  atomic {
    if (list.empty()) {
      retry;
    }
    list.pop();
  }
}
```

This implicitly opens the `list` object, and if the list is empty, the transaction immediately aborts. The system will then monitor the `list` object, and restart the transaction when a modification to the list is committed. When the transaction restarts, it will perform the same test again, and assuming no other transaction has emptied the list again, it will pass the test and pop an element off the list.

Due to the isolation property of transactions, there is no risk of race conditions. The `pop()` operation is performed in the same transaction as the test on `empty()`, and so the state of the list is guaranteed to be consistent between the two.

2.4.2 orElse

The `orElse` operation provides support for *alternatives*, conceptually similar to the *alt* construct in CSP. `orElse` is used to chain two transactions together as one larger compound transaction, so that if the first transaction retries, the second is executed.

A pseudocode example may look as follows:

```
void foo(list) {
  atomic { // first transaction
    if (list.empty()){
      retry;
    }
    list.pop();
  }
  orElse { // second transaction (is also implicitly atomic)
    list.append(x);
  }
}
```

As before, we use `retry` to verify that the list is non-empty before pushing elements off it. However, this time, this transaction is followed by an `orElse` statement — if `retry` is invoked, the transaction does not restart as described in Section 2.4.1, but instead the *alternative* transaction listed after the `orElse` is attempted. If this retries as well, the entire compound transaction retries: when an object opened by either transaction is modified, the process starts over, attempting each transaction in order until one of them succeeds.

This simple construct allows the library user to decide whether or not to block on a call: if the user wishes to block until the operation succeeds, the operation is attempted in isolation. If non-blocking behavior is desired, `orElse` is used to provide an alternative which, for example, simply returns an error code indicating that the first transaction did not succeed.

2.5 Related work

Transactional Memory was first introduced by Herlihy and Moss [8] in 1993 who proposed a set of hardware extensions to enable transactional memory semantics. Shavit and Touitou [14] introduced the term Software Transactional Memory. Herlihy et al. described the first *dynamic* STM implementation[7] in 2003, which, unlike earlier systems, did not require the set of memory locations accessed by a transaction to be specified in advance.

In 2005, Harris et al. introduced the *retry* and *orElse* operations in their Haskell STM system. In addition to the two new primitive operations, this implementation used the type system to prevent side effects occurring in a transaction. The Haskell language already uses a special monad to introduce all I/O operations, effectively making side effects visible to the type checker. The STM system introduced a similar Transaction monad, allowing the type checker to verify at compile time that transactions do not contain side effects. The primary drawback of this system is the somewhat esoteric language used: while Haskell is getting fairly popular, it is still not accessible to mainstream programmers.

Another master's thesis was written here at DIKU by Egdø [4] in 2007 which described his ESTM library. Where earlier work on STM systems in C++ has treated the language very much like C with a few extra features, ESTM used more advanced concepts and features such as generic programming and compile-time polymorphism to create a very clean and elegant system. To my knowledge, this is the first STM system for C++ that allows existing types to be reused as-is.

Larus and Rajwar's book, *Transactional Memory* [11] provides an excellent introduction to transactional memory, and contains a detailed explanation of the basic concepts, as well as a discussion of what goals implementers should strive to achieve, both in terms of features, syntax, semantics and performance, in order to deliver a widely useful STM system. Finally, the book provides an overview of existing implementations, highlighting important milestone systems.

3. ANALYSIS

In Section 2.2.1 the basic usage of an abstract STM system was described. In Section 3.1 we will identify the requirements for a STM system to be considered practically useful. The different implementation strategies and design choices available for the DikuSTM system are discussed in Section 3.2. Based on this analysis, the high-level design of the DikuSTM system is described in Section 3.3.

3.1 Requirements

In order for our STM system to be usable in the real world, it must satisfy a number of requirements. The abstractions, syntax and semantics presented to the programmer must be consistent, usable and intuitive, and the system must be safe and efficient enough to be considered worth using.

3.1.1 Genericity and type constraints

The STM system should be as generic and type-agnostic as possible; if modifications to existing types are necessary, they should be as unobtrusive as possible, but preferably, existing types should be able to be reused unmodified. So far, most STM implementations in C++ have required types to inherit from some kind of base object such as the `transaction_object` of TBoost.STM. This approach causes a number of problems:

- POD types[1], including built-in types, cannot be used in transactions,
- existing types must be modified in order to be used in transactions,
- introducing an additional base class is not always desirable, as it may force programmers to use multiple inheritance, which is often considered bad design, and invalidates certain assumptions about the layout of the class.

We do not actually need the "is-a" relationship usually represented by inheritance. The data we wish to work on does not have to know about transactions, as long as the transaction is able to copy and overwrite the data. There is no reason why it should not be possible to apply transaction semantics to built-in types such as `int`. Apart from the synchronization and locking needed to ensure atomicity, the only thing we really need is the ability to take a copy of the data being modified, and on commit/rollback, write this private copy back into the original object.

The STM system should require only this minimum of functionality from types, without any intrusive changes such as forcing objects to derive from a special transaction base class.

[1] In C++, a POD (Plain Old Data) type is essentially a type compatible with C — that is, it must either be a primitive type, or a struct/class where all members are public, no inheritance is used and containing no nontrivial constructor/destructors, and where all members are POD types. Such types are given special treatment in many cases. For example, the `memcpy` function from the C standard library may only be used on POD types.

For this, a simple class template could be used, which internally stores the object that must be transaction-enabled. RSTM[2] and ESTM both use the name *shared* to describe the object fulfilling this role, so reusing this terminology, we can name the template `shared<T>`, where `T` is the type to be made transaction-enabled.

3.1.2 API design

As mentioned in Section 1.3, part of the objective of this work is to define a good generic interface that does not expose any unnecessary implementation details, so that the STM implementation can be swapped out without affecting the code using it.

C++ is already a rich language with its own distinctive style, idioms and conventions. A STM library that does not fit into this model cannot be adopted in real-world C++ code. Beyond simply adhering to existing conventions, we must also expose an API that is simple and understandable. It must present well-defined and logical abstractions, and it should be hard to misuse the API and thus break the STM system.

Previous C++ STM systems have tended to rely on macros, or required the user to explicitly write large amounts of boilerplate code. For example, to define a transaction in RSTM, we have to use a pair of `BEGIN_TRANSACTION`/`END_TRANSACTION` macros.

TBoost.STM[3], another well-known C++ STM library, does not even provide these macros, and instead requires the user to write all the retry logic manually. The following example is taken verbatim from Gottschlics and Connors [5] paper describing the system:

```
transaction t;
transaction_state state = e_no_state;
int val = 0;
do
{
  try
  {
    t.write(global_int).value()++;
    val = t.read(global_int).value();
    state = t.end_transaction();
  }
  catch(aborted_transaction_exception&)
  {t.restart_transaction();}
}while(state != e_committed)
```

These approaches have several problems: they are error-prone and rely on the user to strictly follow the correct structure for a transaction. The macro approach suffers from all the problems normally associated with macros, while the explicit approach is extremely verbose and cumbersome. Further, they both allow the transaction to easily see all other local variables declared in the enclosing function. While this is not necessarily an error, allowing the transaction to see too much nontransactional state is going to make the transaction harder to reason about, as it can easily modify nontransactional variables. In the macro case, we have also effectively hidden a loop from the user, meaning that code such as the following does not do what the user expects:

[2] Rochester Software Transactional Memory, developed at the University of Rochester [15].
[3] Formerly known as DracoSTM, this STM system developed at the University of Colorado-Boulder[5]. The stated goal of this library is adoption into the Boost libraries, as the new name indicates.

```
for (int i = 0; i < 100; ++i){
  BEGIN_TRANSACTION
  ...
  break;
  END_TRANSACTION
}
```

In this case, the user would intuitively expect to break out of the *visible* loop. But the `break` statement will instead break out of the loop created by the transaction macros. Likewise, `return` statements may have surprising effects, returning from a function in the middle of a transaction.

Since a transaction should occur in isolation, it makes sense to give it a unique scope, in which other local variables will not be visible, and so that the transaction loop only has to be written and maintained in one place. It is worth noting that the entire transaction actually has a fairly simple structure, consisting of a number of lines setting up the loop, then the actual transaction that the user wishes to execute, and finally, the end of the loop and a bit of cleanup code. Both the setup and cleanup code is independent of the actual transaction being executed, and should ideally be factored out to avoid duplication.

In a functional language, this could be easily achieved using higher-order functions. Assuming the user expresses his transaction as a separate function with some type `TxFunc`, perhaps with the signature `void myTx(transaction tx)`, we could simply define a higher-order function taking this user-defined function as its parameter:

```
void Atomic(TxFunc f) {
  while (not committed) {
    transaction tx; // create a transaction
    f(tx); // execute the user-supplied transaction
    commit(tx); // attempt to commit the transaction
  }
}
```

A structure such as this would solve many of the above problems: The scope of a transaction is now a user-defined function, meaning that it cannot accidentally access local variables of the calling function. Nor is the transaction loop visible to the transaction body, so that `break` statements cannot affect the loop. `return` will also behave as expected, simply exiting the transaction body, but without automatically leaving the transaction commit loop.

Fortunately, this approach is not limited to functional languages. C++ does not directly support higher-order functions, but the effect can be approximated, either with function pointers or with function objects or functors. This is a well-known idiom, and is used to implement the algorithms of the STL. If `TxFunc` is made a template type, then `Atomic` can be called with either a function pointer or a function object. With the support for lambda expressions in the upcoming C++0x, we could get even closer to an ideal syntax, as the user-defined transaction body no longer has to be a separately-defined function. Instead, a transaction which increments a transactional object `x` could be expressed as `Atomic([&] (stm::transaction& tx){++tx.open(x);} )`[4]. Using this approach, we have eliminated the explicit loop as well as the macros and the scoping problems.

The explicit `transaction` parameter could perhaps be omitted, if the opening of transactional objects was implicit, but passing such an object to the transaction function solves a couple of other problems: it gives the user a single interface to manipulate in order to interact

[4] The precise syntax of lambda expressions in C++0x is beyond the scope of this text, but in a nutshell, the first `[]` indicates what should be included in the lambda function's closure. The ampersand indicates that a reference to local variables visible in the enclosing function should be added to the closure, effectively making `x` visible to the lambda function. After the bracket follows the function definition, consisting of the function parameters followed by the function body as in any other function definition.

with the STM system — for example, aborting a transaction can simply be done by calling `tx.abort()` inside the transaction body. It also helps us ensure that certain operations can only take place within a transaction function — outside a transaction, the object will not be available, and so there is no way to call its member functions and interact with the STM system.

If transactions were allowed to see inconsistent data, all rollbacks could be deferred until the entire transaction has completed executing. In that case, rather than committing when the user-defined transaction function returns, we would simply revert all modified objects to their canonical values. However, as previously described, we cannot allow transactions to see any kind of inconsistency, and so we must be able to roll back in the middle of a transaction, when it attempts to open a variable. The only way to do this in the C++ language is to throw an exception, unwinding the current function, and being caught in the outer "atomic" loop. Apart from the performance penalty incurred by exception handling, which in most cases is insignificant, a more serious problem is that the user may catch these STM exceptions that are intended to pass through the user's transaction function.

To discourage this, STM exceptions should not be derived from the `std::exception` base class. Of course, a `catch(...)` statement will catch STM-internal exceptions as well as the ones actually intended for the user to handle, so these cannot be allowed inside a transaction body. But assuming the user only attempts to catch exceptions derived from `std::exception`, which is a common best practice anyway, exceptions generated by, and intended to be caught by, the STM system, will pass through the user-defined transaction function as intended.

So far, this API has been described making no references to my actual implementation strategy, which is intentional: one of the goals with the design of this API was to create an interface general enough to be used for *any* STM system for use in C++. The only real constraint this API poses for the library implementation is the explicit `open` operation yielding a reference to the transaction-local copy of an object directly. Some STM systems do not provide this operation, but provide *smart pointer* objects that must be used on *every* access to an object. However, that approach both clutters the syntax and suffers the additional overhead of updating or looking up metadata on every access. Additionally, that approach cannot prevent the user from simply dereferencing the smart pointer to get a reference to the real object being manipulated. As long as the transaction-local object is not moved around in memory, a reference to it can be returned directly from an initial `open` operation, saving the overhead of future accesses. Likewise, the assumption that transactional objects are implemented through composition, as members of some STM-defined wrapper type, rather than intrusive inheritance is safe, as it enables additional type safety and prevents transactional values from being accessed outside of transactions, while allowing predefined types to be used in transactions with no modifications. Systems relying on inheritance or other intrusive approaches provide no real benefits, and so I consider it acceptable that those approaches cannot be expressed using the API described here.

3.1.3 Exception semantics

If a transaction throws an exception, should the transaction roll back or commit? It is tempting to say that exceptions are indicative of errors or failure to complete some operation, and so the transaction should be rolled back before the exception is rethrown to the caller.

However, we believe the opposite to be more consistent and intuitive. An exception may indicate that some user-defined operation failed, but this does not necessarily imply that the *transaction* has failed. In particular, a transaction may consist of multiple operations, the first of which complete successfully, and only the last throwing an exception, and this may be anticipated by the user. We believe transactions should try to mirror the semantics of non-transactional code in this respect: if the user calls two functions `f1()` and `f2()` in

that order, and `f2` throws an exception, he does not expect the effects of `f1` to be rolled back. Instead, the result of the transaction should be "`f1()` completed successfully, and `f2()` threw an exception, ending the transaction."

If we assume that *every* exception that escapes the transactions is indicative of transaction failure, we are second-guessing the programmer, which may lead to surprising semantics in some cases. Transaction failure should only occur when a conflict is detected by the STM systel itself. User code failure is a separate concept, and failure in a user-defined function does not imply failure of the transaction in which it occurred.

A further argument in favor of committing when a transaction escapes a transaction is that if the transaction is rolled back, then the situation that provoked the exception no longer exists. At a conceptual level, the transaction *never happened*, and yet it returned an exception potentially carrying information about the application state at the time it was thrown.

For these reasons, we believe the most consistent and well-behaved semantics to be that a transaction implicitly commits when control leaves the transaction function, regardless of *how* it leaves the function. The transaction should only be rolled back when the STM system detected an inconsistency, or when the user explicitly aborts the exception.

3.1.4 Performance

In an ideal world, the STM system should take full control of the entire application: every variable access, whether a read or a write, should be protected by transaction semantics. In practice, the cost of this would be prohibitive. C++ does not expose a robust mechanism for intercepting arbitrary memory accesses, and given that every modified byte of memory would have to exist in multiple copies (at least one copy containing the transaction's uncommitted modifications, and another storing the original value in case the transaction is aborted), memory consumption would increase dramatically.

Instead, the application must be split into transactional and nontransactional parts. The user should indicate which objects are required to support transactions, so that there is no overhead for nontransactional code, following the common C++ design principle that "you don't pay for what you don't use".

A common source of inefficiency in C++ is excessive copying of objects, and this issue is very relevant to STM systems. After all, the main mechanism for implementing the isolation property is the creation of private copies of the data that is being modified. To achieve reasonable performance, this should be kept in mind when designing the STM system, to keep the number of copy operations at a minimum. A C++0x feature that may be relevant here is that of *rvalue references*, used to implement *move semantics*. For types with expensive copy operations, it is often possible to define a cheaper "destructive" move operation. For example, copying a `std::vector` requires the internal array to be copied. But a *move* can be implemented simply by "stealing" the array from the source vector, effectively a simple pointer swap operation.

We should also consider the number of indirections required to access transactional data, and the impact this has on the CPU cache. Ideally, when opening a transactional object, its data should already be present in cache, and should exhibit good locality. Some implementations have required all transactional objects to be allocated on the heap, scattering them and reducing locality in addition to requiring at least one further level of indirection to access the object.

It also seems unavoidable that "opening" a shared object will incur some overhead: if the object has not previously been opened in the transaction, then we must create some kind of metadata to track the object and ensure isolation while the transaction is running. If the object has been opened previously in the same transaction, we may be able to avoid updating

metadata, but instead we have to look up the existing metadata entry. This means that the *open* operation should be explicit and return a reference to the "raw" data, so that repeated reads or writes of an object within a transaction only has to incur the overhead of the `open` operation once.

3.1.5 Safety

One of the defining characteristics of good C++ libraries is how safe they are. The RAII[5] idiom is used to ensure that acquired resources are released implicitly and automatically, reducing the scope for programmer error. Exception safety guarantees are offered so that users of a class or function can be sure that even if an exception is thrown, the application will not enter an inconsistent or invalid state. Finally, templates and generic programming are exploited to provide extensive type safety at compile time for all operations. All these properties should also apply to a STM library.

Given that the entire program cannot run within a transaction for the reasons explained in Section 3.1.4, there will be a distinction between transactional and nontransactional code, as well as transactional and nontransactional data. As far as possible, this distinction should be enforced at compile time by the STM system. For example, nontransactional code should not be able to inspect transactional data, as this would violate the ACI properties of transactions. In specific cases, it may be meaningful for transactional code to access nontransactional data, but outside of these limited cases, that too should be limited by the STM system. Likewise, it should not be possible to invoke transaction operations such as `commit`, `retry` or `abort` outside of a transaction.

C++ code normally provides one of three *exception guarantees* [1], which help ensure consistency even in the face of unexpected exceptions:

- The *no-throw* guarantee specifies that the operation *cannot* throw any exceptions.
- The*strong* guarantee specifies that if the operation throws an exception, all affected data is reverted to state it was in before the operation. This is similar to the semantics for rollbacks on transactions, and should be provided when possible.
- The *basic* guarantee simply specifies that if an operation throws an exception, the application is left in a valid, consistent state and no resources are leaked. Every operation should at a minimum provide this guarantee.

For an STM system, we must consider which exception guarantees we are able to provide. *no-throw* is out of the picture: a transaction consists of user-defined code, and we have no way of ensuring that it will not throw an exception. The *strong* guarantee can often only be provided through indirection: rather than overwriting an object, which is irreversible, a separate copy should be made, and a pointer updated to point to the new copy instead of the original object. This way, even if creation of the copy causes an exception, we can at least revert to the initial state. If the object was overwritten directly and an exception thrown afterwards, we could try overwriting again with the original value, but this could fail, preventing us from reverting to the initial state.

Unfortunately, this rules out the *strong* guarantee for many implementation strategies. Providing only the basic guarantee in the general case is unacceptable, and the strong guarantee much better models the transaction semantics we wish to create. However, it may be acceptable for the system to provide the strong guarantee in some cases only, and fall back

[5] Resource Acquisition Is Initialization — the accepted name for the idiom of mapping resources to classes, so that resources are acquired in an object's constructor, and released in its destructor. If the object has a deterministic lifetime (if it is either a class member of a function local object), we are ensured that its internal resource will be freed correctly when it goes out of scope.

to the basic guarnatee in others; if the transaction operates on objects whose copy/assignment operations themselves provide the strong guarantee, the transaction as a whole should offer the same. If the user wishes to implement transactions on objects which can only support the weak guarantee, the transaction as a whole may be downgraded to guarantee only weak exception safety.

3.1.6 Transparent nesting

Although some STM systems have considered support for nested transactions to be a kind of optional extra feature, it is essential in order to achieve composability: if two transactional code snippets are composed into a larger transaction, nesting must be supported. A further complication is that this nesting must be transparent to the user — a user of the STM system might not know whether a given function uses a transaction internally, so no special syntax can be required for nested transactions.

Similarly, since the user might not have a reference to the outer transaction, the inner transaction's constructor must not require a reference to the transaction it is nested within. The system must keep track of this implicitly and automatically.

We must also consider the nesting model to be used. These can be categorized as follows:

- *Flattened nesting* describes a model in which inner transactions are "merged" into outer ones: the inner transaction operates directly on objects visible to the outer transaction, and its commit operation is effectively a *no-op*. If the inner transaction attempts to rollback and restart, the outer transaction is restarted as well. Semantically, this is acceptable in the common case as long as each transactions execute in a single thread, and nested transactions always execute in the same thread as their parents. This ensures that although the uncommitted changes of the inner transaction are technically visible in the scope of the outer transaction, the outer transaction does not execute until the inner one terminates, and so ACI properties are maintained. It does mean that the entire "stack" of transactions must be restarted if the innermost one attempts to restart, which may be considered unnecessary overhead — but on the other hand, if the innermost transaction encounters inconsistency forcing it to restart, it will likely encounter the same inconsistency again when the outer transaction attempts to commit. A more important limitation of this model is that if the inner transaction is explicitly *aborted* but not restarted, its speculative changes cannot be rolled back, and will be seen by the outer transaction.

- *Closed nesting* is the most intuitive approach: Changes seen in an inner transaction must be committed to become visible to the outer transaction, and only become visible to the "outside world" once the outermost transaction commits. If the innermost transaction commits, the semantics are the same as for flattened transactions. The difference occurs if the user aborts *only* the innermost transaction: in that case, the outer transaction is able to continue unaffected.

- *Open nesting* allows the committed changes of a nested transaction to be visible immediately and *globally*, even before the outer transaction has committed. This model can enable certain optimizations, but may also result in violations of the ACI properties of transactions. It is also significantly more complicated to implement.

Since flattened nesting does not work as expected if the user wishes to abort only the inner-most transaction, a closed nesting model should be implemented.

3.1.7 N-ary transaction parameters and return values

Transactions cannot be considered in isolation. As indicated in Section 3.1.4, they are going to be part of a larger, nontransactional program, as tools used to ensure thread safety and synchronization for the specific operations that require it without introducing overhead into the nontransactional parts of the program. Therefore, it must be possible to transfer some data between the transactional and nontransactional parts of the code. For this reason, a transaction must be able to take at least one, but preferably an arbitrary number of parameters, just like ordinary functions do.

Transactions must also be able to return data to the surrounding application in some way: A transaction attempting to read data from a shared data structure must be informed of what data to try to read, and return that data in order to be useful. However, return values may not be necessary in every case. For example, an operation that cannot fail, and writes data rather than reading it, may not need to return anything, and so having to specify a return type and -value would be tedious and unnecessary.

We should also consider that some transactions may be required to return large sets of data, rather than individual objects. Since side effects cannot occur inside a transaction, this may become an important operation. Consider a logging system, or perhaps a graphical user interface which must, at some specified interval, display a consistent snapshot of some shared data structure. Because displaying this data is a side effect, it cannot occur inside the transaction, but we also cannot return each member of the data structure individually, as we would then lose the consistency guarantee that the snapshot *as a whole* mirrors the application state at a specific time.

To solve this, some mechanism should be supported for returning an arbitrary number of values of differing types. In normal code, we would typically achieve this by defining a function which takes a number of references or pointers as its parameters, which can then be modified to point to the additional "return values". A transaction could do something similar, although with a small twist: These parameters should not be modified *unless* or *until* the transaction commits. So the simple approach showed here will not work:

```
void myTx(outParam) {
  atomic {
    val v = ...
    outParam = v;
    ...
  }
}
```

If the transaction aborts after assigning to `outParam`, this assignment will still be visible to the calling code, which violates isolation and atomicity. Instead, the STM system must provide a mechanism for specifying assignments to perform *if and only if* the transaction commits. In pseudocode, this could look like the following:

```
void myTx(outParam) {
  atomic {
    val v = ...
    snapshot(outParam, v);
  ...
  }
}
```

Here, the `snapshot()` function can only be called during a transaction. This would register the assignment of `v` to `outParam` to occur atomically if and when the transaction commits. While this *Snapshot* capability could be considered nonessential, I believe it would prove

very useful in practice, as communication between transactional and nontransactional code must be painless for STM systems to gain popularity.

3.1.8 Reversible side effects

Some transactions may be difficult to express without allowing at least some side effects: a simple operation such as appending a node to a linked list requires at the very least a memory allocation, which is considered a side effect. So some mechanism for allowing selected side effects within a transaction would be useful. We can loosely divide side effects into three categories, two of which can be supported by an STM system without too much trouble:

- reversible side effects,
- side effects that can be buffered,
- side effects that cannot be reversed or buffered.

The last group cannot be safely expressed in a transaction. However, reversible side effects can be allowed, as long as they are reversed during a rollback, and assuming that either the rest of the system does not see the temporarily applied side effect, or that this slight breach of the isolation property is acceptable. Dynamic memory allocations fall into this category. A transaction that attempts to insert a new node into a linked list must create the new node immediately in order to manipulate it. But the action can be reversed simply by releasing the allocated memory again. So as long as a mechanism is provided for pairing reversible side effects with their inverse operations, and registering these with the STM system, reversible side effects can be performed during a transaction.

Other side effects can be buffered, so that they are not performed until the transaction commits. An example of this may be the opposite of the above example: Removing a node from a linked list involves releasing the memory allocated for the node. This operation can *not* be reversed. Once memory is released, we have no way to request "the same memory block again" from the operating system. However, it can be buffered. We do not need the memory to be released *immediately*, we simply wish it to occur if and when the transaction commits. So, for operations that can be buffered until commit time, a registration mechanism could be provided allowing the programmer to register individual side effects to be carried out when the transaction commits.

3.1.9 Support for detaching transaction metadata from transactional object

In the typical case, we can colocate a transactional object and its metadata as suggested in Section 3.1.1. A wrapper class is developed which holds both metadata and the object that is being transaction-enabled.

However, in some cases, we may wish the metadata to be located elsewhere. For example, we may be given an array of objects which should all be transaction-enabled for some concurrent operations to take place, and afterwards, the resulting array of objects should once again become non-transactional.

One way of doing this could be through a `detached_shared` class, which, instead of owning the object it protects, contains a pointer to the object to be protected. Using such a class, an array of objects can be transaction-enabled in-place, and can even revert to becoming non-transactional once the associated `detached_shared` goes out of scope. While an object is associated with a `detached_shared` object, its state is indeterminate if accessed directly, as ongoing transactions might modify the object at any time. Once the

`detached_shared` object associated with an object goes out of scope, the object reverts to becoming non-transactional, and should contain the value assigned to it by the last transaction that committed.

3.2 Properties of an STM system

In the previous section we discussed the requirements a user might place on a STM system in order for it to be considered usable. This section will discuss a number of issues more related to the implementation strategy. This section will discuss how the atomicity required for transaction commits should be implemented and how to detect inconsistencies and avoid "dirty reads" and other such *how* questions. While some of these issues will have an impact on the user, the primary concern is what each choice means for the implementation complexity.

3.2.1 Isolation level

Isolation is a premise for any STM system – without it, transactions would be meaningless. However, different degrees of isolation can be provided. While isolation must always be enforced between transactions, it is less clear to which degree it should apply between transactions and nontransactional code. *Weak isolation* provides only the basic guarantee of isolation between transactions, and leaves the semantics undefined for transactions and nontransactional code accessing the same data. In those cases, race conditions may occur, and both threads may see inconsistent views of the application state. So in a weakly isolated STM system, the burden is on the programmer to ensure that transactional data is never accessed outside a transaction.

Strong isolation replaces these undefined semantics with a simple guarantee that transactions must be isolated *even from nontransactional code*. However, enforcing this is almost impossible in a language such as C++ — unless every memory location is monitored for modifications, we have no way to track which objects are accessed at any given time in the execution of an application. Transactional objects can be given specific types, as suggested in Section 3.1.1, which can only be opened within a transaction, ensuring that transactional data is not *unintentionally* accessed outside a transaction. But even so, we can never prevent aliasing: A transaction may open such a transactional object, and pass a pointer to it to the outside world. Since strong isolation is impossible to achieve in C++ without drastically modifying the compiler, and accepting a significant overhead, I have chosen to implement weak isolation. However, while the system cannot offer any hard guarantees about the isolation of transactions from nontransactional code, we can and should make it harder to accidentally violate this isolation. In order to do so, we must consider communication in both directions:

- Nontransactional code attempting to access transactional data can be blocked as described above, by wrapping all transactional objects in a special type that can only be "opened" if a transaction is passed to them. While this is not fool-proof, it would prevent isolation from being accidentally compromised, at least, and require a conscious decision from the programmer.
- Transactional code attempting to access nontransactional data is harder to prevent, as nontransactional data consists of regular C++ data types, with no mechanism to verify that no transaction is ongoing. Scoping rules can be used to prevent local data from being visible to a transaction if the transaction executes in its own function (see Section 3.1.2), but global objects can always be accessed. Further, for transactions to be useful, it must be possible to parametrize them, and these parameters typically come from nontransactional code. So while there is no general method to prevent

> transactions from seeing nontransactional data, a small amount of discipline should keep the problem in check: the only nontransactional data visible to a transaction is whichever global variables are visible, as well as the parameters explicitly passed to the transaction.

Since we can mostly prevent unintended sharing of data between transactional and nontransactional code, I consider weak isolation to be sufficient for my STM system.

3.2.2 Transaction granularity

One of the most basic decisions for an STM system is the granularity at which changes are tracked during a transaction. In a hardware-based TM system, no type information is available, so it makes sense to track fixed-size blocks, either at the level of individual bytes, or larger units such as cache lines or memory pages. While tracking individual bytes would reduce the risk of conflicts, larger blocks reduce the number of objects to be tracked by the STM system, lowering overhead.

Some early software implementations followed a similar strategy, but without significant support from both compiler, hardware and runtime library, this strategy is impossible to implement efficiently. A more common approach in software TM systems is to use the objects defined by the type system: A STM system operating on an `int` should not see a sequence of four bytes, but a single object of type `int`. This gives us fewer objects to track modifications on which should aid performance of several key operations, and allows us to reuse operations already defined for the type, such as its copy constructor and assignment operator.

Even if there was an efficient method for tracking accesses to each byte in memory, and redirecting as needed to either the uncommitted modified version, or the canonical one with only committed changes, the C++ language standard would simply not permit treating memory blindly as a sequence of bytes.

So for a library-only STM system implemented in C++, object-based granularity is the only option that can work.

3.2.3 Contention management and validation

Different STM systems have explored a number of different ways to manage contention for shared resources. When a conflict occurs, whether it is a read-write or write-write conflict, one of the conflicting transactions must be aborted. Scherer and Scott [12] have explored a number of different *contention managers* for resolving these conflicts. However, one premise for such managers to work is that the conflicting transactions can be identified. If a transaction attempts to commit modifications to an object and finds that it is unable to acquire the object, it must be able to determine who is holding the lock on the object. This strategy of using *visible readers* means that whenever an object is opened by a transaction, some data structure must be updated to point to the transaction that opened the object. This is potentially expensive, especially as it must be done every time an object is opened. Other STM systems have used *invisible readers*, where there is no way to determine which transactions have opened a given object. This saves a possibly significant amount of overhead, but also hinders the ability to apply different conflict resolution strategies. When a transaction encounters a conflict, it cannot request the other transaction involved in the conflict to abort, and must be aborted itself. Such a naive approach to conflict resolution may potentially cause some degree of starvation, as long-running transactions may prevent newer, shorter transactions from committing.

Assuming conflicting transactions can be identified, transactions can perform *invalidation*, where, rather than validating their own data-set when attempting to commit, they instead

mark all conflicting transactions as *invalid*. Gottschlich and Connors [5] describe some significant benefits to this approach, but also note that a validation strategy may be more efficient than invalidation for large numbers of threads.

If it is deemed worthwhile, a transaction *could* be made to iterate over the set of all active transactions, scanning the data-set of each to determine which transactions have the object open, and depending on some conflict management strategy, either invalidate the other transaction, or abort itself. However, for this initial implementation, the considerable complexity this would add, and the questionable performance benefits, means that we will stick to the naive validation-based approach where a transaction is responsible for validating its own data-set, and must abort itself if validation fails.

While validation could be implemented simply by testing for equality (if a comparison of a transaction-local copy of an object and its canonical version yields true, the transaction-local object is assumed to be up to date and valid), this approach is both inefficient (comparisons may be costly on user-defined types), and may yield false positives. For example it is vulnerable to the *ABA* problem common in concurrency: the canonical object (starting with value A) may have been updated to a different value (B), and then updated *again*, bringing it back to the original value (A). The comparison will yield true, despite the transaction-local object being a copy of an outdated version, that just so happened to have the same value as the most up-to-date one.

A better approach is to use versioning: associated with each shared object is a version counter, recording a timestamp for the last modification of the object. When a transaction starts, it records the current timestamp as its *read version*, and every shared object it opens is compared against this. If the shared object has a version higher than the transaction's read version, it has been modified since the transaction started, and should not be seen by the transaction which must then restart and receive a new read version.

When a transaction prepares to commit, it records the most recent timestamp as its *commit version*. The version fields of all modified objects are then updated with this commit version if they are still valid. For this scheme to work, it is important that during the commit phase of a transaction t_0, the modified objects are not accessed by another transaction t_1 with higher read versions than the committing transactions commit version. If this happened, t_1 would be unable to distinguish between objects already committed by t_0 and objects whose updates had not yet been performed, since in both cases, the object's version would be lower than t_1's read version and it would be considered valid and consistent.

3.2.4 Ensuring consistency

As described in Section 2.1.2, the C++ language relies on the user to avoid *undefined behavior*. For the STM system to be useful, it must not accidentally exhibit undefined behavior in the absence of user errors, or present behavior that may lead the users code into undefined behavior. Some STM systems have allowed transactions to see inconsistent views of data, which could lead a transaction to reading out of bounds or perhaps enter an infinite loop or otherwise behave enter erroneous states that cannot be recovered from. Accessing memory out of bounds invokes undefined behavior, at which point we have no guarantees of the consistency of other memory in our application, and while an infinite loop could possibly be interrupted by aborting the thread in which it is executing or provoking it into producing an error that will abort the loop, such tricks would also risk aborting the thread halfway through some operation, again leaving the application in an inconsistent state.

So to ensure correctness, we must guarantee that transactions never see any inconsistent state to begin with. This constrains ou implementation somewhat: On opening an object, we must immediately verify its validity, and we must then ensure that the STM system does not update the object while a transaction has it open. If objects are modified in-place, this

restriction means that only one transaction may have any given object open at a time. If modifications occur in transaction-local copies, but overwrite the canonical object when committed, then no commits can occur while other transactions have the object opened.

In managed languages, or given support from the compiler, it may be possible to recover from transactions seeing inconsistent data, and so it may be worthwhile to delay validation until commit-time. But as mentioned, a library-only implementation such as this does not have that luxury: data must be validated before the transaction is granted access.

We must guarantee that once a transaction has opened an object for reading or writing, the version of the object seen by the transaction will not be modified until the transaction commits. Additionally, when the object is first opened, the transaction must verify that the version seen is consistent with all other objects opened by the transaction. This does not necessarily mean that *all* conflicts must be detected immediately, however. If multiple versions can exist of an object, so that the version seen by a transaction is not necessarily the most recent one, then it becomes possible for a transaction to proceed, operating on consistent, but outdated data. Of course, the transaction cannot safely commit in this case, but until it tries to commit, it will behave consistently and predictably, so delaying at least some conflict detection until commit-time is possible.

An argument in favor of delaying conflict detection when possible is that validating the consistency of the entire transaction data set is an $O(n)$ operation. We must verify that every object opened by the transaction is still valid. If this is done every time an object is opened, for example, the total cost of validation throughout a transaction is $O(n^2)$. While allowing an explicit `validate()` function that the user can call may be useful, in order to restart invalid transactions before some long-running operation, the most general solution may simply be to postpone validation until commit-time.

3.2.5 Update strategies

It is clear that to provide transaction semantics on modifying objects, we must either forbid concurrency entirely so that conflicts never occur and rollbacks will never be necessary, or have some kind of copying mechanism in place to ensure that uncommitted modifications can exist along with the original, unmodified values, so that whether a transaction commits or aborts, it can ensure the canonical version of the object is in the correct state when the transaction ends. The first, somewhat pathological, option is obviously not desirable — concurrency was the entire motivation for STM in the first place — so some strategy for copying and updating objects must be defined.

The possible update strategies can be roughly categorized along two axes: direct versus indirect updates, and relying on indirection versus updating in-place

Direct update

A direct update strategy, sometimes described as *undo logging* since it maintains a log used to undo speculative changes, means that the canonical version of an object is modified directly during a transaction. To support rollbacks, a transaction-local copy is made, which contains the original value of the object, and in the event of a rollback, this is used to overwrite the canonical object, bringing it back to the initial state. In a naive implementation, this would make uncommitted modifications visible to other transactions, violating the isolation property. However, some systems allow this, assuming that the error can be detected at commit-time, and transactions that have seen inconsistent state can be aborted. Other systems use locking to prevent other transactions from accessing an object while it is being modified, which preserves isolation, but limits concurrency. However, a key advantage of a direct update strategy is that commits become practically free: all modifications have already

been made directly to the canonical versions of each object, so on commit, a transaction simply has to validate that no inconsistent reads have occurred, and then update the requisite metadata. A rollback, on the other hand, requires every modified object to be reverted to its initial state by overwriting them with private backups created before the objects were modified. This strategy is sometimes described as *redo logging*, since the transaction-local copies containing speculative changes constitute a log that must be applied to the canonical objects in order for the modifications to take effect at commit.

Deferred update

Using a deferred update strategy, transactions apply speculative modifications to private copies of data only, and defer updates of the canonical objects until commit time. This helps ensure that the isolation property is preserved, as the objects seen by other transactions are only modified briefly when a transaction commits its modifications. Of course, to avoid inconsistencies, the commit phase must still be protected somehow to ensure other transactions do not see the canonical object being updated.

Some implementations only create the transaction-local copy if an object is opened for writing, while read-only accesses are done directly on the canonical object. In that case, we must further ensure that no currently running transactions have accessed the object, as inconsistency would then be introduced into those other transactions.

This strategy is also called *redo logging*, since the transaction-local copies of objects containing speculative changes constitute a log to be applied, or "re-done" on the canonical objects when the transaction commits.

Compared to direct updating, this strategy makes commits more costly: both strategies require private copies of objects to be made before they are modified, but in a deferred update strategy, this copy must be used to overwrite the canonical object at commit-time as well, resulting in a total of two copies per modification. However, rollbacks become less expensive, as the canonical objects are not modified until commit-time, and so the transaction-local copies can simply be discarded without the canonical objects ever being affected.

In-place update

In-place updating describes an approach where there is always *one* canonical object stored in one known location. In order to commit modifications, this object must be overwritten with the newly modified one. While this allows for potentially better locality and cache behavior, it also makes it harder to avoid inconsistencies: the same canonical object is known to all transactions, and so while it is being modified, whether that happens gradually throughout a transaction as in direct updating or as part of a discrete *commit* stage at the end of transactions, we must ensure that no other transaction has the object opened for reading *or* writing.

Indirection-based update

To avoid the inconsistency problems with updating in-place, some implementations such as ESTM, have tried an indirection-based approach. Here, rather than there being a single canonical object that must be updated at every commit, only a canonical pointer exists, which points to the instance that is currently the most up-to-date version of the object. Under this approach, every transaction can create a private copy of each object it modifies, and at commit-time simply set the pointer to point to this copy, without overwriting or touching the "old" object. This ensures that other transactions which already have the old version opened will not be affected by the modification: they will continue to see the same old instance,

avoiding inconsistencies.

The problem with this approach is mainly one of performance: pointer indirection is not free, and as objects must be dynamically allocated in order to outlive the transactions creating them, they will exhibit poor locality.

A hybrid approach

It is clear that each of these approaches have advantages: The indirection-based approach neatly sidesteps all the problems of ensuring consistency, as object are simply never modified once they have been committed. Direct in-place updating enables relatively cheap commit operations, at a cost of only one copy per modified object, but limits concurrency. Deferred in-place updating allows transactions to read an object while another transaction is modifying it, enabling more concurrency than direct updating, but makes commits costly at two copies per object.

Ideally, we would like to design a strategy that exploits the advantages from each of these approaches. In order to enable transactions to read from an object while another transaction is modifying it, we must take a leaf from the indirection-based strategy: we do not need an unbounded number of locations in which the canonical version of objects may be stored, which is what destroys locality for indirection-based approaches, but we do need at least two "official" slots into which modifications can be stored: one containing the current canonical object, and another into which modifications can be written during a commit. Since we have a fixed number of update "slots" per object, we can ensure that these are allocated to get maximal locality, eliminating the key disadvantage of indirection-based systems.

When a modification is committed into the backing slot, a flag can be flipped, swapping the roles of the two slots, so that the previously active or canonical slot is now the backing one into which modifications may be written, and the slot into which the last update was written is now the canonical one. Of course, when the active slot is turned into the backing one, we may have lingering readers still accessing the object written there, and until these have terminated, further updates cannot be permitted.

In a direct-update strategy, we may still get a lot of contention, as modifications to this "update-only" slot may happen throughout a transaction, so we would be limited to allowing only one transaction to open each object for writing. In a deferred-update strategy, however, modifications are only applied at commit-time, so we can relax this requirement, saying that only one transaction may *commit* to each object at a time — a requirement that would have to be satisfied in any case in order to preserve atomicity. This effectively rules out the direct update strategy, but we would still like to borrow its one key advantage: that a committing transaction only requires one copy operation per modified object. In order to achieve this, we can exploit C++0x *move semantics*. Unlike copy operations, a move may be destructive, meaning that it modifies the source object. This is not acceptable when we are first creating the private copy — the canonical object we are copying from must not be affected — but when applying our modifications, the private copy is no longer needed can be modified by a destructive move operation. The commit-time copy can therefore be converted into a *move* operation, effectively eliminating the second copy operation on types where copying would normally be expensive. This hybrid approach is described in detail in Section 3.3.

3.2.6 Commit strategy

Committing can be done using a two-phase locking strategy: rather than locking one object at a time and immediately modifying it, which would cause problems if a later object cannot be locked, forcing the entire transaction to roll back, all locks must be acquired first. When this has been done successfully, all modifications can be applied.

Objects opened for reading only do not need to be locked during commit: instead, at the beginning of the commit phase, the transaction must retrieve its *commit version*, the version that will be assigned to all modified objects on a successful commit, and which indicates the atomic time at which the entire commit operation takes place. The system must ensure that all transactions given lower commit versions are fully committed before the transaction is allowed to proceed. Once this version has been determined, each read-only object must be scanned to ensure that it still has a valid version — the object's version must be lower than or equal to the transaction's initial read version. If this scan succeeds on the entire read-set, we know that while the actual transaction was being executed, the entire read-set was valid. If the write-set, once locked, is also still valid, we know that no transaction occurring between the transaction's initial read version and its commit version has modified any object opened by the committing transaction, and so it can safely commit.

3.2.7 Starvation, deadlocks and livelocks

While this is not really a question of implementation policy — deadlocks and livelocks are obviously unacceptable — we must nevertheless ensure that these cannot occur.

Deadlocks are almost trivial to avoid in a STM system: If a transaction fails to lock an object it wishes to modify, it rolls back, releasing any previously acquired locks. This ensures that a deadlock can never occur, but creates the potential for *livelocks* — two transactions could in principle keep trying to acquire the same set of resources, each time acquiring some, but failing to acquire others, so that both are forced to roll back and start over, repeating the same scenario. In order to prevent this scenario, we can ensure that objects are acquired in a globally consistent order. If all transactions follow the same order when acquiring objects, no cycles can occur where multiple transactions block each others by each holding an object needed by the other. In normal lock-based programming, this is difficult to achieve, as locks are typically taken on the fly, when a specific object is needed and so the order in which they are taken is implicitly defined by the order in which the objects they protect are needed. In a STM system, however, locking can be deferred until commit-time[6], at which point the full set of modified objects is known, and so these can be acquired in any order, allowing the system to impose a consistent global ordering, for example based on the address in memory of each object.

However, we still have to consider starvation: in a STM system employing an *in-place* update strategy, we will see some amount of contention between readers and writers. Writers must never commit changes to an object while readers have access to it. This may lead to writers getting starved, being forced to constantly roll back and retry, every time failing to commit because other transactions are reading the objects that the writer is attempting to commit changes to.

A fully indirection-based system avoids this problem by never modifying already committed objects, however the situation is less clear cut in the hybrid scheme sketched out in Section 3.2.5. We know that "lingering readers" may be a problem — assuming that an object's slot 0 is initially the canonical one, several transactions may have it open for reading when another transaction commits an update to slot 1. The roles of the two slots are now swapped, so that slot 1 is the canonical one that new readers must open, and slot 0 is the one into which the next update should be written, but the reading transactions may still have the object in slot 0 open, and so a second commit cannot be permitted yet. However, while this may limit the frequency of commits in a heavily contended scenario with both readers and writers, it is impossible for writers to be starved out completely. Only a finite number of readers can exist on a slot when that slot is switched from a canonical to a backing role. And from this point onwards, no new readers can be added: all new readers will access the new canonical slot. This means that assuming no reading transaction enters an infinite loop, all readers will

[6] At least this is the case for deferred-update systems, as described in Section 3.2.5.

Transaction: action	slot 0	slot 1
