A Software Transactional Memory Library for C++

Master's Thesis

Kasper Egdø
egdoe@diku.dk

Department of Computer Science
University of Copenhagen

January 31, 2008

Abstract

An increasing number of cores (or CPUs) per computer is creating a need for good programming tools for exploiting these cores. Locks are traditionally used but issues with deadlocks and race conditions easily arise and choosing the correct granularity for locking is often not trivial. In addition, the use of locks does not generally enable the programmer to compose existing correct lock-based program pieces into a new larger correct lock-based piece.

An alternative to locks is Software Transactional Memory (STM) which is a concurrency approach that does not use locks as its primary method. Instead STM is an optimistic concurrency control mechanism; using memory transactions similar to transactions used in database systems. Programs built with STM avoid deadlocks and race conditions and enables composition of program pieces from existing pieces.

This work explores design criteria for building a STM system for C++ and for building Standard Template Library containers on top of such a system. We emphasize correctness over performance, generic programming and direct reuse of existing algorithms and data types with our implementation. Using an indirection-based approach for our STM implementation enables us to guarantee strong exception safety for all operations on our data structures. We also develop a new method for nesting STM-based data structures within each other, such that we avoid unnecessary conflicts and copying.

Using compile-time polymorphism we develop a new method for laying out shared data in memory, thereby allowing us to reduce the number of cache misses that are otherwise the primary disadvantage of indirection-based STM implementations. For large datasets with low contention the throughput in our benchmarks is increases by up to 33 percent when using our layout technique.

Resumé

Moderne datamater vil i fremtiden indeholde en stigende mængde CPU-kerner. Det medfører, at sekventielle programmer vil kunne udnytte en stadigt faldende delmængde af datamaternes regnekraft. Der er derfor brug for nogle programmeringsværktøjer til at hjælpe med at anvende flere kerner samtidigt. Traditionelt anvendes låse, men brugen af disse kan nemt medføre problemer med deadlocks og race conditions; det er heller ikke altid trivielt at vælge den korrekte granularitet i forbindelse med låsning. Yderligere medfører brugen af låse generelt ikke at man kan sammensætte eksisterende korrekte programstykker til større korrekte programstykker.

Som alternativ til låse findes Software Transactional Memory (STM), der ikke anvender låse som sit primære virkemiddel. I stedet er STM en optimistisk samtidigheds kontrolmekanisme; der anvendes transaktioner på arbejdslageret i stil med transaktioner som man kender fra databasesystemer. Programmer bygget ved hjælp af STM undgår deadlocks og race conditions og muliggør generelt sammensætning af eksisterende korrekte programstykker til større korrekte programstykker.

Relevante designkriterier evalueres med henblik på at implementere et STM system til C++ samt at implementere datastrukturer til dette; datastrukturerne implementeres så vidt muligt i tråd med de tilsvarende fra Standard Template Library. Korrekthed prioriteres over ydeevne; der fokuseres på generisk programmering og direkte genbrug af eksisterende implementation af algoritmer og datatyper. Ved at anvende en pegeromvejs-baseret tilgangsvinkel til STM implementationen opnås den stærke form for undtagelsesgaranti for alle operationer på de implementerede datastrukturer. Der udvikles også en ny metode til at indlejre STM-baserede datastrukturer i hinanden, således at unødvendige konflikter og kopieringer undgås.

Ved at anvende oversættelsestidspunktstypepolymorfi udvikles en ny metode til at placere data i arbejdslageret, således at antallet af fejlslagne opslag i arbejdslagerets cache reduceres; mange fejlslagne opslag er iøvrigt den primære ulempe ved pegeromvejs-baserede STM implementationer. Ved store datasæt med lav konflikthyppighed opnås op til 33 procent flere transaktioner per sekund når den nye dataplaceringsmetode anvendes.

Contents

1 Introduction **3**
- 1.1 Motivation . . . 3
- 1.2 Goals of the thesis . . . 4
- 1.3 Prerequisites . . . 4
  - 1.3.1 Memory models, cache coherency . . . 4
  - 1.3.2 Exception safety . . . 5
- 1.4 Software Transactional Memory . . . 6
  - 1.4.1 Programming with STM . . . 6
  - 1.4.2 Semantics . . . 9

2 Approaches to STM **11**
- 2.1 Granularity . . . 11
  - 2.1.1 Word-based granularity . . . 11
  - 2.1.2 Block-based granularity . . . 11
  - 2.1.3 Object-based granularity . . . 12
- 2.2 Validation . . . 13
  - 2.2.1 The need for validation . . . 13
  - 2.2.2 Validation methods . . . 15
- 2.3 Update method . . . 17
  - 2.3.1 Undo logging . . . 18
  - 2.3.2 Redo logging . . . 19
  - 2.3.3 Indirection . . . 19
- 2.4 Contention management . . . 20
- 2.5 Priority inversion . . . 21
- 2.6 Related/previous work . . . 21
  - 2.6.1 Hardware-based approaches . . . 21
  - 2.6.2 Existing implementations for C/C++ . . . 22

3 ESTM **23**
- 3.1 API . . . 23
  - 3.1.1 General design considerations . . . 23
  - 3.1.2 Constraints on types used in transactions . . . 24
  - 3.1.3 Transaction interface . . . 24
  - 3.1.4 Returning from transaction blocks . . . 28
  - 3.1.5 Aliasing problems . . . 29
  - 3.1.6 Memory management . . . 30
- 3.2 Object lifetimes . . . 30
- 3.3 Lifecycle of a transaction . . . 31
- 3.4 Internal data structures . . . 32
  - 3.4.1 Transaction group data . . . 32
  - 3.4.2 Shared objects . . . 32
  - 3.4.3 The transaction metadata . . . 32

3.5 Operations . . . 34
3.5.1 Starting a transaction . . . 34
3.5.2 Opening objects in read-only mode . . . 34
3.5.3 Opening objects in read-write mode . . . 34
3.5.4 Creating an object . . . 35
3.5.5 Deleting an object . . . 35
3.5.6 Validation . . . 35
3.5.7 Acquiring an object . . . 35
3.5.8 Performing the commit . . . 35
3.5.9 Aborting a transaction . . . 36
3.6 Retry operation . . . 36
3.7 Memory management . . . 37
3.7.1 Colocating objects with their metadata . . . 38
3.7.2 Deferred deletion . . . 41
3.8 Implementation . . . 42
3.8.1 General . . . 42
3.8.2 Version numbers and memory use . . . 42

4 Data structures and STM **44**
4.1 APIs . . . 44
4.1.1 Iterators . . . 45
4.1.2 Container wrappers . . . 46
4.1.3 Creating and destroying data structures . . . 46
4.1.4 Temporaries . . . 47
4.1.5 The `size` method . . . 48
4.1.6 Allocators . . . 48
4.2 Nesting STM-based data structures . . . 48
4.2.1 Recursive wrapping . . . 49
4.3 General considerations for data structures . . . 50
4.3.1 Running times and starvation . . . 51
4.3.2 Sentinel nodes and false conflicts . . . 51
4.3.3 Storage granularity . . . 51
4.3.4 The `empty` method . . . 51
4.4 Some specific data structures . . . 52
4.4.1 `map` . . . 52
4.4.2 `list` . . . 53
4.4.3 `vector` . . . 53

5 Experimental evaluation **54**
5.1 The benchmark . . . 54
5.1.1 Test environment . . . 56
5.1.2 Test method . . . 56
5.2 Evaluation . . . 56
5.2.1 Restarted transaction . . . 56
5.2.2 Allocations . . . 56
5.2.3 Run time . . . 58
5.3 Large nodes . . . 62
5.4 Adjacent cache line prefetch . . . 65
5.5 Discussion . . . 66

6 Closure **68**
6.1 Conclusions . . . 68
6.2 Future work . . . 68
6.3 Source code availability . . . 69

Chapter 1

Introduction

1.1 Motivation

Modern computers will increasingly contain more cores and/or CPUs. This implies that sequential programs will use a decreasing amount of the computers' power. Some programming tools for using more cores or CPUs are therefore needed. As many tasks that programs work on are not completely independent, the multicore (multithreaded) program requires means for communication of data between the threads of execution. To ensure correct communication and sharing, one or more synchronization mechanisms are necessary. Locks are traditionally used on machines where all cores have equal access to main memory, such that a given core can gain exclusive access to a piece of data in main memory.

The use of locks does however carry some disadvantages with it. Choosing the correct granularity for locking can be difficult — choosing to small a granularity risks using a large part of the running time on acquiring and releasing locks; choosing too large a granularity may result in different threads accessing different data losing their opportunities for concurrency.

In addition it is hard to write lock-based programs that have no data races, and are deadlock free. In particular the use of locks does not generally enable the programmer to compose existing correct lock-based program pieces into a new larger correct lock-based piece. We say that lock-based programs are not *composable*.

Software Transactional Memory (STM) is a concurrency approach that does not use locks as its primary method. Instead STM is an optimistic concurrency control mechanism. Every thread of execution can read and write shared state via the STM system. The individual thread can assume that its work happens independently of the other threads. The work done by a thread is done in appropriately sized transactions. Only when a transaction is *committed* (it successfully completes) does its changes to shared state become visible to other threads. If a thread during its execution has read or written state, that another thread's completed transaction has written to, the former thread's transaction is *aborted* and its changes discarded. The aborted transaction is restarted with the new state.

STM avoids deadlocks, livelocks and makes it possible to compose existing correct program pieces into larger correct pieces, without having to know or alter the inner workings of each piece.

Several existing implementations of STM exist for C++. However, none of them have generic programming as their primary priority, and most of the implementations seem to be (for performance reasons) moving towards an implementation method that either trades correctness for speed, or requires the data types used in transactions to be littered with STM implementation details. We wish to emphasize the generic programming approach, and avoid both the correctness versus speed tradeoff and need for littering the code with STM implementation details. Instead, we accept an increased memory footprint for our implementation.

1.2 Goals of the thesis

We have three main goals in mind in this thesis. First, we wish to describe and analyze the aspects of STM needed for an implementation in C++ such that the idioms and conventions of C++ are respected as much as possible. Second, we wish to do for STM what the standard template library did for memory management: to hide the details of using STM from the programmer and push the details into template-based standard algorithms and data structures, such that the programmer's possibilities for errors is reduced and the clarity of the code is increased. Third, we will implement a STM system based on the results of the analysis, build some of the common data structures on top of the STM system and compare the performance of the new STM and data structures to the performance of the equivalent data structures in the standard template library.

In more detail, we will describe STM systems and concepts and the common approaches to implementing the different parts of an STM system. During this, we will argue for what we believe to be the correct approach in a C++ setting. In particular, we will be interested in using our STM system in generic template-based code that should be capable of coexisting with and reusing existing data structures and objects. The generic approach gives us slightly different priorities than those used by existing STM systems, and affords us some new options for performance improvements.

After the analysis and implementation of the core STM system, we will implement some data structures, whose design will be very similar to the design of those in the standard template library. We will look at how some of the techniques used to implement "normal" versions of these data structures can be transferred to their STM counterparts. Some new challenges with regard to nesting the data structures as values of other data structures arise, that we solve. Solving these problems has to our belief not been attempted before.

As for performance goals, the target is not to outperform, in our microbenchmarks, a lock-based or single threaded non-STM approach, but simply to perform well enough to be a viable alternative in programs that perform significant amount of calculations or operations on non-shared data, but sometimes need to access and update some shared data. We do however explore the performance advantages that the generic programming approach affords.

1.3 Prerequisites

In this section we discuss the concepts needed to understand the challenges related to concurrency in general and STM in particular. We assume the reader is familiar with the simplest concurrency terminology; in particular the reader is expected to understand threads, locks, deadlocks, livelocks and starvation. The reader is also expected to have some familiarity with C++ and the Standard Library [23].

1.3.1 Memory models, cache coherency

When we deal with concurrent programs on multiple CPU or cores, we must be aware of how and when memory writes made by one core becomes visible to other cores. We discuss the relevant aspects of this in the following.

Memory models

When operating under an assumption of a single thread of execution the performance of a given program may be improved be letting both the compiler and the CPU *reorder* individual reads and writes of memory around one another and in some cases remove them entirely, provided the compiler or CPU can prove that this does not change the observable behavior of the program. Performing these reorderings may improve performance by hiding or removing latency in the memory system.

In multithreaded programs where individual threads of execution need to communicate with one another (via the memory system), *memory barriers* can be inserted into the program or

instruction stream to inhibit the reorderings. While the details differ across hardware architectures and programming languages, we assume in this work that the following barrier types exist:

- An *acquire barrier* prevents reads and writes placed *after* the barrier from moving over the barrier.
- A *release barrier* prevents reads and writes places *before* the barrier from moving over the barrier.
- A *full barrier* preventing reads and write specified on either side of the barrier from moving over the barrier.

The amount of freedom the compiler or CPU has in reordering reads and writes and the options for inhibiting it is specified by a *memory model*.

Cache coherency

Multiple CPU may have both individual memory caches and shared caches. We assume that the caches are *coherent*; that is, that different caches caching the same memory location do not contain different values. Many cache coherency protocols exist; we shortly discuss the MESI protocol here (see e.g. [15]). With the MESI protocol every cache line in a cache is in one four states:

- `Modified`: The cache line is only stored in this cache, and it has modifications not present in memory.
- `Exclusive`: The cache line is only stored in this cache, but has no modifications.
- `Shared`: The cache line may be stored in other CPUs' caches, but has no modifications in any cache.
- `Invalid`: The cache line is invalid.

When a CPU wishes to perform a read, its cache can satisfy this read if the relevant cache line is anything but `invalid`. If the CPU wishes to perform a write, the relevant cache line must initially be in the `modified` or `exclusive` state; after the write it will be in the `modified` state. A cache A must alter the state of its cache lines appropriately when the state of another cache's equivalent cache line becomes `modified`, `exclusive` or `shared`; A may have to write a modified cache line back to memory before letting the other cache assume ownership of it.

Relevant to this work is that, if two or more CPUs are repeatedly writing to memory held in the same cache line, the cache line will have to move back and forth between the caches (and be written back to memory as well); this very time consuming phenomenon is called *cache bouncing*.

Memory primitives

Most CPU architectures provide some memory primitives that we can use for synchronization purposes, such that we can avoid using locks. We only assume one such primitive: the compare-and-swap (CAS) instruction. The CAS instruction atomically compares the contents of a memory location with a given value and, if they are equal, replaces the contents of the memory location with another given value. The operation returns the previous value of the memory location, such that we can determine if the operation succeeded. We use this operation extensively.

1.3.2 Exception safety

Programming with exceptions presents some challenges. When a method of an object is terminated by an exception, the method is said to be *exception safe* if the object in question is left in a valid state; this means the object's invariants are preserved despite the exception [24]. A given exception safe method can provide different guarantees about its behavior with regard to exceptions. In decreasing order of desirability:

```
1 atomic {
2   int x=listA.front();
3   listA.pop_front();
4   listB.push_front(x);
5 }
```

Figure 1.1: *Simple example:* A transaction reads and removes an element from `listA` and inserts the element into `listB`.

```
1 atomic {
2   if (listA.empty())
3     retry;
4   //Do something with the contents of listA
5 }
```

Figure 1.2: *Retry example:* A transaction checks if `listA` is empty; if so, the transaction *retries* and is woken up and restarted when another transaction writes to `listA`; otherwise the transaction continues as normal.

- *No-throw guarantee*: The method in question will never throw an exception.
- *Strong guarantee*: If the method throws an exception, the state of the object will be the same as before the method was called. No resources are leaked.
- *Basic guarantee*: If the method throws an exception, the state of the object will be in a valid state after the method finished. No resources are leaked.

1.4 Software Transactional Memory

Software Transactional Memory is a concurrency control mechanism used to control access to data in shared memory. As with database transactions it allows the programmer to specify that a block of code that reads and/or writes shared data must execute independently of other transactions, such that it either completes entirely or has no effect. Simultaneously executing transactions will not see any intermediate states of other transactions, nor will they see inconsistent views of data.

In Figure 1.1 we show a simple example in pseudo-C++ of a transaction atomically removing an element from one list, and adding it to another; the intermediate state where the element is in neither list, is never visible to other transactions. In Figure 1.2 we show the use of the `retry` keyword, allowing a transaction to wait until the list becomes non-empty. In Figure 1.3 we show an example of nesting one transaction within another. In Figure 1.4 we show the `orelse` construct, letting a transaction run one of any number of different blocks; effectively letting the different code blocks be different alternatives. In Figure 1.5 we show a proper code example of a transaction reading one shared variable and writing another.

1.4.1 Programming with STM

STM has only a few programming constructs. We discuss these here. Only the first two are needed for a simple STM implementation; the last four however add some additional functionality.

Starting and stopping transactions

A transaction must begin and end at well defined points. Typically this is done by letting the scope of a transaction coincide with the scope of a block of code; instead of e.g. an `if`-block the block is

```
1  void f() {
2    atomic {
3      int x=listA.front();
4      listA.pop_front();
5      listB.push_front(x);
6      g();
7      listC.pop_front();
8    }
9  }
10 ....
11 void g() {
12   atomic {
13     listC.push_front(42);
14   }
15 }
```

Figure 1.3: *Nesting:* The function `g` which itself starts a transaction, can be called inside another transaction; all the changes from `f` and `g` are executed as one larger transaction; the changes made in `f` are visible in `g` and vice versa. When the inner transaction commits, its changes become part of the outer transaction, and not globally visible unless and until the outer transaction commits.

```
1  atomic {
2    atomic {
3      if (listA.empty())
4        retry;
5      //Do something with the contents of listA
6    } orElse {
7      if (listB.empty())
8        retry;
9      //Do something with the contents of listB
10   }
11   listC.push_back(42);
12 }
```

Figure 1.4: *OrElse example:* A transaction *either* processes `listA` *or* `listB`; in either case, `listC` is updated. If both `listA` and `listB` are empty, the transaction blocks and is automatically restarted when another transaction writes into either `listA` or `listB`.

```
1  ....
2  shared<int>* hx=...;
3  shared<int>* hy=...;
4  ....
5  estm::transaction tx;
6  ATOMIC_BEGIN(tx) {
7     const int& x=hx->openRO(tx);
8     int& y=hy->openRW(tx);
9     y+=x;
10 } ATOMIC_END(tx);
```

Figure 1.5: *Example of a transaction using our implemention:* `hx` and `hy` are handles to shared elements of type `int`. `tx` holds the private transaction data, used to keep track of the transaction. `hx` is opened for reading, and a const reference to its contents is returned (from the `openRO` call). `hy` is opened for writing, and a non-const reference to its contents is returned. When the transaction is completed, the value pointed to by `hy` has been increased with the value pointed to by `hx`. The `ATOMIC_BEGIN` and `ATOMIC_END` are preprocessor macros, containing enough logic to start and stop the transaction, and restart it in case it is aborted.

typically called an `atomic`-block. Code run within such a block either completes atomically with respect to other threads of execution, or is aborted without making visible changes.

Opening shared data

When a transaction attempts to access a piece of shared data, we say that the transaction *opens* the data — depending on the programming language and compiler this can either be done explicitly (with a function call) or implicitly (the compiler in effect inserts a function call into the code). We distinguish between opening an object for reading or writing. In Figure 1.5 we show code explicitly opening for reading and for writing.

Transaction nesting

While not a part of all STM systems, *nesting* transactions is a common useful feature. The most useful variant enables an *inner* transaction to be started while another *outer* transaction is in progress; the inner transaction sees the speculative changes made so far by the outer transaction. When the inner transaction commits, its changes are not made globally visible; instead they are added to the speculative changes of the outer transaction. Aborting the inner transaction also aborts the outer transaction.

Retry

Like transaction nesting the *retry* primitive is not a part of all STM systems, but allows a transaction to abort itself and block until some of the variables read by the transaction have been modified by another transaction; once this happens, the transaction is restarted. This feature allows transactions to signal one another, without having to deal with semaphores, events or condition variables. In particular, the transaction writing to the waited upon variables does not have to be specifically constructed to signal the other transaction.

OrElse

Combining the retry and nesting primitives with the *orElse* construct allows the programmer to create transactions that execute one of several alternative blocks of code: each block in turn checks that its preconditions (if any) are met — if they are not, the block *retries*, its changes

are discarded, and the next block is tried instead. If no block's preconditions are met, the whole transaction is aborted; the transaction is restarted when one or more of the read variables are changed by another transaction. See Figure 1.4 for an example.

Privatization

Using transactions to access shared data has some overhead. To avoid (some of) this overhead in circumstances where some shared data is known not to be accessed concurrently, some STM implementations support *privatization*, which is a mechanism for "cutting out" a set of shared data and making it private to a thread. After privatizing such data it can then be worked on with very little overhead, and the altered data can later be unprivatized. We will not discuss this feature any further.

1.4.2 Semantics

ACI properties

Transactions and associated semantics have been used in databases for a long time (for example, see [19]), and STM is based on these ideas. While "real" databases have the ACID properties, STM systems only have the ACI properties, as the effects of transactions are not durable, since the transactions only effect volatile main memory. We discuss the meaning of these with respect to STM here:

- *Atomicity:* Transactions either complete in their entirety, or have no effect at all. Some STM systems actually violate the atomicity property if the data types used in transactions do not provide assignment operators that give the no-throw guarantee; our implementation *does* have the atomicity property even when used with such types.
- *Consistency:* The consistency property in STM means that memory and data structures are in valid state after a transaction either commits or aborts; however, such guarantees only apply to shared data controlled by the STM system, and only at the most basic level: only individual objects are guaranteed to be consistent. No high-level integrity constraints can be specified via the STM system.
- *Isolation:* Operations by one transaction on shared data is never visible in its intermediate stages by another transaction. Again, this applies only to data controlled by the STM system. More specifically, for STM systems the transaction history or schedule is *conflict serializable*. While it is common for relational databases to allow weaker forms of isolation for performance reasons, STM implementations usually do not, as it makes reasoning about transactions difficult, in particular because data structures manipulated via STM are not limited to relational tables.

Input/output and side effects

None of the ACI properties deal explicitly with input/output (IO) nor manipulating non-STM controlled data. We refer to both IO and non-STM controlled data manipulation as *side effects*. Performing side effects during a transaction will normally violate at least the AI properties, since the transaction may be aborted or restarted at almost any time. Some implementations (e.g. Haskell's [11]) make it impossible to perform IO or alter non-STM data during a transaction. Other implementations, including ours, simply make the programmer responsible for avoiding such behavior.

