Cloned Transactions: A New Execution Concept for Transactional Memory

Vom Promotionsausschuss der
Technischen Universität Hamburg-Harburg

zur Erlangung des akademischen Grades

Doktor der Naturwissenschaften (Dr. rer. nat.)

genehmigte Dissertation

von

Holger Machens

aus

Achim, Germany

2015

Date of Oral Examination	April 1st, 2015
Chair of Examination Board	Prof. Dr. Dieter Gollmann Hamburg University of Technology
First Examiner	Prof. Dr. Volker Turau Institute of Telematics Hamburg University of Technology
Second Examiner	Prof. Dr. Sibylle Schupp Institute of Software Systems Hamburg University of Technology

Abstract

Transactional memory aims to replace mutual exclusion in critical sections with transactions on shared data to improve the scalability of concurrent applications and eliminate traditional issues of parallel programming such as deadlocks/livelocks. One key aspect of transaction processing is the concurrency control, which is responsible to find an interleaving or overlapping schedule for running transactions accessing the same shared data and provide a result equal to some serial execution of the same transactions. In this regard, former research focused on single-threaded concurrency control schemes using a trial and error approach: The transaction repeats executing the critical section until no conflict with other transactions was detected.

This thesis proposes the application of helper threads in transactional memory to achieve a parallelised transaction processing, which is able to compute the same critical section in respect to different serialisation orders with other transactions. While traditional concurrency control has to rollback and start another attempt in the same thread, this approach aims to compute the next attempt in a helper thread in parallel to the current attempt computed by a leading thread. Due to the nature of transactions and concurrent shared memory applications in general, the introduction of helper threads comes with several new aspects to be considered on the conceptional and the implementation level.

The work presents concepts for helper thread aided concurrency controls independently executing the same transaction in parallel instead of sharing information of the leader thread, to lower the contention. For example, the detection of conflicts is delegated to a helper thread which executes the same transaction on the same state of shared data in parallel instead of validating the data read by the leader. In this manner parallelised transaction processing has been proposed which executes a transaction in parallel with different validation schemes: lazy and eager validation. Because the reaction time on conflicts and execution time differ, they drift apart and compute results for different serialisation orders.

The work also presents several requirements and sub-systems needed to run parallelised transactions: The use of so-called transactional wrappers or pure functions to speedup access to thread private data in transactional sections cannot be granted, because the private data of the application thread is now concurrently accessed by the helper thread. Also, a concept has been developed to clone the state of the transaction start and transfer the state of the helper committed transaction back to the leader. The existing sandboxing approach of Dalessandro and Scott to suppress and prevent errors in transactions resulting from lazy validation has been improved and memory management inside transactions was adapted.

While the developed approaches can compete in some cases and show good scalability especially in higher concurrency, the evaluation shows an unexpected large loss of performance, too. An investigation of hardware characteristics in respect to memory and cache read/write latency revealed several scalability issues with concurrent access to the same address. These issues have significant impact on highly concurrent applications, which seriously affects the use of transactions in general and parallelised transactions in particular.

Table of Contents

1. Introduction **1**

2. Concurrent and Parallel Programming **5**
2.1. Concurrency Control . . . 6
2.2. Runtime Environment . . . 8

3. Transactional Memory **13**
3.1. Relationship to Transactions in General . . . 13
3.2. Correctness . . . 15
3.3. Types . . . 16
3.4. Interfaces . . . 17
3.4.1. Transactional Language Constructs for C++ . . . 18
3.4.2. Transactional Memory Application Binary Interface . . . 20
3.5. Concurrency Control . . . 22
3.5.1. Locking . . . 22
3.5.2. Timer-based . . . 22
3.5.3. Multiversioning . . . 24
3.6. Reduce Loss of Progress . . . 24
3.6.1. Nesting . . . 25
3.6.2. Checkpointing . . . 26
3.7. The NOrec CC Algorithm . . . 27
3.8. Parallelising Transactions . . . 27

4. Analysis **33**
4.1. Non-functional Requirements . . . 34
4.2. Draft Approaches . . . 35
4.2.1. Helper Thread Validation . . . 38
4.2.2. Cloned Incremental Validation . . . 39
4.2.3. Cloned Synchronous . . . 41
4.2.4. Cloned Asynchronous . . . 42
4.3. Transparency and Sandboxing . . . 43
4.3.1. Transparency Violations . . . 44
4.3.2. Sandboxing . . . 48
4.3.3. Transparency of Memory Management . . . 52
4.4. Cloning . . . 53

5. Design 59
5.1. TM ABI Layer . . . 61
5.2. Sandboxing . . . 63
5.3. Cloning . . . 65
5.4. Allocators . . . 67
5.5. NOrecHt . . . 68
5.6. NOrecCIV . . . 71
5.7. NOrecCs . . . 74
5.8. NOrecCa . . . 77

6. Evaluation 81
6.1. Build and Test Environment . . . 82
6.2. Benchmarks . . . 83
6.2.1. Genome . . . 83
6.2.2. Intruder . . . 84
6.2.3. Kmeans . . . 85
6.2.4. Labyrinth . . . 86
6.2.5. SSCA2 . . . 86
6.2.6. Vacation . . . 87
6.2.7. Transactional Characteristics . . . 87
6.3. Results . . . 88

7. Investigation of Hardware Influences 99
7.1. Shared Memory Related Hardware Specifications . . . 100
7.2. The MOESI Cache Coherency Protocol . . . 101
7.3. Cache Miss Penalty . . . 104
7.3.1. Single Cache Line . . . 105
7.3.2. Atomic Instructions . . . 110
7.3.3. Main Memory Access . . . 113
7.4. Discussion . . . 115

8. Conclusion 119

Bibliography 125

Index 129

A. C++ Transactional Memory Application Example 133

B. Average Conflict Position 137

C. Raw Measurement Results 141

D. Source Code and Raw Measurement Results 143

Chapter 1

Introduction

Due to physical and economical reasons, clock speed of processors has reached its current maximum. Manufactures of central processing units have therefore started to push towards multi-processor and multi-/many-core architectures for the consumer market. To satisfy the increasing demand in processing power of modern applications this development forces the software industry to apply the techniques of concurrent and parallel programming.

Since the last three decades concurrent and parallel programming is one of the most challenging and complex tasks for software developers and designers, raising several specific issues with deadlocks, livelocks, fairness between threads and scalability of concurrent applications. The root cause for all those issues is in most cases the use of the still popular mutual exclusion pattern on shared data: In a critical section, the running thread acquires a lock for exclusive access to shared data and releases it afterwards.

Transactional memory is considered to be the most promising solution to this dilemma. Its main goal is to replace mutual exclusion with transaction processing on shared data in critical sections, seamlessly integrated with the program code. In the first place, it is a conceptual move, because transactions are just an abstract term for some transparent mechanism which controls concurrent access to shared data. This concurrency control mechanism is supposed to solve or avoid deadlocks/livelocks or issues with fairness, while permitting more concurrency inside the critical section than mutual exclusion and thereby improving scalability. Thus, transactional memory has the potential to solve all the big issues of concurrent and parallel programming just by replacing the concurrency control mechanism without any changes regarding the overall programming paradigm on shared memory. Programmers do not need a special training and the development costs will remain the same and not rise due to increased complexity.

The success of transactional memory obviously depends on the quality of the internal concurrency control mechanism. Its first and foremost responsibility is to find a legal interleaving or overlapping of concurrent operations from multiple transactions with respect to the intended outcome of the corresponding critical sections. The intended outcome of a critical section is simply the result that would have been achieved with mutual exclusion, i.e. some serial execution of running transactions. In these terms, transactions are said to be *serialisable* if their result corresponds to some serial execution. But beyond the task of protecting data consistency in critical sections a concurrency control mechanism should also outperform the mutual exclusion pattern in terms of scalability.

The topic transaction processing is almost as old as concurrent and parallel programming itself, but it was almost exclusively dedicated to database systems and distributed systems in the past. Both domains contributed a lot of research on concurrency control, which has already been evaluated in real world applications for years now. But those systems have different requirements in respect to the persistence or the location of data or resources in general, which is heavily reflected in the concurrency control mechanisms. For example, the most common concurrency control scheme (pessimistic concurrency control) applied in database systems has already been proven to be disadvantageous in transactional memory. Thus, a lot of research in transactional memory dealt with the development of specialised concurrency control mechanisms aiming to further increase concurrency and reduce control overhead in concurrent shared memory applications.

Most of the previous research in concurrency control for transactional memory considered a transaction to be single-threaded. A conflict between transactions always forces at least one thread to abort and retry to execute the same critical section. Complex concurrent application behaviour cannot be predicted, which makes this procedure inevitable in generic concurrency control algorithms which permit concurrency inside critical sections. These aborts and retries theoretically generate a significant loss of progress. The part of the section that has already been processed has to be discarded or even undone and recomputed. Many researchers dealt with progress loss on an algorithmic level, trying to reduce the amount of work discarded using backups (checkpoints) of earlier valid states of the transaction. This involves more work and it is difficult to find proper locations for the checkpoints unless they are placed by the developer. No one considered parallelisation of the concurrency control mechanism as a possible answer yet.

In the early nineties Bestavros and Braoudakis proposed a parallelised concurrency control mechanism for transactions in real-time databases, called speculative concurrency control [Bes93]. Their intention was to utilise a set of threads to perform multiple executions for one particular transaction at once and thereby cover multiple possible serialisations for the given transaction in relation to other transactions, concurrently running. In case of a conflict this

concurrency control mechanism can just discard threads which where following now invalid serialisations and keep proceeding with the remaining threads without abort or retry. At least one of the multiple serialisations originally addressed will remain in the end and provide a valid result for the transaction.

Theoretically, this approach provides the opportunity to have concurrency control without loss of progress due to aborts, which is close to an ideal solution. Of course, this advantage comes at the cost of an increased consumption of processing power in terms of utilised processors/cores but it is just yet another approach which applies increased hardware utilisation to solve a complex problem. Obviously, it is not generically applicable in its pure form because it will most certainly exhaust the available amount of processors of any computer in highly concurrent applications. But the general idea to apply a few more threads in the same transaction to compensate for algorithmic progress loss appears to be legitimate if the amount of threads will be kept at a reasonable limit.

This thesis studies the application of helper threads in concurrency control of transactional memory in general and in terms of speculative concurrency control (SCC) in particular. In the run-up to this thesis, the author gathered some experience with a straight forward implementation of the original SCC approach. Those experiences have shown severe performance issues due to an unnecessary degree of synchronisation between transactions and threads of a particular transaction. The experienced behaviour is a known side effect of the family of pessimistic concurrency control mechanisms which does not meet the requirements of transactional memory according to recently published research [DSS10]. An issue which demonstrates the difference between transaction processing in database and distributed systems on one side and transactional memory on the other side.

To gain more knowledge in the topic of parallelised transaction processing towards a speculative concurrency control in transactional memory, this thesis makes the following contributions: It presents approaches for concurrency control algorithms in parallelised transactions with low communication overhead with respect to the most recent knowledge in the research field of transactional memory. It identifies requirements and usage constraints resulting from side effects of the parallelisation of concurrency control. It identifies and presents solutions for a set of sub-systems required to operate parallelised transactions transparently for the application. It provides a comprehensive evaluation of developed prototypes using state of the art benchmarks. Finally, it also evaluates the capabilities of modern 64 bit end-user multi-core architectures to support concurrent applications in general and parallelised transaction processing in particular.

The thesis has a traditional outline, roughly separating state of the art, analysis of the problem domain, design of prototypes and evaluation of the achievements, bookended by this introduction and a conclusion in the end. The next two chapters will start with an introduction in the most important background knowledge in regards to concurrent and

parallel programming and transactional memory. Readers familiar with concurrent and parallel programming in general and the basic terms of concurrency control, function calling conventions and the execution stack can skip directly to the third chapter. This chapter starts with a brief explanation of the common understanding of transactions before going deeper into the transactional memory topic. The chapter is especially important for understanding the internal functionality of transactions and learn technical terms used in subsequent chapters. It also explains how transactional memory is integrated in concurrent applications and their runtime environment, different kinds of transactional memory and internal mechanisms as well as their advantages and disadvantages. This chapter also introduces the concurrency control, which serves as the basis for the experiments and the general idea of parallelised transactions in more detail. Those readers familiar with transactional memory should at least read the last two sections of the third chapter. The remaining body of the thesis contains the contribution of the author besides some minor sections introducing required technical background knowledge that contextually did not fit in the state of the art chapters. Those sections contain featured notes on the sources. The analysis chapter motivates the outline and design decisions for the approaches for prototypes and their required subsystems. The design chapter puts the formerly roughly introduced approaches of the analysis into a system specification and describes the general interaction scheme of its components. The evaluation is split into two chapters. The first chapter (Chapter 6) presents and discusses the results provided by the comparison of the prototypes in measurement runs with different benchmarks especially developed for applications using transactional memory. The second chapter deals with the influences of the hardware used for the evaluation. This investigation of the hardware influences was originally initiated to get a better understanding of the evaluation results, but it revealed some interesting issues with modern multi-core architectures, which has a significant impact on transactional memory in general.

Chapter 2

Concurrent and Parallel Programming

This chapter introduces briefly important terms of concurrent and parallel programming on shared memory. Starting at the concept level it also discusses required hardware related background on a reasonable level of detail. Some of the provided information is common knowledge about operating systems (see e.g. [Tan92]) and computer architecture (see e.g. [Tan84]) to help you recall.

The term concurrent software refers to applications which run multiple threads of execution concurrently to deal with multiple jobs at once. In this context a thread of execution refers to a process or thread. In today's operating systems processes and threads have the same scheduling properties covered by the same abstract concept called *task*. A task is an individual scheduling unit associated with a virtual address space and a stack to execute function calls etc. A running task is assigned to one processing unit (processor or core of a processor). If the number of running tasks of an application is larger than the number of available processing units, the operating system can switch between tasks to accomplish quasi-simultaneous *concurrent* execution. In contrast, concurrent applications designed to run multiple tasks at the same time and each exclusively on a processing unit are called *parallel*.

Collaborating tasks require mechanisms to exchange data with each other (c.f. *inter-process communication* (*IPC*)). Those mechanisms can be categorised by the general parallel programming paradigms:

- *Message passing*: This programming paradigm comprises all concurrent applications which use messages to exchange information between tasks.

- *Shared memory*: This programming paradigm comprises all remaining concurrent applications where all tasks work on the same shared data to share information.

The main difference of both paradigms is neither the structure of the data nor the location of the participants but the way to access and maintain shared data. While shared memory potentially allows tasks to access shared data concurrently, message passing follows a strict processing model: a task receives one or more messages, processes it and creates one or more new messages to be sent. Messages are logically copied from the sender to the receiver. Thus, a message is at no time visible to more than one task.

2.1. Concurrency Control

In the shared memory model the developer is responsible for appling or implementing some kind of *concurrency control* (*CC*) to protect consistency of shared data during concurrent access. In that, consistency of shared data is a semantic property defined by the application, respectively the developer who designed the application. For example, in one application the concurrent increment of a variable might be consistent only if it is executed entirely exclusively (i.e. exactly once). In another application it might be legal if not all concurrently processed increments are consistently executed (e.g. to achieve at most once semantics). The particular code sections containing the instructions that transform shared data from one consistent state to the next consistent state are denoted as *critical sections* [Lam79]. Concurrency control is applied to exactly these critical sections to protect the consistency of the data transformation.

Most basically, the concurrency control has to guarantee that the result of simultaneous (temporal overlapping) concurrent executions of critical sections equals the result of a sequential execution of the same critical sections. A simple model used to prove consistency in most concurrency control mechanisms is related to the inputs (reads) and outputs (writes) of the critical section: The result of an execution is consistent as long as the data read is of one particular consistent state of the shared data (i.e. not partially modified by a concurrent task) and the data written in the critical section is not partially overwritten by a concurrent task until the critical section is left.

A concurrency control mechanism can simply consist of a mutex for the critical sections, which is manually inserted by the developer or even more complex generic mechanisms such as those hidden inside transactions. Although this section focuses on concurrency control in application code it refers to exactly the same general concept applied in operating systems, distributed systems and database systems.

The methods of concurrency control can be generally subdivided into two categories:

- *Pessimistic Concurrency Control* (*PCC*): Pessimistic concurrency control means to prevent any kind of inconsistency in critical sections beforehand by establishing exclusive access to the required shared data, e.g. by using locks to block concurrent tasks.
- *Optimistic Concurrency Control* (*OCC*): Optimistic concurrency control involves speculative work on probably concurrently modified shared data as long as inconsistencies will be detected and resolved when the critical section is finished.

PCC is generally established through mutually exclusive access by acquiring a lock before accessing shared data and releasing it afterwards. The most conservative implementation is the use of the same single global lock for all critical sections. That way all concurrent executions of critical sections are mutually exclusive and even critical sections that actually access disjoint data, will unfortunately waste time while waiting for the lock.

The probability to block non-conflicting critical sections can be reduced by using more than one lock. For example, the developer manually identifies conflicting critical sections and associates them with individual locks. A more generic approach is to assign locks to portions of the shared data (e.g. data objects). However, both methods introduce the risk of deadlocks. Deadlock prevention is complex and still requires the developer to take care of it manually. In contrast, generic concurrency control mechanisms detect deadlocks dynamically using a deadlock detection algorithm and resolve them by a *rollback* to a non-conflicting state of the critical section, which is in most cases its beginning. This includes restoration of the previous state of the modified shared data and release of the locks.

Locking protocols are usually derived from or similar to *two phase locking* (*2PL*) [Lau09] where each task dynamically acquires all locks on demand during computation of the critical section and releases them in the opposite order at the end. PCC usually writes directly to the shared data object (*direct update*) instead of buffering updates until a final write back is performed (*deferred update*).

All PCCs have one major conceptual issue resulting in loss of progress: Letting tasks wait while a lock is exclusively owned by another task is fine as long as the owner does not run into a deadlock, which will be resolved by a rollback (which can cause livelocks in turn). Then, the work discarded by the conflicting task and the time of tasks waiting for it is just wasted.

OCC introduces speculative work of a task in a critical section. A critical section is speculatively executed under the optimistic assumption that there will be no conflicting modifications by concurrent tasks. This assumption has to be checked for all accessed shared data before the critical section is left. A violation requires the task to rollback to the last valid point inside the critical section, which might be the beginning.

OCCs usually use deferred updates because direct updates cause *cascading rollbacks*: If a task reads data which was written by a task that has not yet finished its critical section, the

follower task has to consider the case that the leading task might rollback. Thus, work of the follower is of speculative manner until the leader successfully finishes its critical section and thereby acknowledges the persistence of its updates. But, if the leader runs into a conflict and has to rollback, the follower and its followers have to rollback too.

2.2. Runtime Environment

Parallel and concurrent programming heavily depends on the support of the runtime environment, meaning the operating system, runtime libraries and the hardware. This section provides a closer look at mechanisms and tools of modern 64 bit runtime environments supporting concurrent and parallel programming.