tx_0: r	a:1 (tx_0)	b:0
tx_1: c_0	a:1 (tx_0)	b:1 (tx_1)
tx_1: c_1	b:1 (tx_0)	a:0
tx_2: r	b:1 (tx_0)	a:1 (tx_2)

Fig. 3.1: Reader counts for subsequent accesses to the two slots of a shared object. Transaction 0 first registers as a reader (r) on the currently active slot (a:1). Transaction 1 then initiates a commit (c_0) by registering on the backing slot (b:1). Transaction 1 completes its commit by swapping the roles of the two slots and at the same time unregisters itself on the slot that is now active. Transaction 2 now registers as a reader by incrementing the active slot. This shows that no new readers can be added to the backing slot, and so, once transaction 0 terminates, its reader count will reach zero, and the next transaction can commit.

eventually terminate, and so a writer will be allowed to proceed with the next commit. This is illustrated in table 3.1.

By placing some limit on the rate of successive commits in scenarios with heavy contention we may also reduce the risk of reader starvation, as readers could otherwise be forced to constantly restart as the objects they access could get updated before the reader transaction had a chance to commit. Detailed benchmarking would be necessary to determine the performance implications of this strategy, but it does at least guarantee that neither readers nor writers are *completely* starved: both types of accesses will be allowed to succeed at regular intervals.

3.3 Design of the DikuSTM library

In the previous parts of this chapter, we have sketched out a number of requirements and a possible implementation strategy. This section will describe in detail the strategy chosen for the DikuSTM library.

As suggested in Section 3.1.2, transactional objects are represented through nonintrusive composition: A class template `shared<T>` is defined which stores the object of type `T` that should be given transaction semantics. But instead of storing a single instance of this object, the `shared` template contains two "slots" into which different versions of the object can be stored. At any given time, one slot is given the *active* role, meaning that it contains the canonical version of the stored object, while the other slot is the *backing* slot. For convenience, the two roles can be abbreviated as slots *A* and *B*. New transactions opening the object should always get a reference to slot A, while committing transactions should store their modifications into slot B. After a successful commit, the two slots switch roles, a double-buffering strategy as is often used in realtime graphics.

The rest of the `shared` object is taken up by metadata required by the STM system, and consists of the following components:

- A version field describes the version number of the `shared` object. Each successful commit updates the version number, and when opening the object, the version number is compared against the transaction's version. If the object has a higher version, it has been modified since the transaction started, and so the transaction must restart.
- A flag specifying the roles of the two slots. This can be stored in a single bit: if it is set,

then slot 1 is active and slot 0 is the backing slot, and if not, the roles are reversed.

- A read counter for each slot: This is necessary to prevent commits to the backing slot while readers are still accessing it. Any time a transaction opens an object, it must increment this counter, and when the transaction terminates, it must be decremented again. A commit can only take place when the read counter for the backing slot is zero.

It is worth noting that no explicit locking mechanism exists for the metadata fields: for a transaction to safely inspect the version number, it must prevent commits from occurring, which can only be done by incrementing the read counter of the backing slot. As long as readers exist on the backing slot, no commits can occur, effectively freezing the version counter and the flag. Further, to ensure that only one commit can occur at any given time, we introduce a rule that transaction may only commit if the backing slot's read counter was zero when the transaction incremented it. If the counter was already nonzero, then the object is either locked for another transaction to commit, or lingering readers still exist from when the current backing slot was active. Since these counters are used as the only mechanism for "locking" an object, access to them must be atomic. This includes both read accesses and the increment/decrements used to update them.

In this approach, readers are not directly visible, so there is no way for a transaction to see who else has a given object opened — only the read counters are visible, and these do not identify readers, they only indicate how many readers currently exist. This limits our options for conflict resolution somewhat. When a transaction encounters a conflict, we have no simple way to identify the other transactions causing the conflict and force those to abort. This gives us a simple LIFO order, where the last transaction to attempt to acquire an object is aborted when a conflict occurs. This simple conflict resolution strategy is not ideal, but the alternative of relying on visible writers also carries significant overhead [3].

As mentioned in Section 3.2.7, we are guaranteed that neither readers nor writers get starved *entirely*, which lessens the need for a more complex conflict resolution strategy.

This layout of the `shared` objects means that both slots are located *in-place*, enhancing locality. Further, since transaction-local copies of objects are transient and must be deleted when the transaction ends, these can be placed into a single contiguous transaction buffer as well so that all objects opened for a transaction are placed sequentially, also ensuring good locality. A simple indirection-based approach would have placed each object copy at a pseudo-random location determined by the call to the `new` operator.

Of course, such a contiguous buffer can only have a fixed size and some mechanism must exist to acquire more buffer space for large transactions. Since transaction-local objects cannot be moved (as the transaction code contains references to them), a `vector`-like data structure is out of the question. A `deque` seems a promising alternative, exhibiting good locality without the need to relocate its contents, but another problem prevents this data structure from being used: the objects copied into the transaction buffer have variable, and unbounded, size, and we have no guarantee that each page of the `deque` will be large enough to store very large objects. This means that the only real option for a resizable buffer is to rely on indirection: each object copy must be dynamically allocated, and a single pointer stored in the transaction buffer. A good hybrid implementation could therefore be to attach a single fixed-size buffer, in the form of a simple array, to handle all reasonable-sized transactions. If this buffer runs out of space, either because a transaction attempts to allocate very large objects, or because too many objects are opened, the transaction can fall back to a linked list implementation, dynamically allocating storage for each new object added. In the vast majority of cases, this backing structure should not be necessary and the high-performance fixed-size buffer is sufficient, but without it, large transactions may fail simply because the buffer runs out of space.

4. THE DIKUSTM LIBRARY

In Section 3 we discussed the high-level strategy for the library, but left many implementation details unspecified. In this chapter, the DikuSTM implementation is described in detail. First, we will describe the high-level structure of the DikuSTM library in Section 4.1, giving the reader an overview of the architecture of the system, and the responsibilities of each component. In Section 4.2, we will walk through the operations performed by a transaction throughout its lifetime, from when the transaction is created, through the execution of the user-defined transaction function, and ending with either a commit or a rollback.

We will then examine the most important components in further detail. Section 4.10 and Section 4.11 contain a precise definition of the interface exposed by the library. This is intended both for users of the library, and as a contract for alternative STM implementations to follow — as long as the described interface is supported, the entire STM implementation can be swapped out for an alternative library, with no changes to user code.

Finally, a number of limitations of the current version of the library are listed in Section 4.12. These are either features that have not yet been implemented, or dependencies on external libraries or hardware requirements that could be eliminated in future versions.

4.1 High level library design

The DikuSTM library is divided into three tiers or layers, with distinct roles:

- the *user* tier contains classes and functions intended to be seen or used directly by the library user, but contains none of the actual transactional logic or infrastructure,
- the *frontend* tier contains the first half of the STM system, and handles operations initiated by the user. The defining characteristic of all frontend components is that type information is available for the values being manipulated. As such, this tier is responsible for generating the metadata necessary for the backend to operate correctly without type information,
- the responsibility of the *backend* is to maintain the transaction-local buffers, and, when a transaction commits or otherwise terminates, updating the canonical objects with the values stored in the local buffer. Since these buffers must store arbitrary types of objects, they are not inherently type-safe, and store objects simply as byte sequences. Because type information is not available, the backend relies on metadata explicitly generated by the frontend in order to carry out type-specific operations such as object destruction or assignment.

These layers are designed so that the outer layers only have dependencies on the inner one, in order to keep a clean separation between the components.

In the following, an overview of the main components of each tier is given. For larger and more complex components, a detailed description is provided in a later section.

4.1.1 The User tier

This component consists only of the classes and functions described in Section 4.11, and require little additional explanation. As classes in this tier just make up the interface to the rest of the system, this tier is largely trivial to understand:

- the `shared`, `shared_detached` and `transaction` class templates are simple forwarding wrappers for associated `shared_internal`, `shared_detached_internal` and `transaction_internal` classes defined in the frontend. Their main purpose is to present a simpler interface than the frontend equivalents, hiding all functionality that is intended for internal use by the STM system,
- the *orelse* functionality is implemented entirely in the user tier, using a simple wrapper template for storing the two transactions to be alternated between, and providing the same interface as user-defined transaction functions, by defining an `operator()(stm::transaction&)`. The only complication is that in order for the `atomic` function to know that it is executing within an `orelse` (in which case the exception thrown by `retry` must escape the executing transaction instead of forcing it to block and retry), `atomic` is passed a second optional parameter.

4.1.2 The STM frontend

The frontend provides the ability to open transactional objects within a transaction and controls the lifetime of a transaction, by initializing transactions and initiating commit or abort operations.

The main components in the frontend are the *transaction manager* and the *internal* classes mirroring the user-tier *shared* and *transaction* classes. These frontend counterparts provide the actual implementation of their user-tier wrappers. The role of the transaction manager is to provide the primitive operations needed to implement higher level STM functionality, without enforcing any specific policy. For example, the transaction manager provides functionality for copying an object into the transaction-local buffer, searching for an object in the buffer, or validating the objects in the buffer. However, it does not define complex operations such as *commit*.

Complex operations are defined by the transaction and *shared* classes,using the operations enabled by the transaction manager. For example, when a transaction commits, the `transaction_internal` class retrieves a list of objects stored in the local buffer from the transaction manager, sorts it, and then locks, validates and updates these objects through the transaction manager. Thus, the *policy* of how a commit operation should be performed is defined by the transaction class, simplifying the transaction manager, which only has to define the building blocks for larger operations, with no concern for how and when they are called.

The *shared* classes used to represent transactional objects are part of a class hierarchy spanning all three tiers. In the user tier, the user-facing class `shared` simply forwards all member functions to the frontend-tier `shared_internal`, which provides access to the two internal object slots. Replacing this class with `shared_detached_internal` redefines these functions to access either the one internal slot, or the external object attached to the *shared* object. The remaining frontend functionality, which is common for all *shared* implementations, is defined in `shared_internal_common` from which `shared_internal` inherits. The CRTP[1] pattern is used to enable `shared_internal_common` to access the

[1] Curiously Recurring Template Pattern: A pattern in which a derived class is used to specialize its base class, as in `Derived : Base<Derived>`. This enables a form of compile-time polymorphism in which the base class can invoke operations defined by the derived class.

object slots exposed by `shared_internal`. Finally `shared_internal_common` derives from the backend class `shared_base`, described in the following. The entire *shared* class hierarchy is also described in figure 4.1.

The transaction manager is tied to the executing thread, and has static storage duration, so that it does not get destroyed between transactions. Since the user may in some cases have several disjoint "groups" of transactions executing, the version counter is placed in a *transaction group* class. A default instance is globally accessible and is used by default when a transaction is started, but other transaction groups can be created, each with its own version counter. In this case, new transaction managers are created as well, giving each thread unique buffers and possibly a different configuration of template parameters for each group of transactions. But in the common case, only one transaction group exists, and each thread has just one transaction manager. Th transaction group has two responsibilities: controlling access to the version counter and retrieving the transaction manager associated with the active thread.

A final set of functions defined in the frontend are responsible for encoding the type-dependant operations for use in the backend. These are functions intended to be invoked directly on an offset in the untyped transaction buffer, and so are referred to as *buffer functions*. When a transaction commits, every object being committed must have its assignment operator invoked in order to be copied out into the canonical locations. This is done using the *assign* function. When an object is evicted from the buffer, its destructor must be called, which is done through the buffer function *destroy*. The implementation of these functions is described in Section 4.6. These functions bridge the gap between frontend and backend: because the functions are strongly typed and operate on objects using their actual types, they are defined in the frontend, but they are intended to be passed to the backend where they are stored as metadata along with the object stored as an untyped byte sequence, on which they are meant to be called.

4.1.3 The STM backend

The backend contains only the few types that have to work without type information: the buffer intself, iterators for traversing the buffer, the *shared* base class containing only the STM metadata with no knowledge of the actual objects stored by derived classes, and a number of small helper functions for manipulating the metadata stored with each object.

A number of requirements constrain the implementation of the transaction-local buffer.

- The buffer must be dynamically sized and resizable, as the amount of transaction-local data stored in a transaction is unknown at compile-time, which rules out at least a simple array implementation.
- The buffer must enable arbitrarily large contiguous allocations. A data structure such as the standard libary `std::deque` cannot work, as it provides fixed-size "pages" contiguously, but allocates new pages to extend the buffer. A single large object allocated in the buffer may overflow such a page.
- Objects placed in the buffer must never be moved, since the user's transaction code contains references to it, so a dynamic array such as `std::vector` is not suitable.
- Performance should not suffer. Access should be efficient, and objects stored in the buffer should exhibit good locality. This makes a linked list implementation like `std::list` problematic.

No perfect data structure exists which satisfies all our requirements. If a single data structure is to be used, a linked list is an acceptable solution, but the performance characteristics are not

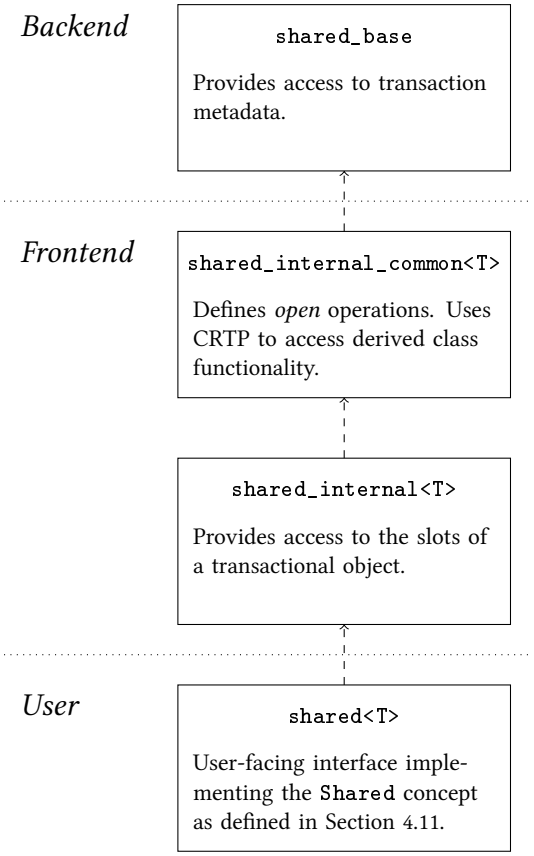

Fig. 4.1: The hierarchy of classes implementing transactional objects. Not shown here are the two classes implementing *detached shared* objects, replacing the bottom two classes to provide an implementation that does not own the protected object.

ideal. Instead, a pair of buffers can be used: most transactions are expected to be small, and so a fixed-size array can be used until it fills up, giving us optimal locality and performance, even avoiding dynamic memory allocations. However, if this array does overflow, a linked list of pointers to dynamically allocated objects can be used as a fallback.

In the current implementation, the linked list buffer has not been implemented, and only an array buffer exists. It is set to a default size of 64KB, which is enough to satisfy almost all transactions, but for a truly general STM system, a resizable secondary buffer must be implemented.

The backend also contains the base class of the *shared* hierarchy, `shared_base`, which, unlike its derived classes, is not templated and has no knowledge of the type of object being protected. Instead, its role is solely to provide access to the metadata associated with the object: the object's version and reader counters.

4.1.4 Utility code

Outside the three tiers defined so far is a small *utility* component, which is used by all three tiers, and contains no dependencies on other parts of the library. This component contains configuration files, and provides the necessary abstractions for the library to work on both Windows and POSIX platforms and be compiled for C++03 or C++0x, using either the Microsoft Visual C++ compiler or GCC.

The component also defines the exception classes used to signal events such as validation errors or requests to retry or abort transactions.

4.2 The transaction lifecycle

Transactions go through several distinct phases, as they execute, from starting up, to executing the user-supplied transaction function, to committing or rolling back. These are described in the following.

4.2.1 Initialization

When the transaction is created, it retrieves the current version from the global version counter and stores this as its *read version*. If the transaction is nested inside another, it is given the parent transaction's version — since the inner transaction must atomically commit as part of the outer transaction, it must use the same transaction version as well. The transaction is also passed information about the number of times it has been restarted so far, and whether it is part of an *orelse* statement. The transaction manager is used to store state that should be remembered across transactions, such as if an outer transaction was part of an *orelse*, since this affects the semantics for `retry()` calls, even in inner transactions.

Finally, iterators pointing to the last read-only and read-write records inserted in the buffer is recorded. This is used to delimit the range of objects inserted by the current transaction from those inserted by parent transactions.

4.2.2 The live phase

Once the transaction has been initialized, it enters the user-supplied transaction function. As long as this is executing, the transaction is said to be in the *live* phase.

In this phase, mainly user-defined code is executed. The only STM operation available, other than those that terminate the transaction in one way or another, is that of opening

transactional objects.

The *open* operations are implemented in a fairly straightforward manner. If the object is opened for reading, the *shared* object containing the object being opened must have its reader counter incremented. If the object is opened for writing, that object must instead be copied into the transaction-local buffer.

In both cases, the reader counter for the currently active slot is initially incremented, and the object's version is verified to be less than or equal to the transaction's read version. If the object is opened for writing, this increment is to ensure that the object does not get modified while we are copying it, and the counter is decremented again when the copy as been created.

Whether the object is opened for reading or writing, the transaction-local buffer's list of objects opened for writing is now scanned to find existing copies of the same object. From this point, the workflows for objects opened for reading and writings diverge.

For objects opened for reading, the following operations are performed:

- if a copy of the object was found in the buffer, a pointer to this is simply returned,
- if no copy was found, a new pointer to the object being opened is pushed onto the buffer's list of opened read-only objects, and a pointer to the canonical object is opened.

For objects opened for writing, the following is done:

- if no copy was found in the buffer, a copy is inserted, and a pointer to this new copy is returned,
- if a copy was found, and it was inserted by a "parent" transaction, one that the current transaction is nested within, this copy cannot be reused directly: closed nesting semantics require that we are able to roll back the inner transaction, leaving the outer transaction in the same state as when the inner transaction started. So in this case, we must create a copy of the copy already found in the buffer, insert that in the buffer, and return a pointed to this *new* copy,
- if a copy was found and it was inerted by the *current* transaction, we can simply return a pointer to this object,
- finally, the canonical object's reader counter is decremented to cancel out the increment performed at the beginning of the open operation, allowing other transactions to lock and modify the object.

The transaction can leave the *live* phase in a number of ways:

- validation of an object may fail, forcing a rollback. This can happen either when the transaction commits, or when an object is opened for reading or writing,
- control may leave the function, either through an exception being thrown, or by reaching the end of the function or a `return` statement. When this happens, the transaction will attempt to commit,
- the user may call `abort()` or `retry()`, forcing the transaction to roll back and, if `retry()` was called, restart again later. An `stm::abort_exception` is thrown from the `atomic` function when the rollback completes if the transaction was aborted.

4.2.3 Rollback

In the case of a rollback, the transaction manager's `release()` function is called on every object opened for reading, to decrement the reader counter indicating that the transaction no longer requires access to the object.

Objects opened for writing do not have their reader counters incremented, and so they do not need to be explicitly released. Instead, we must destroy the transaction-local copies stored in the buffer. The transaction manager's `destroy()` function is used for this purpose, and simply invokes the `destroy` function pointer stored along with each object, as described in Section 4.6.

Once this is done, both the read-only and read-write records can be safely removed from the buffer, which is done with the transaction manager's `pop()` function.

All of the transaction manager functions described here operate on a range of objects, implicitly starting with the last object inserted in the buffer, and ending with an iterator pointed to the last object inserted by a parent transaction.

4.2.4 The commit phase

When control leaves the transaction function, the system attempts to commit the transaction. If the transaction is not nested, changes must be made globally visible, and the following procedure is followed:

1. a *write list* is created, containing references to every object in the transaction's write set, and sorted by the address of the canonical objects they reference. This sorting ensures that all transactions acquire objects in the same order, preventing livelocks.
2. Every object in the write list is locked, by incrementing both of the reader counters associated with the canonical object.[2] If the counter for the backing slot was previously 0, the object has been successfully locked, and no other object will be able to acquire it. If it was nonzero, another transaction has access to the backing slot, and we cannot commit into it, so the entire transaction must abort.
3. If all modified objects were locked successfully, we now lock the version counter and retrieve the transaction's *commit version* which is defined as the current value of the version counter plus one. This is the version that modified objects will be set to when the transaction commits. By locking the version counter, other transactions are prevented from committing as they cannot get a commit version.
4. The entire read set of the transaction is validated, by verifying that all objects have versions less than or equal to the transaction's read version. If the validation succeeds, the transaction can no longer be forced to roll back, and the commit is guaranteed to succeed.
5. If objects have been registered as part of a *snapshot* to be generated at commit (see Section 3.1.7), this snapshot is generated, by performing the assignments registered.
6. Finally, the version counter is unlocked, once all object copies in the buffer have been written out into their canonical locations and their versions are updated and the flag flipped, indicating that the backing slot, which was just updated by the transaction, is now the active slot.

[2] Technically, only the backing slot's reader counter has to be incremented, but as the transaction does not immediately know which slot this is, the simplest and most efficient solution is to simply increment both counters.

If the transaction is nested, there is no need to perform validation, as no changes are being made globally visible. In that case, we only copy objects that were also opened for writing by outer transactions, into their "parent copies" in the buffer. Then the copied objects have their destructors called, and they are removed from the buffer. Objects opened for the first time by the committing transaction are simply left unchanged in the buffer, effectively adding them to the parent transaction's write set.

4.3 Ensuring consistency and conflict serializability

The reader counters associated with each slot in a `shared` object ensure that objects do not get modified while they are accessed by a transaction. The version counter is locked while a transaction applies its changes during a commit operation, ensuring modifications are always applied in order. An update at time t to an object is guaranteed to be visible to transactions with read versions $rv \geq t$. Further, no transaction with a read version $rv = i$ can start until the transaction with commit version $cv = 1$ has terminated.

This ensures that transactions never see inconsistent state: a transaction with read version rv can only see versions of objects whose last modification was at time $t \leq rv$.

This guarantees conflict serializability: A transaction committing with commit version cv is given a unique offset cv in a serial schedule of transactions.

4.4 Starvation and prioritization

The system as implemented now has no mechanism for prioritizing transactions or controlling starvation. As described in Section 3.2.7, objects opened for reading cannot starve out a transaction attempting to commit modifications. However, multiple transactions attempting to commit modifications to the same object may in some cases cause starvation. Since locks are only acquired at the end of the transaction, a long-running transactions t_l is at a disadvantage, as a shorter transaction t_s which modifies the same object x will most likely complete and commit before t_l completes, forcing t_l to rollback and restart. It is impossible to say how significant a problem this is — in real-world code, constant contention for the same objects is unlikely, and we do not know how common longer transactions are going to be. Most likely, code using STM would be structured to get as short transactions as possible.

However, if starvation does become a problem, several modifications can be made:

- one possibility would be to switch a transaction to a pessimistic concurrency model once it has been rolled back a fixed number of times: under this model, objects would be locked immediately when they are opened for writing, preventing other transactions from committing modifications at all until the locking transaction has finished,
- a transaction priority can be encoded into the object's metadata, and consulted before a transaction attempts to lock an object. When an object is opened, a transaction can write its own priority into the object's priority field, and when transactions commit, they check the version field, verifying that no higher-prioritized transaction is using the object.

4.5 The transaction-local buffer

The buffer held by the transaction manager is meant to be replacable and configurable. The current implementation includes only a fixed-size array buffer, which provides good

performance and locality, but may overflow for large transactions. A more complete implementation should either replace this entirely with a resizable buffer, or use a combination of both, filling up the fixed-size buffer first, and silently falling back to the resizable one when an overflow occurs. If both buffer types are implemented, the strategy of using both buffer types can be implemented through a simple wrapper class using the same public interface.

A buffer class must define the following public members:

```
struct buffer {
  typedef implementation-defined iterator_r;
  typedef implementation-defined iterator_rw;

  buffer() throw();
  ~buffer() throw();

  template <typename T>
  T* push_rw(shared_base* src
    , backend::metadata* outer_open
    , T& obj
    , void (*destroy)(backend::metadata*)
    , void (*assign)(const backend::metadata* psrc, void* pdst));
  bool push_r(shared_base* src) throw();

  void release(iterator_r last
    , uint32_t tx_version) throw();

  void destroy(iterator_rw last) throw();

  void pop(iterator_r last) throw();
  void pop(iterator_rw last) throw();

  iterator_r begin_r() throw();
  iterator_r end_r() throw();

  iterator_rw begin_rw() throw();
  iterator_rw end_rw() throw();

  size_t write_count(iterator_rw last = end_rw());
  size_t read_count(iterator_r last = end_r());
};
```

Conceptually, the buffer consists of two stacks, as both read-only and read-write records are stored in LIFO order. `release`, `destroy`, and `pop` implicitly work from the top of the relevant stacks, to the *last* parameter passed to them. The *begin* and *end* functions allow traversal over the read-only and read-write stacks. The *begin* functions returns an iterator to the top of the stack so that traversal from begin to end will iterate in reverse order of insertion.

`push_rw` is called when the user calls `open_rw` on a transactional object, and pushes a copy of the opened object onto the read-write stack. `push_r` is called by `open_r`, and pushes a pointer to the opened object onto the read-only stack. These functions are both able to return failure: this is not because the STM system is expected to be able to handle allocation failures, but because multiple buffer strategies may be combined as described above. If the fixed-size array overflows, `push_r` and `push_rw` can return failure, indicating to the wrapper class that the secondary buffer should be used instead. In general, the rest of the STM system does not assume these functions will return failure.

`push_rw`'s signature is complicated enough to warrant a more detailed explanation. The `src` parameter is a pointer to the source *shared* object. `outer_open` is a pointer to the previously

entered copy of the same object, if a parent transaction already opened the same object, and is set to *null* if the object is not already open. `obj` is a reference to the object from which a copy should be made. This is necessary as `src` only tells us which `shared_base` object owns the object, but not which of the two slots should be copied from, or where the object itself is located. The final two parameters are function pointers for destroying the buffer copy, and for performing an assignment from the buffer copy to the destination when the transaction is committed.

The system is required to track two types of information: pointers to objects opened for reading, and transaction-local copies of objects opened for writing along with their associated metadata generated by the frontend. These could be allocated into separate buffers, but this would prevent us from using allocated memory efficiently. If one of the buffers overflows, an extension data structure must be allocated, while the other buffer is left partially empty. A more efficient approach is to reuse the same buffer for both types of data, so that pointers to read-only objects are placed at one end of the buffer, and transaction-local object copies at the other, both growing towards the middle.

When writing objects into the buffer, we must be aware of the *alignment requirements* for the object. A type may only be allocated on addresses divisible by some integer, typically a power of two. That number is the type's required alignment. On most platforms, built-in types require alignment equal to their size, so that a `char` can be allocated on any address, an `int` only on addresses divisible by four, and a `double` on addresses divisible by eight. Compound types, such as classes and structs require alignment equal to their most-aligned member. This means that objects cannot be written into the buffer at arbitrary offsets: padding must be inserted when necessary, to ensure the object is placed at a well-aligned offset.

The simplest approach is for read-only pointers to grow from the beginning of the buffer towards higher addresses, and object copies placed at the end, growing back towards the beginning of the buffer. These copies have variable size and alignment requirements, and when placing these, we have to insert additional padding to locate the next properly aligned address. If these objects grow from the end of the buffer, this can be done in a relatively simple manner: assuming the last object was placed beginning at offset x, and we now wish to place a structure of type with size s and alignment requirement a, the offset at which it should be placed is computed as $(x - s)\& \sim (a - 1)$. $x - s$ sets aside the number of bytes required by the object, but this may yield an unaligned offset. So a further number of bytes must be subtracted, to find the first location divisible by a. This address will have the lower $lg(a)$ bits masked out, so we can create a bit mask with all 1's, except the lower $lg(a)$ bits — the exact opposite of the mask given us by $a - 1$. So a bitwise *not* is applied to this, and *and*'ed together with the initial offset. Figure 4.2 illustrates this.

In order to store an object in our buffer, we must start at the offset at which the previous object ended, subtract the number of bytes required for the object we wish to store (including its associated metadata), find the next well-aligned offset, and write the object there. Since C++ does not allow arbitrary integer arithmetics on pointer types, we cannot simply mask out individual bits of the pointer, in order to find the next aligned address. Instead, the simplest and most portable solution is to work on byte offsets from the start of the buffer. However, this requires the buffer itself to be allocated with suitable alignment. A simple `char` array only requires the same alignment as a single `char` object, and so may be allocated at any address. The C++ standard does guarantee that `char` arrays allocated with `new[]` are to satisfy the most stringent alignment required by any type on the platform[3], and so the buffer could simply be dynamically allocated in this manner.

An alternative solution is to use the class templates for controlling alignment available in C++0x. In the current version of C++, these are not part of the standard, but are specified in TR1[4]. The two relevant templates are `alignment_of` and `aligned_storage`. Given a

[3] Section 5.3.4, paragraph 10 of the C++ standard [9].

[4] Technical Report 1, a collection of library additions not formally part of C++, but approved by the standardiza-

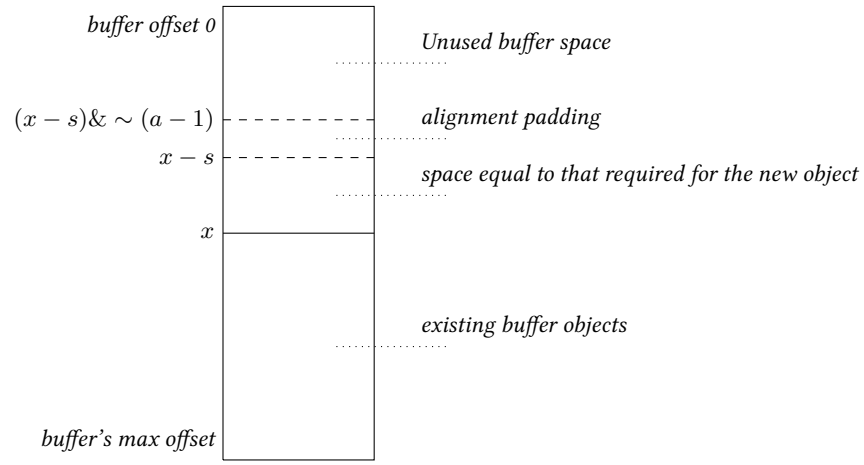

Fig. 4.2: Determining the offset at which a new object should be inserted into the transaction-local buffer. x denotes the previous top of the buffer "stack", while s and a are the size and alignment requirements of the buffer entry. The new object is placed at the topmost dashed line

type `T`, `alignment_of<T>::value` is an integral constant equal to the required alignment for `T`. The `aligned_storage` template is used to allocate storage into which a type can be constructed, given its size and alignment requirements. C++0x additionally specifies a `max_align_t` type, an implementation-defined type requiring the strictest alignment possible on the system.

So to allocate a buffer of size `S` we can instantiate an object of the following type