Exception-safety guarantees

The atomicity property also implies that data structures built with transactions give the strong guarantee. This means that the strong guarantee of exception safety is provided almost effortlessly to the programmer implementing a data structure with STM. Not only does the individual

operation on a data structure give the strong guarantee — if run within a transaction multiple operations on multiple data structures give it as well. This means that STM may have some use even in a single-threaded environment.

Chapter 2

Approaches to STM

We gave a high-level overview of STM systems in Section 1.4. In this chapter, we will discuss the main design parameters of a STM system, and determine the proper choices for our STM implementation.

2.1 Granularity

The first choice we must make for our STM system is the granularity at which it operates; by this we mean the granularity of the individual pieces or units of transactionable data. STM systems are typically divided into two classes: word-based and object-based. However, the term "object-based" is somewhat overloaded, and we will discuss its different meanings in more detail.

2.1.1 Word-based granularity

The *word-based* granularity is perhaps the easiest to understand: when we read or write a piece of data via the STM system, this data occupies a machine-word. The STM system normally does not associate any type or semantics with the contents of the machine-word, and altering or restoring the contents are simple (and typically atomic) machine instructions. This makes the STM implementation quite simple; at least in this regard. In addition, the simplicity can help make the word-based approaches very fast.

However, the word-based approach has some disadvantages: since the STM associates no semantics with the contents of the words, the word-based approach is only of limited value in an environment without garbage collection. If we, for instance, decide to store a string via the STM system, we could store a pointer to the beginning of the string via the STM system. However, if we in a transaction need to update the string, the STM system itself has no functionality in place to help us with the *contents* of the string. Instead the user would have to allocate a new string during the transaction, and ask the STM system to replace the pointer. When the transaction is committed or aborted, we need to have a mechanism in place for freeing either the old or the new string; a word-based STM does not by itself supply such functionality.

2.1.2 Block-based granularity

The *block-based* granularity is quite similar to the word-based granulairy. Each transactionable piece of data is a block of some fixed individual size. As with the word-based granularity, the block-based data has no associated semantics in the STM (in C++ parlance it would be structs or POD-types (plain old data) that could be safely `memcpy`'ed). The Hardware Transactional Memory (HTM) implementations (discussed in Section 2.6.1) are typically block-based, e.g. that described in [17].

The advantages and disadvantages are much the same as for the word-based granularity. However, updating the contents of a block-based entry can usually not be done with an atomic machine

operation (except, of course, when using HTM). In STM implementations, this may complicate matters a lot, compared to the word-based approach, without giving any appreciable advantages apart from a relatively low overhead when the blocks are larger than one word.

The block-based granularity is occasionally referred to as object-based.

2.1.3 Object-based granularity

The *object-based* granularity deals with objects. The objects may have differing size, non-trivial copy constructors and destructors; therefore the objects cannot be `memcpy`'ed. The advantage of this approach is that the objects can own other data, and by using the object's methods for copying and destruction the object itself manages its data, and maintains its invariants.

Depending on the transaction method we use (see Section 2.3), an object may need to be "overwritten" when we commit a transaction; we cannot simply overwrite the object's memory, but the object must instead support assignment. Just as for the block-based approach, during assignment to an object, its invariants may be broken temporarily.

Runtime polymorphism

When an object is opened for writing, we need to create a copy of the current value of the object (independent of the transaction method). Most, if not all, C++ implementations are based on runtime polymorphism, such that every shared object has a virtual function `clone`, that the STM system calls when the object must be copied[1]. The object (or the associated metadata) also has a virtual function for destroying the object. The runtime polymorphism simplifies the STM implementation, since most of the implementation does not have to know about the types of the objects stored; typically only the "frontend" or API of the STM implementation knows the type of the objects, and this is only a convenience for the programmer whose scope for type errors is reduced.

RSTM [18] uses an intrusive scheme, such that every type to be used with the STM system must inherit from a RSTM interface; this makes reusing existing types cumbersome, as they must either be altered, or wrapped in a helper class. DracoSTM [9] uses the helper class (in the form of a class template) directly, and thus makes reusing existing classes easier.

However, the use of virtual functions means that the object (or its metadata) needs to hold a pointer to a vtable[2], and this increases the size of every object. In addition, the use of runtime polymorphism implies that the STM system cannot know nor use the size of the objects, and some interesting memory layout techniques are impossible. Also, the virtual function calls take additional CPU-cycles, and may inhibit some optimizations that the compiler could have performed if it knew the type of objects being copied or destroyed.

Compile-time polymorphism

We introduce a new approach for implementing STM systems, at least within C++, which uses compile-time polymorphism. We have three motivations for this approach: first, we wish to remove the vtable pointer that induces space overhead for every shared object; second, we wish to avoid some virtual function calls and their overhead; third, since we know the type of objects, we also know their sizes, and can use this to lay out the objects in memory in advantageous ways. The layout options will be discussed in more detail in Section 3.7.

The basic insight to this approach is that the type of an object is available when the object is opened for reading, and more interestingly, for writing. When we clone the object, we can now use the type's copy constructor, and thereby avoid the virtual `clone` method. We also remove the virtual method for destroying the object. However, this leaves us with a problem when we

[1] The virtual function does not need to be on the object itself — it may be part of the metadata stored with the object.

[2] A *vtable* or *virtual method table* is part of the mechanism used for calling the correct function at runtime, and is a common mechanism for implementing virtual functions in C++.

wish to delete the object (which happens after the transaction has committed or aborted), since we do not at this point have access to the object's type. Instead, each transaction that opens an object for writing, also stores a function pointer for deleting the same object, in the transaction's metadata. This way, we no longer have to store a pointer per object, but only a pointer per opened-for-writing object, and only for the duration of the transaction. This method also halves the number of indirect function calls.

Compile-time polymorphism is non-intrusive, and in fact, any type that can be used with STL containers, can be used directly in this approach. Should we wish to use runtime polymorphism with this approach, we can do so by storing a smart-pointer that clones its pointee when the pointer itself is copied.

Our implementation will use compile-time polymorphism and object-based granularity.

2.2 Validation

As a transaction executes, it will need to open shared objects for reading or writing. Whenever a new object is opened, there is a chance that the new object will have been modified after the transaction began and that some of the already opened objects have been modified as well. If the transaction makes no effort to detect changes to opened or about-to-be-opened objects, the transaction may see an inconsistent view of memory. To avoid inconsistent views, a transaction must *validate* the objects read or written by the transaction. To do so, the system must support some means of identifying different versions of the same objects; we will discuss some methods of doing this in Section 2.2.2.

Alternatively, the system can use *invalidation*, where a transaction that modifies an object invalidates (i.e. aborts) all running transactions that have opened the object.

Since we obviously need to perform validation before committing a transaction, we could conceivably skip validation as we go, and accept that some transactions would continue work even though they are doomed to abort. However, as we discuss next, such "zombie" transactions cause unacceptable problems.

2.2.1 The need for validation

Letting a transaction run with an inconsistent view of data may cause three distinct problems that we need to avoid. We discuss these problems and how they may arise in the following.

Infinite loops

In Figure 2.1 we show a (contrived) example, taken from Harris and Fraser [10], of how an inconsistent read can lead to an infinite loop. Assuming x and y are equal before either transaction runs, neither transaction T_a nor T_b alter this invariant. However, if T_a reads x before T_b completes, and y after, the invariant will appear broken, and the program will enter an infinite loop.

Several authors [11, 8] have suggested not avoiding the problem, but instead handling it, by having a mechanism in place for periodically checking for doomed transactions and forcibly aborting them. While this may work in a managed programming environment, there is no portable way to do so in C++; it would require either letting the compiler insert special code in every potentially infinite loop, or terminating the thread running the doomed transaction. The latter option includes many problems, in particular cleaning up the stack left behind by the terminated thread.

Unintended side effects

Marathe et al. [18] give an example of how an inconsistent read could cause a function with side effects to be called instead of function with no side effects. In Figure 2.2 we show an example of how such code might look.

Transaction T_a

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

Transaction T_b

```
atomic
{
  ++x;
  ++y;
}
```

Figure 2.1: *Inconsistent read:* Example of inconsistent reads causing infinite loop in transaction T_a. Provided x=y initially, transaction T_b would not change this, but if transaction T_a reads x before T_b writes x and T_a reads y after T_b writes y, transaction T_a would see the inconsistent view and enter the infinite loop. This example is taken from Harris and Fraser [10].

In essence, the example relies on an inconsistent read causing the program to call a virtual function with side effects on an object, based on a boolean flag whose value is based on another object.

Segmentation faults and memory corruption

Another type of problem arises when a transaction reads inconsistent data, and based on the inconsistent values accesses memory addresses that are not valid for the program; e.g. two objects are read, the first containing a pointer to a buffer of data, and the second an offset into the buffer; if the two objects read are not consistent, the program may try to read beyond the end of the buffer. This may have no harmful effect (the address is valid), or the program may experience a segmentation fault. Some authors [7] suggest handling the segmentation fault and aborting the transaction. To perform cleanup of stack variables and the transaction, the only feasible way to do this is to convert the segmentation fault into a C++ exception. However, these segmentation faults may occur almost anywhere, and do constitute undefined behavior. The compiler may have optimized out some exception handling code that does stack unwinding, if the compiler can prove the code is exception free (under the assumption of no undefined behavior). Handling segmentation faults is then not a viable solution.

We have looked at how inconsistent reads of different objects can cause problems. Some STM systems allow the reads of the individual object to be inconsistent, by letting a transaction read an object as it is being updated. We will refer to these types of inconsistent reads as *dirty reads*. Only the log-based update systems (discussed in Section 2.3) are at risk of dirty reads.

Every problem that can arise with the normal inconsistent read can occur with dirty reads as well. Several STM implementations avoid inconsistent reads, but do allow dirty reads; these dirty reads are simple race conditions, and while they probably occur very rarely, they do exist.

In Figure 2.3 we show how a dirty read can cause memory corruption. A transaction T_a reads part of object `x`, another transaction T_b writes to `x` (in the redo case while committing, or in the undo case after locking the object and performing speculative writes), and finally T_a reads another part of object `x`. In effect T_a sees an inconsistent view of the single object `x`, and this causes T_a to overrun a newly allocated buffer, and corrupt the program's memory. The program has no way to recover from this corruption, and may not even detect it immediately.

While the memory corruption can happen with a normal inconsistent read, the dirty read scenario is in some sense even worse, since the dirty reads induce errors into existing correct methods. Avoiding the dirty reads is strictly speaking not a part of the validation problem, but since its failure modes are much the same, they have been introduced here.

```
class P
{
  ....
  virtual void m()=0;
};
class C1 : public P
{
  ....
  int i;
  void m() { ++i; }
};
```

```
class C2 : public P
{
  ....
  void m() { LaunchMissiles(); }
};
```

Transaction T_a:

```
atomic
{
  if (x)
    y->m();
}
```

Transaction T_b:

```
atomic
{
  x=false;
  y=getC2instance();
}
```

Figure 2.2: *Inconsistent read:* Example adapted from Marathe et al. [18]. Object `x` is a boolean variable indicating if we can safely call virtual method `m` on object `y` (which is (a pointer to) some subtype of `P`). Transaction T_a reads `x`, T_b commits both `x` and `y`, and T_a then reads `y` and calls `y`'s method `m`. While transaction T_a is doomed to abort, it has caused irreversible effects.

2.2.2 Validation methods

There are quite a few schemes for validating transactions, and many small variants within each scheme, in particular with respect to the intersection of the validation with other pieces of the system. In the following we discuss some of the more common approaches, leaving out most of the details in the process. The first approach is actually an invalidation approach.

Visible readers

In the visible readers approach every shared object has a list of readers as part of its metadata. When a transaction T_a first opens an object for reading or writing, T_a adds itself to the object's reader list. When another transaction T_b commits, it aborts (invalidates) all transactions listed as readers of objects written by T_b. When a transaction commits or aborts, it removes itself from all the reader lists it had added itself to.

The primary advantage of this method is that a transaction does not need to validate its own readset when the transaction commits or opens new objects; it only needs to check that is has not been aborted by another transaction.

An additional advantage is that a doomed transaction T_a may be aborted sooner; since another transaction T_b can see readers that conflict with T_b's own writes, T_b may decide to abort T_a long before itself committing; such decisions may be taken by a contention manager, which we discuss further in Section 2.4.

A disadvantage is that the reader lists need to be stored somewhere; the list should be adjacent to the object itself for performance reasons, but this takes up space. We do not know in advance the maximum required size of the list, even though we expect the list to be empty or close to empty most of the time. We thus need to either reserve space for an prohibitively large number of readers, or be able to deal with the list becoming full. This requires either dynamically allocating space for a larger list, or falling back to another mechanism when the list is full. In addition we need a mechanism for concurrently updating the list; this is not trivial.

A perhaps larger disadvantage is that the reader list effectively turns every read into a write;

```
1  class string
2  {
3  public:
4  .....
5    string(const string& other)
6    {
7      mStart=new char[other.mFinish-other.mStart];
8      mFinish=mStart;
9      char* pos=other.mStart;
10     while (pos!=other.mFinish)
11       *mFinish++ = *pos++;
12   }
13   char& operator[](int pos)
14   {
15     return mStart[pos];
16   }
17 private:
18   char* mStart;
19   char* mFinish;
20 };
```

Transaction T_a	Transaction T_b
`atomic`	`atomic`
`{`	`{`
`....`	`....`
`  string copy(s);`	`  s[10]='␣';`
`}`	`}`

Figure 2.3: *Dirty read:* Transaction T_a opens `s` for reading, enters `string`'s copy constructor and allocates memory based on the value of `s`'s `mStart` and `mFinish`. It reads `s`'s `mStart` in line 9, and is preempted. Transaction T_b opens `s` for writing, copies it to its redo log, makes a change to its own copy, and during commit writes back to `s` via an assignment operator (not shown), which means that the `mStart` and `mFinish` values change. T_a, which is now reading inconsistent data, proceeds to copy the contents but its termination condition (line 10) is based on the old value of `s`'s `mStart` and the *new* value of `s`'s `mFinish`, causing the termination condition not to fire, thus overflowing the newly allocated memory, and corrupting whichever data follows the buffer. A similar example with an undo log is easily constructed.

this may cause excessive cache line bouncing in cases where an object is read by many transactions; e.g. the root node of a red-black tree.

RSTM uses visible readers, and in case of list overflow it falls back to per object validation, discussed next.

Per object version numbers

The *per object version numbers* approach assigns a version number to every object. Whenever an object is modified, its version number is increased when the transaction commits. Every running transaction holds a private list of opened objects and the version number each had when it was first opened. Validation can then be done simply by iterating over the transaction's list of opened objects and checking if the stored version numbers match the objects' current version numbers; if they do not, the transaction must be aborted.

As discussed earlier, every time a transaction opens an object for the first time, validation has to be performed to avoid inconsistent reads. Unfortunately, this means that the number of validations is linear in the number n of objects to be opened, and the total number of version numbers to be checked is thus $O(n^2)$, which is unacceptable for large values of n.

Global version number

Dice et al. [6] introduced the concept of a *global version clock*, which uses a global version clock, to assign version numbers to objects. Leaving out the details, the idea is to at the beginning of a transaction T_a read the current global version clock and save this number rv (read-version) in the transaction's private metadata. Whenever T_a opens an object O_1, it examines O_1's version number v_{O_1}, and if $v_{O_1} > rv$, the object O_1 has been written to after T_a began, and by continuing T_a would (possibly) see inconsistent data; T_a can then abort itself. When a transaction commits, it first runs through its list of objects and confirms that they are still current (while its view of data may have been consistent, some of the objects may have been updated since they were read by the transaction); it then increments the global version clock; finally all the objects written by the transaction are assigned the new value of the global version clock.

If a transaction is known a priori to be read-only, it is not even necessary to store the read list in the transaction; only rv is required.

Getting back to the case of opening an object in a transaction: as an extension, if $v_{O_1} > rv$, instead of aborting the transaction immediately, we can instead check all objects read by the transaction to make sure they are still current; if so, our view is still consistent, and we update rv to the value of v_{O_1} and continue the transaction. If this happens every time an object is opened, the total time spent opening objects becomes $O(n^2)$, as for the per object version numbers; however, this scenario is quite unlikely: it requires that before every object opening, another transaction commits and updates the object, but none of the other opened objects.

While Dice et al. describe the method in a lock-based setting (we discuss the different update methods in the next section), the method itself does not require such a setting. In fact, in an indirection-based approach to object-updating (indirection is described in Section 2.3.3), where old versions of objects may be available, we may have long running read-only transactions that read the older versions of objects written by other transactions; effectively "snapshotting" the shared data.

Because of the need for never seeing an inconsistent view of data, and that the global version clock approach makes this efficient in the common case, we will use this approach in our implementation.

2.3 Update method

STM systems differ in how they implement writing of shared data such that writes can be rolled back in case a transaction aborts. Common to all of them is that they associate some metadata with every piece of shared data. The type and amount of metadata differs for different STM

systems. However, the point of the metadata is in all cases to ensure that every transaction when it commits has seen a consistent view of memory, and to ensure that both commits and aborts are completed properly.

Three approaches are common: undo logging, redo logging and indirection. They are described in the following along with their advantages and disadvantages. Note that the descriptions only give a high-level view and leave out significant details.

2.3.1 Undo logging

Every object has a lock (often implemented as a bit that can be set to indicate that the object is locked) and normally some metadata describing the current version of the object. In addition the object may have some data identifying the transaction which owns the lock.

When an object is opened for writing, the transaction attempts to lock the object. If this fails, the transaction either aborts or waits for the object to be unlocked. Otherwise the current value of the object is written to the transaction's *undo log*, and new values can be written to the object.

If the object is opened for reading, the lock is not taken; however, the transaction will not proceed while the object is locked by another transaction. When the object is not locked, the object's version data is recorded in the transaction, depending on the validation method.

When faced with a locked object, a transaction can, instead of waiting or aborting itself, choose to abort the transaction that owns the locked object; this would normally involve a contention manager, described in Section 2.4.

Since changes are made directly to the global object, we have to lock the object immediately when we open it for writing; this is referred to as *eager acquire* or *encounter-time locking*.

Committing a transaction is simple: The transaction runs through its list of read objects, and verifies that every object is still the same version that the transaction read. If this is the case, the version numbers of all locked objects are updated (again, depending on the validation method), and the locks released. The undo log is simply discarded. If any read object has changed, the transaction aborts.

Aborting the transaction is done by restoring all the values stored in the undo log to their respective objects, and releasing all the locks.

The primary advantage of using undo logging is low overhead; the object can be stored along with all of its metadata, and therefore not spread out among several cache lines (which, as Ennals shows [7], is good for performance). In addition, very little work has to be performed to access the stored data (since there is no indirection). Assuming more commits than aborts, the total overhead is smaller than for redo logging (described next), as the overhead in a successful transaction almost only consists of the initial copying of the objects to the undo log.

Disadvantages include the risks of deadlock and livelock (depending on whether the transactions wait or abort when faced with a locked object). Mechanisms have to be put in place to detect and handle deadlock situations, and some work has to be done to ensure the solution does not become livelock prone instead.

Aborting a transaction requires a lot of work, since the old data has to be copied back; this means that if a transaction T_a causes another transaction T_b to abort, it has to wait for T_b to undo its work, before T_a can continue. Alternatively, some (typically non-trivial) mechanism can be put in place to allow T_a to undo some of T_b's work.

However, the largest problem is the race conditions caused by the lack of reader locks; since no lock is taken when opening an object for reading, a committing transaction may overwrite the data that another transaction is reading; the problems with this were described in the validation section. A solution to this problem is to verify every memory read of the object, immediately after the read, to ensure that the object has not been altered by another thread (this is the approach used in RSTM for their redo log backend). This either requires to programmer to call the extra validation code (done via a macro in RSTM's implementation) or for the compiler to automatically insert these extra checks. While the manual method of RSTM is feasible, reusing existing types without altering their implementation is impossible. The compiler case, as tempting as it seems, is however insufficient, as it introduces new failure points into the code; these failure points may

not be covered by exception handlers. Letting the compiler insert these checks into the code in the example in Figure 2.3, would instead cause the copy constructor to leak the memory, since the implementation assumes that the only thing that can fail is the initial memory allocation itself. The insertion of these validation checks would also inhibit some compiler optimizations that would otherwise be possible; the checks themselves also incur some overhead.

2.3.2 Redo logging

Redo logging is quite similar to undo logging; however instead of writes happening directly to the public object, objects are copied to a private *redo log*, and the speculative changes happen to the private versions of the objects. When the transaction commits, the contents of the redo log is written back into the public version.

At commit time the modified objects have to be written back ("redone") and hence have to be locked; however, the locking can either happen at encounter time or be delayed to commit time; the latter is called *commit-time* locking or *lazy acquire*.

In order for read-after-write and write-after-write to work properly, every open call must first consult the redo log to see if the transaction has a local copy of the object; implementing this lookup mechanism is critical to good performance for redo logging.

Committing a transaction consists of locking all the objects that we have opened for writing, verifying that every object opened as part of the transaction is still the same version, and finally writing out the values from the redo log to the objects, and unlocking all the locked objects.

Aborting the transaction is very simple: the redo log is simply discarded, and any eagerly acquired locks are released. This means that, unlike redo logging, aborting another competing transaction is quite simple to implement and has low runtime overhead.

Redo logging is a common choice for STM implementations; a redo logging backend has been added to the newest version of RSTM; DracoSTM uses either redo or undo logging; LibLTX uses redo logging — high performance and the possibility of easily aborting other transactions are the main reasons for this. However, redo logging has the same problems with dirty reads as undo logging, and we will for the same reasons not use either in our implementation.

2.3.3 Indirection

The *cloning* or *indirection* based approach is quite different from both redo and undo logging, and there are several quite different subapproaches (see e.g. [18, 11, 13]). However, the main idea is to let the object's metadata have a pointer to a version of the object. See Figure 3.2 on page 33.

When a transaction first opens an object for reading, it reads the object's metadata, which contains a pointer to the current version (the last commited version) of the object. This pointer is stored in the transaction's readset. When an object is first opened for writing, the (pointer to the) current version is found and the current version copied into a transaction private version, which is added to the transaction's writeset, and the private version is then worked on speculatively. When a transaction opens an object, it must of course first check the writeset to see if a private copy exists.

When the transaction commits, it first runs through its read and write sets, and verifies that none of the objects' metadata's pointers have changed; it then commits by atomically changing the metadata's pointers to point to the transaction's own updated version of the objects. Aborting the transaction is just a matter of discarding the private copies.

Verifying the read- and writesets and atomically updating the pointers is a challenge of this approach, and many different schemes exists for doing so. Some of these approaches lock (some of) the objects (by having a lock variable along with the objects' metadata) during the updating; other let only one transaction commit at a time. Most of the approaches rely on the existence of a compare-and-swap operation to change the pointers atomically, and by doing so, avoiding some amount of explicit locking.