Concurrent programming requires the operating system to support multitasking, which allows concurrent or parallel execution of tasks in general which means processes or threads in particular. Managing the limited resources of the hardware the operating system needs to map n running tasks to m cores in terms of scheduling. Thus, a task will occasionally be suspended (removed from a core) by the operating system in favour of another task, which will get this core for processing. During this procedure, which is called *context switch*, the *machine state* (working set/context) of the previous task is saved in a backup and the machine state of the new task is restored to the given core. The machine state mainly consists of the registers of the core. The registers of the core typically contain:

- *Instruction pointer* or *program counter*: A reference to the next instruction in the application code to be executed by the task.
- *Stack* (*SP*) and *base pointer* (*BP*): References to the top and the bottom of the stack frame on the execution stack currently used by the task to execute a particular function.
- Instruction parameters: Machine code instructions require parameters to be passed through registers.
- Temporary variables: Most registers have no specific purpose and are free to be used by the task to temporary store data e.g. for local variables in functions.

The execution stack is especially needed to store the local data of recursive function calls in particular and has become a standard mechanism to implement function calls on most platforms. An execution stack stacks so-called *stack frames* for each function not yet finished by the task. The stack frame contains data such as the return address, a reference to the previous stack frame, the function parameters and local variables.

For the interaction with libraries available in the system, the layout of the stack and the procedure to call functions is specified in a so-called *application binary interface* (*ABI*) of the operating system for each supported architecture type. For example, the reference for 64bit Linux operating systems is the Unix System V ABI for AMD64 architectures [MHJM13]. The ABI also contains other details for binary compatibility with the system such as layout of executables, interrupt handling etc.

Each task is assigned to a virtual memory area available to it. While each process is assigned to a particular virtual memory area, threads of the same process share the same virtual memory area. In detail, a process is started with one task, which establishes a task group consisting of all tasks of the same process (i.e. threads). Virtual memory is maintained in a so called *page table*, which contains the information to translate virtual addresses into physical memory addresses. All threads of a process refer to exactly one shared page table and each process has its own page table.

The structure of the page table is predefined by the *memory management unit* (*MMU*) of the hardware. On x86 systems it is a tree structure, which contains references to so-called page frames in physical memory at its leaves. Each page frame has a fixed size of 64KB. The MMU decodes virtual addresses on the fly by traversing the page table. To improve address translation effort the MMU uses a cache called *translation lookaside buffer* (*TLB*) for page frame references. That way addresses already cached require no additional translation effort. The TLB is reset on each context switch. Some operating systems such as Linux do not reset the TLB on context switches between threads of the same process because they use the same page table.

Another mechanism typically available for threads of a process in modern runtime environments is *thread-specific data* (or *thread-local data*). This refers to a concept to associate a process wide reference (e.g. identifier) for each thread with a different, thread-specific memory location. This allows transparent management of thread-specific data such as locks currently owned by the thread. Memory content is transferred between the core and the memory banks via one or more front side busses. Multiple cores of one processor or even multiple processors have to share a front side bus.

Memory access is typically slow and ranges somewhere between 80 and 120 instruction cycles. To lower memory access latency the hardware comprises caches, which are organised in a hierarchical manner. Usually, each core is aided by a so-called level 1 cache providing space for a few kilobyte of data. A slightly larger level 2 cache (around one or two megabytes) on top of it and in some cases a 3rd level cache providing even more space. Cache space is further subdivided into *cache lines* (usually 64 bytes on x86-64), which defines the lowest granularity memory is maintained in the system. When transferring data from memory to

the cache or vice versa, it is always packaged in cache lines and each cache line is virtually assigned to a fixed address. Thus, the whole memory is logically sliced into cache lines.

Concurrent read and write access of multiple cores to memory at the same location (same cache line) results in inconsistencies. Thus, the hardware applies some *cache-coherency protocol* through cache and memory controllers to maintain a guaranteed *consistency model* (or *memory-ordering model*), the software can rely on. Its main purpose is to detect conflicting concurrent modifications and accordingly update cache lines to the least recent modification. For example, a typical method for the detection of concurrent modifications is sniffing, which was originally achieved by simply listening on the front-side bus to the traffic between other cores and memory. Modern multi-processor architectures with their direct linking between the processors (i.e. Intel QuickPath Interconnect or AMD HyperTransport) require some different methods based on subscriptions and notifications.

The consistency model maintained by the hardware describes which memory content has to be expected after concurrent execution of some fetch and store instructions on multiple cores. The definition of this consistency model is especially required to allow cores and memory controllers to reorder memory accesses for performance increase in terms of instruction pipelining. For example, if multiple accesses to different cache lines occur in some control flow, the core can try to reorder instructions to serve all access to a cache line that has been fetched previously while it fetches another cache line required from memory in parallel.

Due to reordering, concurrent access can result in a memory image presented to a task which might not reflect the given order of instructions executed in parallel. Thus, the consistency model of modern x86 architectures does not support the so-called *sequential consistency* [Lam79], where each instruction is guaranteed to be executed in the order given by the program code. To allow the programmer to force a certain order required by the application, the x86 64bit hardware provides so-called memory fences (also called memory barriers):

- *Load fence* (`LFENCE`): A load fence guarantees that all reads and writes required for the instructions after the fence instruction, will not be served before it is passed.
- *Store fence* (`SFENCE`): A store fence guarantees that all writes to be issued by instructions following the fence instruction will be served after the fence has passed only.
- *Memory fence* (`MFENCE`): This is simply a combination of both, a read and a write fence.

The compiler also reorders load and store instructions, which is why a developer, programming low-level parallelism, requires so-called *compiler fences* (or *compiler barriers*) too. A compiler fence prevents any kind of reordering done by the compiler to cross the fence.

This is still not enough to provide consistency in concurrent programs on application level. To realise mechanisms such as locks, semaphores and higher level synchronisation utilities, the hardware offers *atomic* instructions which execute consistently even if the instruction involves access to multiple cache lines. Most atomic instructions are simply a usual assembler instruction prepended with the prefix `LOCK`, such as `LOCK ADD` for an atomic increment. Other instructions such as the so-called *compare and set* (*CAS*) are implicitly executed with the lock mechanism. The `LOCK` prefix originally stood for *bus lock*, which refers to a special signal issued at the front-side bus to suppress any attempt of memory transfer of other processors. Today it also refers to so-called *cache locks*, which lock one or more cache lines using the cache coherency protocol. Unlike bus locks cache locks allow concurrent locking of disjoint cache lines by different cores.

Chapter 3

Transactional Memory

Transactional memory (*TM*) is the concept of applying transactions to critical sections in the program code of an application to almost seamlessly replace mutual exclusion with a more scalable concurrency control mechanism. This section starts with an introduction into the term transaction to finally explain the different kinds of transactional memory known today, their usage, theory and internal functionality. It will especially introduce the technologies used for the implementation of helper-thread aided CCs developed in this thesis such as the NOrec CC algorithm and the interfaces between concurrent application and TM system. However, the details on transactional memory in this section are focused on the topic of the thesis. A more comprehensive survey can be found in [HLR10].

3.1. Relationship to Transactions in General

In 1977, Jim Gray published a survey [Gra78] which concludes the state of the art of transaction processing in its very early stage. According to this, the term transaction originated from actual transactions in large institutes such as in the finance, insurance or manufacturing sectors. The problem was to keep data located at distributed locations consistent while transactions where concurrently executed and data was stored in plain files in the file system managed by the operating system on distributed nodes. Hence, transaction processing was distributed and they built special operating systems or extensions to operating systems to manage their execution. Later the different disciplines split up into transaction processing in distributed systems and database systems.

Because the original understanding of a transaction [Gra81] was more related to distributed systems, it was explained as an agreement between the distributed parties, which is ***c****onsistent*

with legal protocols, ***atomic***, that it either happens completely or not at all and ***durable*** once it was committed. Later these statements were extended by a property which claims transactions to be ***isolated*** from other live transactions and coined as the ACID criteria [HR83] to prove correctness of transaction processing.

In terms of computing a transaction specifies a procedure to be performed on shared data (or resources) concurrently to other transactions, just as a critical section with some generic concurrency control mechanism. A transaction consists of a sequence of actions, which specify its internal control flow and outcome, and includes actions on shared data. An action on shared data is either to retrieve (read) or modify (write) the shared data. Shared data consists of a set of arbitrary, large data objects (e.g. variables in memory or records in a file). The set of all read actions and their retrieved data is called the *read-set* of the transaction. Likewise the *write-set* is the set of all write actions and associated data to be written.

Each transaction has its local data, which reflects its current state and is (usually) invisible to other transactions. Transaction local data consists of its read and write-set and the initial data obtained during the transaction start. Initial data is provided by the control flow, which initiated the transaction (i.e. process or thread local data). It consists for example of parameters for a query (e.g. databases) or the data currently on the execution stack of a task (e.g. distributed transaction processing).

As part of its output a transaction may provide a result to the caller the same way it retrieved its initial local data. Data exchange of initial data and result can be modelled as data transfers between caller and transaction. Thus, a transaction can be executed by a different task as the caller and even by multiple tasks (e.g. in distributed transaction processing).

The characteristics of a transactions' run-time behaviour can be rendered by the following events:

- A *start*, which occurs exactly once for each executed transaction,
- an *abort*, which marks the discard of the transaction or its current attempt to execute
- a *rollback*, which represents its abort and the return to the start
- a *commit*, which stands for the successful completion of the transaction
- and the *end*, which also occurs exactly once for every transaction.

Generally it is not necessary that a transaction will commit once it was started, because in some systems a transaction might even become obsolete after an abort and does not need to be retried again. However, in transactional memory we usually expect all started transactions to be executed at some point in time and thus to commit.

Even a single global lock is a legal concurrency control for transactions too, but the rollback and retry behaviour is most common.

A rollback usually is the result of a conflict, which may be either a deadlock detected by a deadlock detection algorithm in PCC or a data inconsistency detected by a *validation* needed in OCC.

Transactional memory shares all those properties with transaction processing in distributed systems or database systems except transactional memory is typically neither distributed nor does it operate on persistent data. Transactional memory typically operates in a shared memory system and the shared data is gone once the application is finished. But still it is not impossible to integrate transactional memory with distributed transactions or database transactions.

The fact that transactional memory operates embedded in the program code involves issues with (external) systems that do not support transactions. A well known example is I/O on a terminal. Once the output has been written to a terminal it cannot be undone by a rollback. There are a lot of system functions that do not support transactions. To allow their usage an transactional memory system has to dynamically switch to some kind of mutual exclusion on demand: The affected transaction gets declared as *irrevocable* and cannot be rolled back anymore. Thus, any another concurrent transaction that might get in conflict will be aborted or blocked until the irrevocable transaction has finished.

3.2. Correctness

Section 3.1 introduced the ACID criteria as the common comprehension of the notion of transactions. Beyond that, *serialisability* [Pap79] was established to define a formal model of synchronisation between transactions which allows to prove the correctness of its CC. It very simply states that concurrently executing transactions are serialisable if their effect complies to some legal sequential execution of the transactions. Legal in this case means that the outcome has to be the same as if the transactions would have been executed by a single task and thus it prohibits reorderings of transactions which violate the program order too. However, most literature in the field of transactional memory refers to *linearisability* as the correctness criterion for transactions in transactional memory.

Linearisability has been originally introduced by Herlihy and Wing as a correctness condition for so-called *concurrent objects* [HW90]: objects, in terms of object-oriented programming which offer thread-safe methods to access their state. Method calls on concurrent objects are linearisable if they satisfy two principles (cf. [HS12]):

1. "Method calls should appear in a one-at-a-time, sequential order."

2. "Each method call should appear to take effect instantaneously at some moment between its invocation and response."

The first principle assures that a method call appears atomic and in program order. The second principle mainly demands method calls to take effect before they return considering architectures which may reorder the effects.

Considering executed method calls to be transactions, linearisability seems to be the same as serialisability. As Herlihy and Shavit state in their book: "Transactions must be serialisable" [..], "serialisability is a kind of coarse-grained version of linearisability" and transactions are "blocks of code that may include calls to multiple objects". Serialisability "ensures that a transaction appears to take effect between the invocation of its first call and the response to its last call" (cf. [HS12], Page 421). Thus, linearisability addresses the internal behaviour of a transaction and not just the outcome of complete transactions and it declares that modifications on shared data have to apply to principles of linearisability as the transaction as a whole to serialisability.

There is a lot of discussion about the proper correctness criterion (see e.g. [GK08] for a survey) but the common agreement is, that serialisability is enough to describe the external behaviour of a transaction.

3.3. Types

Transactional memory was first published by Herlihy and Moss [HM93] in 1993 as an extension to the cache coherency protocol in hardware, so-called *hardware transactional memory* (*HTM*). Once a transaction on a core is started, every load to the associated cache of the core is considered as part of the read-set of the transaction and every store to the cache is part of the write-set. Writes are stored in the cache (i.e. not written to main memory or published to other caches) until the transaction is finished. Whenever the cache coherency protocol detects a foreign modification of a cache line which is part of the read-set, the transaction is considered to be inconsistent and aborted. A transaction without conflicts is allowed to commit and publish its write-set as part of the new state of the shared data.

A major problem of hardware transactional memory is the limited capacity of the cache. Thus, if the read- and write-set of a transaction exceeds the capacity of the cache, HTM requires a fallback solution on software level such as a single global lock.

HTM extensions are a recognised topic for most CPU vendors today: Sun (now Oracle) implemented the prototype of the Rock processor [DLMN09] with HTM, AMD proposed the Advanced Synchronization Facility (ASF) [CYD+10], IBM already integrated HTM support in a supercomputer [HT11, WGW+12] built for the US National Nuclear Security

Administration and plans an HTM extension for the POWER8 processor and Intel recently published the Transaction Synchronization Extensions (TSX, [YHLR13]) for its Haswell architecture.

To experiment with different concurrency control algorithms and due to the lack of available HTM implementations, researchers began to develop prototypes of transactional memory in software such as libraries, so-called *software transactional memory* (*STM*) (see e.g. [ST97, HLMS03]). While STM generates more control overhead, HTM has limited capacity. Thus, other researchers started to combine both into *hybrid transactional memory* (*HyTM*) with STM as the fallback for HTM (see e.g. [KCH+06]).

3.4. Interfaces

The boundaries of a transaction in TM have to be marked by the developer the same way it has to be done for critical sections with mutual exclusion, which are wrapped by a pair of `acquire` and `release` operations on a mutex.

To mark the begin and end of a transaction HTM provides additional machine instructions. In STM three different approaches exist:

- `Procedural or object-oriented API`: The first APIs published with STMs were simple APIs to the STM subsystem for example consisting of functions such as `txbegin()`, `txend()` to mark the boarders of the transaction and `txread()` and `txwrite()` to instrument access to shared data inside the transaction.

- `Macros`: Especially in benchmarks (e.g. STAMP[MCKO08]), which should be adaptable to different STMs, the API consists of macros. Macros are a feature of some programming languages (e.g. C/C++) which serve as placeholder for actual source code. Macros are substituted with source code in the pre-compiler phase during the build of the application. In this form there are actually two interfaces to the STM: One for application-level instrumentation, consisting of the macros, and another to the STM implementation, which is utilised by the macros.

- `Language extensions`: The most recent step was to add STM as a feature to the programming language. The language was extended by new keywords and statements to define transactions and control their behaviour. The compiler has to support the extension and generate appropriate instrumentation code to adapt utilise the STM interface, such as the Intel TM ABI [Int08]. Thus, there is again a concept which involves a front-end API to the application and a back-end API to the STM implementation.

All classic APIs for STM had common features, which were one by one integrated in the language extension approaches. Currently, there exists one approach for a language extension to C++ [ATSG12], which will presumably become the standard. It is already supported by major C++ compilers such as the GNU Compiler Collection (GCC) and Intel C Compiler (ICC). Thus, we will focus on this language extension here to explain the most important features. The section thereafter will explain the Intel TM ABI supported by the compiler.

3.4.1. Transactional Language Constructs for C++

This section focuses on the meaning of the language constructs. A complete application example can be found in Appendix A. Transactional code is always covered in a code block. This is either a section in the code covered in a block as depicted in Listing 3.1 or existing blocks such as a function body (see Listing 3.2) or methods of classes (see Listing 3.3).

```
// non-transactional code ..

__transaction_atomic {
   // example of transactional code:
   a = b + c;
   // call transaction safe function f():
   f();
}

// non-transactional code ..
```

Listing 3.1: A transactional block

The only way to define the beginning and the end of a transaction is to write a transactional block (Listing 3.1) where the open bracket (`{`) and the close bracket (`}`) mark both ends. The compiler is responsible for instrumenting the transactional code properly, which means to insert the required method calls of the TM ABI to call the STM in the code.

Transactional blocks can be of two types:

- `__transaction_atomic` blocks start/end a transaction or open/close a child transaction if a running parent transaction exists. Code inside atomic transactions gets instrumented to forward memory access instructions (reads and writes) to the STM. Atomic transactions cannot contain transaction unsafe code, mainly speaking of transaction unsafe functions or functions not declared to be transaction safe as well as transaction relaxed blocks (see below).
- `__transaction_relaxed` blocks also declare start and end of transactions but those transactions will not execute concurrently to other transactions. Thus, transaction

relaxed blocks will be executed mutually exclusive to other transactions on demand and allow even unsafe code to be executed inside.

A transaction can be manually aborted and rolled back by the `__transaction_cancel` statement. This allows for example to implement busy waiting on a certain condition variable.

Transactional code can generally exist in two versions with and without instrumentation. For example functions with the attribute `transaction_safe` will be compiled in one version with instrumentation and another without instrumentation. It depends on the type of transactional code block whether the compiler will insert a call to the transactional version or the non-transactional version of a function in a transaction.

```
// declaration
[[transaction_safe]] void f();

// definition
void f() {
   // transactional code
}
```

Listing 3.2: A transactional function

Transactional functions and methods of transactional classes are generally treated as the same, except methods of objects have an additional parameter which refers to the object instance. There are the following attributes to control the transactional behaviour of a function:

- `transaction_safe` functions will exist in an instrumented and an uninstrumented version and can be called from running transactions.
- `transaction_callable` function will exist in instrumented and uninstrumented versions and can be called from anywhere in the code.
- `transaction_unsafe` functions will exist as uninstrumented functions only and cannot be called from transactions per definition.
- *`transaction_pure`* functions are a proprietary extension by the GCC compiler. Those functions are declared to be uninstrumented but safe to be called from transactions. This is for example useful for integrating functions which do not modify shared data in a transaction such as a string comparison.

Those attributes can be assigned to an entire class (see Listing 3.3) to declare a default for all contained methods. However, the default attribute can be overridden by a different attribute to the method inside the class.

```
class A [[transaction_safe]] {
  // method automatically declared as transaction safe by the class attribute
  void m();
  // another method with a specific transaction attribute
  [[transaction_unsafe]] void n();
}
```

Listing 3.3: A class with transactional methods

3.4.2. Transactional Memory Application Binary Interface

The Intel transactional memory application binary interface (TM ABI) [Int08] defines a standard back-end API to an STM implementation to be utilised by a compiler during compilation of transactional sections. The compiler is responsible for generating the proper instrumentation in transactional code sections marked by the developer. This section contains a brief description of its coverage.