```
aligned_storage<S, alignment_of<max_align_t>::value>::type
```

4.6 Type erasure

In order to store arbitrary types into the transaction-local buffer, all type information must be erased, and the object data stored into a simple byte buffer. This *type erasure* poses some challenges for us in C++, as the system must still be able to invoke type-specific operations even after the type information has been lost, so that the object's destructor and assignment operator can be called.

When an object is opened for writing, the data written into the buffer must, in addition to a copy of the object itself, store certain metadata to compensate for the lack of type information:

- a pointer back to the canonical object is required, so that the copy can be written back when the transaction commits, and to allow the backend to access the object's reader counters and version number.
- A pointer to a function able to destroy the object by invoking its type-specific constructor,
- a pointer to a function able to copy the object back into the canonical location,
- a pointer to the previous object allocated in the buffer, in order to enable traversal of the objects stored in the buffer,

tion committee and supported by major compilers.

- if the object was opened previously in an outer transaction, a pointer to its previous entry in the buffer is required, to properly implement closed nesting semantics.

The two function pointers in particular are key to making type erasure work. The functions pointed to must have a uniform signature, regardless of which type of object they copy or destroy. This can be achieved by defining template functions which take a common pointer type, such as simple `void` pointers, as their parameters, but internally cast these pointers into the types specified by their template parameters, as in the following simplified example:

```
template <typename T>
void destroy(void* object) {
  // Convert the void pointer into the object's actual type
  T* ptr = static_cast<T*>(object);
  // manually invoke the object's destructor
  ptr->T::~T();
}
```

This allows us to store the same type of function pointer in the buffer metadata, regardless of the type of object being stored, while still enabling us to call the type-specific assignment operators and destructors when needed. As in Section 4.5, the `aligned_storage` class template is used to set aside suitably aligned storage for the object copy itself next to the metadata.

4.7 `shared` object metadata

The `shared_base` class template contains the metadata required for transactional objects to be tracked by the STM system, while making no references to the actual objects being stored by the derived classes. This has been described at a conceptual level in Section 3.3, but the precise class layout is presented here.

As described previously, the metadata must contain a version counter field, a flag for indicating the active slot, and two counters which indicate how many readers are accessing each slot and act as locks ensuring exclusive access to the object.

This should be packed as well as possible to minimize cache usage, while ensuring atomic and efficient access to each of the counters. We note that the flag is only flipped when the version counter is updated and both fields are accessed only when the object is locked, so atomicity is not needed. This means that both can be stored as part of the same word. The most significant bit of the version counter can be used as the flag, and the remainder used for the version field. This limits the system to 2^{31} updates before the version counter overflows, but this does not make a significant difference. The version counter will overflow in realistic long-running programs whether it uses 31 or 32 bits, but it will happen rarely enough in both cases that the amortized cost of handling the overflow is negligible. Alternatively, a 64-bit field can be used for the version counter instead, in which case the counter will never realistically overflow, even if one bit is reserved for use as the flag. This gives us a layout such as this:

```
struct shared_base {
  uint32_t version; // msb is used as flag
  int16_t lock[2]; // reader counters
};
```

As long as 32-bit versions are used, this gives us exactly 64 bits of metadata. We assume 16 bits to be sufficient for each slot's reader counter, as this enables us, even with signed

integers[5], to track over 32000 transactions opening the same object. If the version field is extended to 64 bits, 128 bits will be spent on metadata: 96 bits are directly used by the specified fields, and the remaining is necessary padding, as any C++ object must have a size equal to some multiple of its alignment requirement — if that was not the case, then an array of this type would result in some array entries being unaligned. This additional overhead is unfortunate, but should still be tolerable.

4.8 Version counter overflow

Because a global version counter is used to to generate timestamps for transactions and transactional objects, we have to handle the case when this counter overflows. As objects are able to store 31 bit timestamps in the current implementation, the counter is limited to 2^{31} different values. Assuming a million transactions per second, a figure achievable in many STM systems, the counter will overflow in just 35 minutes. Even at a much more conservative thousand transactions per second, the counter will overflow in less than 25 days — not commonly a problem, but something that will occur in long-lived processes. A simple way to avoid this problem could be to extend the version counter to a 64-bit value, giving us 2^{63} versions, effectively eliminating the risk of overflows.

Otherwise, a mechanism for ensuring that the counter only wraps around under safe, controlled conditions would have to be designed, and for handling validation of transactional objects that are left with a high version after the version counter is reset.

The version counter can be extended to 64 bits with a minimum of difficulty, as operations on it are not required to be atomic: the version of a transactional object is only accessed while the object is locked for modifications, ensuring that its version is safe to access, even if the data type is larger than what can be read atomically by the hardware.

4.9 Optimizations and caching

The iterator types exposed by the buffer are only required to be *forward iterators*, since the read-write stack is effectively a singly-linked listed; objects copied into the buffer have variable size, so there is no way to compute the location of arbitrary objects, making random-access iterators impossible to implement without a separate data structure caching the offsets for each object. Bidirectional iterators could be implemented by adding an additional pointer to the metadata for each record, but there would be little benefit from this.

This buffer interface is intentionally minimal, requiring the weakest possible iterators and providing no functionality other than what is absolutely necessary. Many operations could be optimized or cached using additional data structures. For example, whenever an object is opened, it is necessary to search the buffer to see if the same object was already open. With the available interface, only a linear scan is possible, which will become expensive for larger transactions. However, this design decision is to enable different buffer implementations to be combined as described in Section 4.5. If a wrapper is to be developed which contains both a fixed-size array buffer and a resizable linked list, these two buffers should not implement any kind of auxillary data structures for optimizing searching themselves, or those optimized data structures would get duplicated as well.

Instead, the buffer provides only basic forward iterators, and the transaction manager must then be extended to contain a separate data structure for caching or providing efficient indexing of the contents of the buffer. If this is done outside the buffer, there is also the

[5] Signed integers are used because the intrinsics provided by Microsoft's Visual C++ compiler for atomic increment/decrement operations only work on signed values.

possibility that the same caching data structure can be reused for different buffer types.

While such optimizations have not yet been implemented, they are likely to provide a significant boost in performance in larger transactions.

4.10 Type Requirements

The library is designed to be non-intrusive, so that it can be used with existing types that have not been designed specifically for STM use. However, the following requirements still have to be fulfilled for a type `T` before it can be transaction-enabled.

In the following, `t` is some object of type `T`. `u` is an object of some type convertible to `T`.

Expression	Exc. Guarantee	Notes
`T t(u)`	basic	
`t = std::move(u)`	nothrow, basic	`t = u` in C++03
`t.T::~T()`	nothrow	

Detailed description

`T t(u)`

Notes:

The type `T` must be copy constructible, as this is the mechanism used for creating transaction-local copies of the object.

Throws:

The operation must provide the basic exception guarantee. If an exception is thrown, `t` must be left in a valid state and no resources leaked.

Postconditions:

`t` and `u` is equivalent, the precise meaning of which is unspecified, since no `operator==` is required to be defined. However, the transaction semantics depend on `t` and `u` to contain interchangable values.

`t = std::move(u)`

Notes:

The function `std::move` only exists in C++0x. In C++03, the expression is replaced by a simple copy assignment, as in `t = u`. The effect of `std::move` is to perform a move assignment if one is defined, and otherwise fall back to a copy assignment. This operation is used to assign transaction-local objects into the canonical ones.

Throws:

If this expression provides the nothrow exception guarantee for all types opened by a transaction, the entire transaction operation also offers the nothrow guarantee. If one or more types provides the basic or strong guarantees, the transaction only offers the basic guarantee.

Postconditions:

The value of `t` is equivalent with the value `u` held before the expression was evaluated. The precise definition of *equivalence* is unspecified, as no `operator==` is required to be defined. However, the transaction semantics depend on the values being interchangable.

`t.T ::~ T()`

Throws:

T's destructor must provide the nothrow exception guarantee.

Additional requirements

No operation invoked on the type `T`, including both the ones defined above and the ones invoked explicitly by the user during the transaction, may access data visible to other parts of the application. Since changes made during a transaction are speculative until the transaction is committed, these changes must not be visible outside the transaction. To enforce this, any data modified through this object must be exclusively owned and accessed by the object itself. Likewise, it is assumed that copy assignment and copy construction is "deep", in that it copies all relevant members, instead of pointing back to common instances shared with other objects.

4.11 Detailed API reference

The DikuSTM API presents a very small and simple API consisting of only a few classes and functions all of which is contained in the header file `stm.hpp` and is enclosed in the namespace *stm*. These are described in detail here:

4.11.1 Concepts

As the DikuSTM library relies heavily on templates and generic programming, many operations are not simply defined for specific pre-defined types or base classes, but can be invoked on *any* type, as long as it defines the required functions and members. We say that the type must implement a specific *concept*, which is defined by the operations that must be legal on that type, and by the semantics of these operations.

The concepts that are part of the user-facing API are described in the following. Concept names are written in *CamelCase*, to distinguish them from concrete types and functions which are all lowercase. For example, `stm::atomic(TransactionFunction f)` can take any type which implements the `TransactionFunction` concept as its parameter.

While these descriptions also specify the exceptions that may be thrown by an expression, only exceptions that the user must be aware of are listed. If an exception can be thrown by an expression, but the user is required to let it pass through without being caught, that exception is not listed in the following.

TransactionFunction concept

A type that implements the TransactionFunction concept can be executed as a transaction. As such, this concept must be implemented every time the user wishes to define a transaction. It is a function or a function object `txf`, for which the following expression is valid:

Expression	Returns	Throws	Notes
`txf(tx)`	Any	Any	`tx` has type `stm::transaction`. The expression may be evaluated multiple times during a transaction.

Detailed description

`txf(tx)`

This expression is invoked by the STM system, and executes the transaction defined by `txf`. If an exception other than those defined by the STM system is thrown, the system attempts to commit the transaction, after which it will rethrow the exception for the caller to catch. If commit fails, the system will revert any changes to transactional objects made within the transaction, and evaluate the expression again. If the function returns a type other than `void`, the return value is passed to the caller if the transaction commits without throwing an exception.

Shared concept

A type that implements the Shared concept models a transactional type: it exposes the necessary functions for a transaction to retrieve the contained value, and contains the metadata necessary for the STM system to track modifications to the object.

Given an object `sh` of a type that implements the Shared concept and transaction-enables an object `obj` of type `T`, and an object `tx` of type `stm::transaction`, the following expressions must be valid:

Expression	Returns	Throws	Notes
`sh.open_rw(tx)`	`T&`	`std::bad_alloc`, same as `T t0(t1)`	`t0` and `t1` have type `T`. `T` may not be const.
`sh.open_r(tx)`	`const T&`	`std::bad_alloc`	`sh` can be const.

Detailed description

`sh.open_rw(tx)`

Effects:

Opens `obj` for reading and writing.

Returns:

A reference to a modifiable instance of type `T` containing the most recent value of `obj`.

Preconditions:

No transaction belonging to a different transaction group may access `sh` concurrently with the transaction containing this expression.

Postconditions:

The object referenced by the return value of this expression is guaranteed to exist in isolation from the rest of the system and its lifetime lasts until the transaction leaves the *live* phase. No other thread will read or modify the object.

Throws:

`std::bad_alloc` if the amount of data stored in the transaction exceeds the capacity of the transaction-local buffer, and allocation of a new buffer fails. The default size of this buffer in this implementation is 64 KB, so this exception should not occur unless extremely large transactions are executed.

Any exception thrown by `obj`'s copy constructor.

`sh.open_r(tx)`

Effects:

Opens `obj` for reading.

Returns:

A reference to a const instance of type `T` containing the most recent value of `obj`.

Preconditions:

No transaction belonging to a different transaction group may access `sh` concurrently with the transaction containing this expression.

Postconditions:

The object referenced by the return value of this expression is guaranteed to exist in isolation from the rest of the system and its lifetime lasts until the transaction leaves the *live* phase. No other thread will modify the object.

Throws:

`std::bad_alloc` if the amount of data stored in the transaction exceeds the capacity of the transaction-local buffer, and allocation of a new buffer fails. The default size of this buffer in this implementation is 64 KB, so this exception should not occur unless extremely large transactions are executed.

4.11.2 `atomic` function template

The `atomic` function template executes a given function as a transaction. The function has the following signature, where `R` is the return type of the transaction function `f`.