By combining this approach with the global version clock approach for validation, we can guarantee that a transaction never performs inconsistent or dirty reads, which, as we have seen,

in a non-managed language like C++ is a necessity. While this can be achieved with the other approaches at the cost of constant validation of almost every read, and by breaking encapsulation, it is inherent to the indirection approach.

The indirection approach also gives the transactions the strong exception guarantee; since no existing data are modified during the transaction, it is only at commit time, when all the changes are ready, that state is altered. This can be a very valuable property, as it makes reasoning about programs' failure modes easier. Redo and undo logging can only give this guarantee if all the types used in the transaction have assignment operators that give the no-throw guarantee. With the indirection approach, if a copy constructor throws an exception while we are creating a local copy of an object for speculative updates the transaction fails; however, the global state is unaltered and the transaction has made no changes that need to be rolled back.

The disadvantage is the cost of the indirection; we must traverse the pointer to the current object, which itself has a small performance cost; worse is the high probability that the object itself resides in a different cache line than the metadata. This may double the number of cache misses, which in a memory bandwidth constrained application may come close to halving the program's performance. For this reason, most new STM implementations are avoiding the indirection approach.

Another disadvantage is related to memory management; the other approaches use a transaction-local buffer for all objects written, and since their private copies are never seen by other threads, and are discarded all at once, they may use special memory allocators optimized for this behavior; the indirection approach makes the local copies public at commit time, and the objects allocated must be deallocatable individually and by any thread.

A third disadvantage is that the amount of metadata is higher for the indirection approach; in addition to the "common" metadata consisting at least of a pointer to the current object, most implementations store per-copy metadata as well.

However, the indirection approach is the only one that can allow us to reuse existing types unaltered and correctly in our STM implementation, and we choose it for this reason. In order to reduce the cost of cache misses and allocation overhead of the indirection approach, we introduce a new technique allowing us in the common case to be able to store objects in the same cache line as their metadata, provided the objects are of limited size. Due to our use of compile-time polymorphism, we know the size of the objects, and can lay them out in memory, such that we alongside the metadata have room for storing n objects. Provided $n > 1$, this is sufficient for updating an object, without having to store either version in a different cache line; however, since we cannot immediately delete old versions (since running transactions may be reading them, see Section 3.7.2), we may sometimes need n to be larger than 2. In such cases we fall back (at runtime) to allocating new space from the normal heap (and accept that it happens in a different cache line); in practice, as we show in Chapter 5, the need to allocate extra space happens rarely provided there is little contention.

This approach then trades a slightly higher memory use for the benefits of correctness, reusable code, while getting some of the performance benefits of the logging approaches. Section 3.7 explains the details of the approach.

2.4 Contention management

When an STM system uses lazy acquire and invisible readers, a transaction T_a may run until it starts acquiring objects, only then to discover that it must be aborted because another transaction T_b has updated some of the objects read or written by T_a. Using eager acquire we may discover some write-write conflicts before either transaction has acquired. When a transaction discovers such a conflict, it has two options for resolving it: it can either abort itself, or the competing transaction. Likewise, if a STM systems uses visible readers, a transaction that writes an object may find itself to conflict with one or more readers of the object; here there is again a choice of which transaction to abort.

STM implementations use a *contention manager* to make this decision. They may base their

decision not only on the immediate conflict, but on the relative priorities of the involved transactions, the amount of work "lost" by aborting a given transaction, the number of times a transaction has been aborted etc. In addition the contention manager may let the aborted transaction wait a while before restarting to avoid the transaction restarting and immediatly achieving the same conflict (and in the process causing needless contention). Scherer and Scott [21] discuss different contention managers.

Our implementation presently only discovers conflicts at commit-time, and a transaction can only abort itself; this is basically a very naive contention manager.

2.5 Priority inversion

Priority inversion is a well-known phenomenon within normal lock-based programming, and can occur if a low-priority thread holds a lock on a resource that a high-priority thread needs to access. The high-priority thread will then have to wait on the low-priority thread, thereby "inverting" their priorities.

In STM systems, depending on their implementation, the contention manager can be used to let a high-priority transaction abort a low-priority transaction that it has conflicts with. However, short low-priority transactions may perform writes that cause a longer running high-priority transaction to abort later. Some STM implementations (e.g. DracoSTM [9]) deal with this by invoking the contention manager just before a transaction commits, and can use this to detect and avoid priority inversion that could result by completing the commit.

We do not consider priority inversion further.

2.6 Related/previous work

The idea of Software Transactional Memory is not new — Shavit and Touitou introduced the term in 1995 [22].

Recently Harris et al. [11, 12] introduced the primitives `retry` and `orElse`, along with their STM implementation in Haskell. One of the significant advantages is that their implementation statically (via Haskell's type system) guarantees that only memory operations are performed, such that programmer cannot, even by accident, write programs that do not have ACI properties.

In 2006 Dice et al. [6] introduced the *global version clock* approach to transaction validation, based on the *lazy snapshot algorithm* of Riegel et al. [20]. We use a variant of their approaches.

2.6.1 Hardware-based approaches

To reduce the overhead of the transactions, some attempts have been made to design hardware transactional memory (HTM) (e.g. [17]). HTM normally uses cache lines as the level of granularity, adding additional metadata to each cache line to indicate its use and state in a transaction. One problem with HTM is unnecessary conflicts due to false sharing (more than one data element per cache line). In addition since the cache is used to hold the speculative values, problems arise when the transaction size outgrows the cache size, or when the distribution of the memory addresses fill a cache line set, due to the set associativity of the cache. Long running transactions will also have a high probability of being preempted, leaving the question of if/how to store the current state of the transaction.

Lie, in his thesis [17], describes an extension of the MIPS instruction set, which, by requiring operating system support, can use main memory when the cache would otherwise overflow; his design thus supports arbitrarily large transactions.

Sun Microsystems have been working on a hybrid approach. HTM is used as the "default" and the system falls back on STM if the HTM approach fails [5]. Hybrid transactional memory is set to debut in Sun's "Rock" processor [25].

2.6.2 Existing implementations for C/C++

The following contains a discussion of some of the existing STM implementations for C/C++.

RSTM

The Rochester Synchronization Group at the Department of Computer Science at the University of Rochester has released several versions of their STM implementation called RSTM [18]. RSTM is object based and indirection based, though the newest version has optional support for redo logging. The implementation uses visible readers for validation checking, and can use either eager or lazy acquire. The use of visible readers in effect turns every read into a write, thus hurting performance. RSTM has no support for the `retry` and `orElse` primitives. RSTM does support privatization. RSTM requires all types used in transactions to be derived from a specific base class, making reuse of existing code harder.

LibLTX

The Lightweight Transaction Library (LibLTX) is discussed by Ennals [7]. For the stated cause of good cache locality, the indirection approach is not used in LibLTX. Instead redo logging is used. LibLTX is object based and has only the most rudimentary STM primitives. The main problem with the library is that transactions may see inconsistent views of data.

DracoSTM

DracoSTM [9] is a quite new STM library. It is object based, and offers both undo and redo logging. It supports nested transactions. The authors claim it is significantly faster than the RSTM library. It seems DracoSTM avoids transactions seeing an inconsistent view of individual objects as the objects are being updated by blocking other transactions while an update is in progress; the mechanism is however not explained in detail. Unlike RSTM, DracoSTM supports use of existing classes directly.

Chapter 3

ESTM

In the following we present the details of our STM implementation (named "ESTM" in a moment of vanity). We first present the API. We then expand on the global version clock approach that we discussed earlier, and show that the transactions are conflict serializable. We then discuss how the transactions are implemented, including how we implement the layout technique we mentioned earlier. Finally we discuss some practical details of the implementation.

3.1 API

3.1.1 General design considerations

In designing the API for our STM system, we have focused on making the API strongly typed, such that e.g. operations for creating or opening objects return a reference with the exact type of the object. Given that we use compile-time polymorphism internally in the implementation, the API naturally reflects the exact types.

In addition, we believe the use of transactions should be explicit, such that if a method is meant to be called only as part of a transaction, its type signature should reflect this fact. This prevents at compile-time calling such methods without actually being in a transaction. While it would be possible to let such methods automatically start a (nested) transaction when called, doing so could cause surprises for the programmer if he is not aware that the data manipulated by such a method are shared. This would mean that a sequence of operations may not be correct if using such implicit transactions. Consider:

```
1 estm::list<int>& l = ...;
2 l.push_back(42);
3 assert(!l.empty())
4 l.pop_back();
```

The assertion in line 3 may not be true, since another transaction may have removed the element. This basically reintroduces race conditions into the program; these are the same race conditions the use of transactions was to meant to avoid. The correct solution is to clearly specify where transactions begin and end, and to only allow the above methods to be called during a transaction. In this implementation we do so by requiring all methods that work on shared data to accept an extra argument, which is the private transaction descriptor (`transaction` — discussed in Sections 3.1.3 and 3.4.3):

```
1 estm::transaction tx;
2 ATOMIC_BEGIN(tx) {
3   estm::list<int>& l = ...;
4   l.push_back(tx, 42);
```

```
5     assert(!l.empty(tx))
6     l.pop_back(tx);
7   } ATOMIC_END(tx);
```

The `ATOMIC_BEGIN` and `ATOMIC_END` macros (discussed in Section 3.1.3) make use of the private transaction descriptor to start, stop and restart transactions as needed.

3.1.2 Constraints on types used in transactions

Standard library containers impose a few constraints on the types they work with in order for the container's methods to provide their documented exception guarantees (we discussed exception safety guarantees in Section 1.3.2). Though some template parameters specify e.g. memory allocators used by the containers, we are only interested in the parameters that specify the type or types stored in each node of the containers. Though some containers, e.g. `std::map`, store more than one type (the key and the value types can be different), the constraints apply to all the types. The following discussion applies to all the types stored.

The constraints on standard library containers are in all cases that the destructor of the type `T` gives the no-throw guarantee. In addition, the assignment operator of `T` typically needs to give at least the basic exception guarantee in order for the container itself to give the guarantee. In addition, when an object a is copied to give object b, this operation is expected not to change the state of a. In other words, `T`'s copy constructor must take it argument by `const` reference.

For a type `T` to be used with our STM implementation we impose slightly different constraints. These constraints form the minimum set of constraints any indirection-based STM using compile-time polymorphism must demand of `T` for the STM implementation to work correctly. They are enumerated in the following along with their justifications.

- *No-throw destructor*: A non-throwing destructor is needed, since we, when committing or aborting a transaction, need to be able to destroy either the original version of an object, or the new version; if either fails, we cannot fully commit or abort the transaction.
- *Copy constructor*: `T` must have a copy constructor such that we can clone the object when it is opened for writing. The copy constructor may also be used when creating new shared objects, although any constructor may be used during construction.
- *Reentrant read operations*: As more than one transaction may open a shared object in read-only mode and perform read-only operations on it (including reading its public variables), the read-only methods need to be able to be executed concurrently. Note that this does not prevent an implementation from mutating its internal state during a read-only operation, as long as this mutation can correctly run concurrently with other read-only operations, perhaps by using locks internally. This could be relevant e.g. for caching results of lazily evaluated expensive calculations. In C++, `T`'s methods should be marked with the `const` qualifier if and only if they are read-only.
- *No global side effects*: Calling a method of `T`, even a non-const method, should not modify any global state nor perform IO. More specifically, we require changes to be limited to the state of the object in question, any data it may own, and any transaction-local data passed by reference or pointer to the method.

3.1.3 Transaction interface

In Figure 3.1 we show a summary of the API. In the following we discuss the different pieces.

```
//Type thrown when a transaction is aborted:
class abort_exception;

//Type thrown when a transaction should be retried:
class retry_exception;

//The transaction group which owns a set of shared data.
//This has user visible members:
class transaction_group;

//The type used for keeping track of a single transaction.
//Its use is normally encapsulated by the helper macros shown
//later:
class transaction {
  transaction();
  transaction(transaction_group&);
  transaction(transaction&);
  void start();
  void commit();
  void reset();
  void abort();
  void retry();
};

//The type that makes an existing type T sharable across transactions:
template<typename T>
class shared {
  //Construct a shared instance of T, using T's default constructor:
  static shared<T>* construct(transaction&);

  //Construct a shared instance of T, using (one of) T's one-argument
  //constructor; this may be its copy constructor if T and A1 are the
  //same type:
  template<typename A1>
    static shared<T>* construct(transaction&, A1&);

  //Several overloads of construct which take more parameters (not shown).

  //Destroy this shared instance:
  void destroy(transaction&);

  //Get read-only access to this shared instance:
  const T& openRO(transaction&) const;

  //Get read-write access to this shared instance:
  T& openRW(transaction&);
};

//The helper macros:
ATOMIC_BEGIN(transaction)
ATOMIC_END(transaction)
ATOMIC_ELSE(transaction)
```

Figure 3.1: Summary of the API

Transaction groups

Shared data belongs to a *transaction group*; normally only a single group will be in use in a given program. The group holds what are essentially shared global data, used to coordinate the different transactions: the global version clock and other bookkeeping information. The use case for more than one group is to allow disjoint data sets in a program to belong to different groups; this may reduce contention on the global version clock and the support structures. At a minimum one group must be created. As a convenience, a default group is provided by the library and is used when the programmer does not specify otherwise.

Transaction class

Bookkeeping data for a single transaction is stored in the `transaction` class. It has five methods:

- `start`: Tells the `transaction` instance to begin a new transaction. If the tranaction's constructor was supplied with a reference to another `transaction` instance, this transaction becomes a nested transaction.
- `commit`: Tells the `transaction` instance that the transaction should be committed. An `abort_exception` is thrown if the transaction cannot be committed due to a conflict.
- `reset`: Used to restart an aborted transaction; the internal data structures and transaction-local copies of objects of the aborted transaction are cleaned up, and the transaction is restarted. If the transaction is an inner transaction, another `abort_exception` is thrown to abort the parent transaction as well.
- `abort`: Used to clean up internal data structures and transaction-local copies of objects of an aborted transaction; note that this does not throw an exception.
- `retry`: Tells the `transaction` instance that it should be restarted when some of its read data have been altered; or, if the transaction is nested, to abort its changes, and continue with the next alternative. If no alternative exists, it retries the parent transaction (by throwing a new `retry_exception`).

The `abort_exception` can also be thrown by the `openRO/RW` calls discussed next, when opening an object would cause inconsistent reads. When a transaction wishes to retry, it should not call the `retry` method above (it merely handles that a transaction is being retried); instead it should throw a `retry_exception`. The proper way to use the above functions without the helper macros (described later) is as follows for the simple non-nested case:

```
1  //Create the transaction object
2  estm::transaction tx;
3  tx.start();
4  while (true) {
5    try {
6      //The actual work goes here:
7      dosomething(tx);
8      tx.commit();
9    } catch (estm::abort_exception&) {
10     tx.reset();
11     continue;
12   } catch (estm::retry_exception&) {
13     tx.retry();
14     continue;
15   } catch (...) {
16     //Let non-transaction exceptions escape
17     tx.abort();
18     throw;
```

```
19      }
20      break;
21  }
```

Shared data

To make a shared type from an existing type `T`, the class template `shared` is used. Creating a shared `int` is done not by `new`'ing a new instance of the `shared` template, but instead calling the static `construct` method on the `shared` template (which has several overloads, so different overloads of `T`'s constructor can be called. The `construct` method returns a pointer to the new `shared` instance. Pointers to `shared` instances may be stored between transactions, and as part of other shared objects; the address of the data itself should never be stored between transactions.

To open an existing object for reading the `openRO` method is used; it returns a `const` reference to the current version of the object, or if the object has been modified in the transaction, a reference to the transaction-local copy. Likewise `openRW` returns a non-`const` reference to a transaction-local copy of the object. If the object cannot be opened without the readabilty constraints (discussed in Section 3.2) being violated, an `abort_exception` is thrown. Last, an object can be deleted (or more specifically be scheduled for deletion after the transaction commits), by calling the handle's `destroy` method. An example:

```
1  estm::transaction tx;
2  ATOMIC_BEGIN(tx) {
3    int k=....;
4    //Construct a new shared int, with the value of k
5    estm::shared<int>* si=estm::shared<int>::construct<int>(tx, k);
6    //Access the new shared int in read-write mode:
7    int& i=si->openRW(tx);
8    //...and alter its contents:
9    ++i;
10   //Open an existing shared std::string for reading:
11   estm::shared<std::string>* ss=...;
12   const std::string& s=ss->openRO(tx);
13   //Open the same std::string for writing:
14   std::string& sw=ss->openRW(tx);
15   //We can edit the string as normal:
16   sw.append(....);
17 } ATOMIC_END(tx);
```

Helper macros

Writing the exception handling shown earlier can get quite tedious, in particular when using nested classes and retry alternatives. To avoid this, we provide a few preprocessor macros which contain all the necessary logic to perform this. They are `ATOMIC_BEGIN`, `ATOMIC_END` and `ATOMIC_ELSE`. They are used as follows:

```
1  estm::transaction tx;
2  ATOMIC_BEGIN(tx) {
3    dosomething(tx);
4    estm::transaction txnested(tx);
5    ATOMIC_BEGIN(txnested) {
6      //First alternative
7      dosomething1(txnested);
8    } ATOMIC_ELSE(txnested) {
9      //Second alternative
10     //Called if first alternative retries
11     dosomething2(txnested);
```

```
12    } ATOMIC_ELSE(txnested) {
13      //Third alternative
14      //Called if second alternative retries
15      dosomething3(txnested);
16    } ATOMIC_END(txnested);
17  } ATOMIC_END(tx);
```

They create the necessary loops and exception handlers and invoke the necessary calls to the `transaction` instances' `start`, `commit`, `reset`, `abort` and `retry` methods.

3.1.4 Returning from transaction blocks

None of the examples shown so far return a result to their caller. Returning results may be somewhat tedious and error prone. The RSTM and DracoSTM implementations use the somewhat clumsy approach of storing the return value in the scope enclosing the transaction; with our implementation, it looks like this:

```
1  int f() {
2    int retval;
3    estm::transaction tx;
4    ATOMIC_BEGIN(tx) {
5      retval=calculate(tx);
6    } ATOMIC_END(tx);
7    return retval;
8  }
```

It seems that we should be able to return the value directly in line 6 with a normal `return` statement; however, doing so would return before the `commit` function is called. Returning would leave the transaction still active, and thus neither committed nor aborted. What we need is for the transaction to be (attempted) committed just before the `return` statement. There are three approaches that almost make this work. We describe next why they fail to work in all cases:

Return macro

The first approach is to define a macro `ATOMIC_RETURN`, that is used instead of the `return` statement:

```
1  #define ATOMIC_RETURN(tx, exp) { tx.commit(); return exp; }
```

This way, the transaction is committed before the value is returned; however, since the expression may have side-effects (and may use the transaction) this will not work.

Smarter return macro

Instead, we can devise a method to evaluate the expression, store the result, attempt the commit, and finally return the result of the expression. We would prefer to do so without requiring the programmer to specify the return type explicitly (this may be quite tedious in template code). To this end we introduce a helper function template:

```
1  template<typename T>
2  T return_helper(estm::transaction& tx, T returnvalue) {
3    tx.commit();
4    return returnvalue;
5  }
6  #define ATOMIC_RETURN(tx, exp) { return return_helper(tx, exp); }
```

This approach works under normal circumstances — if the commit fails, the `commit` function throws an `abort_exception` and the result is never returned; if it succeeds, the value is simply returned. The compiler can deduce the template type automatically.

However, the approach causes problems in the case of nested transactions. When an inner transaction commits, its state is simply rolled into the outer transaction, but the `return` statement may exit the scope of the parent transaction as well (which would then not be committed). We could recursively commit the parent transaction as well, though this would be premature if the `return` statement does not exit the scope of the parent transaction; this would be the case if the inner transaction is defined in a separate function.

This approach may then be usable in some cases, but in general it does not work.

Stack guard

The last approach is to use a stack guard. The stack guard is a small object that we place on the stack as part of the `ATOMIC_BEGIN` macro, that performs some operations during stack unwinding when its destructor is called. The idea is to let the guard commit the transaction. There are two reasons the guard can go out of scope: an exception has been thrown and is in progress (it may be either a normal exception thrown by the user code, a `retry_exception` thrown by the user, or an `abort_exception` thrown by the STM system due to a failure to commit or to open an object). Alternatively, no exception is in progress, and the transaction needs to be committed (we assume for a moment that the explicit call to `commit` does not happen in the `ATOMIC_END` macro as it would normally do). In the former case, the transaction is doomed and the normal exception handlers work fine. In the latter case, the transaction needs to be committed; the guard simply calls the commit function. The guard can use the somewhat obscure `std::uncaught_exception` function in the standard library to determine if an exception is in progress.

The stack unwinding (and hence the invocation of the guard) does not happen until after the `return` expression has been evaluated, and we thus know that no more work is to be done in the transaction. If the `return` statement exits more than one transaction block, every transaction will be committed in turn by their own guard. We can then write the above example naturally:

```
1 int f() {
2   estm::transaction tx;
3   ATOMIC_BEGIN(tx) {
4     return calculate(tx);
5   } ATOMIC_END(tx);
6 }
```

This is a great solution except for one problem: if the commit fails and an `abort_exception` is thrown (as expected) the return value is constructed before the exception is thrown, but is never destroyed; if it contains resources that its destructor would normally dispose of, these resources are leaked[1].

Neither approach works all the time. Instead we use the clumsy approach. We could instead use the stack guard to assert at runtime (in debug version of our programs) that we never leave a transaction block without it being committed except when processing an exception.

3.1.5 Aliasing problems

Like other C++ implementation of STM, ours cannot prevent *aliasing errors*. These arise if a transaction opens an object in read-only mode thereby getting a reference to the current version of the object, and next opens the object in read-write mode and gets a reference to the transaction-local copy of the object. If care is not taken to avoid using the old reference, an aliasing error occurs; this is essentially a read-after-write error, where the write is not visible to the transaction. We did nothing to avoid this problem.

[1] To the best of our knowledge, the C++ standard does not specify what the correct behavior is; we did however observe the described behavior on every compiler used.

3.1.6 Memory management

We stated in Section 3.1.3 that shared data should only be created through the static `construct` method of the handle template, and only deleted through a call to `destroy`. This allows the STM system to manage the actual lifetime of the objects. This also means that if a transaction aborts, any new objects created by the transaction are deleted by the STM system. The programmer does not have to deal with cleaning up such objects.