The ABI was primarily designed for STMs written in C/C++ and languages with compatible function calling convention such as Fortran and for applications running on Linux or Windows operating systems. The ABI mainly consists of function prototypes. Because Windows provides just slow access to thread-specific storage most functions for Windows expect a reference to the transaction context as the first parameter.

The entry of a transactional section is signalled by `_ITM_beginTransaction()`. Via its parameters the STM gets the machine state of the current task at the entry point of the transaction (checkpoint). It is used later to reset the transaction to the begin during a rollback. The STM has to detect if a parent transaction already exists and the incoming call signals the start of a child transaction. Another parameter of the function provides information about the code inside the transaction, for example whether its reads and writes are instrumented or not (e.g. for relaxed transactions). The STM has to configure the transaction properly. The return value indicates whether the application has to run instrumented code or not and if local variables have to be restored by the application after a rollback. Because the transaction will return to the `_ITM_beginTransaction()` function after a rollback too, the return value is also used to inform the application about the recent abort. The compiler is responsible for generating all the different control paths to handle the return values.

The end of a transaction is signalled by a call to `_ITM_commitTransaction()` or `_ITM_tryCommitTransaction()`. This function may or may not return to the calling function context depending on whether the transaction commits or rolls back.

For memory reads and writes to locations other than the local variables of the function the compiler has to insert appropriate calls to read and write functions of the ABI based on the

data type and of the access history inside the transaction. The ABI supports all primitive data types of the C language such as pointers, bytes, floating point numbers, signed and unsigned integer up to 64 bit floating point, 256 bit integer and complex 32 bit. It supports five different access types:

- Read: A variable is read. Every read, even the special read functions below, expects a pointer to the location to be read whose value will be returned.
- Write: A variable is updated. Every write function expects the location and the value to be written.
- Read for write: A variable is read to be updated later.
- Read after write: A read of a variable which was recently updated by the transaction itself. In deferred update STMs those reads have to be served from the write-set of the transaction.
- Write after write: A variable is updated at least twice in the same transaction. The STM can use this information for example to look-up and overwrite the log entry in the write-set and thereby reduce the size of the write-set.

For each combination of access type and data type one function exists.

Access to larger portions of memory is supported by special versions of the C runtime library functions `memcpy` and `memmove`, which also consider different access histories such as those explained above.

To save the state of local variables the compiler is free to utilise logging functions of the ABI, which forward the location and value of the local variable at the start of the transaction to the STM. In case of a rollback, the STM is responsible for writing the logged values back and thereby reset the state of the local variables.

To support application forced abort such as through `transaction_cancel` (see Section 3.4.1) the ABI provides an `_ITM_abortTransaction` function, which resets the transaction to the begin without rollback and an `_ITM_rollbackTransaction` function, which triggers a regular rollback.

Irrevocable transactions are supported by the function `_ITM_changeTransactionMode`, which allows switching the execution mode of a transaction to irrevocable (serial).

The remaining functions are for error reporting, initialisation and similar internal purposes of the STM implementation and to support exception handling. The latter functions differ a lot in the current implementations of GNU and Intel and are expected to change before the specification will become part of the standard.

3.5. Concurrency Control

Published CC algorithms for STM are very diverse but their properties can be categorised by three basic concepts: Either they are PCC approaches based on *locking*, or OCC approaches, which are either *timer-based* or some kind of *multiversioning*. A *timer-based* OCC essentially associates consistent states of shared data with a timestamp (version number) retrieved from a logical timer (counter). In *multiversioning* OCC each data object of the global data may exist in multiple versions at the same time, keeping older states, too. The following subsections will discuss the fundamental building blocks of these concepts.

3.5.1. Locking

The most famous example of a locking CC and also the counter example for good scalability is a *single global lock* (or *common global lock* (*CGL*)) approach, which establishes mutual exclusion between transactions. As formerly explained in Section 2.1, CGL does not allow any kind of concurrency in critical sections and thus it is nowhere suggested as STM implementation.

The first STM implementation that was published under the term software transactional memory [ST97] by Shavit and Touitou used locks. To prevent deadlocks, its use is restricted to *static transactions* where the locking order is predefined and will not change dynamically. Locks are acquired in a predefined (for example ascending) order at the start of the transaction and released after commit. Locks (ownership) were stored separately in so-called *ownership records* (*orec*). This term was later used for data structures which store any transaction related metadata associated with a data object in general.

In the first STM approach locks were occupied for the whole execution time of a transaction. While non-conflicting transactions can execute concurrently others have to wait for the commit of the current lock owner. As mentioned in Section 2.1 the time other transactions have to wait for a lock, in cases where the owner fails and rolls back, adds to the loss of progress in the STM. We call this *transaction queuing*. Researchers tried to reduce the waiting time by postponing the lock acquirement to the commit [DS06, DSS06] or allowing transactions to steal locks (*lock stealing*, e.g. [WRFF10, LLM+09]) owned by other transactions.

3.5.2. Timer-based

Lock stealing and late locking introduces OCC behaviour because data read earlier might have been modified by other transactions in the meantime when the transaction tries to commit. Thus, before a transaction can commit it has to *validate* that every value of the read-set belongs to the same valid state of shared data and was not modified. If a transaction is detected to be invalid it aborts.

Simply comparing the actual value with the value first read is not always enough because other transactions might have changed the value from one state A to another state B and back to state A, which is called the *ABA-problem* [MS98]. A legal method is to apply version numbers to data objects, which are incremented on each committed update. Thus, a version number sampled during the read of the object can be compared at commit time for validation.

When validating at commit time (so-called *lazy validation*) a transaction risks to work on inconsistent data in the meantime. This can lead to several serious issues such as to access memory locations that do not exist (e.g. deleted objects) or to run in an endless loop due to an inconsistent termination condition. If lazy validation is applied the STM has either to prevent those errors to occur or hide them from the application by handling them, which is called *sandboxing*. In other words transactions have to transparently deal with inconsistent states and resulting internal errors. Some of these errors can cause the transaction to run in endless loops. To prevent this case, a viable approach is to validate at certain critical instructions such as jumps or run a timer which triggers interrupts to perform a validation apart from application code (*timer-triggered validation*).

In contrast to lazy validation *eager validation* implementations validate every read and thereby make sure that the whole stored read-set is of the same global state and by definition valid. Thus, if the transaction works on this read-set it should not run into issues with internal errors. Such STMs comply with the so-called *opacity* [GK08] criterion, which entirely prohibits inconsistent states of one transaction to be visible to other and even non-committed transactions.

Eager incremental validation forces the transaction to revalidate the whole read-set on every read (so-called *incremental validation*) because otherwise it cannot be sure, whether parts of the read-set might have been modified and thereby do not belong to the same global state. Eager incremental validation unfortunately has a runtime complexity of $\Omega(n^2)$ where n represents the number of reads. Several proposals have been published to reduce the complexity in incremental validation. The most common and most basic is a *global version clock* [DS06] (also called timer), which prevents transactions from iterating through their whole read-set while there was no commit and thus no conflict possible. The global version clock is incremented on each commit. This allows a transaction to sample the clock at its start and check it before it runs a full validation.

As implied above the global version clock can reduce effort of incremental validation if the overall contention probability is low only. In its worst case the runtime complexity is still $\Omega(n^2)$. But, if we can not reduce the complexity we might reduce the number of reads to be validated. One way to do this is to increase the granularity of memory locations assigned to versions, for example associating objects in terms of object oriented programming with a version instead of every byte in memory. Those *object-based* STM implementations

potentially require fewer versions to be validated than *word-based* or even *byte-based* STMs but the amount of memory to be copied to the read-set is higher.

One issue of all CC mechanisms discussed so far is fairness between transactions. Considering a very long running transaction which conflicts with short transactions, the long running transaction will repeatedly and probably forever have to abort in favour of the shorter transactions. This is more likely in OCC than in PCC. In OCC especially with deferred update the shorter transactions will have a high probability to commit and force the longer transaction to roll back. With direct updating STMs there is also an issue with livelocks because two transactions can cause each other to abort repeatedly. In PCC there are cases where a deadlock will be generated over and over again, for example if all transactions work on an array and while the long transaction iterates in forward direction the short transactions might iterate in backward direction and thus will always cause a deadlock. Because the shorter transactions are earlier ready to commit they will always win over the long transaction. Thus, there is a need for a so-called *contention management* [SS05], which controls the order in which transactions should commit to prevent fairness and liveness issues.

3.5.3. Multiversioning

The third known method for OCC applied in TM is multiversion concurrency control (MVCC) or multiversioning for short (see e.g. [FC11]). Multiversioning aims to reduce read conflicts by providing earlier values of a shared data object as long as they are needed. Considering for example a simple read-only transaction T_r, which reads a variable concurrently to a transaction T_w, which updates it before T_r can commit. In the OCC mechanisms explained so far T_r will be forced to abort. But actually T_r could have been considered to occur earlier than T_w, the same order that would have been forced by locking, and thus the abort was not necessary. To circumvent this case a multiversion CC keeps the version of the data object which was present at the start of the read-only transaction and the updating transaction just adds a new version to it. Thus, the read-only transaction can still commit while the update was already performed.

3.6. Reduce Loss of Progress

We already mentioned loss of progress in TM caused by the CC algorithm in Section 3.5.1. Roughly explained we define loss of progress as the difference in progress between an existing CC and a theoretical ideal CC, which finds the optimal interleaving/overlapping of transactions to provide the shortest response time. Reasons for loss of progress are twofold:

- Overhead of the CC such as caused by copying data to a transaction's working-set, resetting the state of a transaction during rollbacks or additional abstraction layers such as the application programming interface.
- CC algorithm related loss of progress through waiting while an ideal CC would not have to wait or abort to positions in the transaction that are far away from the actual operation which caused the conflict.

The first category heavily depends on the implementation of the CC and the capabilities of the hardware such as the memory infrastructure. In most cases it is hard to predict this overhead until the implementation is evaluated. Ownership records have been identified [SMSS06] as one major source of such overhead in TM in cases where orecs are shared between transactions such as in almost all locking and multiversioning CCs. The fact that the transaction has to access the actual shared data and the shared orec as well, simply doubles the amount of shared memory accesses and thereby increases the amount of cache misses and traffic on front-side bus or inter-processor links. Thus, a simple CGL CC has in many cases a better performance than those CCs which require shared orecs. An STM implementation which explicitly avoids shared orecs is NOrec [DSS10], which can be considered as one of the fastest STM implementations today.

The second category is more related to the underlying concept of the CC. For example coarse-grained locking causes transactions to wait even if the shared data they need may not be in conflict. In contrast OCC does not wait and reduces the loss of progress by speculatively processing ahead. This way an OCC potentially avoids the loss of progress observed in coarse-grained locking. But it depends on when the transactions get aware of a conflict if they waste processing time during speculative execution or not. For example eager validation will detect conflicts earlier than lazy validation and more likely avoid this type of loss of progress.

Transactions generally resolve conflicts by rollbacks, which cause the other type of loss of progress in this category. Reason of a conflict whether it appears as deadlock or data inconsistency is always one particular access to shared memory. The ideal position to rollback a transaction and thereby resolve the conflict would be this particular data access and not the beginning of the transaction. Thus, all the work the transaction has to redo from the beginning up to this conflicting data access is another loss of progress. The following subsections explain the major two concepts applied to CC in TM to avoid this kind of loss of progress.

3.6.1. Nesting

In the first place *nested* transactions are a necessary structural feature when a transaction has to run code that is covered in its own transaction (e.g. a subroutine). Thus, a so-called *parent*

transaction may have several embedded *child transactions*, which have to be considered by the CC. The simple concept of *linear* or *flat nesting* just handles the nested transactions as part of the parent transaction and a conflict will rollback the parent transaction to its beginning.

In 1981 J. Eliot B. Moss introduced a method for nested transactions [Mos81] that runs sub-transactions with their own synchronisation and recovery domain inside of a parent transaction. In case of a conflict a nested transaction can rollback and reset the state of the parent transaction to the beginning of the nested transaction. This method was called *closed nesting* (see e.g. [TRS12]).

From an abstract point of view not all conflicts on memory access are actually conflicts on application level. For example a hash table and two transactions where one inserts a value A and another removes another value B. The hash table may have a counter to maintain the amount of elements contained, which is modified by both transactions. Thus, on memory access level there is a conflict on this counter but on application level both operations insert A and remove B are commutative and not conflicting. *Open nesting* [NMAT^{+}07] addresses exactly this. It introduces abstract locks to detect whether concurrent operations are commutative or not. Based on this concept it allows nested transactions that do not conflict on those abstract locks to commit before the parent transaction. In case of a rollback it requires so-called *compensating actions* to revert the modification. In the hash table example above the compensating action is the inverse action of the transaction, which is either to remove A or insert B again.

To permit commutative actions to commit increases concurrency on those transactions but it does not reduce the loss of progress in case of a rollback. Also, it introduces additional contention on the abstract lock and development gets more complex. Considering the loss of progress overall improvement achieved through closed or open nesting is the partial rollback to the beginning of the first conflicting child transaction, which is usually not the exact position of the conflict.

Every child transaction increases management overhead. It requires a snapshot of the executing task to allow the task to return to the begin of the child transaction and its own data structures for read and write-set and so on. Increasing the number of child transactions would allow to rollback closer to the conflicting memory access but it increases the management overhead instead.

3.6.2. Checkpointing

Another approach to reduce rollback effort is checkpointing [WS08]. A *checkpoint* marks any position inside the transaction or nested transaction as a possible rollback target. It stores the current machine state of the task and the transaction it is running. This potentially allows

jumping to exactly the location of a conflict if it has a checkpoint. But again: increasing the amount of checkpoints would increase the probability to have a matching checkpoint for each conflict but it would also increase the overhead. To overcome this issue Waliullah and Stenström [WS08] experimented with prediction methods to dynamically determine proper locations for checkpoints but still the benefit is low.

3.7. The NOrec CC Algorithm

This section introduces the NOrec CC algorithm, which will be the basis for all helper-thread extensions developed in this thesis.

The NOrec STM system is implemented in a C/C++ library providing C API functions to be called such as to mark beginning and end of a transaction in the application code and to perform reads and writes of memory locations in terms of transactions. It is based on an optimistic CC, which uses no additional orecs. The STM system uses a timer called *single global sequence lock* (`sgsl`), which is used for two purposes: first, it is a commit counter, acting as a clock which reflects the number (multiplied by 2) of commits since system start and secondly it is a global mutual exclusion lock on main memory for commits: Each transaction doing a commit first acquires the global lock by incrementing the `sgsl` by one and increments it again when it has finished its commit. Hence, the lock is odd if set and even if not. The values read from memory and values to be written to memory are logged in a transaction's read-set and write-set. Following the deferred update method the actual write to the memory is performed at commit time. A transaction's state is said to be valid (consistent) as long as the values logged in the read-set are still equal to the original values in memory. To observe the memory state, the transaction reads the `sgsl` at transaction start as a *timestamp* and compares it with the `sgsl` at commit time before it writes its write-set into the memory. A difference in the `sgsl` indicates possible modifications in the read-set of the transaction and requires the transaction to perform a full validation, which means to compare each value of the read-set with its original location in memory. If this fails, the transaction aborts and restarts from the beginning. During execution the transaction performs eager incremental validation on every read, too. The validation updates the transaction's timestamp each time a full validation has proven the transaction to be in a valid state for a given `sgsl` value.

3.8. Parallelising Transactions

So far we considered a transaction to be executed by a single task or a set of tasks executing different parts of the transaction in a sequence, such as in distributed transaction processing.

This section will address parallelisation of transactions, which means executing a single run of a transaction by multiple tasks simultaneously. We differentiate three types of parallelisation that may be applied to transactions:

1. Parallelisation of the code in the transactional code section in terms of *thread-level speculation*.
2. Delegation of internal management overhead of a transaction to helper threads.
3. Execution of different instances of the same transaction concurrently by multiple threads to cover multiple *serialisation orders* at once.

The first type is to identify parallelisable tasks inside the code of the critical section and execute them concurrently by different threads. An example is to parallelise a loop by delegating the iterations of the loop to different threads and execute them in parallel. This concept is part of a discipline called thread-level speculation. The challenge here is to deal with the additional parallelism inside the transaction. The additional parallelism adds special issues the transaction has to deal with (see e.g. [BBKO10]) but it does not aim on the optimisation of the CC algorithm, which is the case in the next two types.

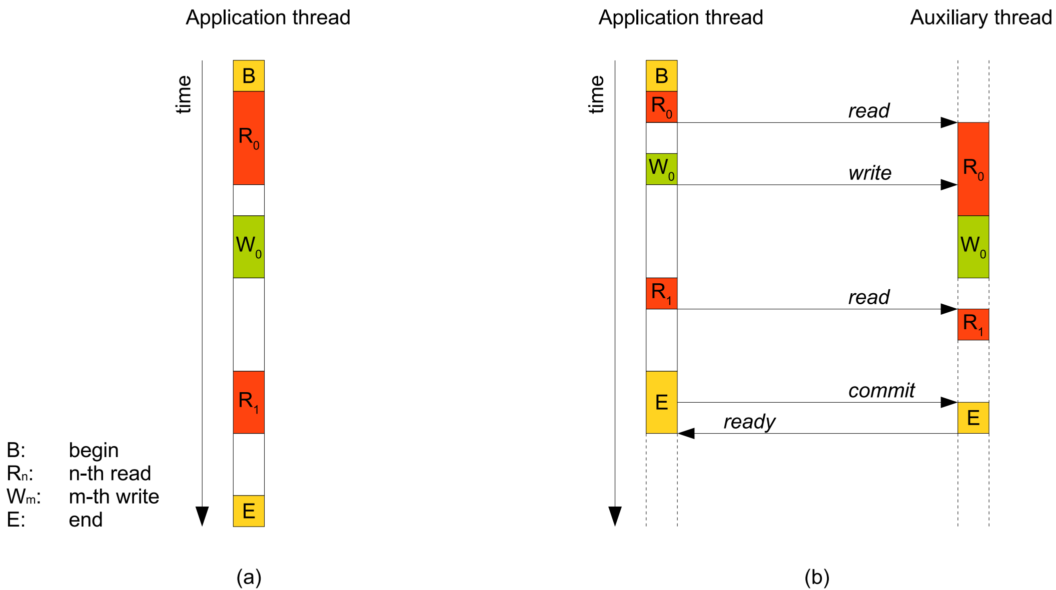

Figure 3.1.: Example for delegation of work to an auxiliary thread. (a) single-threaded execution, (b) application thread delegates to auxiliary thread.

Delegation of management overhead to helper threads was first addressed by Kestor et al. [KGH+11]. Their STM^2 implementation delegates validation, rollback and bookkeeping

(management of orecs) to a so-called auxiliary thread which asynchronously processes the delegated tasks. The application thread which actually executes the transactional section just forwards appropriate signals about the transaction's start, end, read and write events to the auxiliary thread (see Figure 3.1). While the application thread does no validation at all, the auxiliary thread validates and aborts the transaction in case of detected conflicts. Due to the lack of validation in the application thread, it requires sandboxing. Their evaluation demonstrated great performance improvements, which was mainly achieved through the delegation of the validation effort. As discussed in Section 3.5.2, validation generates a lot of overhead in eager validation. The alternative is to use lazy validation with sandboxing and a timer triggered validation in an interrupt handler, what reaches almost similar results (cf. [DS12]).