```
R atomic(TransactionFunction f);
```

Detailed description

`R atomic(TransactionFunction f)`

Effects:

Executes `f` with transactional semantics in the default transaction group.

Returns:

The result of calling the transaction function `f`.

Throws:

`stm::abort_exception` if `f` terminates by calling `tx.abort()`.
Any exception thrown by `f`.

Postconditions:

If `f` terminates by calling `tx.abort()`, the transaction is rolled back.
If `f` terminates by calling a non-STM exception, the transaction is committed and the exception rethrown.
If `f` terminates normally, the transaction is committed and the return value of `f` is returned.

Notes:

If a non-`void` return type is desired, it must be specified explicitly as a template parameter:

that is, for a transaction to return an `int`, `atomic<int>` must be called.[6]

4.11.3 `shared` class template

`shared` implements the `Shared` concept, and is the default class for storing a transactional object. The object stores a single object, and ensures that this object can only be accessed by transactions. In addition to the operations defined by the `Shared` concept, the class defines the following members:

```
template <typename T>
class shared {
public:
  explicit shared(T val);

private:
  T obj; // exposition only
};
```

Detailed description

`shared(T val);`

Effects:

Constructs an instance of a `shared<T>` containing an object `obj` initialized through its copy constructor, with `T obj = val`.

Throws:

Any exception thrown by `T obj = val`.

4.11.4 `shared_detached` class template

`shared_detached` implements the `Shared` concept, but unlike `shared<T>`, this type does not own the object it transaction-enables. Instead, it stores only a pointer to the object. This is to allow for the STM metadata to be located separately from the data it protects, so that existing data layouts such as raw arrays of data can be transaction-enabled. In addition to the operations defined by the `Shared` concept, `shared_detached` also defines the following members:

```
template <typename T>
class shared_detached {
  T* ptr; // exposition only
public:
  shared_detached();
  // see description
  explicit shared_detached(non-const T& val);
  ~shared_detached();
  // see description
  void set_source(non-const T& val);
};
```

[6] This requirement can be avoided in C++0x using the new *decltype* keyword, but the current implementation does not support this.

Detailed description

`shared_detached();`

Effects:

Creates a new instance of a `shared_detached<T>` which does not transaction-enable any object. The only operation valid on such an object is calling the `set_source` member function.

`shared_detached(non-const T& val);`

Effects:

Creates a new instance of a `shared_detached<T>` protecting the object referenced by `val`. `ptr` is set to point to `&val`.

Preconditions:

The object referenced by `val` must not be protected by another `Shared` object.
The lifetime of the object referenced by `val` must be at least as long as the lifetime of `*this`.

Postconditions:

During the lifetime of `*this`, the value of the object referenced by `val` is undefined and must not be modified.
Calling **this**`->open_r` or **this**`->open_rw` returns `val` or a reference to a transaction-local copy of it.

`~shared_detached();`

Effects:

The object pointed to by `ptr` is released and made non-transactional again.

Postconditions:

the object pointed to by `ptr` contains the value assigned to it by the most recent transaction that opened `*this`.

void `set_source(non-const T& val);`

Effects:

If `*this` was initialized with the default constructor, `set_source` specifies an object to transaction-enable.

Preconditions:

`*this` must not protect another object.
The object referenced by `val` must not be protected by another `Shared` object.
The lifetime of the object referenced by `val` must be at least as long as the lifetime of `*this`.

Postconditions:

During the lifetime of `*this`, the value of the object referenced by `val` is undefined and must not be modified.
Calling **this**`->open_r` or **this**`->open_rw` returns `val` or a reference to a transaction-local copy of it.

4.11.5 `transaction` class template

The `transaction` class template is the main interface for transaction functions to interact with the STM system. The transaction function is passed a `transaction` object as its parameter, which can be used to open transactional objects, abort the transaction or otherwise interact with the STM system.

```
struct transaction {
  void abort();
  void retry();
  template <typename T>
  void snapshot(const T* src, T& dest);
};
```

Detailed description

void `abort`();

Effects:

Aborts the currently executing transaction, rolling it back and reporting failure to the caller.

Postconditions:

All modifications to opened transactional objects are reverted. The transaction function does not retry, but throws an `stm::abort_exception`.

void `retry`();

Effects:

Indicates to the STM system that the currently executing transaction should be retried later, when some of its read set has been modified. The implementation is free to regard this as a hint only, and the transaction may be restarted spuriously.

Preconditions:

The transaction has opened at least one transactional object.

Postconditions:

All modifications to transactional objects made by the executing transaction are rolled back. The transaction is restated at some later time.

template <**typename** `T`> **void** `snapshot`(**const** `T*` `src`, `T&` `dest`);

Effects:

Indicates to the STM system that a consistent snapshot of a set of variables as described in Section 3.1.7 is desired when the transaction commits, and adds `src` to this set. `src` typically points to a previously opened transaction-enabled object owned by a `Shared` object.

Throws:

`std::bad_alloc` if space for storing the source and destination pointers cannot be allocated.

Preconditions:

The lifetime of `*src` must last at least until the transaction commits.

Postconditions:

`dest` contains the value that `*src` held when the transaction committed.

4.11.6 `orelse` function template

The `orelse` function wraps two `TransactionFunction` objects in a single compound transaction, here named `OrelseFunction` for exposition purposes. This is a refinement of the `TransactionFunction` concept, and can be used wherever a `TransactionFunction` concept is expected. Additionally, the `operator ||` can be used as a shorthand syntax for chaining more than two `TransactionObjects`.

```
OrelseFunction
 orelse(TransactionFunction lhs, TransactionFunction rhs);
OrelseFunction
 operator || (TransactionFunction lhs, OrelseFunction rhs);
OrelseFunction
 operator || (OrelseFunction lhs, TransactionFunction rhs);

OrelseFunction
 orelse(TransactionFunction lhs, TransactionFunction rhs);
```

Effects:

Constructs a new `OrelseFunction` as a compound transaction consisting of `lhs` and `rhs`. When the compound transaction is executed, `lhs` is first invoked, and if it calls `tx.retry()`, `rhs` is immediately invoked instead. If `rhs` also calls `tx.retry()`, the effect is as if the compound transaction had called `tx.retry()`.

Returns:

A `OrelseFunction` compound transaction wrapping `lhs` and `rhs`.

Postconditions:

When executed, the returned `OrelseFunction` invokes `lhs`.

- If `lhs` retries, `rhs` is invoked.
- If `lhs` does not retry, the compound transaction behaves as if the user had simply invoked `lhs` in its place.
- If `rhs` is invoked and retries, the behavior is as if the compound transaction had called `tx.retry()`. The read set of the compound transaction is the union of the read sets of `lhs` and `rhs`.
- If `rhs` is invoked and does not retry, the compound transaction behaves as if the user had simply invoked `rhs` in its place.

```
OrelseFunction
 operator || (TransactionFunction lhs, OrelseFunction rhs);
```

Effects:

Syntactic sugar for `orelse(lhs, rhs);`.