3.2 Object lifetimes

We introduced validation with the global version clock in Section 2.2.2; here we look at the details. Our goal is to prove that our implementation has the isolated property. The global version clock also helps us determine when it is safe to delete old versions of objects.

We use a variant of the global version clock systems as described by Dice et al. [6] and Riegel et al [20]. Our notation is adapted from the latter. When a transaction T starts, it reads the version clock and stores its value as the transaction's *read version* (rv). Every non-read-only transaction that commits is assigned a unique version number, the *write version* (wv), by reading and incrementing the global version clock. A new version of the shared state is created by committing the transaction, since new versions of objects constructed or updated by the transaction are created.

A particular version of an object is labelled with the write version of the transaction that created the version. A particular version of an object is said to be *valid* in a *version range*. This version range denotes the first and last versions in which the object version is the up-to-date version. When an object is first created and committed, the beginning of the range is equal to wr and the end of the range is unbounded (infinite). We denote a range beginning at version a and ending version b by $R_{a,b}$; in the unbounded case we write $R_{a,\infty}$. Only when the object is destroyed or written by another transaction is the range ended. For every object in the shared state we store (a reference to) the newest version and the version's version range; older versions are also reachable for a period of time.

A read-only transaction with read version rv may only read object versions where the version range $R_{a,b}$ satisfies $a \leq rv \leq b$. This guarantees that the transaction sees a consistent view of memory; it sees the view of the shared state at version rv. An updating transaction must only read (or copy for writing) versions that are unbounded and satisfy $a \leq rv$. The updating transaction needs the version ranges to be unbounded in order to avoid violating serializability when committing. We refer to object versions satisfying these constraints as *readable*; the constraints are called *readability requirements*.

A running transaction T has a readset R_T, a writeset W_T and a createset C_T of versions of objects. The version ranges in C_T are unbounded in both directions, as the objects that they are versions of, are not yet part of the shared state. Our implementation does not allow blind writes, so $W_T \subseteq R_T$.

When a running updating transaction T with a readset R_T attempts to open an object o with a version range $R_{a,b}$ that does not satisfy the readability requirements specified above, the transaction can either abort, or it can attempt to adjust its rv value such that all version ranges of $R_T \cup \{o\}$ satisfy the readability requirements. A read-only transaction may also choose to open a different version range of o (an older version). An updating transaction cannot do so, as it would prevent the transaction from committing. To update rv, the updating transaction reads the current version v of the global version clock, and iterates over R_T verifying that every version range is still unbounded. If so, it can assign the value of v to rv and continue[2].

When an updating transaction commits, it must validate its readset. This is trivial if the current version v of the global version clock is equal to rv; every range in O_T is unbounded, since

[2]In practice, since we cannot atomically update both the global version clock *and* commit a transaction, the global version clock is updated *after* the transaction writes the new version number to the objects it updates. The newly opened object may then have a range $R_{a,\infty}$ where $a = v + 1$ (v is the current value of the global version clock); in this case we actually assign a instead of v to rv. This is safe, since all objects in a transaction are committed atomically, and all the version ranges in R_T are still unbounded.

this was the case when the last object was added to O_T. Otherwise, the same procedure as for updating rv is followed.

We mentioned earlier that older versions of objects are reachable even after new versions are created. Apart from allowing read-only transactions to read old versions, this is necessary to avoid the problems that arise if we do not perform validation nor avoid dirty reads, as discussed in Section 2.2.1. We cannot delete an object version that could be part of the readset of a running transaction. We can only delete an object version with range $R_{a,b}$ once every running transaction T has $rv > b$. We use a deferred deletion mechanism described in Section 3.7.2 to implement this.

In the following we prove that both read and update transactions are conflict serializable by showing the equivalent serial schedule; adjacent read-only transactions in the serial schedule are however not ordered with respect to one another. In the proofs we assume for the sake of simplicity that the global clock is increased by 2 instead of 1 during commit, such that all rw are even. The first commit has version 0. The proofs assign all transactions an offset in the serial schedule.

Theorem 1. *An update transaction T is conflict serializable.*

Proof. When T successfully commits, its rv is increased to the value of the global clock, and, due to the readability requirements, all object versions $o \in O_T$ are unbounded and have range $R_{a,\infty}$ where $a \leq rv$; all o are thus current. T is assigned the next value of the global clock ($rw = rv + 2$) and its offset in the serial schedule is equal to rw. □

Theorem 2. *A read-only transaction T is conflict serializable.*

Proof. T has rv which is even. Due to the readability requirements, every object version $o \in O_T$ has range $R_{a,b}$ where $a \leq rv \leq b$. T's offset in the serial schedule is equal to $rv + 1$. □

3.3 Lifecycle of a transaction

In our implementation a transaction has 2 distinct phases: the *active phase* and the *committing phase*. In addition, it can enter the *committed state* or the *aborted state*.

During the active phase, the transaction reads existing committed version of objects and can make speculative changes to these objects by creating *transaction-local* copies of the object. In addition new objects can be created and existing objects can be marked for deletion. During the active phase, none of the transaction's changes are visible to other transactions. If a transaction during the active phase discovers that one of the objects it has read (and perhaps has local modification to) has been modified, the transaction is aborted, its local modifications are discarded and the transaction restarted.

Once the transaction has performed its work, it switches to the committing phase, which itself consists of two separate phases: the *acquire phase* and *final validation phase*.

During the acquire phase, the transaction *acquires* the objects that it has modifications to. Acquiring an object is essentially a method of locking the object and preparing to replace the current version with the transaction's own version. More than one transaction can be in the acquiring phase at the same time. If a transaction T_a attempts to acquire an object that another transaction T_b has acquired, transaction T_a aborts. In order to avoid both T_a and T_b aborting (and thus wasting work and potentially causing livelock), the objects are acquired in a global order (based on the objects' addresses). Acquiring an object does not prevent other transactions from opening the object; they can access the current version. Acquiring the object just prevents another transaction from preparing to replace it with its own version.

Once a transaction has acquired all the objects it has transaction-local copies of, it enters the final validation phase. Only one transaction can be in this phase at a time. As the name suggests, the purpose of the final validation phase is to ensure that none of the objects read by the transaction have been modifed (the acquired objects need no checking, as they cannot be modified since they have been acquired). Once all objects have been validated, the transaction is assigned the next version number from the global clock, the transaction is marked as committed and the

transaction-local copies atomically replace the existing versions of the objects. Finally the global clock is updated.

We discussed contention managers in Section 2.4; this implementation has no explicit contention manager, but it implictly implements the naive contention manager.

3.4 Internal data structures

Here we discuss the data structures used inside our implementation.

3.4.1 Transaction group data

A program using transactions has, as described in Section 3.1, one or more instances of a transaction group. A transaction group is visible to all transactions operating on the group's data. The group contains the following data:

- The global version clock for the group. It is a simple unsigned integer.
- A mutex used during commit.
- An array of the running transactions' read-versions.

The mutex is a spinlock, as it is only held for a short time. An operating system mutex which deschedules the thread when waiting to get the lock is in practice not a good idea. In our tests this caused lock convoys when using more than one thread; the lock convoys caused benchmarks with more than one thread to run approximately 10 times slower compared to running with just one thread.

The array of read-versions is used to decide when an object version can be deleted. Every `transaction` instance is assigned an offset in this array, and while the transaction runs its read-value exists in the transaction's offset in the array.

3.4.2 Shared objects

Every shared object controlled by the STM system is represented via a *handle*. This handle is in fact the `shared` class template we discussed in Section 3.1; we do however refer to it as a "handle" in the following since "shared" is not a noun. The handle holds two pointers. The first points to a list of waiting transactions (which we describe in much more detail in Section 3.6). The second points to a *container* which holds a version of the object along with some metadata related to the version. These metadata are the write-version of the version of the object, a pointer to the public transaction descriptor (discussed shortly) of the transaction that has created the version, and a pointer to the previous version of the object (`NULL` if this is the first version of the object). When the object is acquired (as discussed in Section 3.3) the handle points to a speculative version of the object, which can be identified as such by inspecting its version number and the contents of the transaction descriptor that the container points to; this is discussed further later. Client code uses pointers to the handle; the addresses of the containers are never revealed to the client code. See Figure 3.2.

3.4.3 The transaction metadata

Every transaction has some metadata; both a public part and a private part.

The public part of the transaction descriptor contains:

- The transaction status
- Version number (zero until the transaction is actually commited)

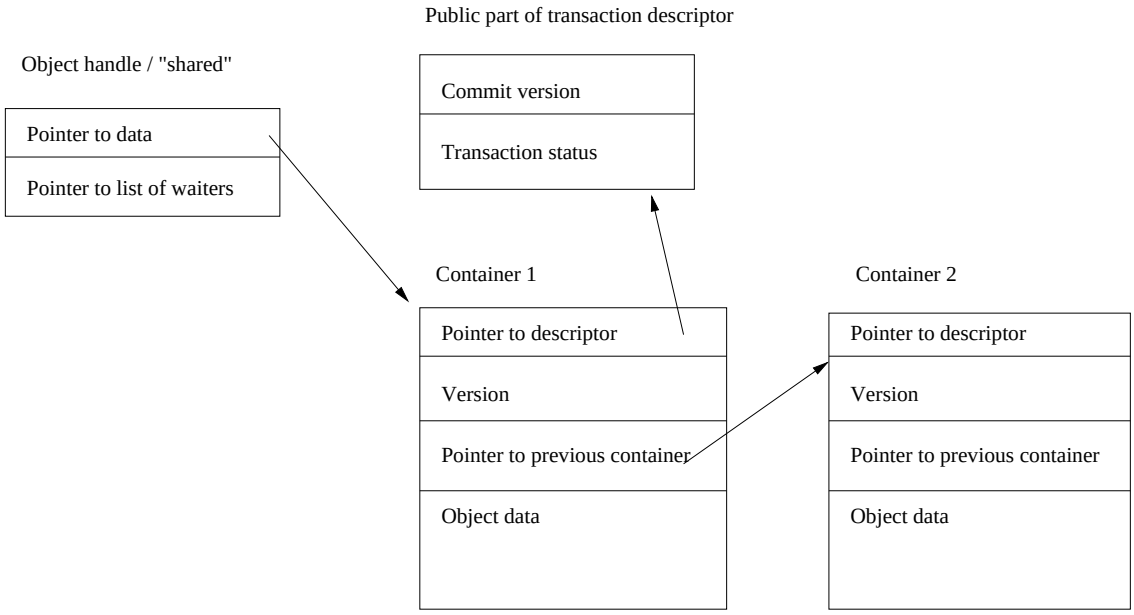

Figure 3.2: *Transactional object structure:* Depending on the value of the version-field in container 1 and the transaction status in the public part of the transaction descriptor, either container 1 or container 2 holds the current version of the object.

The transaction status can be either `active`, `committed` or `aborted`. The version number is assigned once the transaction is commited; it is 0 otherwise. The public part of the transaction descriptor can be seen by other transactions, and is used to implement commit of a transaction such that every object in a transaction is committed atomically.

The private part of the transaction descriptor contains:

- The transaction's readset, represented as an array of pointers to handles. The readset may contain duplicate entries.
- The transaction's write- and createset in a single array. Every entry in this list consists of:
  - A pointer to the object's handle.
  - A pointer to the container holding the version of the object that was current when the object was opened; `NULL` if the object has been created by the transaction.
  - A pointer to the container of the transaction's speculative copy; `NULL` if the object is marked for deletion.
  - A function pointer used to delete the object's current version; `NULL` if the object has been created by the transaction
  - A function pointer used to delete the transaction's speculative copy; `NULL` if the object is marked for deletion.
  - A function pointer to delete the handle; `NULL` if the object is not new and not marked for deletion.
- A pointer to public part of transaction descriptor
- The read-version number of the transaction
- A pointer to the parent transaction; `NULL` if this is not a nested transaction;

The function pointers are used when aborting a transaction or during the defered deletion, and are used instead of virtual destructors, as discussed in Section 2.1.3.

If the array holding the write- and createset grows past a certain size, the implementation switches to a hash-table to hold the set. The expected lookup time in the write- and createset is therefore $O(1)$.

3.5 Operations

In this section we discuss how the individual operations of the STM implementation are performed.

3.5.1 Starting a transaction

When a transaction starts, it creates a new public transaction descriptor, and sets the descriptor's status to `ACTIVE`. It then reads the global version clock, which becomes the transaction's read-version; the transaction enters this read-version into the transaction group's array of running transactions. This is somewhat simplified; special care has to be taken to avoid a race condition between reading the global version clock and writing the entry in running transaction list; we do not discuss the details here.

3.5.2 Opening objects in read-only mode

When we need to open an object in read-only mode, we first inspect the transaction's combined write- and createset, to see if we have a local copy of the object. This is done simply by iterating over the set and looking for a match. If this is a nested transaction, we must also recursively inspect the write- and createsets of the outer transactions.

Otherwise, all we normally have to do, given the object's handle, is to follow the handle's pointer to the container holding the current version of the object. However, since another transaction may have acquired the object, the version held in the container may in fact not yet be committed. To determine if the container is committed, we inspect its version field; if it is non-zero, this version is committed. Otherwise, we traverse the pointer to the container's transaction descriptor, and inspect its status; if the status is `committed` the transaction and hence the container is committed. If it is not, we follow the containter's pointer to the previous version, which is guaranteed to be committed. The important thing to note here is that the container's write version may be zero even though it holds the committed version; this will rarely be observed, as committing transactions update the containers write-version immediatly after committing.

After finding the newest committed version, we compare its (actual) version number v with the transaction's read-version rv. If $v > rv$, we must perform validation; the details are descriped later in Section 3.5.6.

Finally we add the object's handle to the readset. We allow duplicates in the readset, as our experiments have shown that only about 1 to 10 percent of objects are opened more than once. Detecting the duplicates would take more time than we can expect to save (our experiments confirmed this as well).

3.5.3 Opening objects in read-write mode

If we already have a local copy of the object (in the write- and createset), this can be used with no further action. Otherwise we find the current version (by the same procedure we use when opening in read-only mode) and create a copy for the transaction, and add the copy to the transaction's write- and createset. The object's container's pointer to the previous version is set to point to the container whose contents we just copied; however, if we copy an outer transaction's version, we must instead point to *its* previous version.

3.5.4 Creating an object

Creating an object is trivial: a new handle and a new container are created, and the container's pointer to the previous version is set to `NULL`. We add the new object to the write- and createset.

3.5.5 Deleting an object

Obviously we cannot delete an object during the transaction, since the transaction could abort, and since other transactions might concurrently be reading the object. Instead, we can schedule objects for deletion.

We do this by finding the object's entry in the transaction's write- and createset; if it does not exist, we insert it. We then make sure the object's entry specifies a `NULL` value for the transaction's speculative copy; if there is a speculative copy we delete it immediately. If the object has been created during this transaction we can simply delete the object (both the handle and the container) and remove the entry from the list. If we do so, we must remove the corresponding entries from the readset; otherwise we access invalid memory when we later perform validation.

3.5.6 Validation

When we perform validation, we first read the current value of the global version clock gv. We then iterate over the transaction's readset, and for every entry we retrieve the version number v of the current version of the object. If v is larger than the transaction's read-version rv the readability requirements are violated and the validation fails and the transaction is aborted. Otherwise, if every object passes this test, we store the previously read value of gv as the transaction's new read-version rv.

If the transaction is nested, we also recursively validate the objects in the parent transaction.

3.5.7 Acquiring an object

Acquiring an object serves two purposes: locking the object so no other transaction can acquire it and preparing for atomically replacing all committed objects with their transaction-local counterparts. This is done simply by performing a compare-and-swap (CAS) operation on the pointer in the handle. The compare value is the address of the container holding the version that was current when the object was opened. The to-be-swapped-in value is the address of the container holding the transaction's speculative copy. If another transaction has acquired the object the CAS will fail and we abort the current transaction.

After the CAS operation the transaction-local copy will be "visible" to the other transactions, but other transactions that open the object will instead use the committed version as described in Section 3.5.2. In other words, the transaction-local copy will only be visible to the internal parts of the STM; not to any client code.

3.5.8 Performing the commit

Depending on whether a transaction is nested or not, the commit procedure is quite different. The two cases are described in the following.

Committing a nested transaction

When we commit a nested transaction, no changes are made public. Instead the parent's transaction table is updated to reflect the changes in the nested transaction. This is simply a matter of running through the nested transaction's table of objects and inserting or overwriting these entries in the parent's table.

No validation needs to be performed.

Committing a non-nested transaction

When the outmost transaction decides to commit, it enters the acquiring part of the committing phase. We first build a write list; this holds the necessary data to acquire the objects, as described earlier. The list is sorted by handle address. All the objects in this list are then acquired in order. If/when this completes, the transaction enters the final validation phase — only one transaction can be in this phase at once, and the transaction group's mutex is used for this. Validation is performed. If this succeeds, this transaction will complete. The following steps are then performed in order:

1. The public transaction descriptor's version number is set to the next value of the global version clock.
2. The public transaction descriptor's status is changed to the `committed` state. This state change is what atomically lets the transaction-local copies replace their previous versions.
3. The global version clock is incremented.
4. The transaction group's mutex is released.
5. The version number in every container created by the transaction is updated to the version number assigned in step 1.
6. The transaction resets its entry in the transaction group's list of running transactions.

At this point, all that remains is to make sure that old containers and the transaction's own data structures are scheduled for deletion. The mechanisms for memory management are described in Section 3.7.

3.5.9 Aborting a transaction

Aborting a transaction is trivial if no objects have been acquired. We merely delete all transaction-local copies and the transaction descriptor and finally reset the transaction's entry in the group's list of running transactions.

If any objects have been acquired, we need to delay their (and the public part of the transaction descriptor's) deletion until we are sure no other transaction reads them. Again, this is done by handing them to the memory management system as described in Section 3.7.

Aborting a transaction does not require other work than updating pointers and it therefore cannot fail. Since no shared objects have been altered there is no other state that needs to be rolled back; this means that a transaction abort is always completed in its entirety; this fact, and the atomic commit operation, is what makes the transactions atomic.

3.6 Retry operation

In the following we discuss the data structures we can associate with every shared object to implement the retry logic.

Ignoring for a moment the use of retry with alternatives and nesting, we have created the retry-implementation with a few considerations in mind. First, the performance of the transactions performing retry is considered unimportant, since the thread they run on will block on a (operating system) synchronization mechanism and therefore be descheduled and rescheduled; we expect the runtime costs of the resulting context shifts to dominate the running time of the retry operations. Second, we expect that few objects will ever have waiting transactions associated with them; we therefore do not create the retry support-structures for every objects until needed. The waited-upon objects will typically be sentinel objects of data structures, pointing to e.g. either the first or the last element in a list; we do not expect most transactions to wait on arbitrary members of a data structure. Third, some transactions may wait for a long time and we need to make sure

that they do not cause deferred deletion (discussed in Section 3.7) while they are sleeping. Fourth, we do not want a committing transaction to spend undue amounts of time waking up waiting transactions.

The retry logic works, in general terms, by associating with every shared object a list of transactions to be awakened when the object is written to; when a transaction retries, it enters itself into the list of every object in its readset. In fact, it is not the transaction that is entered into the list, but instead a pointer to the transaction's syncronization primitive (a (binary) semaphore). When a transaction is awakened, it removes itself from the lists it had entered itself on. Since we make sure that a waiting transaction does not cause deferred deletion, some of the waited upon objects may be deleted before the transaction has removed itself from their lists; this complicates the implementation a bit.

Recalling that we expect few objects to have waiters, and that the cost associated with waiting and being awoken is unimportant, we associate with every shared object that has (or has had) waiters a linked list stored apart from the handle itself. The handle has a pointer to this list; the pointer is `NULL` when the object is initially created and only once the first waiter enters itself is the list created. The list itself is reference counted; every entry in the list increases the reference count and the handle itself does as well. By the reference counting, we can ensure that even though the shared object is deleted, the list is not, until every waiter has removed itself from it. We store a spin mutex in the list to avoid race conditions on the list contents.

A transaction adding itself as a waiter to a set of objects proceeds as follows. For every object it first installs itself as a waiter (temporarily holding the lock on the list). If no waiter list exists for the object, a new is created and inserted by a compare-and-swap operation. After inserting itself on the list, it verifies that the object has not been written (to avoid a race where the object is written before the waiter is on the list), by comparing the current version with the transaction's read version. If the object *has* been updated, the transaction proceeds to the wake-up phase (described shortly) immediatly. The transaction stores a pointer to the waiter list of every object is has added itself to. Finally, the transaction performs the normal abort-logic and removes itself from the transaction group's list of running transactions and finally waits on the semaphore. Once awakened it enters the wake-up phase during which the transaction removes itself from every list; finally the transaction is restarted.

A transaction writing to an object with associated waiters will, after committing, simply lock the waiter list's mutex, iterate over the list signalling the semaphores, and finally unlock the mutex. If a transaction deletes an object with waiters, no action is taken during commit. When the handle is deleted (by the deferred deletion mechanism), it decreases the reference count of the waiter list, and deletes the list if the reference count has gone to 0. No waiters are signalled when the object is deleted; the waiting transactions cannot just have been waiting on the deleted object. Had they been, they would not be correct as they would attempt to access a non-existant object.

In the case of nested transactions, the outmost transaction (the transaction with no parent) contains a read-list containing all objects seen by all transactions (including the nested transactions that retried). Only when the outer transaction retries is the "normal" retry method described above used.

3.7 Memory management

We've previously in Section 3.2 discussed the use of the global version number and how it relates to objects' life-times. In this section we discuss how memory is managed, in particular with emphasis on placing objects alongside their metadata to avoid cache misses, and when and how we delete objects.

There are in fact two distinct problems here: how do we perform allocation and deallocation, and how to we avoid deallocating memory in use by other threads. We have already shown in Section 3.4 that we store explicit function pointers for freeing handles and containers; here we show where these functions are provided.