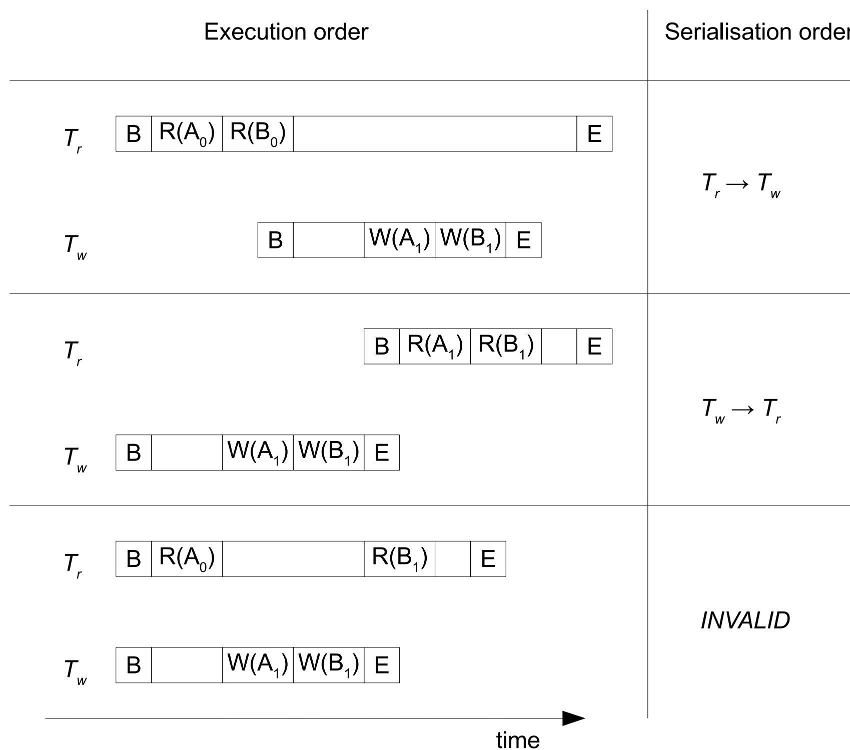

Figure 3.2.: Example serialisation orders for two transactions T_r and T_w operating on variables A and B. Indices of variables represent their version. At start both variables have an initial value indicated by version 0 (A_0 and B_0). T_w updates variables with another value generating version 1 (A_1 and B_1).

The third type of parallelism applied in a transaction aims on following multiple serialisation orders for a particular transaction at once. Consider for example an STM with OCC using lazy validation and direct update. When a transaction T_r optimistically reads shared variables

A and B, which may be modified by another transaction T_w concurrently, it speculates on a non-conflicting legal serialisation order. In this example the two transactions offer two legal serialisation orders, which are $T_r \rightarrow T_w$ (T_r occurs before T_w) or $T_w \rightarrow T_r$ (see Figure 3.2). That means either T_r reads A and B before T_w modifies them or after both have been modified. If only one of the reads follows a different serialisation order the state of T_r gets inconsistent. We already discussed these circumstances in conjunction with MVCC (see Section 3.5). There, a transaction can stay on one serialisation order it has originally chosen, even if concurrent updates already occurred. Considering the invalid case in Figure 3.2, T_r could have chosen to read B_0 instead of B_1 and thereby stay on the serialisation order it has originally speculated on, which was $T_r \rightarrow T_w$. If a transaction would have been able to follow both serialisation orders, applying a helper thread for the alternative serialisation order, the resolution of the conflict would just consist of a discard of the conflicting execution order and both transactions could commit. In this simple example, an MVCC could also provide a solution, but if more transactions occur concurrently there are multiple possible serialisation orders and every transaction chooses one of them. If the chosen serialisation orders of all transactions do not conform to each other, MVCC has to rollback conflicting transactions to solve it. A CC considering parallelised transactions could instead just discard all the clones of transactions that followed the wrong serialisation order.

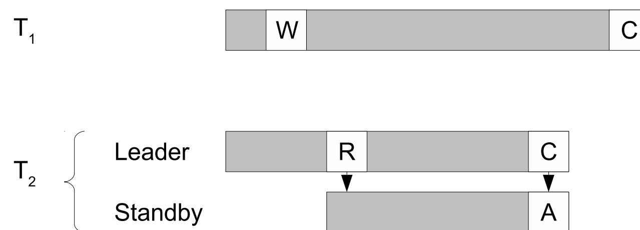

Figure 3.3.: SCC: Optimistic task successfully commits ($T_2 \rightarrow T_1$)

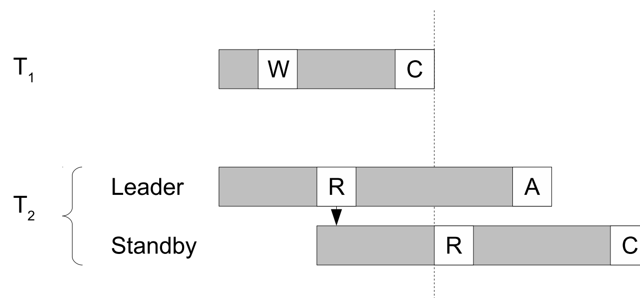

Figure 3.4.: SCC: Standby takes role of the first task ($T_1 \rightarrow T_2$)

The above example demonstrates just one possible approach to utilise parallel transaction execution. The first approach which actually addressed this kind of parallelisation of transactions to reduce the conflict rate was called *speculative concurrency control* (*SCC*) [Bes93, BB93, BW93] proposed for realtime databases. They applied multiple threads to execute in the same transaction and follow different serialisation orders. The first task (called *leader* from now on) runs an optimistic concurrency control with lazy validation and deferred update. Data is associated with orecs to indicate write attempts, which allows other transactions to identify possible conflicts. The remaining tasks (so-called standby shadows) of the transaction are spawned at conflicting read accesses to shared data. Those tasks wait at the orec until the owner has performed its update on the data. If this update occurs before the own transaction could commit then the optimistic task has failed. It will detect this inconsistency later when it tries to commit. The task which was waiting for the update, will instead immediately become the new leader. Recalling the example above, both tasks follow different serialisation orders: The leader speculates on a serialisation order where the own transaction commits first (see Figure 3.3) and the standby follows the serialisation order where the conflicting foreign transaction commits first (see Figure 3.4). Thus, both serialisation orders have been covered at once.

Because the transaction can spawn multiple standbys, it is theoretically possible to cover all serialisation orders and thereby entirely eliminate the loss of progress through rollbacks. But this approach has two algorithmic flaws: (1) It cannot deal with conflicts that occur when the leader already passed the read. In the example above, consider another read before the one that has been identified as possible conflict. In this case the standby reflects an inconsistent state too and none of the attempts is valid. (2) It possibly spawns multiple standbys for the same serialisation order. In the example above, if another read occurs in the transaction which conflicts with the same foreign transaction, another standby is spawned, which follows the same serialisation order as the first standby.

The original approach considers to spawn up to k standby shadows at the first k detected conflicting reads to limit the consumption of computing power. Covering all possible serialisation orders requires $(k+1) \cdot n$ tasks using this approach. Thus, on current hardware it is not possible to achieve full coverage even if the above flaws will be solved. But still it has the potential to reduce the rollback effort if the amount of standby shadows is bound to a proper limit. Due to this downside Bestavros also proposed an SCC with a maximum amount of two threads per transaction called two-shadow SCC [Bes93].

On closer examination the standby shadows functionality in this particular approach is very similar to checkpointing (see Section 3.6.2): They are instantiated at possibly conflicting memory access operations to resume at this position after failure. The main difference here is that they are autonomously observing the state of the memory which allows them to proceed

instantly when a modification occurs without having to rollback first. In single-threaded checkpointing the transaction needs some time to detect the inconsistency and rollback to the checkpoint before it can resume.

More generally the approach outlines the use of helper threads to cover multiple serialisation orders at once and this approach has the potential to greatly reduce the rollback effort over single-threaded transactions at the expense of a configurable amount of additionally utilised cores.

Chapter 4

Analysis

Bestavros' first published algorithm for SCC was a combination of a pessimistic and an optimistic concurrency control. The leader thread of a transaction executes optimistically and standby shadows perform busy waiting at a possibly conflicting read until the concurrent write is committed. In other words, the standby shadows are spinning on read/write locks.

Our first approach for an SCC for TM was an adaption of the two-shadow SCC design with slight deviations. The leader task follows a pessimistic approach using read/write locks (orecs) while one single helper thread per transaction optimistically reads even locked data, speculating that it is able to commit earlier than the owner of the lock[1]. Each transaction acquires a ticket (positive integer number) at its start and releases it after commit. The ticket represents the transactions priority in deadlock situations and decides which of the involved transactions has to abort. The read/write locks were stored in orecs and represented by a 128 bit integer. A lock could be either locked by multiple readers or a single writer. The ticket number was of the range $[1 - 127]$, which allows a compact representation of multiple (read) owners as flags in the 128 bit lock for deadlock detection.

In the evaluation of prototypes during the development of approaches all designs with PCC performed much worse than the OCC implemented in NOrec, which was chosen for comparison. After careful analysis two properties of the design have been identified as the major issues:

1. Use of orecs caused additional cache contention between tasks (c.f. Section 3.6).
2. Use of locking caused transaction queuing and cascading rollbacks (c.f. Section 3.6).

[1] The exchange of leader and standby role is just implementation related.

Obviously, locking and PCC are the root causes for both issues, which lead to the insight that a viable approach has to be based on OCC in general when targeting multi-core architectures without special extensions. Because known concepts for multiversioning also rely on orecs, the decision was to take a timer-based approach as the foundation. As mentioned in Section 3.6, NOrec [DSS10] is one of the best timer-based STM implementations today, thus ideal as the basis for our experiments with extensions for helper threads. Furthermore, this way we can easily evaluate the value of the extensions by comparison to the original NOrec implementation.

The first section in this chapter will introduce a few important non-functional requirements to be considered during the design and the development of an STM system due to its highly concurrent behaviour. Thereupon, the following section will introduce different proposals for possible designs to apply helper threads in terms of SCC inside transactions using a timer-based CC as its foundation. This section will also identify the peculiarities of SCC in TM which raise different problems: The problem to prevent internal errors from getting visible for the application through sandboxing with minor overhead and how to efficiently parallelise a transaction through cloning of threads. The remaining sections in this chapter will develop and discuss different approaches to those problems to take decisions for the design.

4.1. Non-functional Requirements

Besides the CC algorithm, the runtime performance of an STM is significantly influenced by the properties of the runtime environment (cf. Section 2.2). Before developing approaches for the application of helper threads in TM, the essential constraints driven by the runtime environment have to be explained, which significantly influence the design of the CC approaches. Literature studies and experiences with earlier projects condensed in the following non-functional general requirements for the development of an STM with helper threads:

1. **Avoid cache misses:** A significant amount of time in concurrent applications gets lost due to memory transfers between main memory and caches and between caches of different cores. Thus, it is generally important to keep the amount of additional memory accesses (such as access to orecs) low.
2. **Avoid inter-cache communication:** Helper threads will potentially increase the overall amount of memory accesses because they need to communicate with the leader thread at some points. This will induce cache to cache communication. This overhead can be for example reduced when placing leader and helper threads of the same transaction on neighbouring cores of the same processor that share some caches in the cache hierarchy. This can also reduce cache misses because the helper thread can access data that has for

example been fetched from memory by the leader thread earlier. But communication with threads of other transactions will still cause inter-cache communication and shall be limited.

3. **Avoid context switches:** Context switches are expensive too and should be avoided as well. Thus, threads should get a permanent assignment to a particular core. Also, the transaction should use the calling thread and thereby avoid another unnecessary context switch.

4. **Low level programming:** Efficient programming of internal behaviour in CC requires fine-grained control over memory access ordering, the location where temporary data is stored and other aspects of parallel and concurrent programming. Also, the runtime environment of the programming language should have no unnecessary influences on measurement results in the evaluation later. Therefore, the chosen programming language is C, which is a system oriented low level language supporting embedded assembler, too.

Today's multi-core hardware usually provides multiple cores on a CPU but with increasing number of helper threads the amount of accessed memory and inter-cache communication rises. Because this effects conflict with requirements 1 and 2, the approaches proposed here use one helper thread per leader at most. To meet requirement 3 the helper threads will be bound to a particular core. To reduce inter-cache communication helper threads should be placed close to the leader thread, too. Thus, a helper thread will be dedicated to one particular leader thread for its entire life, to fulfil requirement 3 properly. Requirement 4 has no major effects on the design of the approaches but on the details of required sub-systems as explained later.

4.2. Draft Approaches

As mentioned in the introduction of this chapter, an implementation of the original approach for SCC requires orecs associated with shared data objects to realise locks. Unfortunately, orecs double the contention between tasks, and locks introduce queuing transactions and waste time through waiting (see Section 3.5.1). The decision was to find OCC approaches without orecs to realise SCC for TM.

OCC leaves just a few possibilities to realise SCC on top of it. As already discussed in Section 3.8, checkpointing is very similar to SCC with the difference that it is single-threaded. In this regard we differentiate between distinct levels of parallelisation towards SCC, where

checkpointing marks the lowest level, and a CC which executes all possible serialisation orders in parallel the highest level.

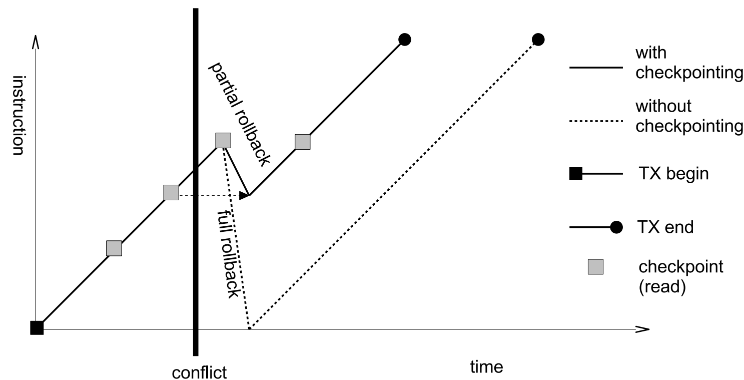

Figure 4.1.: Checkpointing vs. incremental validation: ideal STM

Although checkpointing does not simultaneously process different serialisation orders for the transaction, every checkpoint represents the entry point for an alternative serialisation order. This alternative serialisation order gets available only if a concurrent transaction modifies the memory location associated with the checkpoint. The transaction will get aware of the modification with the next validation. Considering eager validation, as applied in NOrec, this will occur with the next read or the next try to commit. Thus, a single threaded transaction with checkpointing could rollback and enter the checkpoint to follow the new serialisation order that was established by the concurrent commit.

Figure 4.1 compares the progress of a transaction with checkpointing with another transaction which uses incremental validation and full rollbacks. The horizontal axis represents the progress in *time* and the vertical axis the progress in lines of code (*instructions*) inside the transactional section. In this regard, a transaction begins at the lower left corner, and moves in positive direction along both axes and ends if it reaches the last instruction, the top most edge of the graph. Accordingly, a *rollback* results in a negative progress on the vertical axis but a positive progress in time on the horizontal axis. The thick vertical line indicates the time when a *conflict* occurs. Depending on the validation scheme operated by the transaction the rollback will occur either earlier (e.g. with the next read in eager validation) or later (e.g. with the next attempt to commit in lazy validation). As shown in the figure, in an ideal STM implementation and runtime environment the transaction with checkpointing will finish earlier due to the partial rollbacks.

Unfortunately, the read which will cause a conflict in future is unknown, and thus a checkpointing mechanism will either have to checkpoint every read in a transaction or predict

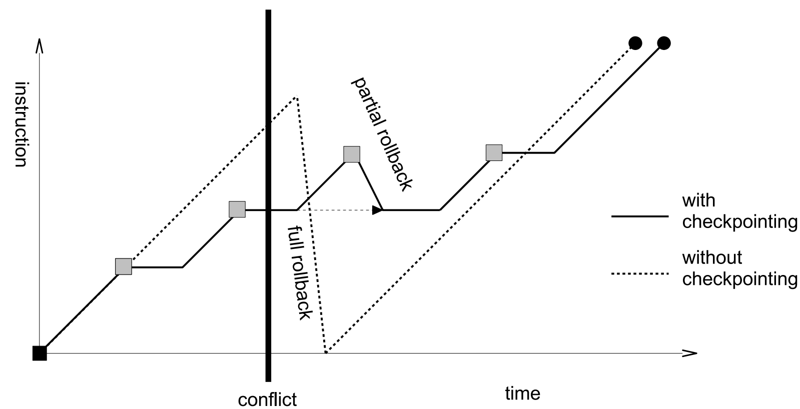

Figure 4.2.: Checkpointing vs. incremental validation: realistic STM

conflicts and place checkpoints at reads with the highest conflict probability. Checkpointing inside a transaction is expensive, because it requires more than just a copy of the processor registers. Considering checkpoints inside function calls entering a checkpoint also requires to restore stack frames that might have been left already. Thus, a checkpoint in this case would require a copy of the stack as well and the overhead of the checkpoint is even higher. As a result, checkpointing is significantly slower (see Figure 4.2). Thus, checkpointing every read is no option. The prediction instead could have a positive effect but still it generates additional effort to gather statistics for the prediction and it depends on the success rate whether it will be significantly faster than a simple rollback.

Deferred updates have the advantage of generating less overhead in the average case compared to checkpoints. Essentially, in case of a rollback the transaction has to reset the index in the write log to its first position, restore the processor registers and recompute the critical section up to the conflicting read. Hence, the most effort is spent in recomputing of the critical section after a rollback. Additionally, the average location of a conflict between multiple transactions moves more and more towards the begin of the critical section with increasing number of concurrent tasks (see Appendix B). Consequently, the rollback effort decreases if the probability for a contention gets higher. In contrast, checkpointing generates a constant overhead even in lower contention scenarios. Hence, checkpointing will be avoided in favour of full rollbacks in our approaches.

The following sub-sections outline several draft approaches towards a realisation of SCC with two parallel tasks for one transaction (i.e. two-shadow SCC, cf. Section 3.8) based on NOrec.

4.2.1. Helper Thread Validation

The next level closer to SCC than checkpointing is to delegate the validation to a helper thread almost similar to Kestor et al. [KGH+11] (see Section 3.8). Applying this approach to NOrec, the helper thread has to constantly iterate over the read-set of the leader each time it detects a commit that was signalled through the `sgsl`. Delegation of the validation will probably have two positive effects:

1. It allows running without read validation in the leader thread, which improves the response time for the first serialisation order, i.e. the serialisation order without conflicts but it requires sandboxing.

2. It provides earlier detection of conflicts because the time to detection with eager validation depends on the time to the next read or commit in the leader thread. The helper thread can signal the conflict to the leader and cause an immediate abort, which reduces the time required for the alternative serialisation order.