```
OrelseFunction
 operator || (OrelseFunction lhs, TransactionFunction rhs);
```

Effects:

Syntactic sugar for `orelse(lhs, rhs);`.

4.12 Limitations

Certain features were considered, but have not yet been implemented. These are described here. While we do not generally believe these to be essential, they should be implemented for the system to be as complete, general and usable as possible.

4.12.1 Implementations of the `Shared` concept should be copyable

The DikuSTM system relies on transaction-enabled objects being copyable. However, in some cases, we may also wish to nest different `Shared` types within each others. If the outer object is copied, for example because it is opened by a transaction, all its members must be copied as well. This could be handled on an ad-hoc basis by explicitly defining a copy constructor for each type that contains a `Shared` object, but this would be tedious and error-prone. Instead, the `Shared` concept should specify that objects implementing this concept are required to implement a transcationally safe copy constructor and assignment operator. These operations should create a new transaction, and within it, copy the object protected by the `Shared` wrapper.

4.12.2 Convenience access of transactional values

In some cases we may wish to simply retrieve the most recent value from a transactional object. We could write a transaction function which opens the object and returns a copy of the retrieved value, but this is needlessly verbose. Instead, this operation could be added to the `Shared` objects. For example, `operator*` could be defined, "dereferencing" the shared wrapper in order to retrieve the object contained within it. As with the copy constructor, this would have to be transactionally safe, creating a small transaction to open and copy the object before returning it to the caller.

4.12.3 Efficient *retry* implementation

The *retry* operation is intended to delay the transaction restart until a member of its read set has been modified by another transaction. In the current version of DikuSTM , this delaying mechanism is not implemented, so the transaction will continuoulsy restart, execute and rollback until the condition for proceeding is satisfied.

4.12.4 Portability and dependencies

The DikuSTM system makes use of a number of features and operations not defined by the C++ standard. Platform-specific atomic operations are used to control access to transactional objects, and thread-local storage to store each thread's transaction manager. Currently this is provided by the Boost libraries, ensuring a high degree of portability. However, these features are all part of C++0x, so once better C++0x support is available from compilers, these dependencies can be eliminated. When the library is compiled for C++03 support, the features defined in TR1 are also used, as these are assumed to be at least as portable as the equivalent functionality defined in Boost.

The system also uses a small number of utility headers from Boost: `noncopyable.hpp`, `iterator_facade.hpp`, which could be easily redefined by the DikuSTM library itself, so complete independence of third-party libraries is achievable once full C++0x support is available from major compilers.

5. EVALUATION

As described in the introduction, good performance is critical for STM systems to gain mainstream acceptance. However, until we get experience with STM systems in real-world code, we do not know which scenarios to optimize these systems for.

This chicken-and-egg problem is difficult to solve. Developing comprehensive test cases using STM for a variety of real-world tasks is beyond the scope of this project, and while benchmarks of insertion and removal on linked lists or binary trees are used for most STM systems, we feel that these provide little real value: most STM systems test these operations against home-made data structure implementations, making direct comparison difficult, but more importantly, most real-world code is not going to consist of a large number of threads constantly inserting and removing elements in a binary tree. Even if an application does need thread-safe manipulation of a binary tree, we do not know the degree of contention that is going to be typical.

So instead, this chapter will present a number of smaller synthetic benchmarks. These are not intended to be representative of real-world usage in any way — instead, they are designed to illuminate very specific aspects of the STM system, allowing us to see how it performs in specific situations, with as little noise as possible. This means that the following benchmarks will all consist of "no-op" transactions: the transactions open one or more transactional objects for reading or writing, and then attempt to commit. Most of the tests are performed at varying transaction sizes, defined by the number of objects opened by the transaction. A transaction of size 1 opens only a single object before committing, and one of size 100 opens 100 objects.

The goal of these benchmarks is not to show the system to be "fast enough for real-world use", but merely to highlight areas to focus on in further work.

All tests are run 5 times, and unless otherwise noted, the median of these results is used. All tests are run on an Intel Q9300 Core 2 Quad processor with 4GB of RAM.

5.1 Overhead with zero contention

Before looking at how the STM system performs when multiple threads are accessing the same data, we will examine the overhead introduced by STM operations under ideal circumstances, when only a single thread is executing. If this overhead is prohibitive, the STM system is unlikely to be usable in more complex scenarios where contention may further constrain performance.

Figure 5.1 shows the time taken to to execute and commit a transaction opening a variable number of objects, divided by the transaction size. This gives us an estimate of the "per open" overhead. We expect some fixed overhead from initializing and committing the transaction itself, and the commit operation itself is linear in the transaction size, so the amortized cost for an *open* operation should decrease with larger transactions, as the constant part is shared between a larger number of opens. However, as described in Section 4.9, a simple linear search through the transaction buffer is performed on every open, to check if a private copy of the object already exists. Using an $O(n)$ algorithm for this is extremely inefficient (as

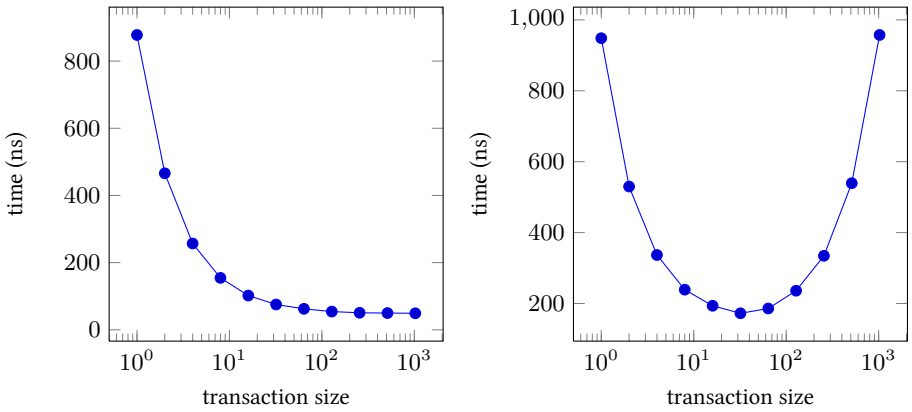

Fig. 5.1: Average transaction duration per object opened, for varying transaction sizes. *Left: objects opened for reading. Right: objects opened for writing.*

this is performed when each object is opened, yielding $O(n^2)$ time complexity in total. For sufficiently large transactions, we therfore expect the amortized cost per *open* operation to increase again.

We see that the average cost for `open_rw` for transactions of size one is 948ns, allowing a single thread with no contention to commit just over a million transactions per second. As the transaction grows, the cost per open decreases as expected, so that for transactions opening 32 objects for writing, the average cost is only 172ns, after which the cost starts increasing again due to the time spent scanning the transaction buffer. At 1024 open operations per transaction, an average open takes nearly the same time as a transaction opening only one object. If typical transactions open only a few objects, the performance may be near optimal already, as implementing a faster lookup mechanism would require an additional data structure to be maintained, introducing additional overhead. If real-world transactions get too large, however, implementing such a data structure for efficient lookups would improve performance significantly.

For `open_r`, the picture looks somewhat different: because this test performs *only* `open_r` operations, no copies are ever added to the transaction buffer, and so every lookup completes in constant time, which allows the cost per open to decrease steadily as the amortized transaction overhead decreases. Because the overhead for opening an object for reading is also lower than when opening for writing (no copy has to be created in the buffer), the average cost peaks for transactions with a single open is 877ns, some 10% lower than the equivalent transaction opening an object for writing. From here, the cost decreases quickly as the transaction size grows. At 1024 objects opened, the amortized cost is only 49ns, allowing over 20 million objects to be opened per second by a single-threaded application.

5.2 Read-write contention

A common scenario is that of a single writer, and multiple reader threads accessing the same data. Here, both reader and writer threads will experience conflicts, although for different reasons: the writer thread will occasionally be unable to lock the objects that should be modified during its commit phase. As mentioned in Section 3.2.7, this cannot starve out the writer thread *entirely*, but we nevertheless wish to see how much it affects the throughput of the thread.

The reader threads never need to lock objects, but instead require the object's version to stay

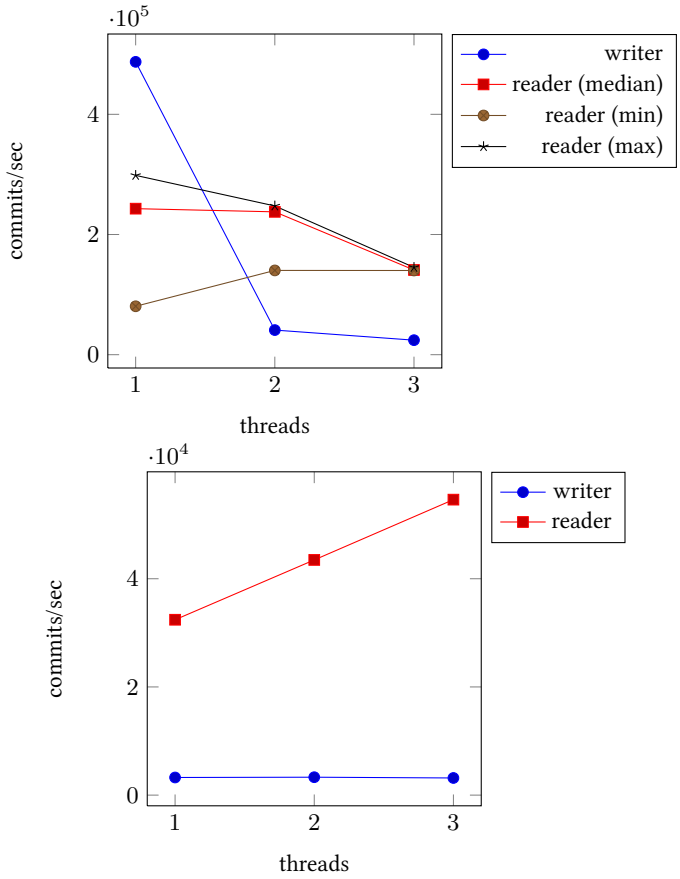

Fig. 5.2: Number of transactions committed by the writer and reader threads. For reader threads, the graph shows total number of commits for all threads combined. *Top: transaction size 1. Bottom: transaction size 500.*

the same throughout the transaction's lifetime. If the writer thread commits a modification, the reader must abort, either when opening the object, or when committing. Figure 5.2 shows the throughput for the writer thread and for all reader threads combined, with varying number of reader threads.

With transactions opening only a single shared object, around 490000 write transactions are committed per second when a single reader thread is running concurrently. As the number of readers increase, this drops dramatically, and with four reader threads, only around 24000 write transactions can be committed per second. With larger transactions of size 500, the throughput for the writer thread is in the order of a few thousand transactions per second. This shows that while read-write contention cannot completely starve out even large writer transactions, it can dramatically limit the number of modifications that are committed, especially for larger transactions.

Throughput for the reader threads is less affected. As readers only get aborted if the writer commits while they are executing, the reader threads experience much better throughput under contention. However, for small transactions, we also see large variations between test runs, showing that their throughput is determined by the very nondeterministic interactions with the writer thread. As well as the median of the test runs, minimum and maximum are also showed in the figure to illustrate this.

For larger transactions of size 500, this nondeterminism is gone, and we see throughput for

reader threads increase linearly with the number of threads. As the transaction size mainly limits throughput of writer threads, the writer spends less time in the commit phase, and so fewer reader transactions are aborted due to concurrent modifications.

Because the drop in writer throughput as contention increases is so severe, we also examine how long transactions typically take before they successfully commit (giving us an indication of whether the system chokes on a few transactions that for some reason keep failing, or the performance decrease is evenly spread across all the transactions), and if the number of rollbacks increases as expected or if some other factor is limiting throughput. We could also have examined the causes for rollback, but since we are mainly interested in the loss in throughput on the writer thread, which can only abort for one reason (inability to lock an object because a reader is accessing it), there would be little point in examining this metric.

We measured the duration of transactions with a resolution of 2ms, giving us a rough indicator of how many transactions stood out as extremely slow. However, it turned out that when computing the median of the results, it showed every single read and write transaction to commit within 2ms. Out of the 5 test runs, each running transactions constantly for 4 seconds on 4 threads, just 43 read transactions took more than 2ms, ranging from 2 to 32ms. For write transactions, the results were similar. When so few transactions show up as taking longer, it likely due to context switching by the operating system or page fault, rather than contention in the STM system. We can conclude that as expected, the loss in throughput is shared between all transactions, and is not due to a few transaction getting "stuck" due to some unforeseen boundary condition.

Finally, we examine the number of rollbacks per successful commit, shown in figure 5.3. Surprisingly, while the number of successful commits drops significantly as the level of contention increases, the number of writer rollbacks is very stable. With two reader threads, the number of rollbacks does increase to some 20000, from around 8000 with a single reader thread running concurrently. However, compared to the dramatic decrease in successful write transaction commits, we would have expected far more rollbacks.

This indicates that validation and object locking conflicts are *not* the main cause for the lowered throughput. One possible alternative explanation could be that when a transaction commits, it locks the global version counter. The high number of committing read transactions could cause contention on this lock. Instead of the write transactions being forced to roll back because the modified objects cannot be locked, they would then be blocked trying to lock the version counter.

As predicted in Section 3.2.7, our double-buffered scheme — where modifications are stored into the backing slot not normally accessed by readers — does seem to prevent write transactions from being starved out. The starvation that does occur seems to be due to other factors: the global version counter being locked too aggressively, and inefficient algorithms used for searching the transaction buffer.

Interestingly, for larger transactions, the number of aborted write transactions actually reaches zero. As discussed above, writer transactions write into the backing slot of modified objects, and so, only experience a conflict if reader transactions from two commits ago are still registered on the object. As the transaction size grows, the performance of writer transactions decreases dramatically, and so this scenario never occurs.

5.3 Write-write contention

We also need to investigate how performance is affected if multiple threads attempt to modify the same objects. We would expect a high throughput of commits even with the contention caused by multiple threads running. Figure 5.4 confirms this. With transactions opening and committing just one shared object, we see the total throughput fall by around

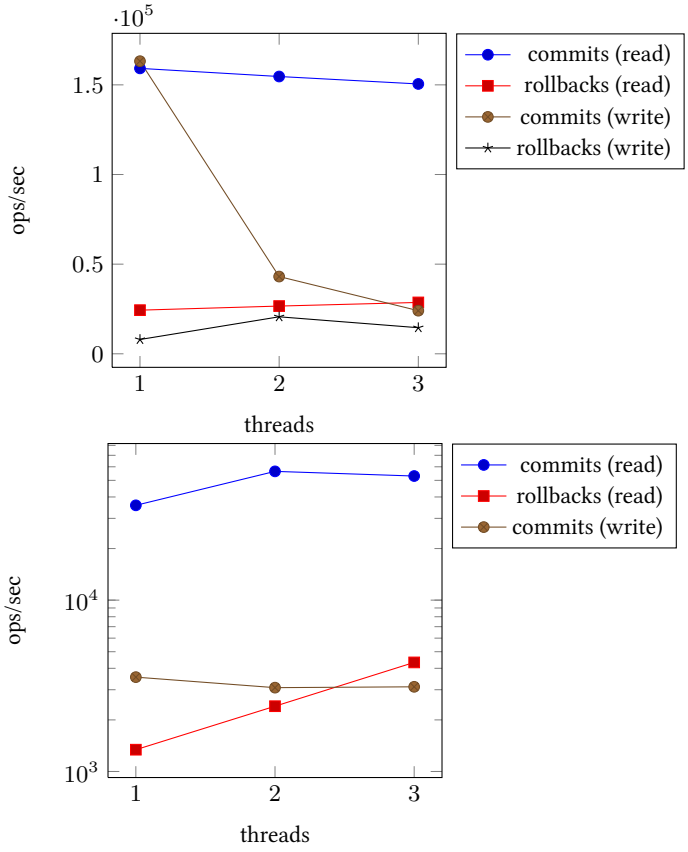

Fig. 5.3: Number of committed and aborted transactions per second for reader and writer threads respectively. *Top: transaction size 1. Bottom: transaction size 500.*

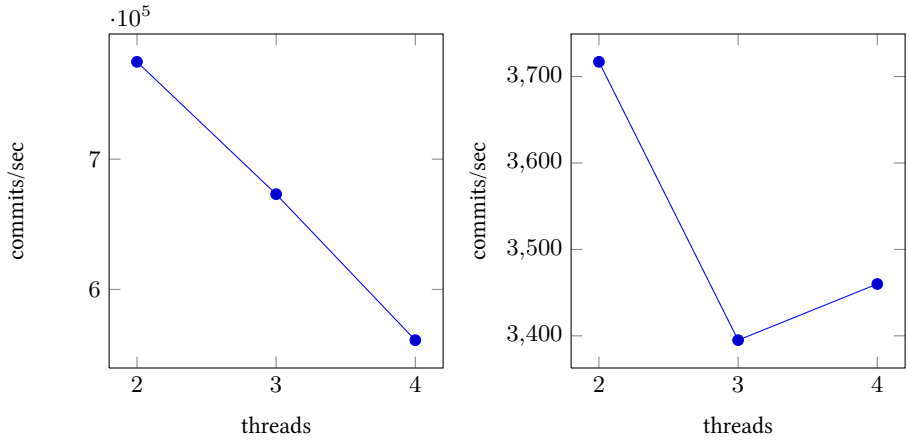

Fig. 5.4: Number of committed writer transactions per second, with varying number of threads writing concurrently. *Left: transaction size 1. Right: transaction size 500.*

28% as the number of thread increases from two to four. Since this is such a tight loop with maximum contention, this is reasonable: in computationally heavy transactions, we would have expected the throughput to scale upwards when more threads are running, but since our transactions in this test are effectively just *no-op*s which open a shared variable for writing, and then immediately commits, the contention prevents upwards scalability. Instead, we get a gradual decrease in throughput, which is acceptable, as long as there are no sharp drops.

For larger transactions, we get a very low throughput of just a few thousand transactions per second. This must be attributed partly to the inefficient algorithm used for scanning the transaction buffer when objects are opened and the long period in which the version counter must be locked for all modifications to be applied, which together make the transactions slow enough to be very vulnerable to contention.

6. CLOSURE

6.1 Conclusion

In this thesis, the DikuSTM transactional memory library has been developed and presented. It presents a very clean and usable interface while supporting a large number of operations beyond the core functionality. The *retry* and *orelse* operations first defined in Haskell STM are implemented, and new functionality for taking consistent snapshots of unbounded sets of transactional objects is introduced.

The interface is designed to be as generic and easy to use as possible, and is, to our knowledge, the first C++ implementation to incorporate functors for representing transactional code. This solves a number of semantic problems by giving transactions a more intuitive scope, and enables C++0x lambda expressions to be used for defining transactions, resulting in a clean and concise syntax eliminating many potential sources of user error.

We consider the presented STM interface to be near ideal for a C++ STM system, and suggest that other STM implementations should be adopted to use it as well. This would encourage real-world adoption of STM, as different library implementations can be tried out without requiring changes to user code, but would also be a first step toward producing more and better benchmark suites.

The DikuSTM system is entirely non-intrusive, requiring no changes to existing types in order to transaction-enable them, another key to the simplicity of the presented interface.

The underlying implementation of the STM system has explored a new "double-buffered" deferred-update scheme which can potentially overcome many of the disadvantages of both direct-update and deferred-update systems. However, crucial optimizations have not yet been implemented, and so the actual performance of DikuSTM leaves something to be desired: the overhead for basic STM operations is low under ideal conditions, but larger transactions currently incur a noticeable performance penalty, and since no mechanism for prioritizing transactions or performing some kind of contention management is implemented, some high-contention scenarios cause performance to drop to unacceptable levels.

However, many of these shortcomings can be fixed by applying simple, well-understood optimizations. Other STM systems have explored a wide range of contention management policies, and faster lookups in the transaction-local buffer is a simple question of defining an auxillary data structure such as a hash table allowing faster lookups of buffer entries.

6.2 Future Work

The performance of the system has only been tested with very small synthetic benchmarks, and while this gives us some clear ideas of the precise areas in which the performance of the system is lagging, more comprehensive and "realistic" benchmarks should be developed.

To improve the relevance of such benchmarks, other STM systems should also be adapted to use the interface provided here. This would also simplify mainstream adoption of STM — if multiple systems use the same interface, they can be swapped out with ease if performance

turns out to be unsatisfactory.

Once this is done, we can get a much clearer idea of which aspects of a STM system's performance are really critical, and adapt our systems for those. Until then, there are, as mentioned above, several optimizations that can be applied to the DikuSTM system with little risk, but many others depend on usage patterns, and as such, should not be applied until we have a better idea of how STM is going to be used.

One possible optimization that has not be discussed would be to eliminate the global version counter lock during commit operations. If transactions locked their entire readset during commit, it may be possible for the commit operation to complete without requiring the version counter to be locked at any point, while still preserving conflict serializability. While this would likely increase best-case overhead somewhat, it would also reduce contention for the shared version counter.

Existing libraries should also be adopted for STM usage — for example, common data structures should integrate STM to provide high-performance thread-safe access. A STM-enabled version of the STL would undoubtedly reveal many weaknesses of this and other STM systems, but would also be a crucial step on the path towards mainstream acceptance of STM.

BIBLIOGRAPHY

[1] D. Abrahams, *Exception-Safety in Generic Components*, Springer Berlin / Heidelberg (2000).

[2] B. Dawes, D. Abrahams, N. Josuttis, et al., *The C++ Boost Libraries*, `http://boost.org`.

[3] D. Dice and N. Shavit, What really makes transactions faster?, *Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, ACM (2006).

[4] K. Egdø, A software transactional memory library for c++, M. Sc. Thesis, University of Copenhagen, Denmark (2007).

[5] J. Gottschlich and D. A. Connors, Dracostm: A practical c++ approach to software transactional memroy, *Proceedings of the 2007 ACM SIGPLAN Symposium on Library-Centric Software Design (LCSD)*, ACM (2007).

[6] T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy, Composable memory transactions, *Communications of the ACM* **51**, 8 (2008), 91–100. (An earlier version appeared at PPoPP '06.)

[7] M. Herlihy, V. Luchangco, M. Moir, and I. William N. Scherer, Software transactional memory for dynamic-sized data structures, *PODC '03: Proc. 22nd ACM Symposium on Principles of Distributed Computing* (2003), 92–101.

[8] M. Herlihy, J. E. B. Moss, J. Eliot, and B. Moss, Transactional memory: Architectural support for lock-free data structures, *in Proceedings of the 20th Annual International Symposium on Computer Architecture* (1993), 289–300.

[9] JTC1/SC22/WG21, *ISO/IEC 14882:2003 FDIS, Programming Language C++*, ISO, Geneva, Switzerland (2003).

[10] JTC1/SC22/WG21, *ISO/IEC 14882 Working Draft, Programming Language C++*, ISO, Geneva, Switzerland (2010).

[11] J. Larus and R. Rajwar, *Transactional Memory (Synthesis Lectures on Computer Architecture)*, Morgan & Claypool (2007).

[12] W. N. Scherer III and M. L. Scott, Advanced contention management for dynamic software transactional memory, *Proceedings of the 24th ACM Symposium on Principles of Distributed Computing*, Las Vegas, NV (2005).

[13] N. Shavit and D. Touitou, Software transactional memory, *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, ACM (1995), 204–213.

[14] N. Shavit and D. Touitou, Software transactional memory, *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, ACM (1995), 204–213.

[15] J. Sreeram, R. Cledat, T. Kumar, and S. Pande, RSTM: A relaxed consistency software transactional memory for multicores, *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, IEEE Computer Society (2007), 428.

[16] A. Stepanov and M. Lee, The standard template library, Technical report, WG21/N0482, ISO Programming Language C++ Project (1995).

[17] B. Stroustrup, The design of c++0x, *C/C++ Users Journal* (2005).