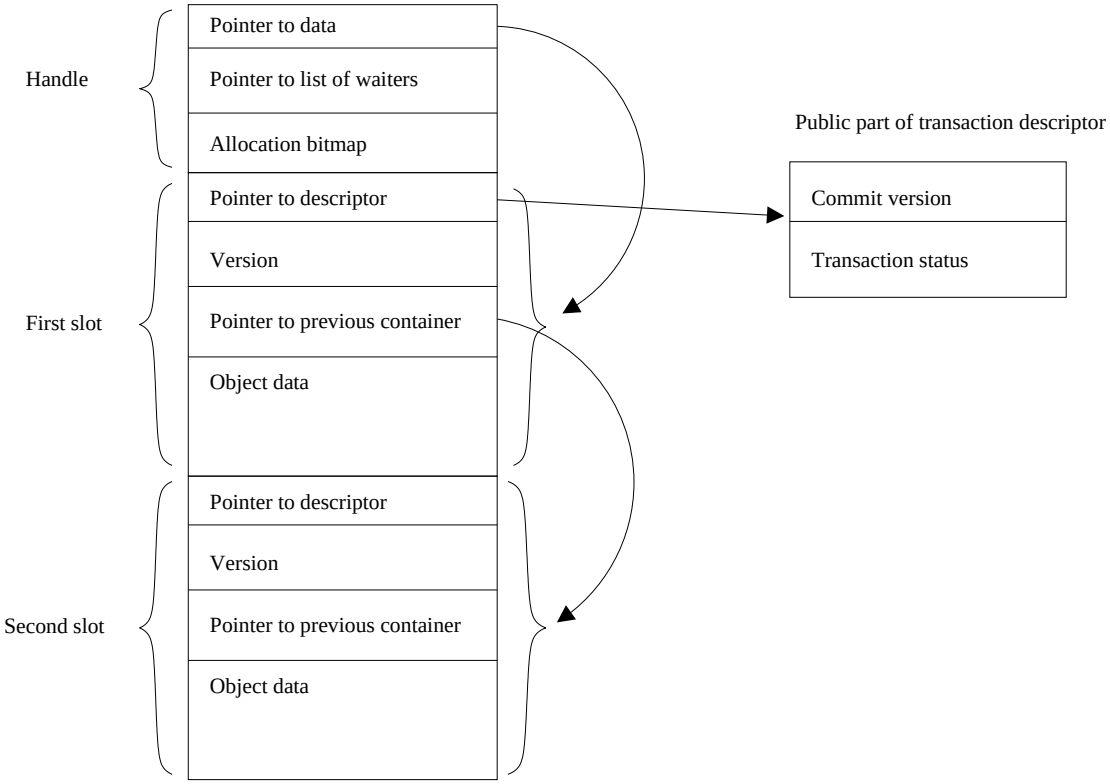

Figure 3.3: *Handle colocated with two slots:* Depending on the value of the version-field in container 1 stored in the first slot and the transaction status in the public part of the transaction descriptor, either container 1 or container 2 holds the current version of the object. Note the existance of the allocation bitmap field in the handle, which only exists when colocation is used.

3.7.1 Colocating objects with their metadata

In order to avoid an extra cache miss caused by first accessing the handle and then the container, we would like to store the containers in the same cache line as the handles. Obviously the number of containers that will fit into the same cache line as the handle is limited and depends on the size of the held objects. If we can find room for two objects, a single transaction updating a value will not require additional memory for the new value.

When we store containers in the same cache line as the handle, we do not initially construct all of the containers; we only make sure there is room for them. The location of a container in the same cache line as the handle is refered to as a *slot*. See Figure 3.3.

It is not always advantageous to have more than one slot; if an object is known never to be written once created, having more than one slot is simply a waste of space. If an object is very large at most one will fit in the same cache line as the handle, and there would be no advantage to having more slots. As the size of objects go up (filling up multiple cache lines), the relative cost of the extra cache miss will shrink, further reducing the need for more than one slot.

The optimal number of slots then often depends on the object's type. We therefore control how many slots to store alongside the handle with a traits class [1]. This `tm_traits` class has an enumeration `slot_count` specifying the number of slots for a given type. For a value of zero, there is nothing special to be done — whenever we need room for a new container, we allocate it on the heap. However, if we have one or more slots, we need a mechanism for allocating and deallocating these. We also need a way of making sure the objects are properly aligned. We do this via some template metaprogramming. In the following, we describe first the basic layout of data, and then

the associated functions for allocating and deallocating slots.

We only discuss allocation and deallocation; once this is taken care of, constructing and destroying the objects is simple.

Layout

When we have more than one slot, we need a mechanism for specifying if a given slot is free. To this end, we introduce a bitmap (simply by using an integer) which is stored adjacent to the handle. Every slot has an associated entry in the bitmap, specifying if the slot is free.

The basic idea of our layout is to create a new class template that contains the handle, the bitmap and the slots. We do this by creating a small class heirachy. The depth of this heirachy depends on the number of slots. The layout is shown in Figure 3.4. First, we derive, from the `shared` class template, a `shared_bitmap`-subtype, whose main purpose is to hold the allocation bitmap. It also holds, as a compile time value via a template parameter, the number of slots n which will be contained in the derived classes. From this we derive $n + 1$ `shared_slot`-subtypes of which the last n have a container as a member variable. The first has no member variables and serves only to end the type recursion of the class heirachy.

The last n classes contain (space for) the required number of slots; however we may not be able to instantiate them (nor do we wish to), since the objects, and hence the containers, are not required to have default constructors. In fact, we do not wish to default construct the objects; we will construct the objects only when they are needed. So, when the user creates a new object (and we therefore need to create a handle and a container), we allocate space for the most derived class, via a call to `new char[sizeof(most-derived-class)]`. We then perform a placement new on this space with the type of the bitmap class. Since the `new char` call is guaranteed to be properly aligned for any type (C++ Standard 5.3.4/10), so are the slots. We then set the bitmap such that the slots are marked as free. In addition to indicating which slots are free, the bitmap also holds a bit for the handle itself, indicating if it has been "deleted". Due to the deferred deletion of both handles and containers, and the fact that the containers write to the bitmap when deleted, we cannot free the memory until all of the container's *and* the handle's deferred deletion has happened.

Allocation

When we need to allocate space for a new container for a given object we proceed as follows. The allocation call is provided with a pointer to the handle of the object for which we are creating a new container. For the case where no containers are colocated, we simply perform an allocation on the heap. Otherwise we call the `allocate` function on the `shared_bitmap` class template. As stated earlier, this knows the number of colocated slots, and casts the handle (`shared`) pointer to the most derived `shared_slot` class. The derived class itself has an `allocate` function, which is called. This checks its own bit in the bitmap to see if the slot is available, and if so attempts to update the bitmap. This is done with a compare-and-swap operation, and one or more threads can safely attempt to allocate the same slot. If the CAS succeeds, the address of the slot is returned and the allocation is done. If the slot is not free, the allocation function recursively calls the `allocate` function in the class from which it derives. If the base case class is reached, it simply allocates from the heap.

Deallocation

Deallocation is a bit more involved than allocation. There are two reasons for this. First, when we deallocate, we are normally either committing or aborting a transaction, and at this point we do not have the types of the objects to be deallocated. Second, we cannot tell just by looking at a pointer to a container if it was allocated on the heap or allocated from one of the colocated slots (and in the latter case, not even which slot). Normally this would not be a problem if we allocated all objects on the heap and used virtual destructors. However, this is not the case here.

```
//For every type X that wishes to use colocation, the
//tm_traits struct needs a specialization. This example
//specifies that for ints two slots should be colocated.
struct tm_traits<int>
{
  enum {slot_count=2};
};

//This is the normal shared/handle class template:
template<typename T>
struct shared;

//This class contains the bitmap for the allocations.
//It holds the slot_count via a template parameter:
template<typename T, int slot_count=tm_traits<T>::slot_count>
struct shared_bitmap : shared<T>
{
  int allocation_bitmap;
};

//This class holds one slot, and is derived recursively
//from itself.
template<typename T, int slot_count>
struct shared_slot : shared_slot<T, slot_count-1>
{
  container<T> slot;
}

//This ends the above type recursion by inheriting from the
//shared_bitmap class.
template<typename T>
struct shared_slot<T, 0> : shared_bitmap<T>
{
};
```

Figure 3.4: *Class structures used to implement colocated data:* When colocation is used, the memory allocator allocates room for a `shared_slot<T, tm_traits<T>::slot_count>` class but only constructs the `shared_bitmap<T>` class. The slots (containers) can then be allocated and constructed as needed. This allocation happens via methods (not shown) on the `shared_slot` classes; they know their own offsets in the bitmap and they use this to determine if they are free. Equivalently methods exist for deallocating individual slots.

To solve this, we make sure that a running transaction gathers the necessary data to perform deallocation, while it still has access to the relevant data. In particular, at any time we create a new object, open an object for writing (thus creating a copy), or schedule an object for deletion, we save a number of pointers to functions, that know how to deallocate the objects. We store a pointer for the handle, one for the current version of an object and one for the speculative new version of an object. We will refer to these functions as *deallocator* functions.

When we allocate a new handle or a new container, we simply return a deallocator function pointer along with the pointer to the new memory. When we copy an existing container, we thus trivially get a deallocator function pointer for the new copy. Getting the pointer for the existing container works like this: We have both the type of the copied container, and the address of the handle. We can then determine if the existing container has the same address as one of the slots. This is done recursively by calling a function on the most derived class of the handle. If the addresses match, the address of a corresponding function in the derived class is returned. Otherwise, the container must have been allocated on the heap, and a corresponding function is returned. Getting a function for deallocating the handle itself is simple, as there are only two possibilities and therefore only two possible functions. Either the object has colocated objects, and a deallocator function which handles the allocation bitmap properly is returned, and otherwise a function is returned which simply frees the handle to the heap.

In the colocation case, only when the handle and all the containers have been deallocated, is their memory released to the heap.

3.7.2 Deferred deletion

As discussed in Section 3.2, we need to delay the actual deletion of objects that may be read by running transactions. We handle this by the following method. When a transaction commits and objects have been scheduled for deletion (either entirely or they have been replaced by a new version), we remember the transaction's write version number wv. The deleted objects are then possibly valid for transactions reading version numbers x where $x < wv$. Thus, as long as transactions reading version $x < wv$ are running we cannot delete the objects. Instead we store a list of objects to be deleted. Every entry stores the highest version number that they are valid ($vw - 1$ in this case), a pointer to the object and a function pointer to the deleter/deallocator function, as described earlier. The public part of the transaction descriptor also merits an entry - it however is valid up to and including version vw. When no transaction reads the objects, they can be deleted.

This leaves us with a decision about when to iterate through the list, and how much of the list to iterate. We can easily keep the list sorted by the last valid version number, and this means we can stop iteration of the list when we reach an element that may still be read in some transaction.

Some of the possible approaches to iterating the lists are:

- Whenever a transaction commits or aborts, iterate the list.
- As above, but iterate no more than some constant number of elements than have been opened during the transaction. In the low or no contention case, the cleanup time will be at most linear in the length of the transaction, which may prove less surprising to the user.
- Use one or more dedicated threads for cleaning up. This may allow a higher CPU-utilization, but introduces problems with concurrent access to the deletion lists, and may suffer from the to-be-deleted objects being in the cache of the wrong CPU-core, thus hurting performance.
- Instead of assigning the elements to the thread's own list, assign them to one of the threads which is causing the elements not to be deleted. This introduces new problems with concurrent access to the lists.

In practise, not much time is spent iterating over the list, so iterating early on on the thread that caused the deletions is probably a good idea — the elements are more likely to be in the cache, and they may be using up free slots. In addition the implementation is simple.

When we decide to iterate over the list, we first need to find the smallest read-version of all the running transactions; we find this in the transaction group's list of running transactions.

Any object which has had no chance of being visible to other transactions can of course be deleted immediatly — this could be the case for an object both created and destroyed within the same transaction.

3.8 Implementation

In this section some practical details of the STM implementation are discussed.

3.8.1 General

First, we need to get a few unpleasant details out of the way. The primary is that C++ has no memory model. This means that there is no way for our implementation to be correct according to the C++ standard. However, most compilers specify to some extent how they deal with multithreading issues, and how to ensure that they behave as we need. In particular, we need some operations or language constructs that give us memory barriers with either acquire or release semantics, such that when we CAS in a newly commited object, we can be sure the object's entire represention is present in memory (in particular, that writes to the object are not reordered past the CAS operation).

Intel have recently released their "Threading Building Blocks" library [14], which includes, among other things, a template type named `atomic`. This provides compare-and-swap operations (and several other operations) on a contained integer or pointer, and allows the user to specify that these operations should have either acquire or release semantics. The implementation of `atomic` "knows" how to give these guarantees on several popular compilers, including Microsoft's Visual C++, GCC and, of course, the Intel C++ compiler. Using this library means we do not have to worry about such details of each compiler. In addition, a spinlock implemention is also provided by this library, that is useful for the STM library. So is the cache line aligned allocator, which we use for allocating handles with colocated containers, as described in Section 3.7. As the name suggests, this allocator allows us to make memory allocations that are aligned on cache line boundaries.

The Boost project [2] provides us with platform independent functions for managing threads and a library of random number generators; we use both in our benchmarks.

The present implementation has some flaws. It currently only runs on Microsoft Windows as we use their `AutoResetEvent` for the retry implementation; we have not yet ported this code to other platforms although it should be quite simple. The implementation of nested transactions is currently broken as well. The API in the current implementation does not quite match the one described in Section 3.1; this is however mainly a question of naming. Though we discussed read-only transactions in Section 3.2 the implementation treats all transactions as read-write transactions.

The implementation only runs on 32-bit machines at the moment, as the implementation's use of the lower bits of pointers as markers (discussed in Section 4.4) is hardcoded to 32-bit variables. This is also easy to fix. However, moving to a 64-bit platform will double the size of pointers, and this will reduce the number of types for which we can store two slots in a single cache line.

3.8.2 Version numbers and memory use

The current implementation uses 32-bit integers for the version numbers. The container class's version number and its pointer to the transaction descriptor are stored in the same variable to save space. To differentiate between the two, all version numbers are odd, such that the lower bit can be consulted to distinguish between version numbers and pointers. This of course reduces the maximum number of transactions to approximately 2^{31}. If the global version clock overflows, the behavior of the implementation is undefined; we should move to a 64-bit version number if the

implementation is to be used in practice. In the experimental evaluation described in Chapter 5 we achieved a maximum of about 1.7 million transactions per second; at this rate a 63-bit version number will last about 172.000 years; even allowing for improvements in hardware and software, this should be sufficient for most purposes.

For handles with colocated containers, the allocation bitmap can be stored in the lower bits of the pointer to the list of waiting transactions (provided the list is suitably aligned — assuming 8 bytes alignment, the lower three bits are free, and we can therefore use it for two slots). If we do so, the handle itself uses 8 bytes. Every container has 8 bytes of metadata (the combined version number and pointer to transaction descriptor, and the pointer to the previous version) and assuming a 64 byte cache line, we can store 2 objects with a maximum size of 20 bytes each in a single cache line ($8 + 2 * (8 + 20) = 64$). We take advantage of this in our data structures in the next chapter.

Chapter 4

Data structures and STM

In this chapter, we will discuss issues related to building data structures using the STM primitives discussed earlier. In particular, we will discuss APIs for the data structures, how to nest these data structures such that we can for instance create a list of red-black trees and use this without error or unnecessary copying, and finally some general considerations about using STM with data structures.

To the extent possible, we model our data structures and their interface on those in the Standard Template Library (STL). The goal is to be able to use our STM implemented data structures with the algorithms of STL. In addition, our data structures give the strong exception guarantee for *all* operations.

We are going to distinguish between simple types as those stored as individual objects, and data structures composed of multiple instances of such types. We say that data structures (containers in STL parlance) are *composed*, while the simple types are *non-composed*.

4.1 APIs

We have discussed the API for opening individual objects in Section 3.1. Here we wish to build complete data structures on top of the primitive STM constructs. The goal is to have an interface much like that of the data structures in the STL; in particular we wish to be able to use our data structures with the standard algorithms in the STL. Note that in STL parlance, data structures such as lists or trees are referred to as *containers*.

To begin with, we will let all access to these containers happen with an explicit transaction object reference parameter, such that the interface to all functions make it explicit that using the functions requires a running transaction. We use red-black trees (`std::map` or `std::set` in STL terms), lists (`std::list`), and arrays (`std::vector`) as our running examples. Given a red-black tree mapping from `int` to `int`, we could write:

```
1  estm::transaction tx;
2  ATOMIC_BEGIN(tx) {
3    estm::map<int, int>& m=....;
4    //Insert mapping from 1 to 2:
5    m.insert(tx, estm::map<int, int>::value_type(1,2));
6    //Get an iterator to the first element in the map:
7    estm::map<int, int>::iterator iter=m.begin(tx);
8    //Dereference the iterator to the first element in the map:
9    estm::map<int, int>::value_type& val=iter.deref(tx);
10   //Update the value of the first element in the map:
11   val.second=42;
12   //Get a const iterator to an element with 3 as the key:
13   estm::map<int, int>::const_iterator citer=m.find(tx, 3);
```

```
14    //Get the corresponding key:
15    int x=citer.deref(tx).second;
16  } ATOMIC_END(tx);
```

There are four things to note here: first, inserting in line 5, does not require the programmer to make any calls to `openRO` or `openRW`; this is all handled by the `insert` method. Second, all the methods take the transaction object reference as a parameter, thereby making it part of the method signature that the methods require a transaction; there is no scope for the programmer forgetting that, and correspondingly, this serves as a "warning" that an `abort`-exception may be thrown by any of these methods. Third, while the `insert` and `begin` methods have become a bit clumsy by the added parameter, the iterator cannot be dereferenced via the normal `operator*` as iterators normally are, since this operator cannot take an argument. This prevents the use of the iterators with standard algorithms; we will remedy this in a moment. Fourth, the difference between a `const_iterator` and a normal `iterator` is now not only that the former cannot be used for updating the contents: dereferencing the former causes the underlying node to be opened for reading, while the latter causes it to be opened for writing.

The last point is quite important: passing a normal iterator instead of a const iterator to a function or algorithm that does not require writing, imposes a large performance penalty: all underlying objects must be opened for writing (thus copying them), and transaction conflicts become more likely.

4.1.1 Iterators

The iterators shown above let the programmer iterate over the data structures and they can themselves be stored in transactional variables. To use the iterators with standard algorithms like `std::for_each`, we need to enable the use of the normal operators (dereferencing, increment, decrement etc.). We do this by creating two iterator wrapper class templates: `read_iterator` and `write_iterator`. They both hold a copy of the transaction object reference and an instance of the iterator type they are wrapping; the `read_iterator` wraps the const iterators; the `write_iterator` wraps the non-const iterators. An excerpt from the `write_iterator` is shown here:

```
1  template<typename T>
2  class write_iterator
3  {
4    estm::transaction& tx;    //Reference to transaction object
5    T                  inner; //The wrapped iterator
6  public:
7    typedef typename T::reference reference;
8  ...
9    write_iterator(estm::transaction& tx, T inner);
10   reference operator*()
11   {
12     return inner.deref(tx);
13   }
14   write_iterator& operator++()
15   {
16     inner.pre_increment(tx);
17     return *this;
18   }
19 ...
20 };
```

This allows us to write code like the following:

```
estm::transaction tx;
ATOMIC_BEGIN(tx) {
  estm::vector<int>& v=...;
  //Negate every element in the vector:
  std::for_each(
    estm::write_iterator<estm::vector<int>::iterator>(tx, v.begin(tx)),
    estm::write_iterator<estm::vector<int>::iterator>(tx, v.end(tx)),
    std::negate<int>());
} ATOMIC_END(tx);
```

However, writing the type of the `write_iterator` gets tedious, so we add helper function templates `make_write` and `make_read`, allowing the following instead:

```
estm::transaction tx;
ATOMIC_BEGIN(tx) {
  estm::vector<int>& v=...;
  //Negate every element in the vector:
  std::for_each(
    estm::make_write(tx, v.begin(tx)),
    estm::make_write(tx, v.end(tx)),
    std::negate<int>());
} ATOMIC_END(tx);
```

Of course, `read_iterator`s and `write_iterator`s should not exist outside of a transaction, since they hold a transaction object reference.

4.1.2 Container wrappers

Just as we can wrap the iterators and store a reference to the transaction object, we can wrap the containers along with a transaction object reference; however, the wrapper class must be written specifically to each container type, as different container types have different interfaces. Additionally, we need two wrappers for every container type: one that exposes the read-only (const) methods, and a second (inheriting from the first) that exposes the write methods as well. The iterators returned from these wrappers are themselves either `read_iterator`s or `write_iterator`s; this way, all we have to do in a transaction is instantiate such a wrapper for each instance we wish to use; the interface then becomes exactly like the interface for the non-STM implementations:

```
estm::transaction tx;
ATOMIC_BEGIN(tx) {
  estm::vector<int>& v=...;
  //Create a read-only wrapper:
  estm::vectorRO<int> vro(tx, v);
  //Sum the elements; begin() and end() return read_iterators:
  int sum=std::accumulate(vro.begin(), vro.end(), 0);
  //Create a read/write wrapper:
  estm::vectorRW<int> vrw(tx, v);
  //Negate every element in the vector; begin() and end() return write_iterators:
  std::for_each(vrw.begin(), vrw.end(), std::negate<int>());
  //Add a value to the end:
  vrw.push_back(42);
} ATOMIC_END(tx);
```

4.1.3 Creating and destroying data structures

A container holds a number of objects, possibly including some sentinel objects. Creating a new container is then a matter of starting a transaction and in the container's constructor creating these

sentinel objects. However, this means the constructor must take a transaction object reference as an argument, as the transaction object is needed for creating the sentinel objects. As an example, the constructor of a list (with relevant definitions) looks something like the following :

```
1  template<typename E>
2  class list
3  {
4    struct list_node { .... };
5    struct list_sentinel
6    {
7      estm::shared<list_node>* p;
8    };
9    shared<list_sentinel>* m_first;
10   shared<list_sentinel>* m_last;
11   list(estm::transaction& tx)
12   {
13     m_first=estm::shared<sentinel>::construct(tx);
14     m_last =estm::shared<sentinel>::construct(tx);
15     //Make m_first and m_last point to nothing
16     //such that the list is empty.
17     ....
18   }
19 };
```

When we insert into or remove values from the container, these insertions change either the sentinel objects, or other existing objects; the container class' own pointers themselves are never altered.

Once we decide to destroy a container (and this can only happen as part of a transaction), we need a method for deleting the stored objects and sentinels. The normal destructor is insufficient, as it does not have access to the transaction object. Additionally, actually deleting the container itself during the transaction will not work: if the transaction aborts, the deletion cannot be undone. Instead an explicit `destroy` method is added to the container; when a transaction wishes to delete the container, it calls the `tx::destroy` method, which in turn calls the container's `destroy` method, which then iterates over all the elements and sentinels, and calls `tx::destroy` for each of them. The container instance is deleted later as part of the normal deferred deletion; the actual destructor does no further work.