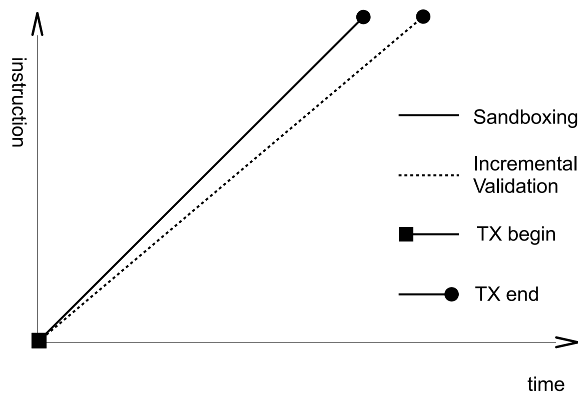

Figure 4.3.: Timer-triggered vs. incremental validation: without conflict

Figures 4.3, 4.4 and 4.5 demonstrate the advantage of delegated validation over eager incremental validation in general. A transaction running with incremental validation is slower due to the in-flight validation (see Section 3.5.2). Thus, a run without conflict will always be faster with sandboxing. With the traditional sandboxing approach (see Section 3.8), validation is not delegated and the timer-triggered validation will occur comparably seldom. Consequently, the validation usually happens at the end of the transaction unless it runs in an endless loop. Thus, the incremental validation will detect the conflict earlier and a conflict is fixed faster (see Figure 4.4). With delegated validation (see Figure 4.5) the transaction is constantly validated and the conflict will actually be detected earlier than in incremental validation. Additionally, the leader will be faster without the incremental validation.

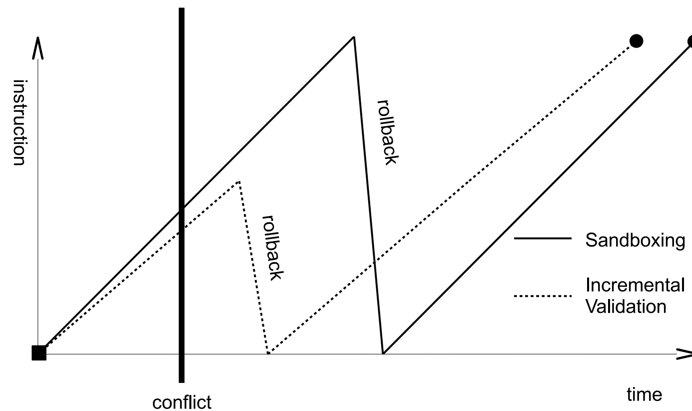

Figure 4.4.: Timer-triggered vs. incremental validation: with conflict

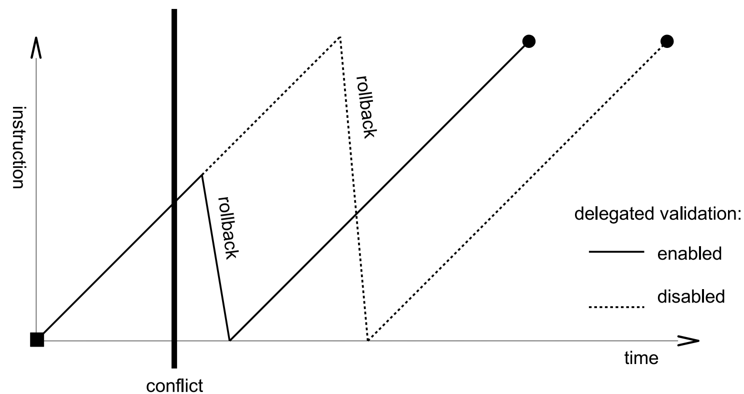

Figure 4.5.: Delegated validation

Unfortunately, the delegation requires lots of communication between leader and helper thread. It aggressively accesses the `sgsl` (read-only) and the read-set, which is concurrently modified by the leader on each read. Thus, it causes permanent contention, which will probably influence the whole application performance. The main purpose to adapt the approach of Kester et al. to NOrec is to have a demonstrator to evaluate the next approach, which is an alternative of this one.

4.2.2. Cloned Incremental Validation

A different approach to delegate validation without additional contention on the read-set can be derived from the consistency model we have originally specified for transactions (see Section 3.2): A transaction transforms a given consistent state of shared data into another consistent state. Any deviation from the consistent state present at start will result in inconsistencies

meaning a read from a different state causes the inconsistency. Consequently, two threads running the same transaction starting with the same state must perform the exact same transformation unless a conflict occurs. In other words, they will read the same values and perform the same writes until either a commit or an abort occurs. Therefore, a thread cloned during the start of the transaction can execute the same transaction and will thereby establish a read-set, which is identical to that of the leader thread as long as no inconsistency occurs. A viable concept for delegated validation without additional contention on the read-set is therefore a cloned thread which executes the same transaction with eager validation while the leader thread performs lazy validation at commit time. If the helper thread detects a conflict it signals it to the leader, forcing him to abort and aborts itself, too. This is required to prevent the leader thread to run into consistency issues such as endless loops. Thus, both threads return to the start position again.

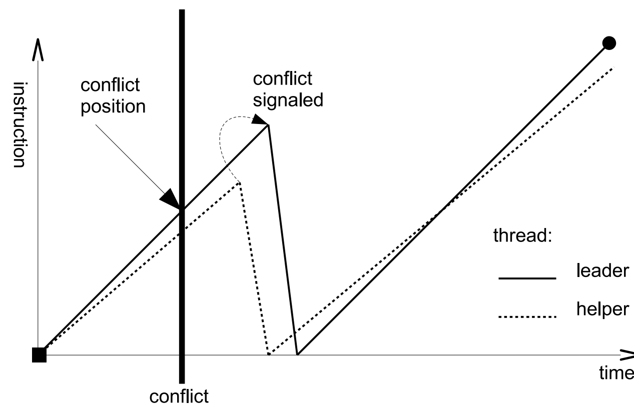

Figure 4.6.: Cloned incremental validation

Cloned incremental validation has the theoretical advantages, modelled for delegated validation in general in the previous paragraph (see Figure 4.5). Due to the latency caused by the incremental validation the helper thread will have a slight backlog compared to the leader thread and the detection will have a bit of an offset (see Figure 4.6). Also incremental validation has a theoretical disadvantage over constant helper thread validation due to the fact that an incremental validation occurs on every read only. But compared to constant read-set iteration it will cause less contention in the system, which will improve runtime performance.

The two previous approaches still do not execute multiple serialisation orders for one transaction in parallel but reduce the time until a contention is detected and accelerate non-conflicting transactions.

4.2.3. Cloned Synchronous

Eager validation has the chance to fix an inconsistent read due to its post read validation. If a currently performed read is detected to be inconsistent but all previous reads are valid, the transaction can redo this particular read and proceed in the new serialisation order (see Figure 4.7). This is a rare case but it represents a case where the transaction can instantly switch to another serialisation order without rollback. The approach proposed here extends the previous approach to benefit from this case. Again the leader thread will apply lazy validation and sandboxing and the helper thread uses eager incremental validation and signals inconsistencies to the leader. Additionally, the helper thread fixes read inconsistencies if possible and it is allowed to perform a commit if the leader thread has not done so far.

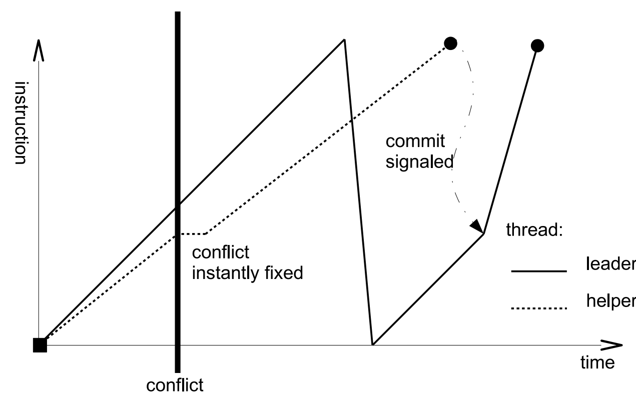

Figure 4.7.: Cloned synchronous: instantly fixed conflict

Because the helper thread notifies the leader about conflicts both will rollback and restart synchronously in most cases. Most of the time, they will run on the same consistent state of the shared data. The expected result is that the leader will still finish earlier if no conflict occurs because it does no validation. The helper thread will instead finish earlier if it can instantly fix inconsistent reads.

There is no contention on the read-set but increased contention on the `sgsl` due to the additional attempts to commit through the helper thread. Also, the optimised case of an instantly fixed read is so rare that the benefit will be low. The main purpose of this approach is to demonstrate a way to instantly adapt to a different CC on the fly while current approaches to switch between different CCs require rollbacks (c.f. [MSS05]).

4.2.4. Cloned Asynchronous

The synchronisation between both threads in the cloned synchronous approach forces both threads to restart on the same state and consequently on the same serialisation order. Removing all synchronising mechanisms would instead allow both threads to run on different serialisation orders and thereby establish another level of parallelisation towards SCC.

This approach is almost similar to the previous approach except the leader thread now applies timer-triggered validation itself to solve endless loops as in [DS12] for single-threaded CC. Thus, even if the helper thread detects conflicts, it does not need to and will not abort the leader thread but itself and both run independently and on different serialisation orders.

The approach still lacks control over the serialisation orders selected by the threads, which seems to be ineffective. To improve the success rate of the helper thread, the following alternatives closer to the original SCC approach have been considered:

- The helper thread could wait at the start and observe the `sgsl`. On modifications, it starts its own run on the possibly new serialisation order. It cannot know whether the leader is actually conflicting with the current serialisation order and thus it might produce the same result as the leader. Additionally, the contention on the `sgsl` will be significantly increased, which influences all transactions.

- It could instead validate the read-set of the leader to actually check if a new serialisation order is beneficial. This would again introduce a lot of contention on the read-set.

- It could also validate the run of the leader thread by executing the transaction on the given state by itself as proposed for cloned incremental validation earlier. This would allow the helper thread to detect a conflict without additional contention and start reliably on a new serialisation order.

The last alternative is most promising. Thus, the helper thread will just start with incremental validation on the same state as the leader and it will automatically switch into a new serialisation order in case of a conflict.

The expected effect of this approach is that the leader thread will still finish earlier in non-conflicting situations (cf. Figure 4.3, considering timer-triggered as leader and incremental as helper thread). If a conflict occurs it will detect this inconsistency usually at commit time except the timer-triggered validation occurred earlier. The helper thread will start on the same state but aborts earlier on conflicts and thus it will switch to the new serialisation order sooner. Until the leader detects the inconsistency, the helper already runs on the new serialisation order (cf. Figure 4.4, considering timer-triggered as leader and incremental as helper thread again). When the leader finally rolls back, the helper will be already far ahead in the transaction. The

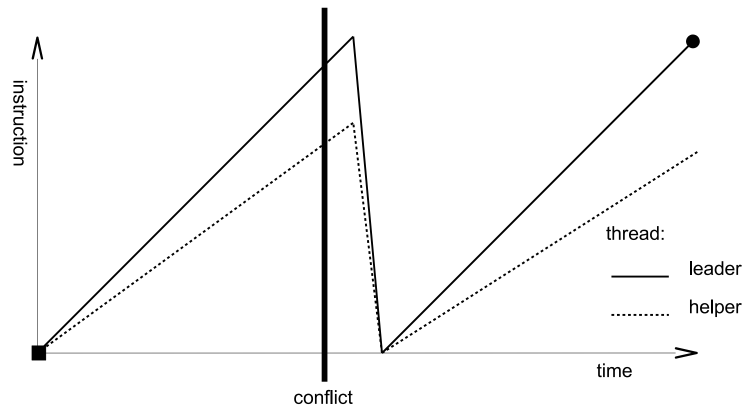

Figure 4.8.: Cloned asynchronous: late conflict

	NOrecHT	NOrecCIV	NOrecCs	NOrecCa
Leader validation type	lazy	lazy	lazy	lazy + timer-triggered
Parallel transaction execution	no	yes	yes	yes
Helper signals conflicts	yes	yes	yes	no
Helper commits	-	no	yes	yes

Table 4.1.: Overview of all approaches and their features

opposite might happen if a conflict occurs late in the run of the leader. Because sandboxing is faster in general the leader will finish earlier (cf. Figure 4.8). Generally, both will drift apart and follow different serialisation orders as soon as conflicts occur. This will in turn increase the probability of cases where the helper thread will run a non-conflicting serialisation order and successfully commit in favour of the leader: the leader can simply discard its current run and switch to the committed state.

Table 4.1 provides an overview of the CC approaches planned and their features in comparison to each other.

4.3. Transparency and Sandboxing

The drafts developed in the previous section require several subsystems besides the actual CC algorithms. Because the leader threads in all approaches do lazy validation they will require a sandboxing mechanism. The sandboxing concept is highly controversial due to the expected overhead. Thus, we revised the only published approach of Dalessandro and Scott by a careful analysis of actual reasons for transparency violations and the solutions they have chosen, to identify the overhead and develop possible improvements.

4.3.1. Transparency Violations

A transaction working on an inconsistent state is doomed to suffer from several possible errors that would not happen with mutual exclusion. The following list covers all possible application visible errors raised by doomed transactions running on x86 64 bit multi-core architectures. Dalessandro and Scott [DS12] gave a similar list of errors but we added more errors and details, removed redundancy considering an OCC and C as the programming language and developed a graph to identify cause and effect relationships, which offers ways to prevent or suppress them.

1. Hardware exceptions: Those are the interrupts (POSIX signals on application level) generated by the hardware on these exceptions:
   a) Memory access violations while using inconsistent pointers or pointers on inconsistent data such as deleted data.
   b) Arithmetic exceptions such as division by zero or overflows.
   c) Illegal instruction errors when entering a memory section that does not contain executable code for example caused by mistaken jumps or inconsistent modifications to code, considering self-modifying code.
2. Incomplete transaction exit: A transaction may accidentally escape the critical section without a proper commit. Not committed writes are no longer visible to the task, the task can be asynchronously rolled back leaving shared data in an inconsistent state due to the lack of transaction support and the next entry of a transaction is interpreted as a nested transaction resulting in more errors. Reasons for these escapes are inconsistent function pointers or jump targets such as the return address saved on stack or dynamically computed jump targets of `switch-case` or `goto` statements.
3. Computation of non-transactional code: A transaction may accidentally enter non-transactional code such as an uninstrumented function, while inside the critical section. Again, uncommitted data is not visible and unexpected rollbacks may occur leaving shared data in an inconsistent state as in case 2 above. Reasons are the same as for incomplete transaction exits but here the thread will at least return to the transactional code.
4. Infinite loop/recursion: Inconsistent termination conditions may result in endless loops or recursions. This endless loop or recursion may as well contain no instrumentation so the STM cannot become aware of this state.

5. Resource exhaustion: Inconsistent parameters to allocation requests for resources such as main memory or space on a hard drive (files) may result in an exhaustion of that resource.

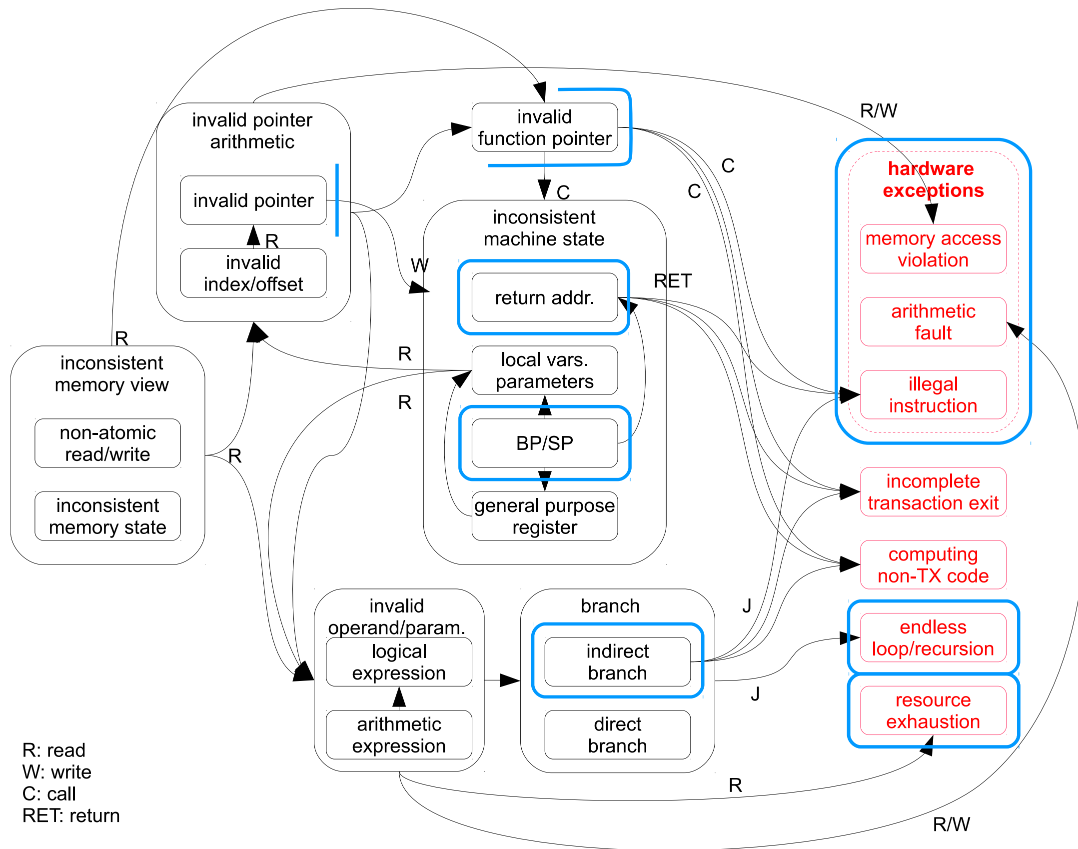

Figure 4.9.: Error model for doomed transactions

The error model given in Figure 4.9 provides an overview of the dependencies between effects of doomed transactions to the left and application visible errors marked in red to the right. Considering inconsistent memory states and non-atomic reads/writes as root causes for all possible transparency violations the arrows point towards subsequent errors. Each dependency is associated with one or more low level instructions such as read (`R`), write (`W`), jump (`J`) and function call (`C`) or return (`RET`). The following paragraphs will explain the characteristics of the effects of inconsistencies depicted in Figure 4.9.

Inconsistent Memory State An inconsistent memory state exists when a transaction reads data from different serialisation orders. For example a transaction reads two variables that are updated by another transaction. If the transaction reads the first variable before

it gets updated and the second from the updated state, both reads do not belong to the same consistent memory state. Instead the reads reflect an inconsistent memory state where just one variable has been updated. The result of such an inconsistent state are invalid operands or invalid pointer arithmetic. A well known class of consistency issues of this kind is caused by *publication* and *privatisation*. Publication is the procedure a thread follows to publish information and make it visible to other threads (see Listing 4.1). For example the thread assigns a new allocated memory area to a pointer, e.g. the end of a linked list. Privatisation is the opposite: a thread removes the information from the shared data area and uses it locally (private) (see Listing 4.2). Privatisation and publication do not necessarily require pointers, but some variable which indicates whether there is data available (i.e. public) or not. If a transaction reads from an inconsistent memory state, the variable might still indicate that the data is available while it actually is used locally by another thread without a transaction or has been deleted already.

```
/* create a private data structure */
void* local_ptr = malloc(..);
init(local_ptr);

/* publish data */
shared_ptr = local_ptr;
```

Listing 4.1: Example of publication

```
/* privatise ptr */
void* local_ptr = shared_ptr;
shared_ptr = NULL;

/* reclamation */
free(local_ptr);
```

Listing 4.2: Example of privatisation

Non-atomic Reads and Writes Read of native data types is not atomic. For example a read of a 64 bit integer which spans over different cache lines may overlap a write of the same integer and results in an inconsistent read. Direct effect of inconsistent reads are invalid pointers, invalid indices in arrays or invalid operands in general.