Non-composed objects have no such special constructors or destroy methods; it is only when creating or destroying composed objects that these methods should be called. To this end we introduce a new member into our traits-class from Section 3.7.1: `tm_traits::is_composed`. For composed types its value is non-zero. As an example, for list the following partial template specialization is used:

```
1  template<typename E>
2  struct tm_traits<estm::list<E> >
3  {
4      enum {is_composed=1};
5  };
```

This way, the transaction object knows whether to add the transaction object reference to constructor calls, and whether to call the type's `destroy` method.

4.1.4 Temporaries

`std::vector`'s `push_back` method takes (a reference) to a value to be inserted into the container; this value is copy constructed into the new position in the `vector`. In fact, most of the insertion methods in the STL need a value to copy; very few default-construct a value. This requirement

often means that the programmer must create a temporary variable serving no other purpose than to be copied. However, when dealing with nested data structures (discussed in Section 4.2), creating and destroying such a temporary may require several heap allocations and deallocations, making this very expensive. To reduce this cost, we add overloads of such functions that simply default-construct objects instead of copying them; in the case of composed types, the transaction object reference is passed to the created types' constructors as well.

4.1.5 The `size` method

Most STL containers have a constant time implementation of `size`, which returns the number of elements in the container[1]. However, adding such a function to the STM versions means that every insertion or deletion must update a counter variable in the container. All transactions concurrently inserting or deleting from a container will conflict, since they all need to write this counter variable. For this reason, we do *not* expose a `size` method on our containers.

4.1.6 Allocators

The containers in the STL all take a template parameter specifying which memory allocator should be used. For our STM implementation, this is not relevant. All allocation and deallocation is done through the STM system; depending on the types used, different number of slots may be allocated. This means that the sizes of the allocations may differ quite a lot from the sizes that a user supplied allocator would normally expect if used with the STL.

4.2 Nesting STM-based data structures

While it is quite common to build data structures like for instance red-black trees on top a STM implementations, the elements of the tree (the keys and values) must normally be simple value types themselves (`int`s, `string`s, etc.). However, letting the values be (STM-based) containers is usually not supported; this is because the values are in such cases not proper value types. Indeed, few implementations store the containers themselves as a transactional variable; instead the container is a static global variable and only the nodes are accessed via the STM system. Here we wish to be able to nest our STM data structures as values of one another; we wish to be able to create for instance a list of red-black trees or a red-black tree with lists as values:

```
estm::list<estm::map<int, int>>
estm::map<int, estm::list<int>>
```

Considering the latter example, a naive solution would, when opening a map node (holding a list) in write mode, copy the entire list — the constraints on objects as specified in Section 3.1.2 demand that we copy the entire object opened for writing. However, as the individual objects of the containers are already individually copied when opened for writing, copying the whole container is unnecessary. In fact, in our implementation the container classes themselves only hold pointers to handles to sentinel objects; these pointers never change.

The solution is to allow the container classes to be copied without copying their contents; we simply copy the pointers to the sentinel objects. As discussed in Section 4.1.3 the container's destructor does nothing, so there is no risk of double-destruction of the contents.

While allowing the shallow copy avoids copying every element in a container, consider the following code, which inserts an element into a list held in a red-black tree (the code does not use the wrapper classes):

[1] `std::list`'s `size`'s runtime may be linear in the number of elements.

```
estm::transaction tx;
ATOMIC_BEGIN(tx) {
  estm::map<int, estm::list<int> >& m=....;
  //Get an iterator to the first element in the tree:
  estm::map<int, estm::list<int> >::iterator iter=m.begin(tx);
  //Dereference the iterator to get a non-const reference to the list:
  estm::list<int>& l=iter.deref(tx);
  //Add an element to the back of the list:
  l.push_back(tx, 42);
} ATOMIC_END(tx);
```

Although the only actual change happens to the end of the list, we still need a non-const iterator in order to get a non-const reference to the list, such that we may write to the list. However, dereferencing the iterator causes the underlying node in the tree to be opened in read-write mode, even though we under no circumstances make changes to the container instance held in the node. This means that we waste resources creating the copy of the tree node, and risk "false" conflicts on the tree node. Again the traits class is useful: we create a special-case of the container type's non-const iterator, such that if it iterates over a composed type, dereferencing is done by opening the node in read-only mode, and casting away the constness before returning the reference. In the above example, this means that the only objects opened in read-write mode are the list's last-sentinel and the last element of the list; this is the minimum possible. This all happens completely transparently to the programmer using the data structures.

4.2.1 Recursive wrapping

We introduced the idea of wrapper classes to avoid explicitly passing the transaction object reference to every method on a container and associated iterators. We would like to let these behave properly on the nested containers as well; writing the following should be possible:

```
estm::transaction tx;
ATOMIC_BEGIN(tx) {
  estm::map<int, estm::list<int> >& m=....;
  //Get a RO wrapper of the map:
  estm::mapRO<int, estm::list<int> > mro(tx, m);
  //Iterate over the map:
  for (estm::mapRO<int, estm::list<int> >::const_iterator iter=mro.begin();
       iter!=mro.end(); ++iter)
  {
    //Dereferencing the iterators returns RO wrappers of the lists:
    int sum=std::accumulate((*iter).begin(), (*iter).end(), 0);
  }
} ATOMIC_END(tx);
```

In the above examples, dereferencing the `mapRO` iterators does not result in a reference to `list<int>` but instead an instance of `listRO<int>`. This is done by modifying the iterator wrappers `read_iterator` and `write_iterator` such that their return value depends on the type iterated over. For non-composed types, they simply return whatever their inner iterators return when dereferenced. For composed types they create an instance of the container's wrapper class. This is done by adding a `tx_wrapper` class template; the primary template looks like this:

```
template<typename T>
struct tx_wrapper
{
    typedef T& wrapper;
    static wrapper get(estm::transaction& tx, T& l)
    {
```

```
7           return l;
8       }
9   };
```

This simply defines that in the non-composed case, the return type from the read and write iterators should be a reference to the type iterated over; a simple identity function that ignores the transaction object reference is supplied as well.

For the composed type, two partial specializations are used; the ones used for list are:

```
1   template<typename E>
2   struct tx_wrapper<const estm::list<E> >
3   {
4       typedef estm::listRO<E> wrapper;
5       static wrapper get(estm::transaction& tx, const estm::list<T>& l)
6       {
7           return wrapper(tx, l);
8       }
9   };
10  template<typename E>
11  struct tx_wrapper<estm::list<E> >
12  {
13      typedef estm::listRW<E> wrapper;
14      static wrapper get(estm::transaction& tx, estm::list<E>& l)
15      {
16          return wrapper(tx, l);
17      }
18  };
```

The `operator*` for `write_iterator` is modified to the following (see 4.1.1 for the non-nesting version):

```
1   template<typename T>
2   class write_iterator
3   {
4     estm::transaction& tx;     //Reference to transaction object
5     T                  inner; //The wrapped iterator
6   public:
7     typedef typename T::value_type         value_type;
8     typedef typename tx_wrapper<value_type>::wrapper reference;
9   ...
10    reference operator*()
11    {
12      return tx_wrapper<value_type>::get(tx, inner.deref(tx));
13    }
14  ...
15  };
```

This way all the logic for wrapping is contained within the iterator wrappers, the container wrappers and the helper templates; none of the containers have this logic inserted into them. This makes it feasible to construct a distinct wrapper-mechanism without altering the containers at all.

4.3 General considerations for data structures

In the following we discuss some general points necessary for creating good STM-based implementations of data structures.

4.3.1 Running times and starvation

Some common implementations of common data structures have methods with good amortized running times, but poor worst-case running times; e.g. `std::vector::push_back` is often implemented to run in amortized constant time, but with a worst-case time linear in the number of elements (since its underlying array is doubled in length when it fills up). Consider the case of two transactions being run repeatedly and concurrently: the first updates the first element in a vector, while the second appends elements to the vector. The first transaction must open for reading a pointer pointing to the underlying array, and then open the first element for writing. The second transaction also reads the pointer to the underlying array, reads a value describing the capacity of the array, and finally updates the used length and inserts the new element; all provided there is spare capacity in the array. If not, a new array must be allocated, and all of the elements copied. However as the vector becomes larger, copying the elements takes longer time; while the copying is happening, the first transaction writes to the first element, thus forcing the second transaction to (eventually) abort. This means the second transaction will be virtually guaranteed never to commit; the second transaction is victim of starvation. While some steps can be taken to avoid problems with starvation by using contention managers (see Section 2.4), it seems better to avoid the problem to begin with: containers with methods with bad worst-case behavior should be avoided if possible.

4.3.2 Sentinel nodes and false conflicts

Several data structures (e.g. lists and red-black trees) are often described and implemented with the use of sentinel-values [4], to reduce the number of special cases in the algorithms when dealing with the "outer" elements in the data structures (first and last nodes in a list; root and leaf nodes in red-black trees). The red-black tree described by Cormen et al. [4] uses just one common sentinel for *all* leaves. However, doing so for STM-based implementations could cause otherwise unconflicting transaction to have conflicts on the sentinel. STM-based data structure implementations should therefore have separate sentinels or avoid them altogether, even though this can complicate the implementations[2].

4.3.3 Storage granularity

The nodes in many data structures, e.g. lists, store navigation pointers in addition to the actual data in the node. The navigation pointers may be updated without the actual data being altered; in a list this happens when an existing node gets a new neighbor. If the node exists as a single shared variable, altering the pointers means that the whole node must be copied when it is opened for writing. While this is not a problem for a list of `int`s, it may be for a list of a data type that is more expensive to copy. In such cases it may be advantageous to store the actual data in a separate shared variable, and let the node have a pointer to this instead of storing the data inline. It may be reasonable to make this choice type specific. This can be done by adding another entry to the traits class, indicating if the type should be stored separately; we have not done so in the current implementation.

Indeed, this idea can be taken further: the individual navigation pointers could be stored as separate objects; this can in some cases reduce the number of conflicts, at the expensive of more shared variables that must be opened.

4.3.4 The `empty` method

While we have decided not to implement (constant time) `size` methods on our containers, due to the contention it would generate, it would still be desirable to have an `empty` method on the containers. However, such a method is often built by testing for `size` to return 0. Instead we need

[2]Even in a lock-based approach, using a single sentinel would force the cache line holding the sentinel to move between different processors' caches, which could actually slow down the lock-based version as well.

to check for the existence of nodes in the data structure. Using lists as an example, this requires either opening the first or the last sentinel, and checking if it points to an actual node. While this works, an implementation must choose *which* sentinel to check. If the first sentinel is checked and the element at the end of the list is then accessed, this means the transaction opens *both* sentinels, even though opening the last sentinel would have been sufficient. In case another concurrently running transaction writes to the first element in the list, this would cause the first transaction to unnecessarily abort. However, unless we implement different versions of `empty`, thereby letting the user specify which sentinel to check, we cannot choose the correct sentinel in all cases.

Instead, it may be advantageous to add to the container a boolean variable (separate from the sentinel objects) indicating if the list is empty: this variable is only written when the list goes between being empty and non-empty, and reading this instead of the sentinels to determine emptiness is then not a cause of contention. Of course, such an additional boolean adds a bit of space overhead and runtime overhead when written. It does not however generate any conflicts by itself.

4.4 Some specific data structures

Most implementations of STM have an implementation of a red-black tree. The standard library also has such an implementation used for the `set` and `map` class templates. We have therefore implemented an STM-based version of the `map` class template. In addition we have implemented the `list` data structure as well. We discuss the `vector` structure too.

4.4.1 `map`

Our implementation of `map` uses the red-black tree algorithms as described by Cormen et al. [4]; however we do away with their sentinel object, and use `NULL`-pointers where Cormen et al. would have used the sentinel. In addition, in order to support the `begin` and `end` methods in constant time, and to iterate "backwards" from the `end` (one past the last) position, we introduce a different sentinel type from that of Cormen et al. One sentinel object points to the first node, another points to the last node and a third points to the root node (or they point to each other, in case the tree is empty). The first (left-most) node in the tree has a pointer to the first sentinel and the last (right-most) node in the tree has a pointer to the second sentinel. All the relevant methods are updated to maintain these pointers. Finally, the deletion routine differs from Cormen et al.'s in that ours never copies element values from one node to another; instead we only alter pointers during deletion. We do this to maintain iterator validity for all the elements that are not removed.

A common `std::map` implementation technique is to divide the nodes into two classes:

```
1  struct node_base
2  {
3    node_base* parent;
4    node_base* leftchild;
5    node_base* rightchild;
6    color_t    color;
7  };
8  template<typename K, typename V>
9  struct node : node_base
10 {
11   K key;
12   V value;
13 };
```

This way the normal nodes are full `node<K,V>` instances and the sentinels are simply `node_base` instances. This way most of the code navigating the tree does not need to distinguish between nodes and sentinels. However, this will not work in a STM context, as transactions need to open

the objects either for reading or writing. In either case we need the object's actual type; if we open the object with the wrong type passed to `openRO/RW`, the behavior is undefined and certainly wrong when opening for writing. In STM contexts this base-class trick simply does not work by itself. Our implementation instead handles all the special cases explicitly.

Our iterator type, when referring to a valid element in the tree, has a pointer to the node in the tree; when referring to the `end` element, it instead points to the second sentinel. Since the nodes and sentinels are different types, we use the least significant bit of the iterator's pointer to determine if the iterator refers to a proper node or the `end` sentinel.

To make the nodes as compact as possible, we store only three pointers per node, plus of course the key and value of the node. The three pointers are `parent`, `leftchild` and `rightchild`. The color of the node is encoded into the least significant bit of the `parent` pointer; likewise, the first and last nodes which are the ones pointing to the sentinels, uses their `leftchild` and `rightchild` pointer for this, and use the least significant bit of the pointers to indicate that they point to a sentinel instead a proper node. This way a node of `map<int,int>` only needs $(3 + 2) * 4 = 20$ bytes; this means (see Section 3.8.2) that we can store two node versions in the same cache line as their metadata. This compaction trick is also described by Brönniman and Katajainen in [3].

4.4.2 `list`

Our `list` implementation uses much the same sentinel tricks as the `map` implementation. The implementation is quite minimal though: none of the sort, reverse, merge, splice etc. methods are implemented.

4.4.3 `vector`

In Section 4.3.1 we discussed the disadvantages of implementing data structures with bad worst-case running time. For this reason, implementing a `vector` with the normal array-doubling technique would be a poor choice. In fact, the reason the array-doubling technique is used in the normal `std::vector` implementations is the requirement that the elements be laid out contiguously in memory, such that the underlying array can be used with C-style functions. Since this is impossible with the STM-based version anyway, there is very little justification for using the array-doubling technique. Instead the *levelwise-allocated piles* method (e.g. as described by Katajainen and Mortensen [16]) could be a reasonable choice, as the method performs no copying when inserting at the end of the vector. Random access in this case is still a constant time operation, but the constant is larger than for the array-doubling technique; this cost may well be insignificant compared to the cost of opening the elements. However, we have not implemented the data structure and therefore have no benchmarks to back up this speculation.

Chapter 5

Experimental evaluation

In this chapter we look at performance of the STM library, with emphasis on determining the benefits of being able to layout objects in the same cache line as their metadata.

We have implemented a microbenchmark that we use throughout this chapter. The benchmark consists of operations on a red-black tree. To provide some perspective of the performance of the STM implementation of the red-black tree, we compare against the implementation from the standard template library (STL) where we use coarse-grained locking when using more than one thread.

In the following we describe first the benchmark and the environment; we then analyse the results of the benchmark.

5.1 The benchmark

The benchmark is quite simple: we perform a constant number of operations on a red-black tree whose keys and values are both integers. The operations are insert, erase and lookup; each is performed in turn. In all runs we perform a total of 20 million operations. This means that most tests run in between 10 and 120 seconds on our test machine. This quite large runtime serves to make start up and timing delays insignificant although we still do try to reduce them as much as possible.

For every operation on the tree we first determine a random number k that serves as the key for the operation. The range of the random numbers is between 0 and $n-1$ where n is one of the benchmark parameters. The insertion operation inserts the key/value pair (k, k) into the map; if the key/value pair already exists in the tree, the value is written to (it is a blind write, which does not actually change the contents; however, the node in the tree is opened in read-write despite this). The erase operation looks up a node with key k and if it exists, it is removed; otherwise the operation alters no data. The lookup operation simply performs a lookup of k and ignores the result.

When m elements exist in the tree, the probability of a random value $k \in \{0, 1, \ldots, n-1\}$ being in the tree is m/n. The probability of a given insert operation increasing the number of elements in the tree is then $p_i = 1 - m/n$; and the probability of an erase operation removing an element is $p_e = m/n$. Initially the probability of insertion is high and erase is low; since there are as many insert as erase operations, they converge at $p_i = p_e = 0.5$, and the number of elements in the tree is then $m = n/2$. The exact size of course fluctuates a bit during the benchmark.

We ran our benchmarks with 1 to 8 threads and $n \in \{2^8, 2^{10}, 2^{12}, 2^{14}, 2^{16}, 2^{18}, 2^{20}\}$; for the STL version we also ran with $n = 2^{22}$. Apart from the STL version with coarse grained locking, we have run five different configurations or setups of the STM version. By configuration or setup we mean the number of slots allocated for the different types in the program. Recalling that the red-black tree has two types of shared variable (nodes and sentinels) that have different sizes, the following configurations make sense to test (the configuration's names are in parenthesis):

- `(n02)`: No slot for the nodes and two for the sentinels. The nodes' handles are allocated with a normal heap allocation. This is the configuration that a non-compile-time polymorphic implementation would use for the nodes. The sentinel handle is allocated with the cache line aligned allocator.
- `(c02)`: As above, except the memory for the nodes' handles is allocated from the cache line aligned allocator, and the handles thus use a full cache line. We expect this to be worse than `n02`, but include it so that we can show that the (expected) benefits of the next setups are not caused by the cache line aligned memory allocator itself being quicker than the heap allocator.
- `(c12)`: For the node type one slot exists in the handle's cache line; for the sentinel there is two slots. This setup would be particularly relevant if only one slot would fit in a cache line. In any case at least every other node allocation has to be from the heap.
- `(c22)`: For the node type two slots exist in the handle's cache line; for the sentinel there are two slots.
- `(c23)`: For the node type two slots exist in the handle's cache line; for the sentinel there are three slots. This is the maximum number of nodes and sentinels that can be colocated in a single cache line. We use this setup to see if there is any advantage to having three sentinel slots instead of just two.

The sentinels have two slots in all but the last setup; any difference in performance between the first four setups is therefore just the result of the different allocation methods for the nodes.

In all but the first configuration, when a new shared node object is created or destroyed, space for the handle is allocated from or deallocated to the cache aligned allocator. When space is needed for a new version of an object, and there are no (free) slots to allocate from, the allocation happens from a free list stored for each thread; we do not use the global heap for this. This means that the number of global heap allocations is the same in all runs; any differences in run time between the different setups are thus *not* caused by differing amounts of time spent in the heap allocator. `n02` however differs from the other configurations in that it uses the normal heap allocator instead of the cache aligned allocator for the handles. It has the advantage of a smaller memory footprint than `c02`; however the cost of the allocations themselves (which may differ between the two allocators) has not been examined.

The retry logic was disabled in all tests, primarily because the implementation does not currently store the allocation bitmap in the lower bits of the pointer to the linked list used for waiters. Removing the retry pointer allowed two node slots in a single cache line. Had the bitmap been stored in the lower bits of the pointer to the waiters, the performance of the benchmark would have been unchanged.

We use the `mt11213b` random number generator provided by the Boost Random Number Library [2]; it is reasonably fast and has only a small memory footprint, so it will not cause a lot of cache line misses or evictions. The C++ runtime's `rand` function is not an option as it is not guaranteed to be thread safe. Profiling (on our development machine) showed that in the single threaded case, the random number generation accounted for about 20 to 25 percent of the total runtime depending on the value of n; it does therefore increase the scalability of the benchmarks somewhat. The random numbers are generated *outside* of the transactions; a random value is reused if a transaction restarts.

For every run we recorded the wall clock time to complete the operations, the number of transaction aborts/restarts and the number of allocations that could not be satisfied with a free slot.

As the test machine (discussed next) has eight cores we ran out tests with between one and eight threads.

5.1.1 Test environment

The test machine used is a Dell PowerEdge SC1430, with two Quad-Core Intel Xeon E5310 processors running at 1.6 GHz. Each Quad-Core processor is essentially two dual-core processors on the same die. Each dual-core processor has 4 MB of shared level 2 cache. The machine runs 32-bit Windows XP SP2 and has 2 GB RAM.

We used the C++ compiler and STL implementation that are part of the Microsoft Visual Studio 2005 suite; the programs were compiled with the `/O2` optimization flag.

The Intel Xeon processors have a feature known as *adjacent cache line prefetch* (ACLP); this was disabled in the primary run of tests. We discuss ACLP and how it affects performance in Section 5.4.

5.1.2 Test method

Every test was run three times. A few obvious outliers have been removed manually from the dataset and additional runs performed to replace the outliers. The outliers were caused by the machine having occasional non-benchmark activity which could not be avoided. The average values from the tests have been used.

All statistics collected during the benchmarks was collected on a per-thread basis, and accumulated after the operations were completed. Critically, the statistics collection caused no cache line bouncing, which could otherwise have had a large effect on the runtimes.

5.2 Evaluation

5.2.1 Restarted transaction

Recalling that the average size of the red-black tree during the benchmark is $n/2$, we expect that the number of conflicting transactions is large when n is small; the probability of two transactions inserting or erasing a node with equal key is high. Since an insert or erase operation also alters pointers in the adjacent nodes in the tree (and sometimes alters even more nodes' pointers when the tree is reordered during the fixup phase of insert and erase), we expect conflicts even when the keys are just almost equal. In Figure 5.1, we show the number of restarted transactions per million operations for different values of n for the `c22` configuration. As expected for small n the number of restarts grows large. However, even in the case of 8 threads and $n = 2^8$ only approximately 15 percent of the transactions are restarted; this despite the tree only holding an average of 128 elements at a time. This behavior is similar for the other configurations (not shown).

5.2.2 Allocations

In Figure 5.2 we show the number of allocations that could not be satisfied with an empty slot, and thus had to be allocated from the threads' free-lists instead. Note the correlation with the number of restarted transactions in Figure 5.1: the number of non-slot allocations is approximately linear in the number of transaction restarts (the factor is around 3.5). This is quite logical: when two (or more) transactions conflict, they often have write-write conflicts. When they both attempt to create a new version of the same object, they both need to allocate room for a new version; however, there will rarely be more than one free slot available. Since we do not detect the conflicts before one of the transactions commits, often at least one of them will have to allocate from its free-list instead.

The `n02` and `c02` configurations obviously have many more non-slot allocations, as there are no node slots; likewise the `c12` has about half the non-slot allocations of `n02` and `c02` as at most half of the allocations can be satisfied from a slot.

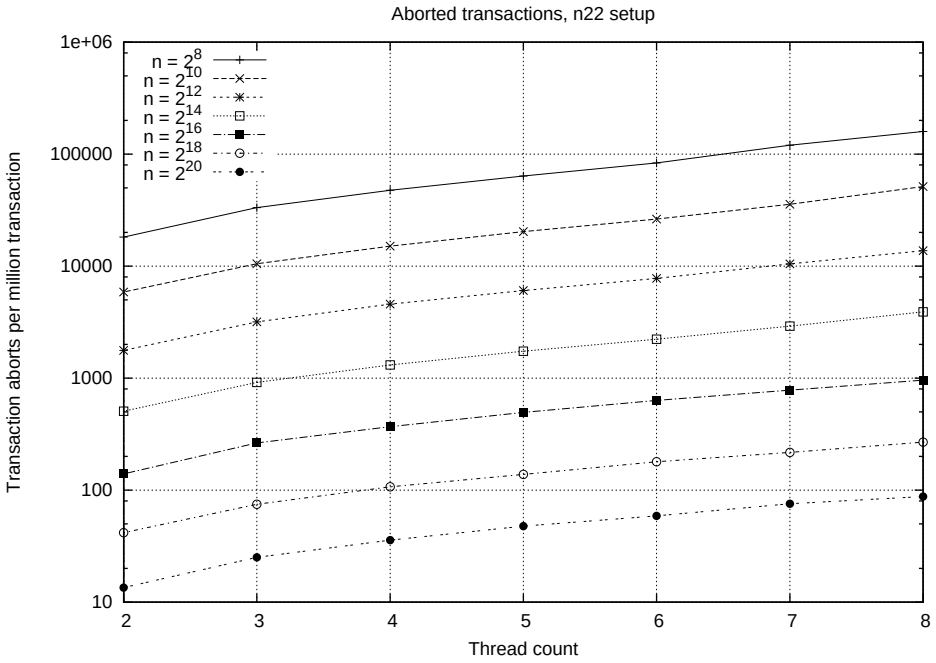

Figure 5.1: *Aborted/restarted transactions:* The number of restarted transactions per million operations. Note the y-axis is logarithmic. As the data sets grow, the number of restarts decreases. For runs with only 1 thread, there are no restarted transactions (not shown).

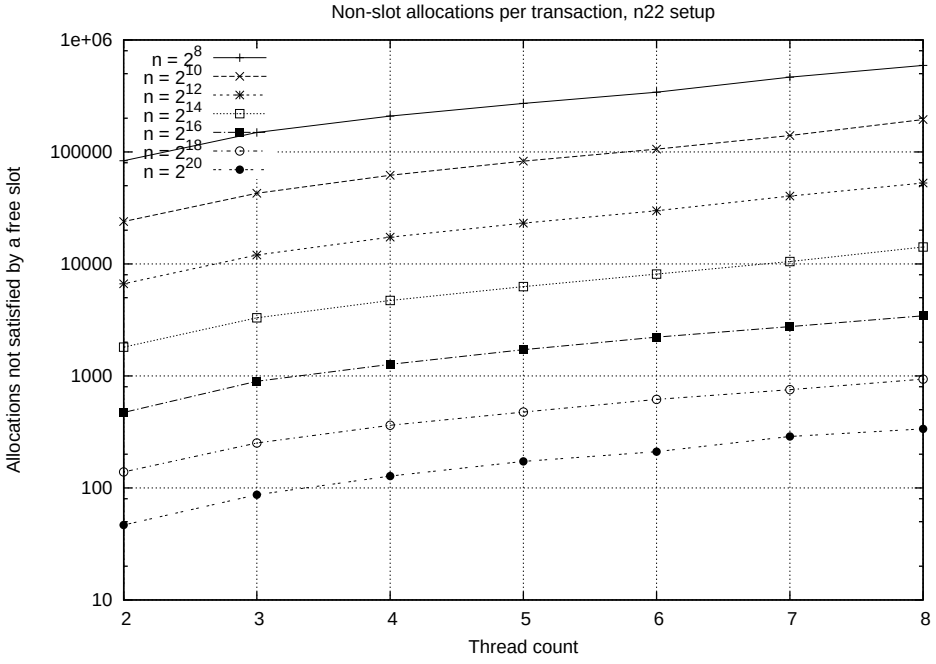

Figure 5.2: *Non-slot allocations:* The number of allocations which could not be satisfied by an empty slot, per million allocation attempts. Note the y-axis is logarithmic. As the data sets grow, the number of non-slot allocations decreases. For runs with only 1 thread, there are no non-slot allocations (not shown).

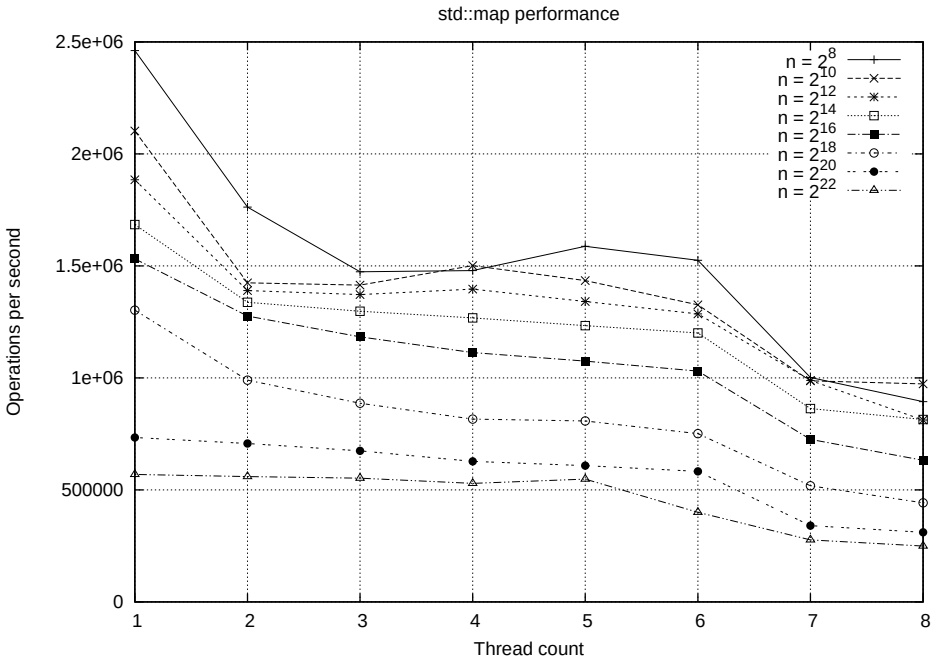

Figure 5.3: `std::map`: The number of operations per second when using the STL-version of the benchmark. Coarse-grained locking is used, so adding threads mostly serves to increase cache line bouncing. For small n the entire dataset fits in the level 2 cache and multithreading is thus relatively worse.

5.2.3 Run time

STL results

To put the performance of the STM configurations into perspective, we ran benchmarks with a non-STM implementation of the red-black tree (using the `std::map` implementation that came with the compiler used). In Figure 5.3 we show the performance for the different values of n and different number of threads. A coarse grained spin lock is used to allow only one thread at a time to access the tree.

There are a few interesting things to note here: since the benchmark's calculation of the next random value happens while the lock is not held, the performance should increase as the number of threads increase. This actually happens for the smallest values of n when the number of threads increases from two to three or four. However, when more than one thread works on the data, the data has to be moved between the different CPU core's caches (both the individual level 1 caches and the level 2 caches shared by two cores). Since all of the insert operations and half of the erase operations actually perform writes to the data, a lot of cache line bouncing occurs, and the performance degrades as more threads are added. There is quite a large drop in performance when more than six threads are used; we do not have a good explanation for this.

For the single-threaded case, there is a large drop in performance as n goes from 2^{18} to 2^{20}. Every node in the tree in the STL version uses 24 bytes, plus perhaps some overhead from the allocator (which may be caused by book-keeping or internal fragmentation). When $n = 2^{20}$, the tree uses on average at least $24 \cdot 2^{17} = 3$ MB; when $n = 2^{20}$, the tree uses on average at least $24 \cdot 2^{19} = 12$ MB. As the level 2 cache is only 4 MB large, at $n = 2^{20}$ the tree no longer fits in the cache, and performance decreases as a result. As expected the performance difference between $n = 2^{20}$ and $n = 2^{22}$ is much smaller than between $n = 2^{18}$ and $n = 2^{20}$, since for the former the cache size is exceeded in both cases.

The STL version is included here for perspective for the STM version; the remaining graphs show the performance results of the STL version along with the STM version.

STM results for large data sets

Recalling that the primary goal of the benchmarking is to determine if the layout technique is worthwhile, we are most interested in the relative performance of the STM configurations. In Figure 5.4 we show the number of operations per second for $n = 2^{20}$. The `c22` and `c23` setups perform about the same; this is expected as the number of non-slot allocations for $n = 2^{20}$ is small as shown in Figure 5.2 and there is thus very little benefit to having an additional sentinel slot. The `c02` setup performs a bit worse than the `n02` setup; this is also expected as the handles in the `c02` setup fill an entire cache line, while the handles in the `n02` setup only use 4 bytes (plus perhaps some allocation overhead); the lower memory footprint of the `n02` setup allows more handles to occupy the same cache line, and this results in fewer cache misses overall since the total memory footprint is smaller.

Most interesting is the difference between the `n02` and `c22` setups: the `c22` setup manages about a third or fourth more operations per second than the `n02` setup (which is the setup that indirection-based runtime-polymorphic STM implementations would use). This is of course due to the handle and the container being in the cache line; despite the larger total memory footprint compared to `n02`, accessing a shared object causes at most one cache miss, whereas in `n02` it may (and often does) cause two cache misses.

The `c12` setup goes in between the `n02` and `c22`; only on every other access on average will the container be in the same cache line as the handle. However the `c12` setup has the disadvantage of a larger memory footprint than both `n02` and `c22`, as a whole cache line is taken for a handle with just one slot, and that on average the slot is unused half the time, and a non-slot allocated container is used instead. The `c12` setup may still be interesting though, as had the red-black tree's nodes been larger, there would have been space for just one slot in a cache line. We look at this case in Section 5.3.

In Figure 5.4 we show also the performance of the coarse grained locking STL version. The STL version outperforms the STM version on a single thread by a factor of 1.75 for $n = 2^{20}$. This is expected as the STL version has about half the memory footprint of the STM version, and does not have the STM system's overhead. That the factor is not larger is mainly due to the large cost of cache misses; the cost of the cache misses is significant compared to the overhead of the STM version. In Figures 5.5, 5.6, 5.7, 5.8, 5.9 and 5.10 we show the performance for smaller values of n. As n decreases the STL version performs much better relative to the STM version in the single threaded case. The relative overhead of the STM system (copying, indirection, validation etc.) becomes larger as an increasing part of the red-black tree fits in the cache.

For the larger runs we see that the number of operations per second increases as we increase the number of threads; for $n = 2^{20}$ up to 6 or 7 threads can be used to good effect; for $n \in \{2^{12}, 2^{14}, 2^{16}, 2^{18}\}$ up to 6 threads improves performance. Adding the 7th or 8th thread decreases performance. As the number of threads increase, the memory bandwidth of the system as a whole probably becomes a bottleneck; adding more threads at this point then only serves to increase the number of cache lines that must be moved between the CPUs.

Considering $n = 2^{20}$, we see that with 3 threads the STM version outperforms the singlethreaded STL version; with 6 threads it runs at twice (1.96) the speed of the single threaded STL version. A small part of this is due to being able to calculate the next random numbers concurrently; however, this does not explain the whole difference. The only plausible explanation for the increased performance is better use of the memory bandwidth of the system. When the STL version is descending through the tree during an operation, the first several levels of the tree can be expected to be in the cache; it is only when the higher levels (those farthest from the root) are accessed that main memory must be consulted. Memory requests are then only made during the higher levels, and the memory bus sits idle the rest of the time. In the STM version the memory bus is in use a larger portion of the time.

For the large datasets there is no or very little advantage to having an extra slot for the sentinel.

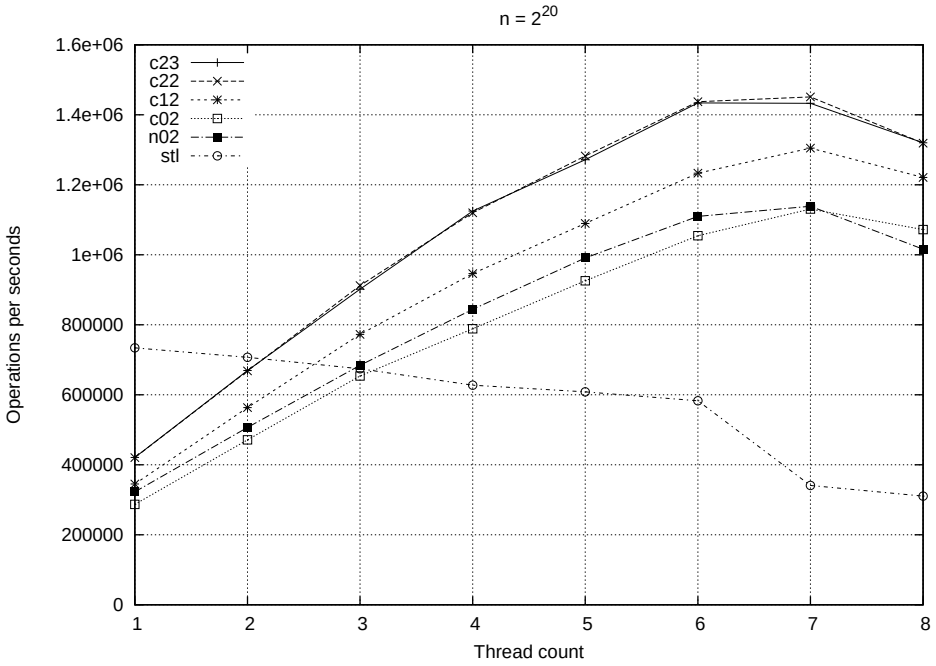

Figure 5.4: *Operations per second:* On large datasets the STM version scales well up to seven threads, and outperforms the STL-version when using more than two threads. There is significant advantage to using a good layout.

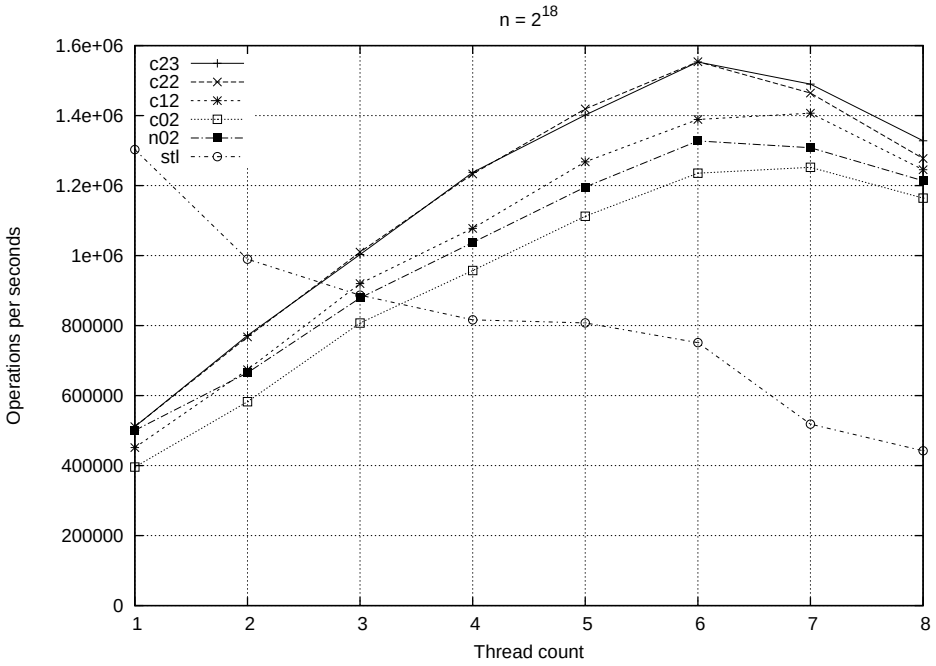

Figure 5.5: *Operations per second:* The STM version scales well up to six threads, and slightly outperforms the STL-version when using five, six or seven threads. There is significant advantage to using a good layout.

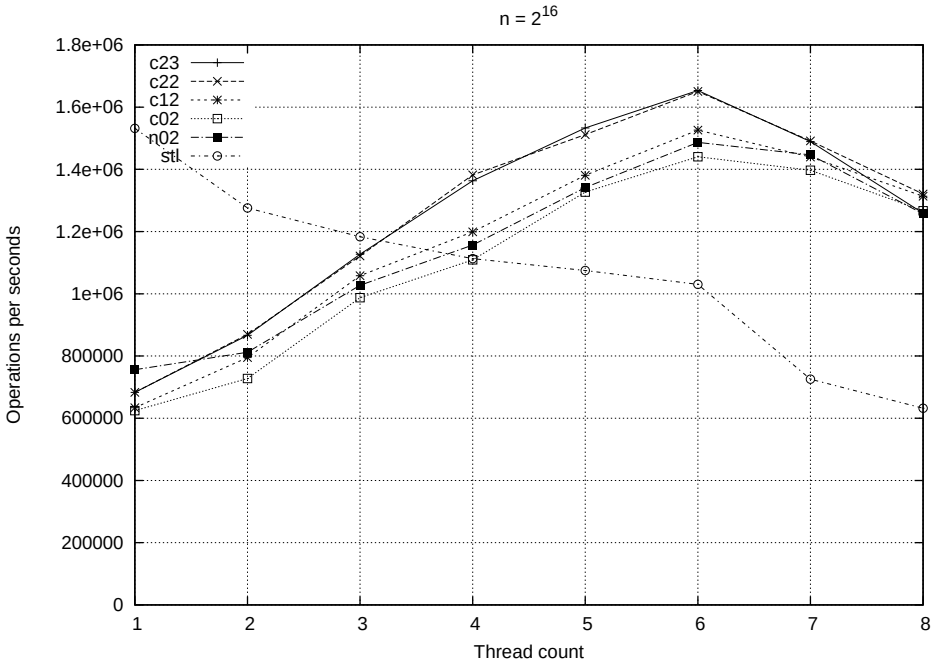

Figure 5.6: *Operations per second:* The STM version scales well, and slightly outperforms the STL-version when using six threads. There is some advantage to using a good layout.

This is not surprising as the sentinels are very rarely written to. The additional slot is therefore unused most of the time; its presence does not hurt performance for the large datasets.

STM results for small data sets

We note first that as the problem size shrinks, for the single threaded STM version the number of operations per second increases up to approximately 1 million operations per second in the case of $n = 2^8$ and $n = 2^{10}$. We also see (Figures 5.4 through 5.10) that in the single threaded case the relative performance advantage of the STL version increases as the problem size shrinks case, even when the entire tree fits in the cache. As the amount of "real" work per operation decreases, the constant part of the overhead of the STM version grows relatively large.

We discussed the number of transaction restarts shown in Figure 5.1; as the number of restarts increase the scalability is decreased: in Figure 5.10 we see that for $n = 2^8$ there is no gain to using more than one thread. In fact for some of the setups, moving from one to two threads means the program runs slower. This is partly due to approximately 15 percent of transactions being restarted, and partly because adding the second thread means that data has to shared between the two processors; in the single threaded case most, if not all, of the tree could be stored in the cache (perhaps even in the processors level 1 cache) as the tree only takes up around 8 KB. Moving from two to three or more processors has a slight advantage, as data must in all cases move between processors and there is some gain from concurrently calculating the random values. In all cases it is however hard to tell if the poor scaling is a result of increased transaction restarts or due to cache line bouncing.

When $n \in \{2^8, 2^{10}, 2^{12}, 2^{14}\}$ and only a single thread is used, the `n02` and `c02` setups are at a small advantage compared to the other configurations. As the entire tree fits in the cache in all these cases, there are no or few cache misses, so these setups do not have their usual disadvantage. The difference is probably a result of allocation from a free slot being more expensive than allocating from the free list. In the latter case, the only work to be done is to pop the first

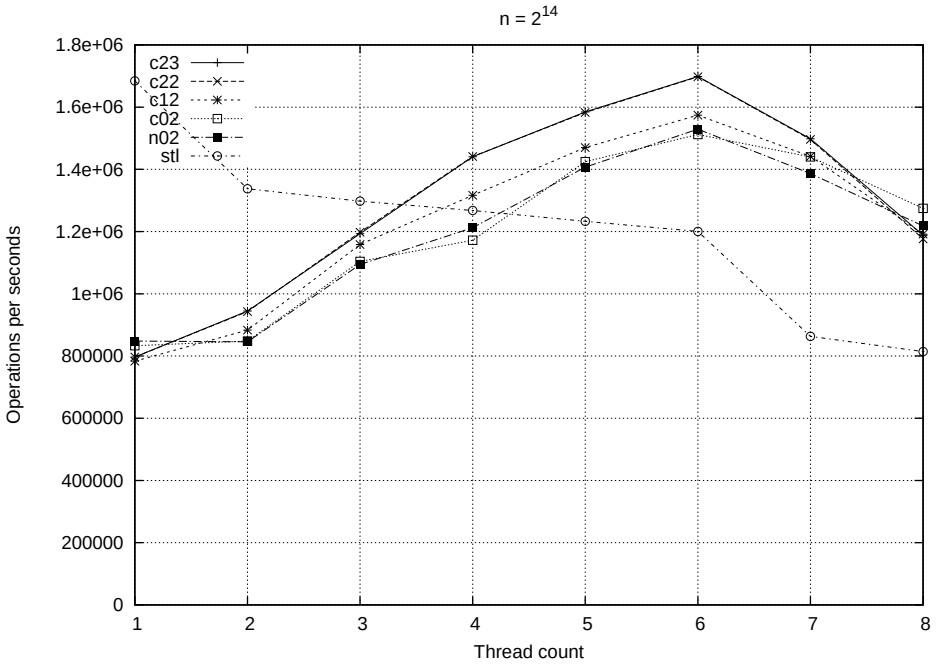

Figure 5.7: *Operations per second:* The STM version scales reasonably up to six threads, but cannot outperform the STL-version. There is some advantage to using a good layout.

free memory from the free list. The free list is a singly linked list, so removing an element is very cheap. By comparison, allocating (and at some point deallocating) a slot requires the use of a compare-and-swap operation; this is a relatively expensive operation, and may account for the difference.