Invalid Pointer Arithmetic This category addresses all kinds of errors resulting from invalid pointer arithmetic such as use of a pointer on freed memory, invalid pointers in general and invalid indexes or offsets used in pointer arithmetic.

Memory referenced by an invalid pointer may be non-existent or freed by another thread. This can result in a memory access violation detected and signalled by the hardware through an interrupt. The same memory area might have been issued to another thread in the meantime or the pointer just refers to some completely different location and the doomed transaction just accesses chaotic data.

Invalid pointers are the result of inconsistent reads on shared data only. Read of private data, such as local variables on stack, can be considered to be valid as long as reads from shared data are consistent. Direct write access via invalid pointers obviously results in unpredictable effects on shared and local data affecting the whole application.

In the programming language C, indexes and offsets are the same. Both can have the same effects as invalid pointers in the first place. Additionally, STM implementations usually do not apply their CC on data in stack such as local variables, because they are considered as thread-private. Thus, inconsistent indexes or offset into data on the stack may result in uncontrolled modifications on it too, which can cause an inconsistent machine state (see below).

Invalid Operands Invalid operands are the result of inconsistent reads from shared data, data referenced by invalid pointers or inconsistent machine states (see below). Invalid operands may affect logical or arithmetic expressions or computed jump targets causing access to non-transactional code, arithmetic faults, resource exhaustion or infinite loops or recursions.

Inconsistent Machine State An inconsistent machine state may be caused by an invalid function pointer or inconsistent stack content for example caused by off-by-one errors using invalid indexes or offsets on local variables or arrays. As introduced in Section 2.2 stack pointer (SP) and base pointer (BP) of the previous stack frame may be stored on the stack. For example an inconsistent write access to an array on the stack can cause BP to be overwritten. If the task restores the previous stack frame when leaving the function, it uses these erroneous BP and the whole stack frame is wrong. Just as BP, the return address can be modified too. When leaving the function the task consequently jumps to this address, which may point to any position in the virtual address space resulting in all kinds of errors such as entering non-transactional code or even sections that contain no proper instructions at all. Before entering a function the task also stores several registers on the stack to be restored on return. Those may be affected by inconsistencies the same

way as BP and the return address above, resulting in inappropriate inconsistent local variables when restored.

4.3.2. Sandboxing

The only comprehensive sandboxing approach for C and C++ existing so far was published by Dalessandro and Scott [DS12]. It is based on a TM-aware C compiler, which provides automatic instrumentation against a proprietary ABI according to a proprietary simple TM API almost similar to the features of the C++ language extensions (see Section 3.4.1). The proposed sandboxing concept is basically to either catch and undo errors of doomed transactions or prevent them by additional validation and possible rollbacks. Referring to our error model in Figure 4.9 the goal is to cut out dependencies until the errors in the red boxes on the right cannot be reached from the root causes to the left.

They used a special extension to the compiler to insert pre-validation hooks for dangerous operations. Dangerous operations are basically in-place stores whether on stack or shared data and indirect branches (dynamically computed jump targets) that result from shared reads. Every read from shared data is considered to be inconsistent unless it was validated. If the data read is used in a subsequent dangerous operation, it requires a pre-validation to prevent possible errors. Hence, the compiler extension instruments almost every in-place store and indirect branch with pre-validation code. The inserted code based on a colouring algorithm decides at runtime whether a validation is necessary or not and aborts in case of inconsistency.

The interrupts raised by hardware exceptions are validated in special signal handlers, too. If the transaction is proven to be valid, the interrupt is raised to the application, otherwise it is suppressed and results in a rollback.

A timer-based periodic validation is applied to escape from infinite loops and recursions. It is simply triggered by a POSIX timer, which raises an interrupt with a dynamically adapting frequency between 1Hz and 100Hz adapting to the frequency of detected errors. The interrupt is processed in another signal handler which performs the validation.

For each function call to function pointers the compiler inserts a lookup of the corresponding transactional function. If a non-transactional function is found, the STM switches to its irrevocable mode, which implies a validation prior to it. Hence, an invalid function pointer without a match to a transactional function is pre-validated implicitly.

These four mechanism pre-validation of in-place stores and indirect branches, catching interrupts, periodic validation and function pointer validation properly sandbox all errors depicted in the error model except the resource exhaustion. Additionally, they considered some STM specific characteristics. They had to add a final validation to read-only transactions while eager validation would not need it. They considered intentionally non-transactional

code sections such as pure functions or so-called *waivered code* sections by additional pre-validation. They also considered system and library calls to be irrevocable, which implies a pre-validation unless they are declared to be transactional.

The remaining disadvantage of the approach of Dalessandro and Scott is the instrumentation of in-place stores. Because NOrec uses deferred updates, in-place stores happen on local variables only. Access to variables is buffered in the write-set and written back to its original location after a final validation during commit. Thus, the only in-place stores to be validated are those via pointers into backups of the machine state in the stack, particularly the BP and the return address field. This dependency is depicted in Figure 4.9 by the write access relationship between *invalid pointer arithmetics* and *inconsistent machine state*.

An analysis of the GCC TM compiler revealed that it instruments every kind of write via pointers with a call to a write function of the TM ABI whether it addresses locations in the stack or not. Thus, the GCC compiler already provides sufficient instrumentation to insert a validation of write access to the stack and a protection of critical parts by suppressing or redirecting the write access in the write-log.

Because access to this critical content cannot be part of a C program, unless it uses embedded assembler blocks, which are not allowed inside transactional sections (see [ATSG12]), we can consider them as non-intended by the programmer. Thus, access to critical parts in the stack is invalid per definition, it does not need a full validation of the read set and immediately aborts the transaction, which is the main difference to validation of the return address proposed earlier. In case of an actual application error this can result in an endless loop. For debugging purposes the transaction could check if it actually works on consistent data (i.e. if no other thread performed a commit) and terminate the process to interrupt the endless loop. But actually such an application error can result in an endless loop anyway and we don't have an advantage adding the validation here. This will significantly reduce the required validation overhead compared to the approach of Dalessandro and Scott.

The interception points in the development of application visible errors through the graph in Figure 4.9 are depicted by blue lines. The machine state protection by validation of invalid pointer arithmetic related to the stack represents the main difference to the sandboxing approach of Dalessandro and Scott and suppresses a lot of subsequent errors caused by the inconsistent machine state.

We do not see more efficient alternatives to concepts such as out-of-band validation to terminate endless loops/recursions and validation of function pointers or signal interposition to handle hardware exceptions, and will use them in our approach as well. However, function pointers and signal interposition do not need a full validation in our approach.

There are just two reasons for dangerous signals such as segmentation faults or floating point exceptions to occur:

- The transaction is working on inconsistent data and requires a rollback to hide the signal.
- The application is actually erroneous in which case the signal will cause a termination of the process.

In a correct application the second case will never occur and thus we decided to optimise the first case. If the globally shared timer indicates that another transaction has committed and potentially modified globally shared data, the signal is considered as invalid and the transaction aborts immediately without a full validation of the read set. In case of an actual application error the signal will be produced again while the transaction is valid and finally terminate the process.

The unsuccessful lookup of transactional clones for function pointers can be treated similar to occurring signals and access to sensitive parts of the stack. Those are by definition invalid and must be errors of the application or the result of an invalid state of the transaction. Thus, on unsuccessful lookups the transaction aborts considering the state to be invalid. Again, for debugging purposes a validation helps here but is not needed for correctly implemented applications.

Resource exhaustion can be either prevented through pre-validation or resolved on demand. Because pre-validation would decrease the advantage of sandboxing again, we will analyse possibilities to solve the exhaustion on demand. Most of the possible resource exhaustion incidents will be pre-validated anyway, because the system library functions are usually not declared to be transactional and the STM has to switch to the irrevocable mode, which implies a validation. But there are functions needed to support transactional behaviour and acquire system resources such as `malloc` for memory management.

We will use `malloc` as an example to explain the issues in resource exhaustion. It is easy to detect a resource exhaustion: The `malloc` function returns an error code, which indicates a lack of available memory. System memory is limited by the available main memory and possibly existing file system space to swap out pages. Thus, the system memory is shared by all processes and all threads and the incident might be shared by multiple transactions consequently. This means, the transaction that experiences the lack of system memory might have a valid state, while another transaction wrongly allocated a huge memory area. Of course the invalid transaction will detect this inconsistency later and resolve it by a rollback releasing the allocated memory, but it causes a lot of interference in the system: The system starts to swap pages of other tasks and the invalid transaction causes the whole system to slow down. Thus, there is a fairness problem that can be solved by a system wide control mechanism only.

The development of a system wide control mechanism for resource allocations is far beyond the topic of this thesis. Hence, we decided to use a hybrid method for memory allocation,

which applies pre-validation in cases where a given limit of requested memory is exceeded, only. This still does not solve the actual problem but it reduces the risk of resource exhaustion sufficiently to allow a proper evaluation of the developed CC algorithms.

The other remaining issue is caused by computed jump targets, which might result in computation of non-transactional code. We have found no alternative solution to pre-validation in this case either. However, the use of such dynamically computed targets in indirect branches is very rare, and thus we decided to prohibit them entirely in transactional sections because otherwise we would have to develop a compiler, which supports it.

The design decisions made so far establish a sufficient sandboxing approach for our CC approaches, which is less intrusive then the approach of Dalessandro and Scott.

The only issue still not addressed are non-transactional sections inside transactions such as pure functions (see Section 3.4.1). While pre-validation of non-transactional sections would be enough for single-threaded transactions, they will cause inconsistencies in parallelised transactions. Because pure functions are not instrumented they have the following two restrictions:

1. Pure functions cannot access shared data.
2. Pure functions shall not have any side effects visible to other transactions.

Because write access of the transaction to shared data is buffered in the write-log, uninstrumented reads to shared data will not reflect the writes of the transaction itself. Also, updates of other transactions may not be complete and an uninstrumented read may be inconsistent. Likewise uninstrumented writes with deferred update would result in direct writes and violate the isolation of transactions. In fact any kind of side effect would be visible to other transactions and thus it has to be prohibited.

However, some STM systems consider pure functions or waivered sections to be uninterruptible and allow side effects that are legal in terms of application behaviour. Such a *legal* side effect could be to allocate memory or acquire a lock as long as it is released in case of a rollback. For example the memory allocator `malloc` uses a lock to protect its internal data. If an allocation or release of memory is interrupted, the lock will be left acquired by the caller. Thus, the periodic validation required for sandboxing has to be suspended for those STM implementations, because a rollback would violate the second constraint.

Pure functions actually allow to work efficiently on private data. For example a task might have just allocated a large data structure and started a transaction to initialise it from shared data and publish it afterwards. If the transaction is executed by multiple tasks at once, for example using the asynchronous cloned approach proposed in Section 4.2, both tasks will directly access the same data structure concurrently. While one task may commit the other

might still run on an inconsistent state, directly writing inconsistent values to the 'private' data structure. The unacceptable result of this case is a committed inconsistent data structure.

Although properly applied pure functions provide significant improvements to the response time of transactions in general, they are error prone and complex in use especially in terms of maintainability. This might also be the reason why pure functions are still not part of the proposed language extensions for C++. Therefore, and because this is not part of the goals of this thesis, the STM systems developed here will not support pure functions with side effects.

4.3.3. Transparency of Memory Management

Today's STM systems support memory management inside transactions. Because memory management is actually not compatible with transaction processing, this section is dedicated to analyse different issues resulting from memory management and find proper solutions. First, memory management causes issues with privatisation and publication schemes. The second issue is to support the integration with transactional rollbacks and deferred updates.

As mentioned in the Section 4.3.1, memory which is reclaimed by the memory management can cause memory access violations in concurrent transactions, another form of transparency violation. A memory reclamation causes a modification in memory, which is not reflected in the contents of the reclaimed memory, but in the pointer referencing it: An application which needs to reclaim shared data has to use the privatisation scheme first, to signal concurrent tasks that this data is no longer publicly available. Thus, the reclamation problem is actually a privatisation problem and should not be an issue with eager validation.

The eager (incremental) validation in NOrec accesses the original memory location to compare its current state to the value originally read by the transaction. If a reclamation occurred right after the validation of the pointer to the reclaimed data, this access to the memory location can cause a violation. To prevent those access violations to occur, the NOrec implementation uses an epoch-based memory reclamation [HSATH06, Fra03]: The basic idea is, to defer the reclamation to a point in time where it is absolutely clear that there cannot exist any concurrent transaction still running, which considers the now private data to be public. This point in time is called *quiescence* time [WCW+07].

In epoch-based reclamation, the execution time of a concurrent application gets logically partitioned into so-called global epochs to find the quiescence times for memory reclamations. The first global epoch begins at the start of the application. The lifetime of each thread is partitioned into thread-specific epochs as well. A new epoch of a thread begins with the start of the thread and every start, restart and end of a transaction. A new global epoch begins each time when all threads have switched into a new thread-specific epoch. Every memory reclamation request of transactions gets associated with the currently running global epoch

and stored in a global data structure called *limbo*. The limbo is checked by each committing transaction for pending reclamation requests that have reached the point of quiescence (older than two global epochs). Those requests will then be executed by the currently committing transaction.

This approach occupies memory for a longer period than required, which prevents memory access violations but may lead to memory exhaustion, too. This can be a serious issue as we have found in a study on NOrec (see author's publications [MT13]). Because our sandboxing approach in Section 4.3.2 already handles memory access violations, we will not need the epoch-based reclamation any more. But to align memory reclamation properly with the deferred update and rollback scheme of NOrec, it requires a modification of allocations and reclamations in transactions:

- Allocations are directly executed. Additionally, every allocation has to be registered in a log for reclamation in case of a rollback.
- Reclamations have to be deferred until the transaction is committed. Otherwise the reclaimed data would be lost in case of a rollback.

This allocator approach can also be combined with a simplified sandboxing approach, which just handles signalled memory access violations, to replace the epoch-based reclamation of the original NOrec implementation with eager validation as proposed by us in [MT13].

4.4. Cloning

Allowing a helper thread to enter the same transactional section as needed in cloned synchronous and asynchronous execution (see Section 4.2), requires a mechanism to clone tasks. A cloned task has to be a duplicate of the original task and therefore requires the machine state and the contents of the stack to be the same.[2]

Every STM implementation requires the machine state of the current task to be saved when entering the transaction to perform rollbacks later. Another task could use this backup of the machine state to enter the same transaction as a clone. Hence, the remaining problem is the access to the stack. Preferably, access to stack contents which represents private data such as local variables etc. should be direct and without a detour via read/write functions of the STM library.

Modified local data has to be reset at rollback too and needs to be backed up somehow. For example the GCC compiler generates at each start of a transaction code to duplicate local variables of the current stack frame on the stack and uses the copies during execution (see

[2]For the sake of simplicity we ignore aspects such as the thread ID and thread-specific storage here.

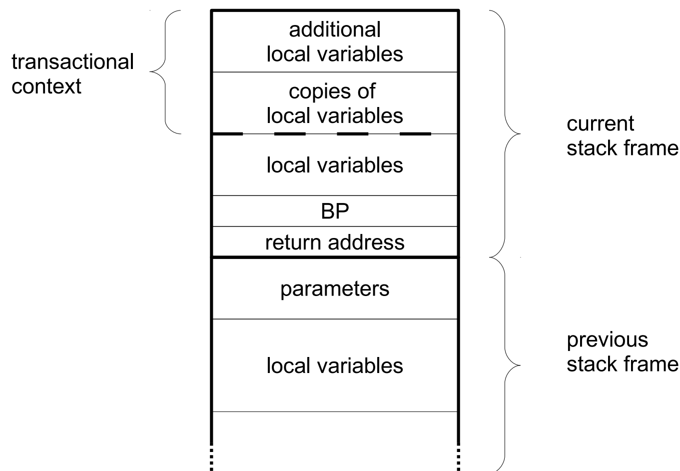

Figure 4.10.: Stack contents during execution of a transaction

Figure 4.10). Additionally, contents of the previous stack frame can be saved in a redo-log of the transaction on demand using the `log` functions of the TM ABI.

Generally it is possible to share variables located on a stack. For example a pointer to a local variable on the stack of the main function (process entry function of each C program) can be provided to multiple threads for shared access. Thus, not the whole stack is private, which implicitly forces the transaction to use transactional read and write access to the stack outside of the context of the transaction itself. The stack frames inside the transaction are guaranteed to be private: Those stack frames exist during the lifetime of the transaction only because they are left when the transaction is finished. Due to deferred update the transaction effectively does not publish data until it is committed, which implicitly prevents it from publishing local variables for shared use, too.

Considering the conditions given above, there are three approaches to clone a task for an STM system similar to NOrec:

No direct stack access: When redirecting every read and write into the read and write-sets of the transaction there will be no direct access to the stack (see Figure 4.11). Read and write-log will serve as a temporary replacement for the stack in this case. According to the calling conventions specified in the UNIX System V ABI, function calls will also require to store registers and parameters of the function on the stack. A realisation of this approach requires to prevent any direct access to the stack, which would be possible with a modification of the ABI only.

Stack copy at different location: The cloned task uses its own stack, which is located at a separate location (see Figure 4.12). Before entering the transaction it has to copy

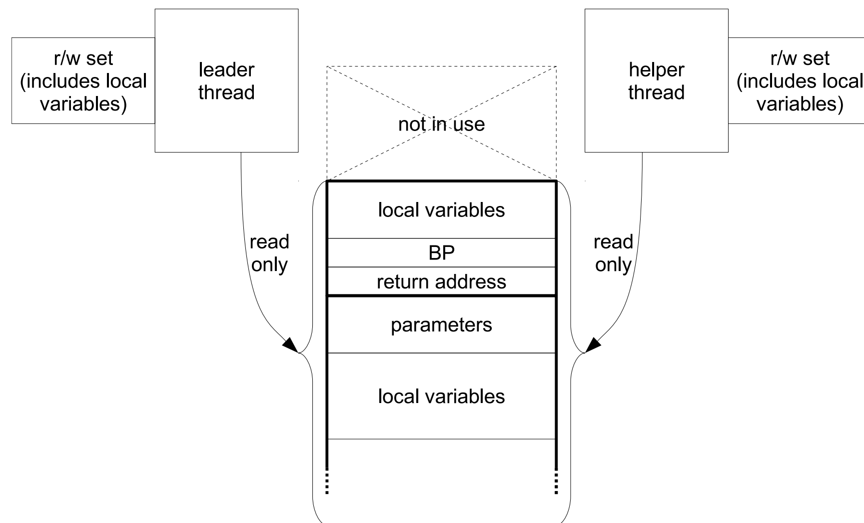

Figure 4.11.: Transaction without direct stack access

the part of the stack which is accessed directly such as local variables, return address and parameters to its own stack. When cloning the machine state, SP and BP registers have to be preserved to keep pointing to the stack of the clone. Access to local data via pointers have to be instrumented with calls to read and write functions of the STM as done by the GCC compiler. That way, access to the stack outside the copied section will be automatically redirected to the original stack and direct access is prevented through the deferred update. After the execution of the transaction the stack-frame may have been modified in content and size. Thus, the final result has to be copied back during commit, too.