For the small datasets, the sentinels are written to more often than for the large dataset. This means that adding an extra slot for the sentinels might be beneficial. However as for the large datasets there is hardly any observable benefit to adding the extra slot. In fact, the number of non-slot allocations is hardly different between `c22` and `c23`. In this benchmark at least, there seems to be no benefit to the third slot for the sentinel values.

5.3 Large nodes

To determine if the `c12` setup is the best choice when only one slot can fit in a cache line, we ran parts of the benchmark again with a larger node size; now the key-type in the tree is no longer an integer, but 8 integers. This means that there is room for exactly one slot alongside the handle in a single cache line. We have only run the benchmark with $n = 2^{20}$ and the `n02`, `c12` and `c22` setups; the sentinel size is unchanged and we colocate two slots for the sentinel in all the runs.

In Figure 5.11 we show the number of operations per second for the different setups. As only one node slot can fit into the cache line with the handle, we expect the `c12` to perform best; it has only about 1.5 times the footprint of the `n02` setup, and approximately every other node access will be done without a cache miss. The `c22` has approximately twice the memory footprint of `n02` setup and still approximately every other node access will be done without a cache miss, since the second slot in `c22` is in a separate cache line.

As expected `n02` performs worse than the other setups. However, the difference is not as large as for the smaller nodes; in particular when the number of threads is small. As the number of threads increase the relative performance disadvantage of `n02` increases; this is somewhat surprising since

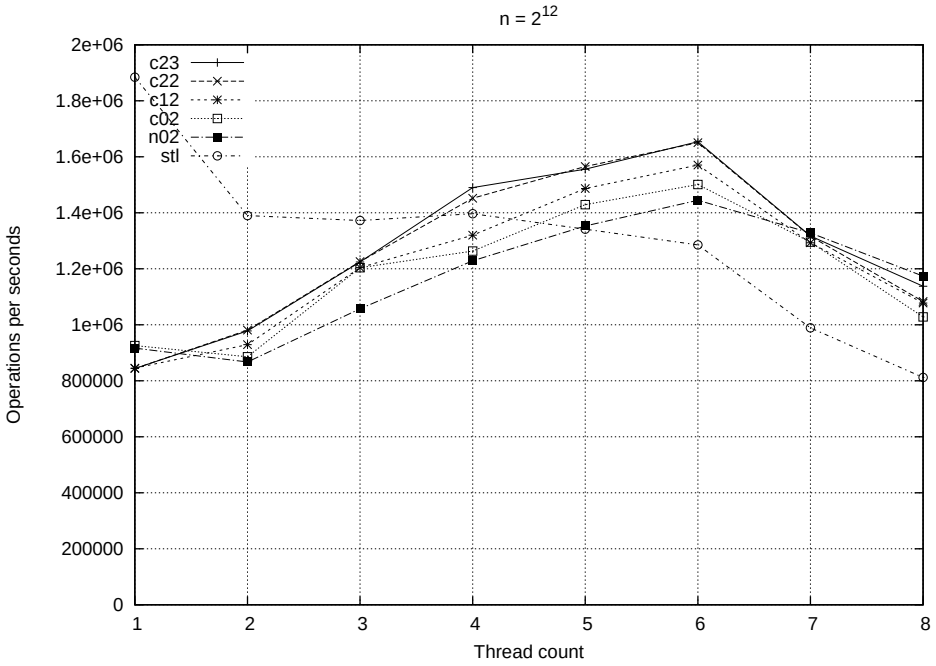

Figure 5.8: *Operations per second:* The STM version scales reasonably up to six threads, but cannot outperform the STL-version. There is some advantage to using a good layout. In this particular instance the `n02` performs worse than the `c02` setup.

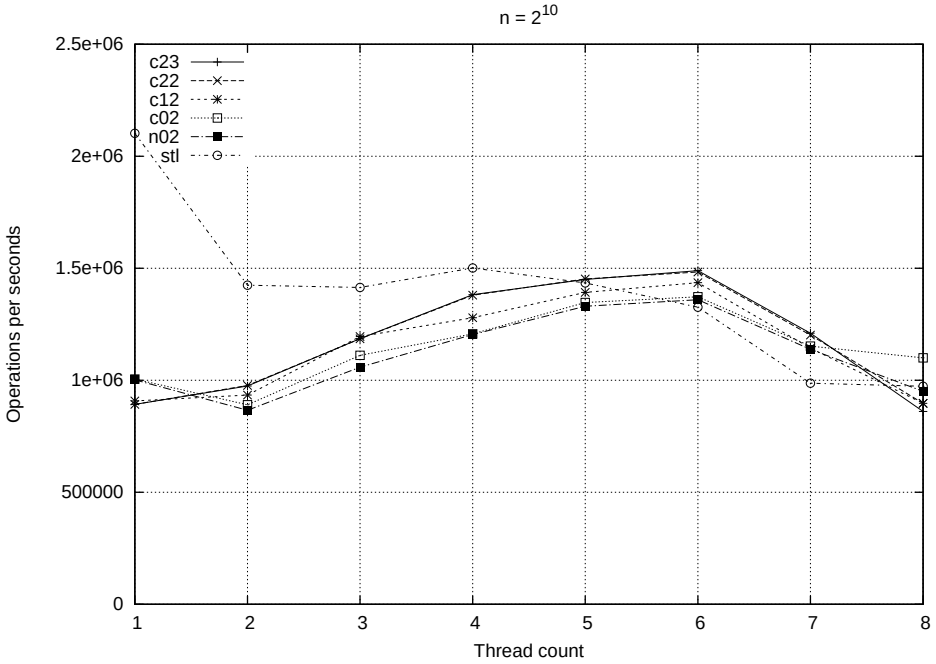

Figure 5.9: *Operations per second:* The STM version does not scale well. There is a small advantage to using a good layout.

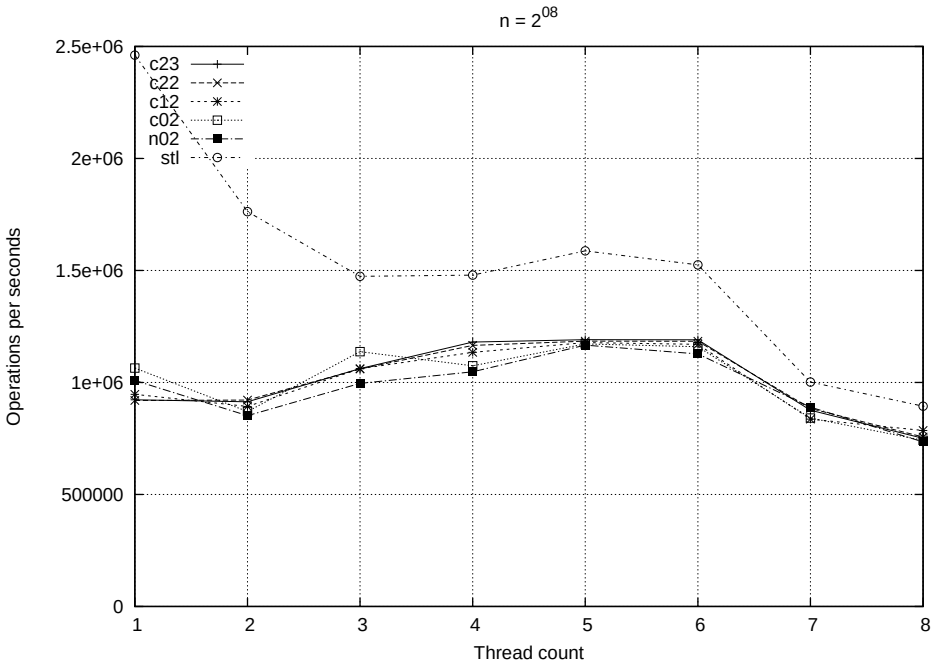

Figure 5.10: *Operations per second:* The STM version does not scale well. The different layouts perform about the same. There is considerable variations in the runtime from run to run (not shown).

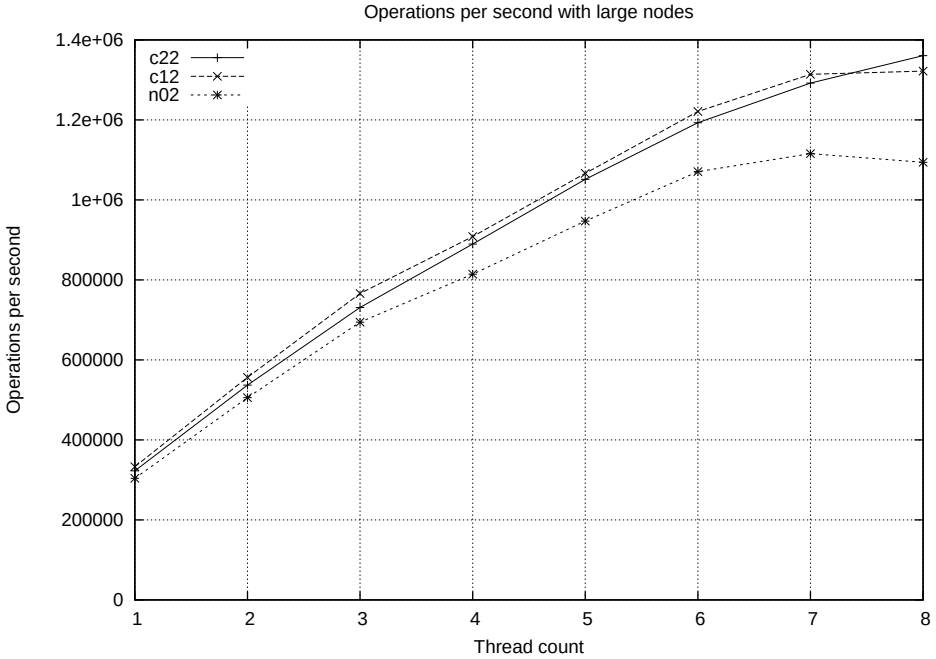

Figure 5.11: *Operations per second:* Large nodes are used; only one node slot can fit in a cache line with the handle. The `c12` layout performs best, though the `c22` layout is close.

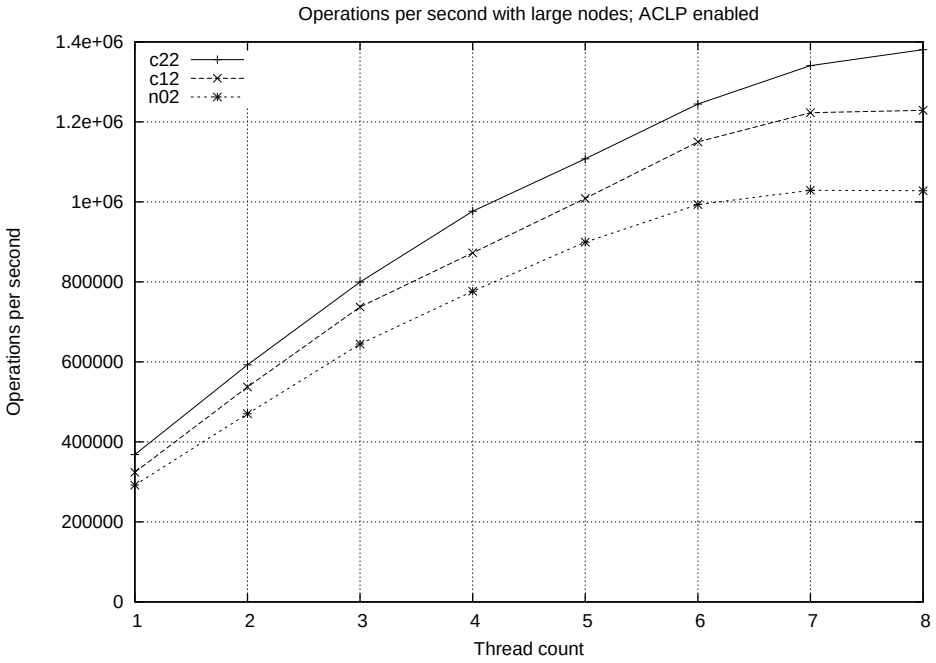

Figure 5.12: *Operations per second:* Large nodes are used and the ACLP feature is enabled; two node slots can fit in two adjacent cache lines. The `c22` layout is significantly better than the others.

the number of cache misses should be about the same independent of the number of threads.

As the `c12` setup is faster than the normal `n02` setup, our layout approach is still relevant even for large node sizes.

5.4 Adjacent cache line prefetch

We mentioned earlier that the test machine has a feature known as *adjacent cache line prefetch* (ACLP), which was turned off during all the benchmarks discussed so far. When this feature is turned on, the CPU will in most circumstances fetch two cache lines at a time; specifically it will fetch the cache line following the cache line we are currently accessing. For our use this means that when we attempt to access some data in a handle (i.e. the pointer to the current container), the following cache line will also be retrieved into the cache; this happens independently of whether we actually access the next cache line or not. We reran all our benchmarks with ACLP turned on, and for the small nodes the difference is very small; the benchmarks as a whole run a bit slower with this feature turned on. The relative advantages of `c12` and `c22` over `n02` are about the same as when ACLP is turned off.

When we use the large nodes, this feature means that there again may be some benefit to using two slots; since 128 bytes are retrieved (and we "pay" for this under all circumstances) both slots are retrieved into the cache, and the `c22` experiences no additional cache misses; the `c12` will every other time experience an additional cache miss. In Figure 5.12 we show the performance for $n = 2^{20}$ with the large nodes with ACLP turned on. We see that again the `c22` setup performs best. Comparing with the large node run with ACLP disabled, we see that the fastest ACLP-enabled version ourperforms the fastest ACLP-disabled version.

The STL version using the small nodes (shown for $n = 2^{20}$ in Figure 5.13) shows a small but consistent performance gain when ACLP is turned off, for up to six threads. Since the nodes take up less than a cache line, the effect of ACLP is mainly negative for our benchmarks: it uses time

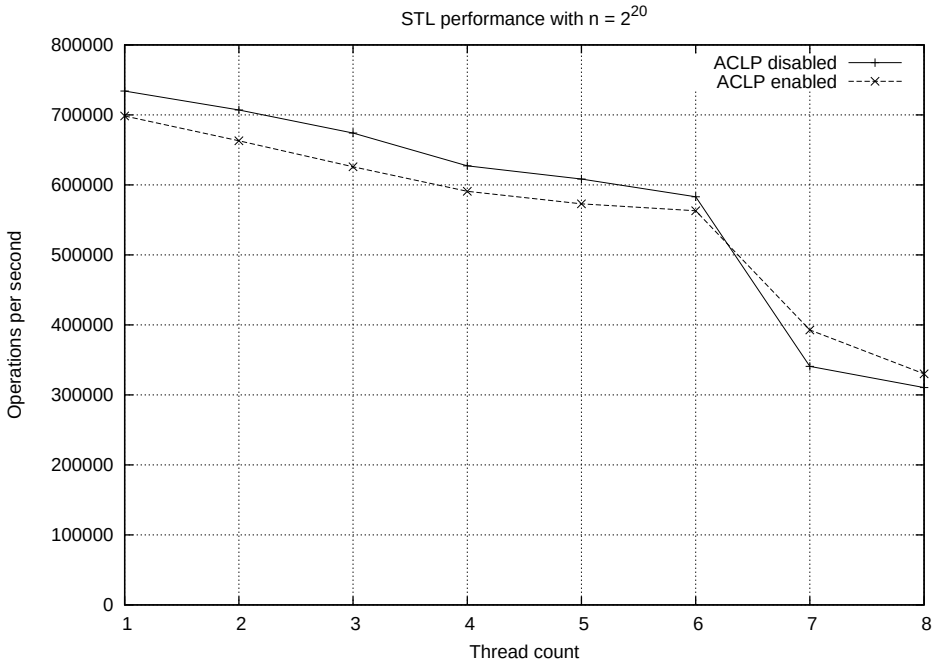

Figure 5.13: *Operations per second:* Comparing the STL runs with and withour ACLP enabled. Small nodes are used. ACLP is a disadvantage with less than seven threads.

and bandwidth retrieving the next cache line, and it must necessarily evict another cache line to make room. Both with and without ACLP the performance drops suddenly when we use more than six threads; we do not have an answer for why the ACLP enabled version performs relatively better than the ACLP disabled version with more than six threads.

5.5 Discussion

This benchmark has not shown a benefit to being able to store more than two slots in a single cache line; additional benchmarks need to be constructed to determine if this is true generally.

Ennals [8] claims approximately a 50 percent increase in performance when comparing his lock-based implementation to Herlihy et al.'s indirection-based DSTM implementation. However, the DSTM implementation requires *two* indirections when opening an object. Ennals cites a reduced number of cache and TLB misses as the cause of his improved performance. For large datasets with low contention we manage to increase the performance in our benchmarks with up to 33 percent when using the best layout. Considering that with our worst layout we only suffer one indirection, our 33 percent performance improvement suggests that we gain a similar performance advantage by using our layout as Ennals does by avoiding indirection altogether.

The test machine we used had four separate level 2 caches shared between eight cores; this makes it somewhat hard to reason about the cache behavior, in particular because the operating system's decision about which cores to schedule a given thread on may significantly effect performance: if two threads operate on disjoint datasets we would like them placed such that they do not share a level 2 cache, but if they do work on the same data we would often prefer that they share the level 2 cache, as this would mean fewer level 2 cache line invalidations. Testing on a machine with many more cores would be interesting as well; in particular to determine if it is our implementation or the test machine that prevents us from achieving higher performance with more than seven threads.

Our performance evaluation would have benefitted from collection of processor statistics of the number of cache line misses; this might have helped us better understand the behavior, in particular when the number of threads increase above six.

A comparison with existing STM implementations would have been interesting as well. However, using just our own implementation, we have been able to show that choosing a proper layout is important for large datasets with little contention.

Chapter 6

Closure

6.1 Conclusions

We set out to choose proper design parameters for a STM implementation for C++. Our analysis provided us with a design that allowed us to create an implementation that provides strong exception guarantees for operations using transactions and that entirely eleminates the problems with inconsistent or dirty reads. Our implementatation also supports the *retry* and *orElse* programming constructs. To our knowledge this is the first C++ implementation to support these constructs.

We also set out to build STL-compatible data structures on top of our implementation, such that most of the STM details are hidden from the user of the data structures. We have accomplished this and have implemented several data structures with interfaces almost identical to those in the STL; with very little effort we can use our data structure implementations with the algorithms in the STL and enjoy the benefits that come from programming with transactions. We have analysed the problems that STM can introduce when implementing data structures and removed the most problematic methods on the data structures to avoid these problems. We have introduced a method of nesting STM-based data structures within one another, such that unnecessary conflicts and copying overhead are avoided.

We have introduced a new layout technique for the transactional object structure such that we reduce the cache performance disadvantages of the indirection-based approach compared to the lock-based approaches to STM. Our benchmarks have shown that we for large datasets with low contention can get up to a 33 percent increase in performance by using a good layout compared to a standard layout. Only in single threaded cases with small datasets does our layout have a small disadvantage. However, we need to implement more benchmarks to determine the general usefulness of the layout technique.

6.2 Future work

A significant amount of the litterature on STM discusses contention management and demonstrates large advantages to choosing a good contention manager. We there believe that if our implementation is to be put to practical use, it needs to be outfitted with a proper contention manager; this should in fact not be terribly difficult to do.

The benchmark we have implemented only uses red-black trees. To determine if our layout technique is generally useful, some additional benchmarks must be constructed. In particular, in the presented benchmark all transactions are short and approximately the same size; longer running transactions may prove to reduce the usefulness of the layout technique — they might also show colocation of more than two slots to be usefull.

Our original thoughts for the implementation also touched on 2 phase commit, such that we perhaps could use our STM implementation along with a proper database. We still consider

this relevant. However, proper databases measure their performance in thousands of operations per second; STM systems measure it in millions per second. A two phase commit needs to be constructed such that non-conflicting STM transactions can complete even though another STM transaction has agreed to commit and is waiting for the coordinator to send the commit or abort message. This would require a very different internal commit protocol than the one described for our implementation.

6.3 Source code availability

We intend to make the source code for the implementation and benchmarks publicly available once it has been cleaned up and altered to fully reflect the described API.

Bibliography

[1] A. Alexandrescu, *Modern C++ Design*, Addison-Wesley (2001).

[2] Boost, *Boost*, `http://boost.org/`. (Accessed January 31, 2008.)

[3] H. Brönnimann and J. Katajainen, Efficiency of various forms of red-black trees (2006).

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Second Edition*, MIT Press (2001).

[5] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum, Hybrid transactional memory, *ASPLOS-XII: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM (2006).

[6] D. Dice, O. Shalev, and N. Shavit, Transactional locking ii, *DISC* (2006).

[7] R. Ennals, Cache sensitive software transactional memory.

[8] R. Ennals, Software transactional memory should not be obstruction-free.

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

[10] T. Harris and K. Fraser, Language support for lightweight transactions, *OOPSLA'03* (2003).

[11] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy, Composable memory transactions, *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, ACM Press (2005).

[12] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy, Composable memory transactions, (Post publication version: August 18, 2006.) (2006)

[13] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, Software transactional memory for dynamic-sized data structures, *PODC 2003* (2003).

[14] Intel, *Intel Threading Building Blocks*, `http://threadingbuildingblocks.org`. (Accessed January 31, 2008.)

[15] L. Ivanov and R. Nunna, Modeling and verification of cache coherency protocols, *ISCAS (5)* (2001).

[16] J. Katajainen and B. B. Mortensen, Experiences with the design and implementation of space-efficient deques (2001).

[17] S. Lie, Hardware support for transactional memory, (Available at `http://supertech.csail.mit.edu/papers/lie-thesis.ps`.) (2004)

[18] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer, and M. L. Scott, Lowering the overhead of nonblocking software transactional memory (2006).

[19] R. Ramakrishnan and J. Gehrke, *Database Management Systems, Second Edition*, McGraw-Hill (2000).

[20] T. Riegel, P. Felber, and C. Fetzer, A lazy snapshot algorithm with eager validation, *DISC* (2006), 284–298.

[21] W. N. Scherer and M. L. Scott, Advanced contention management for dynamic software transactional memory, *PODC'05* (2005).

[22] N. Shavit and D. Touitou, Software transactional memory, *Symposium on Principles of Distributed Computing* (1995).

[23] B. Stroustrup, *The C++ Programmering Language, Third Edition*, Addison-Wesley (1997).

[24] B. Stroustrup, *Appendix E: Standard-Library Exception Safety. The C++ Programming Language, Special Edition*, Addison Wesley (2000).

[25] Sun, *Transactional Memory*, `http://research.sun.com/spotlight/2007/2007-08-13_transactional_memory.html`. (Accessed Dec 19, 2007.)