Stack copy at the same virtual address: This is possible only if tasks are represented by separate processes with separate stacks on the same virtual address and shared data is located in shared memory segments (see Figure 4.13). Even though management of shared memory in the page tables is slightly different from shared use of the same page table with multiple threads, it does not generate significant overhead during access. A major issue is access to data on the stack of a different task. As mentioned earlier, data on stack may have been shared such as global variables from the context of the main function. If the accessed part of the stack is not available in the same process it requires an IPC mechanism to perform the operation such as shared memory again. The whole approach would end up having the same setup as the previous approach using copies at different locations.

Specifying a new ABI would allow more efficient cloning of tasks for transactions, such as

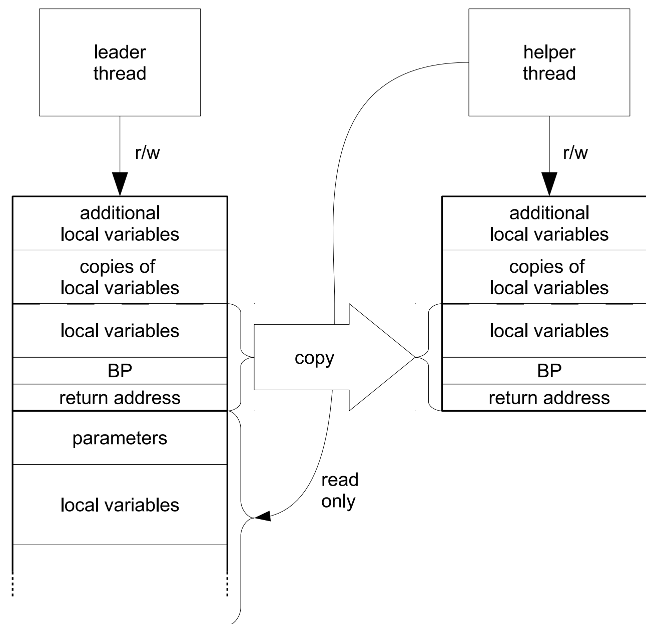

Figure 4.12.: Stack copy at different location

this: Every transaction start causes a switch to a new stack frame, the same way as a function call. The machine state is saved on the previous stack frame. All required local variables of the previous stack frame can be treated as parameters to the transaction and remain there. Local variables to be modified get copied to the new stack frame for temporary modification. To allow parallelisation in the upper stack part, which represents the context of the transaction, BP and SP could be used differently: The BP register is used to provide read access to the previous stack frames, e.g. to read the parameters, and the SP register is used to access local variables in the upper stack part inside the transactional section. That way, a task can share the previous stack frames with other tasks and it can use task-specific private stack frames for the local variables of the transaction referenced by the SP register. A commit requires just to copy back those modified local variables which are actually needed subsequently. This approach allows direct access to local variables without interference with a cloned task and it reduces the overhead at commit and rollback. But the development of a new ABI is out of the scope.

Because the sandboxing approach will benefit from the use of GCC too, the second approach will fit best.

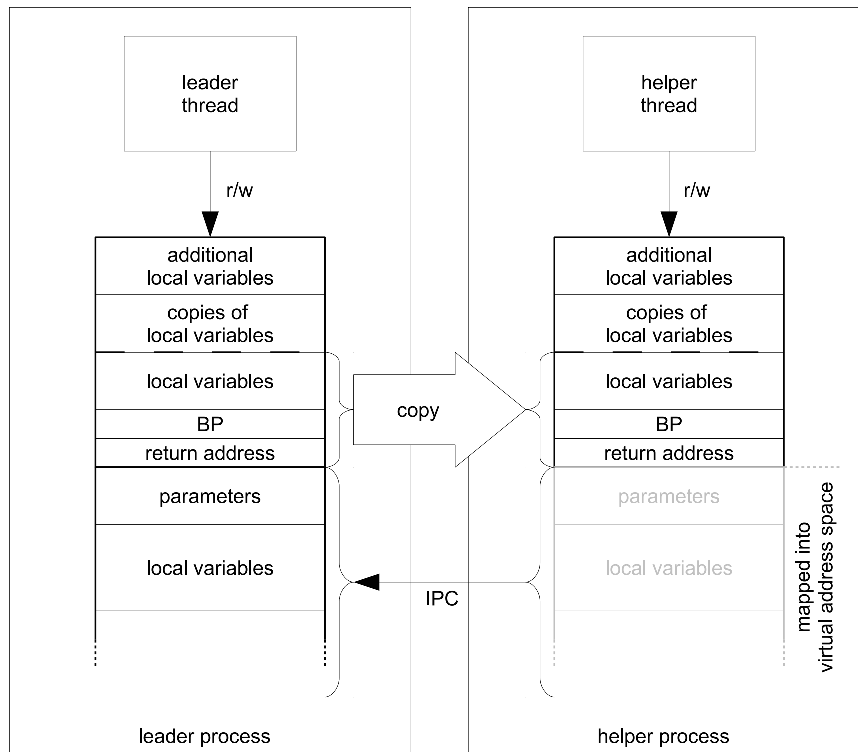

Figure 4.13.: Stack copy at the same virtual address

Chapter 5

Design

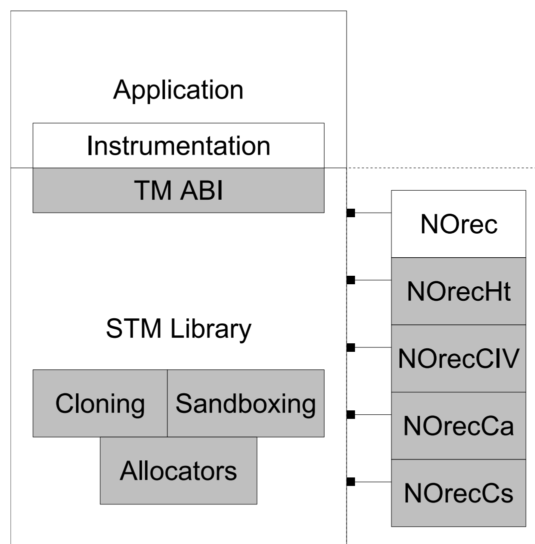

Figure 5.1.: Architecture overview

This chapter contains the system specification based on the design decisions made during the analysis in Chapter 4. Figure 5.1 provides an architecture overview of a concurrent application (upper part) using the STM library (lower part). The STM system is based on the framework developed in the University of Rochester, which contains the NOrec algorithm. All extensions to be made to the STM system are depicted in grey components.

In the analysis section different runtime models have been discussed to implement concurrent applications such as through multiple threads or processes. The runtime model supported by the STM system developed by the Rochester University is based on a multi-threading

concept following different threads of control in a single process and accessing memory of the same virtual address space to interact with each other after the shared memory model.

Each thread is associated with a thread-specific data structure called `TxThread`, which carries transaction processing related information such as the read and write logs, the transaction ID, a priority if required and so on.

The STM system provides a large variety of CC algorithms. The Figure 5.1 depicts just the `NOrec` algorithm and our modified versions.

- `NOrecHt` implements the variant with the validation delegated to the helper thread (cf. Section 4.2.1).
- `NOrecCIV` implements the cloned incremental validation approach (cf. Section 4.2.2).
- `NOrecCs` implements the cloned synchronous approach (cf. Section 4.2.3).
- `NOrecCa` implements the cloned asynchronous approach (cf. Section 4.2.4).

A CC algorithm can be selected at the start through an environment variable or on demand for example to fall back to the CGL algorithm to support irrevocable transactions. Each CC algorithm is implemented by a set of functions to start, abort or commit a transaction or perform reads and writes and so on. A `TxThread` data structure contains pointers to those functions which have to be changed when switching to a different algorithm.[1]

The STM system developed in the Rochester University supports different application interfaces such as the ABI of the Intel compiler and sets of C macros to attach different benchmarks. According to our design decisions in Section 4.4 we will add a support for the GCC ABI to be called by the instrumentation generated in transactional sections by the GCC compiler. The developer has to use the TM language extensions for C++ (see 3.4.1), which are supported by the GCC compiler. The ABI layer transforms calls to functions of the TM ABI (see Section 3.4.2) into function calls for the currently active CC algorithm via the `TxThread` data structure.

Further facilities added to the STM system implement the different features required by the CC algorithms:

- The `Cloning` facility supports creation and control of thread clones (cf. Section 4.4).
- The `Sandboxing` facility provides different levels of sandboxing (cf. Section 4.3.2).
- The `Allocators` facility provides different types of allocators (cf. Section 4.3.3).

The following sections will explain further details on all extensions and modifications to the original system.

[1] Function pointers are faster than abstract base classes because of the dynamic lookup of virtual methods.

5.1. TM ABI Layer

The TM ABI layer receives the calls from the instrumented transactional sections and transforms them into calls to the functions of the currently active CC algorithm of the current thread. The set of functions has been subdivided into functional groups that will be explained in the following paragraphs.

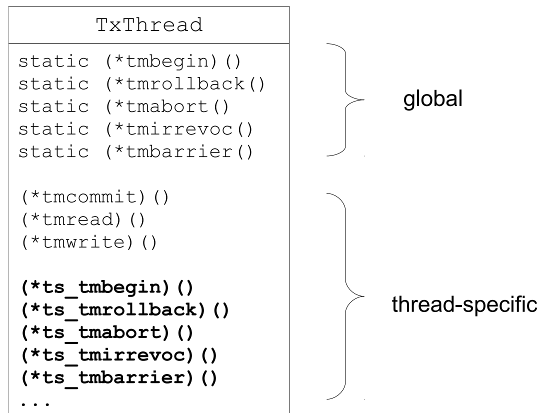

Figure 5.2.: Global and thread-specific function pointers in `TxThread`

Thread-Specific Concurrency Control The `TxThread` data structure (see Figure 5.2) contains global and thread-specific data and function pointers for the CC algorithm. The set of global function pointers (defined as static members of the data structure) has been extended by a set of thread-specific functions with the prefix `ts_` to support cloned execution of the same transaction with different CC.

Figure 5.2 shows the function pointers for the major CC functionalities:

- `tmbegin` is called on the start of a transaction.
- `tmrollback` is called by the `tmabort` implementation to rollback the transaction to its original state at start. It does not include the reset of machine state and the instruction pointer though.
- `tmabort` can be called from any location whenever there is a reason to restart the transaction. Unlike all other functions `tmabort` has to be implemented by the TM ABI layer to realise a proper restart based on the stored machine state.
- `tmirrevocable` is called if a switch to the irrevocable mode is required or requested by the application via the `_ITM_changeTransactionMode` function.

- tmbarrier is used to guarantee a valid state of the transaction at the location of the call to this function.
- tmcommit is called when the TM ABI layer detects a parent transaction to be finished, considering nested transactions.
- tmread is called on every instrumented read in the application.
- tmwrite is called on every instrumented write in the application.

Execution Control The execution of a transaction is controlled by three functions of the ABI: _ITM_beginTransaction, _ITM_commitTransaction for a regular finish and _ITM_abortTransaction for an application triggered abort.

On each start of a transaction the application calls the implementation of the function _ITM_beginTransaction in the TM ABI layer. The function saves the machine state of the current thread as a checkpoint for possible rollbacks determines the current nesting depth and forwards the call to the respective CC algorithm.

The checkpoint is saved in the stack frame of the caller and a reference is stored in the TxThread data structure. When a transaction requires to restart, it can use the machine state to reenter the _ITM_beginTransaction function. The checkpoint is also used to setup and control clones of the thread later.

The ABI layer supports flat nesting only due to the higher effort of checkpoint creation mentioned in Section 3.6.1 and 4.4. Thus, there are no partial rollbacks and no intermediate checkpoints to be supported.

The functions _ITM_commitTransaction and _ITM_abortTransaction are responsible to update the nesting depth accordingly and forward the call to the respective functions referenced by tmcommit and tmabort.

Data Access Control and Stack Protection The TM ABI layer implements all variations of the transactional read and write functions called by the instrumentation. While the STM system works on a 64 bit word-basis internally, the Intel TM ABI allows even byte-based access. This means that every data entry in read-set and write-set is stored in 64 bit words. Whenever a call to a read or write function arrives at the ABI layer, it creates a bit-mask according to the size and position of the referenced data element, which is used to limit access to exactly the region of the data element later. This functionality is also used to protect sensitive parts of the stack during access via pointers for sandboxing (see Section 4.3.2).

Undo Log The log functions of the TM ABI are used to save a backup of data located on the stack that has to be restored during a rollback. Because this functionality is not CC

specific it is implemented in the TM ABI layer. The logs have the same granularity as read and write-sets (64 bit).

Mapping to Transactional Functions Whenever a function call occurs inside a transactional section of the application the generated instrumentation performs a lookup of an instrumented counterpart of the function. The TM ABI is responsible for providing functions to register and lookup the code position of transactional functions associated to a function pointer of a non-transactional function. In case of sandboxing, the ABI Layer will also run a validation of the function pointer if the lookup has failed.

Memory Management Functions of the C/C++ Runtime Library The TM ABI layer overrides memory management functions such as `malloc`, `free` and the C++ specific variations called whenever a `new` or `delete` is used. These replacements mainly forward the allocation or reclamation request to the transactional allocator when called inside a transaction.

Functions to Copy, Move or Initialise Memory The runtime library provides a set of functions to copy (`memcpy`), move (`memcpy`) or initialise (`memset`) larger memory areas. The TM ABI layer implements replacements for these functions to support their use in transactions too. In case of a running transaction these functions just convert the access into calls to the `tmread` and `tmwrite` functions.

5.2. Sandboxing

The sandboxing facility provides several features for CC algorithms that need sandboxing.

Stack Protection It provides filtering of direct access to data in the sensitive parts of the stack. Direct access is permitted in the current stack frame starting above the position of the return address and the previous BP or the position of the SP at the start of the transaction and ends at the current SP. However, direct access results in an abort if addressing sensitive parts of the stack, i.e. BP or return address. If stack protection is enabled the ABI redirects access accordingly either to the functions `tmread` and `tmwrite` or performs direct access.

Signal Validation During the initialisation of the sandboxing facility a CC algorithm can request global or a thread-specific signal handling for the validation of signals and provides an appropriate validation function[2]. The signals to be validated are listed in Table 5.1.

[2]This slightly differs from POSIX thread signal handling, which considers thread-specific signal delivery but globally unique signal handlers.

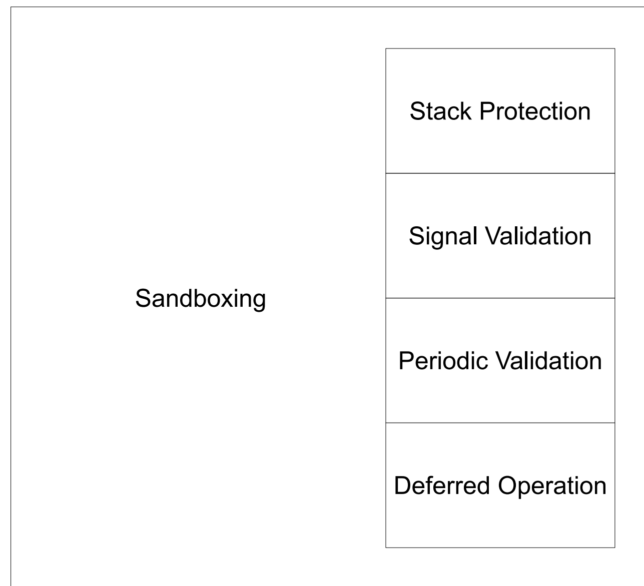

Figure 5.3.: Sandboxing function group overview

Signal	Reason	Default behaviour
SIGILL	illegal instruction	process exit
SIGABRT	immediate process abort	process exit
SIGBUS	memory access violation	process exit
SIGFPE	arithmetic exceptions	process exit
SIGSEGV	memory access violation	process exit

Table 5.1.: Signals validated for sandboxing

The sandboxing facility provides two standard global signal handlers, which use the validation function provided by the `TxThread` data structure to determine, whether the signal is the result of an application error or a doomed transaction. Based on the result the sandboxing facility either forwards the signal to the application or requests an abort of the transaction via the `tmabort` function. If the CC algorithm requires thread-specific signal handling (e.g. for helper threads) the sandboxing facility installs a global signal handler, which forwards the signal to the signal handler set in the `ts_process_signal` member of the `TxThread` data structure. If the thread-specific signal handler of the CC algorithm detects an application error, the signal is returned to the sandboxing facility to be properly forwarded to the application. The forwarding of signals to the application considers even application set signal handlers,

when registered through the function user_signal, which is a replicate of the POSIX signal function and disabled signals, which have to be ignored.

Periodic Validation A CC algorithm can request periodic validation support. The sandboxing facility installs a POSIX timer with the requested frequency and a signal handler for the SIGALRM signal send on each tick of the installed timer. The default signal handler uses the validation function provided by the CC algorithm to validate the running transaction and requests the abort in case of an inconsistency. This mechanism supports thread-specific signal handlers too.

Deferred Operations The asynchronous behaviour of periodic validation or aborts requested by a helper-thread can lead to inconsistencies of transaction internal data such as read- and write-set or wrong results of the periodic validation. Also, the support for memory allocation and reclamation inside transactions modifies data maintained by the C/C++ runtime library, which will be left inconsistent in case of an abort. To suppress periodic validation and abort in critical cases the sandboxing facility supports defering the execution of these operations to a non-critical point in time. This functionality is provided through flags in TxThread and appropriate macros to suspend and resume abort and validation and to register operations (validate or abort) to be executed with the next resume.

5.3. Cloning

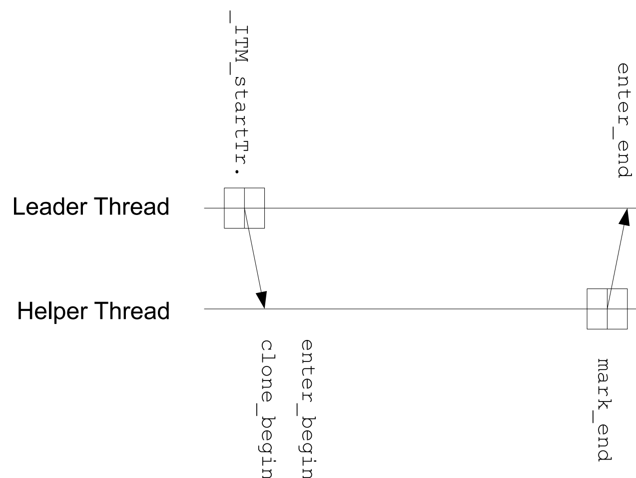

Figure 5.4.: Control transfer between leader and helper threads

CC implementations using cloning are supposed to apply the following concept: A leader thread starts the transaction with an arbitrary OCC algorithm based on deferred updates. The checkpoint created by the leader thread in the `_ITM_beginTransaction` function is considered to be unmodified until the leader leaves the transaction. Because of possible modifications any helper thread first has to create a clean copy of the checkpoint before it can enter it. The helper thread has to use its own separate stack and log all writes into the stack of the leader (e.g. via pointers in previous stack frames) according to the deferred update concept. When the helper successfully commits a transaction it can return control to the leader through another checkpoint created during commit. This end checkpoint has to contain the contents of the stack used by the transaction context in the helper thread, too. When the leader enters the checkpoint using the cloning facility, it also receives the modified content from the helper-thread. Modifications to the stack of the leader outside of the stack frames of the transaction will be transparently written to the correct location during the commit by the helper thread, because the references received during the start of the transaction (via registers, local variables etc.) still point to the original location.

The cloning facility provides functions to create and enter checkpoints. The copy of the machine state in a checkpoint has the same format as the checkpoint created during the start of a transaction in the `_ITM_beginTransaction` function. However, for the implementation of cloned transaction execution, a checkpoint for the transfer of the control from a committed transaction in the helper thread to the leader thread was needed, which also requires the modified content of the stack frame of the helper thread. The transfer of control is realised by such a checkpoint created during the commit of the helper thread and entered by the leader for the control transfer. To differentiate the cases, there are appropriate functions for each of them:

- `clone_begin`: Create a copy of the checkpoint of the start of the transaction.
- `enter_begin`: Enter the checkpoint at the begin of the transaction by a different thread (e.g. helper-thread).
- `mark_end`: Create a checkpoint during commit for the control transfer to another thread (e.g. leader thread).
- `enter_end`: Enter the checkpoint created during the commit of another thread of the same transaction. The new stack content is automatically transfered to the current stack frame of the calling thread.

To satisfy the first and second non-functional requirement given in Section 4.1 to avoid cache misses and inter-cache communication as far as possible, leader and helper threads will be usually placed on neighbouring cores which share most of the caches, i.e. cores of the same

CPU. This applies to all developed approaches with helper threads but NOrecCa. To keep the development effort low, the placement is hard coded and not dynamically decided based on the given architecture.

5.4. Allocators

Figure 5.5 depicts a class diagram for the required allocators. To support different allocators used by leader and helper thread of the same transaction an abstract base class Allocator was introduced and the TxThread data structure has a pointer to an allocator, which can be set thread-specific.

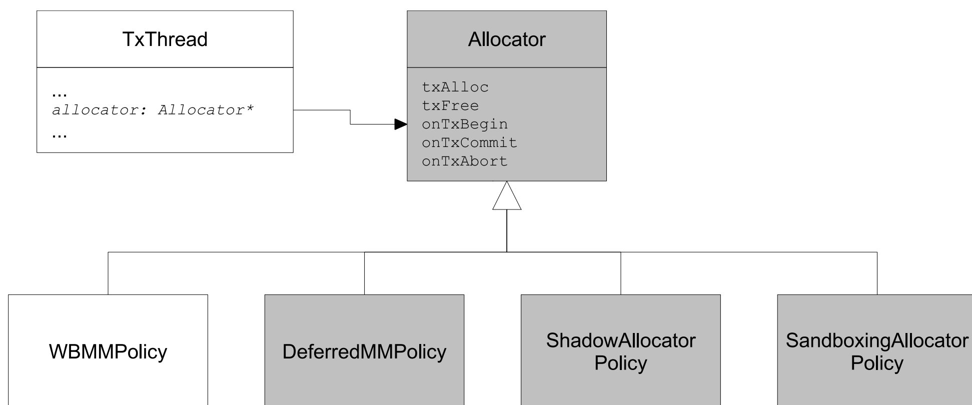

Figure 5.5.: Overview of the allocators facility

The abstract base class Allocator defines the standard interface of allocators, containing all methods available to the CC and the ABI layer[3].

- txAlloc is called by the ABI layer on a call to an allocation function such as malloc.
- txFree is called by the ABI layer on a call to a reclamation function such as free.
- onTxBegin is called by the CC to signal the begin of a transaction.
- onTxCommit is called by the CC to signal the commit of a transaction.
- onTxAbort is called by the CC to signal the abort of a transaction.

The different allocators are implemented in the subclasses of Allocator:

[3]Methods of allocators are less frequently called than read and write functions for example. Thus, we used proper OO here.

- `WBMMPolicy` implements the allocator which applies the epoch-based reclamation originally used by NOrec.
- `DeferredMMPolicy` implements an allocator which simply defers reclamations until the commit of the transaction. This allocator requires sandboxing for segmentation fault signals to maintain transparency during validation, as explained in Section 4.3.3.
- `ShadowAllocatorPolicy` is a variant of the `DeferredMMPolicy` allocator for cloned helper threads that execute the transaction without intention to commit such as in the cloned incremental validation approach (cf. Section 4.2). This allocator does not log or execute reclamations. It observes allocations to signal possible resource exhaustion to the CC as requested in Section 4.3.3. It also applies a lazy validation in case of an unsatisfied allocation request using the `tmbarrier` function set in `TxThread`.
- `SandboxingAllocatorPolicy` is another variant of the previous allocator called `ShadowAllocatorPolicy` for cloned leader and helper threads which fully execute and commit a transaction. Thus, this allocator now maintains allocations and reclamations and defers them until it commits.

The class hierarchy is kept flat to reduce the overhead during calls to the virtual methods of the base class.

The internal logs for allocation and reclamation requests contain pointers to the respective versions of the library functions for memory reclamation to be called for its actual execution during commit or abort (to revert allocations).

5.5. NOrecHt

The `NOrecHT` module implements the CC approach presented in Section 4.2.1. The leader thread implements a modified version of the NOrec CC with lazy validation and sandboxing and the helper thread provides periodic validation of the read-set. Both reside in separate modules (see Figure 5.6): `NOrecHT` contains the functions of the leader thread and `NOrecHT_sdw` the functions of the helper thread. Even if the helper thread will not execute transactional sections it uses the `TxThread` structure to benefit from the functionalities provided by the `Sandboxing` facility.

The leader thread uses full sandboxing support besides the periodic validation which is performed by the helper thread. Consequently it uses the `SandboxingAllocatorPolicy` of the allocators facility and implements a thread-specific signal handler. Periodic validation is suppressed during modification of internal data structures of the transaction such as read and write-sets.

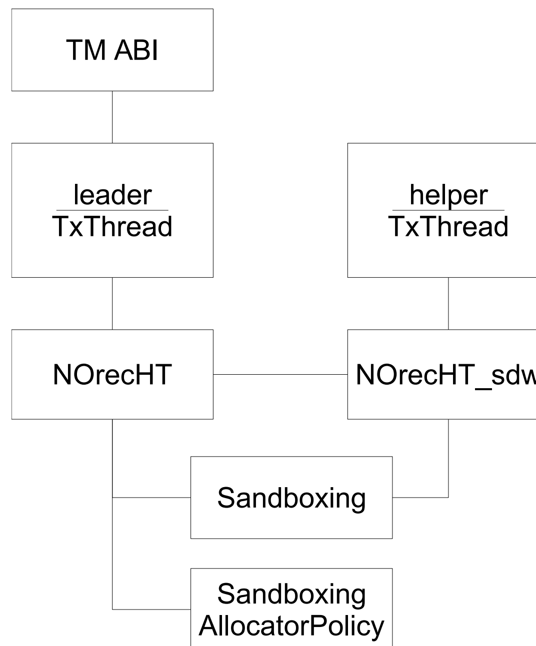

Figure 5.6.: NOrecHT component diagram

The helper thread is created when the first transaction is started. While the leader executes a transaction the helper thread repeatedly validates its read set and communicates inconsistencies and the time (in terms of the `sgsl`) the leader has been validated last. When the leader finishs a transaction, it resets its read-set and the helper thread consequently validates an empty read-set, which equals an idle loop. The helper thread is not suspended to prevent context switches.

Leader and helper thread use three shared variables to communicate the different states of the transaction:

- `sgsl` is used as a globally shared clock, which reflects the current state of shared data used in transactional sections, as known for NOrec.
- `ts_valid` indicates at which time (`sgsl`) the read-set has been proven to be valid by the helper thread.
- `ts_cache` has two purposes. It indicates at which time (`sgsl`) the leader performed its last read or when the read-set has been considered to be invalid by the helper thread last time. The latter case is indicated if the least significant bit is set too.

The control variables `ts_valid` and `ts_cache` are shared between the leader and the helper thread only. The leader thread uses these information to streamline its validation to

be performed at commit time. If `sgsl` equals `ts_valid` the leader thread can determine that its current state is valid and proceeds with the commit. If `ts_cache` equals `sgsl` and the least significant bit is set, the leader thread concludes that its current state is invalid and performs an abort. If none of the above cases is true, the leader has to perform a full validation itself.

The concurrent access to the read-set of the leader thread can cause several race conditions to be handled:

- Read-set inconsistency: If the leader thread resizes its read-set due to a lack of space the helper thread may read inconsistent data or receive a segmentation fault signal. To prevent the first case, the helper thread double-checks the state of the read-set before it considers its validation result to be true. To recover from possible segmentation faults it creates a checkpoint at the start of its validation loop and installs a thread-specific signal handler via the sandboxing facility, which instantly enters the checkpoint on segmentation faults. The read-set grows exponentially by the power of two, which is a common memory management strategy. The read-set keeps its size for the lifetime of the process because it is assumed that certain critical sections will be executed multiple times. Thus, read-set inconsistencies occur with the first execution of some critical section only.

- Update latency: Another case is that the helper thread works on a state of the read-set which does not yet reflect the state of the read-set seen by the leader. Thus, the read-set may actually contain new entries which are not yet visible to the helper thread due to the latency and reordering on hardware layer. To cover this case, the leader updates the `ts_cache` with a copy of the `sgsl` each time it performs a read. By comparing the `sgsl` to the copy of the leader thread, the helper thread can determine if the read-set reflects the state seen by the leader thread at the given state of the `sgsl` it is currently validating. If those values differ, it will wait for the next update and validate the remaining reads.

The mechanisms described above do not cover cases where the leader thread remains in an endless loop due to a doomed transaction. The `ts_cache` variable is checked by the leader after each read. The leader thread aborts immediately if the least significant bit is set. Otherwise it updates `ts_cache` with the current value of `sgsl`. But it cannot react if the endless loop does not contain transactional reads. To detect this case, the helper thread observes `ts_cache`, which has to be updated by the leader thread during a restart. If the copy is not updated in a time frame of one millisecond it considers the leader to be stuck and sends it a POSIX signal. The leader thread receives the signal through its thread-specific signal

handler, checks its own condition by a validation in respect of the control variables and aborts if necessary.

5.6. NOrecCIV

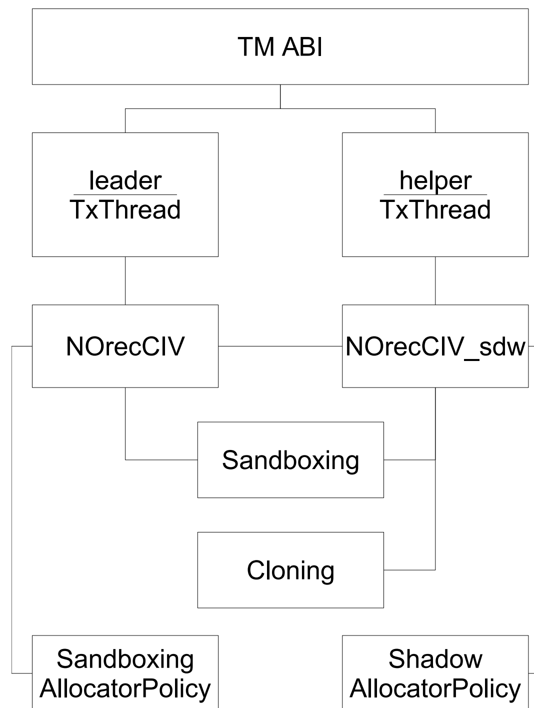

Figure 5.7.: NOrecCIV component diagram

The `NOrecCIV` module implements the CC approach presented in Section 4.2.2. The leader thread implements almost the same modified version of the NOrec CC as `NOrecHt`, applying lazy validation and the features of the sandboxing facility except of periodic validation. Periodic validation is delegated to the helper thread. It executes the same transaction with eager validation starting on the same consistent state to detect inconsistencies and signal them to the leader. Both reside in separate modules (see Figure 5.7): `NOrecCIV` contains the functions of the leader thread and `NOrecCIV_sdw` the functions of the helper thread. Both use their own `TxThread` structure to keep their transaction contexts separated and set their own CC functions and signal handlers. Because the helper thread does not attempt to actually commit transactions it uses the `ShadowAllocatorPolicy` (see Section 5.4) while the leader requires the `SandboxingAllocatorPolicy` again.

Leader and helper thread have to follow the same serialisation order on the same transactional section otherwise the validation results of the helper thread will not reflect the state of the leader too. This is achieved through two member variables of `TxThread`:

- `start_time`: The copy of the `sgls`, which reflects the current serialisation order. A difference in this variable indicates different serialisation orders of leader and helper thread and appropriate actions have to be taken to fix it.
- `order`: A counter for transactions which is incremented on transaction start and commit. A difference in this variable indicates if the leader executes a different transactional section. Additionally, the least significant bit implicitly indicates whether the leader thread currently executes a transaction or not.

The helper thread executes an idle loop while the leader thread does not execute a transaction. In this case the helper observes the `order` variable of the leader to detect the entry of a new transaction. When the leader entered a new transaction, the helper copies the `order` variable of the leader and tries to get a clean copy of its machine state using the cloning facility. A change in the `order` variable indicates an inconsistent copy because the machine state may have been overridden with a new machine state, while cloning it. In this case the helper restarts its attempt to clone the transaction start. On success it enters the cloned state and executes the same transaction in parallel and performs eager validation on every read.

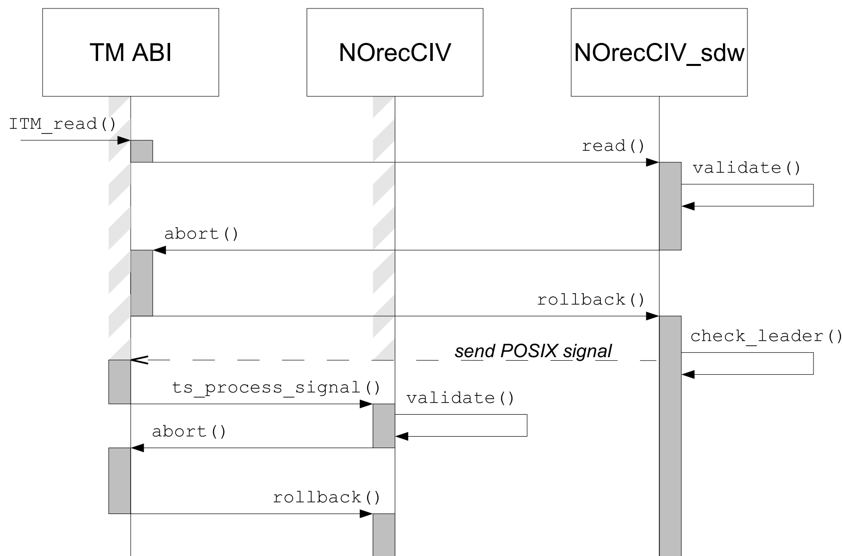

Figure 5.8.: NOrecCIV sequence diagram: abort case

Figure 5.8 depicts a UML sequence diagram of an abort case detected by the helper thread. The grey boxes depict thread activity. Grey boxes to the left of the life line of a component

indicate activity of the leader thread in this component and grey boxes to the right indicate activity of the helper thread. Unknown or arbitrary activity of the respective thread is shown by the boxes in light grey with white stripes. When the helper thread detects a conflict through validation of its own state during read it performs a rollback. Part of the rollback is to check the state of the leader by comparison of the `order` and `start_time` member variables. If the leader still executes the same transaction on the same serialisation order it sends a POSIX signal to report the inconsistency to the leader. This way, the leader cannot get stuck as a doomed transaction in endless loops and the time to detection is reduced.

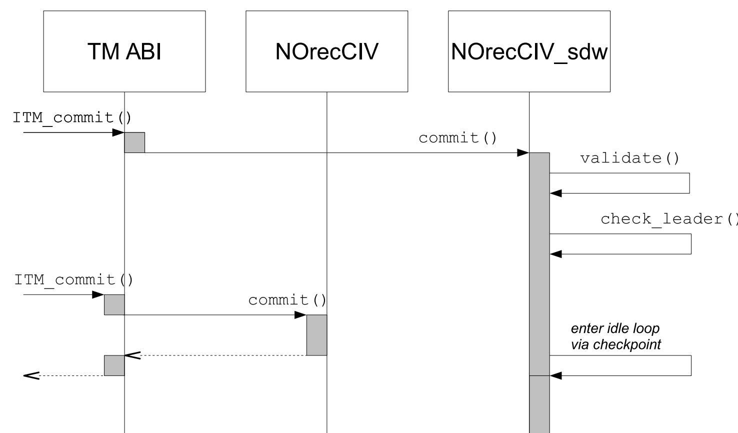

Figure 5.9.: NOrecCIV sequence diagram: leader commits before helper

In case of a non-conflicting transaction the leader will commit on its own, performing a usual validation of its own read-set. The helper thread will be unaware of it until it reaches the end of the transaction itself, but it will not have serious side effects because the helper performs eager validation.

In rare cases the helper thread might finish the transaction earlier than the leader. The sequence diagram in Figure 5.9 shows this case: After successful commit the helper enters `check_leader`, a loop where it keeps validating the consistency of the read set until the leader has finished. If an inconsistency occurs the helper thread will again report it to the leader and perform an abort.

After each successfully finished transaction the helper thread uses a checkpoint to immediately jump to the idle loop and not proceed through the control flow of the application code following the transactional section.

5.7. NOrecCs

The `NOrecCs` module implements the approach for cloned synchronous parallel execution of a transaction drafted in Section 4.2.3. The function sets of leader and helper thread are implemented in two separate components called `NOrecCs` and `NOrecCs_sdw` (see Figure 5.10). Both threads execute a single transaction in parallel using different CCs: The leader thread implements lazy validation and deferred update while the helper thread uses eager validation but deferred update too. Leader and helper thread utilise features of the sandboxing facility. The leader requires full signal validation but no periodic validation support, which will be performed by the helper thread. The helper thread needs just limited sandboxing support to deal with segmentation faults occurring during eager validation of concurrently deleted memory areas as described in Section 4.3.3. Both use the functions of the cloning facility to realise cloning of the checkpoint at transaction start and control transfer for a transaction committed by the helper thread according to the concept presented in Section 5.3.

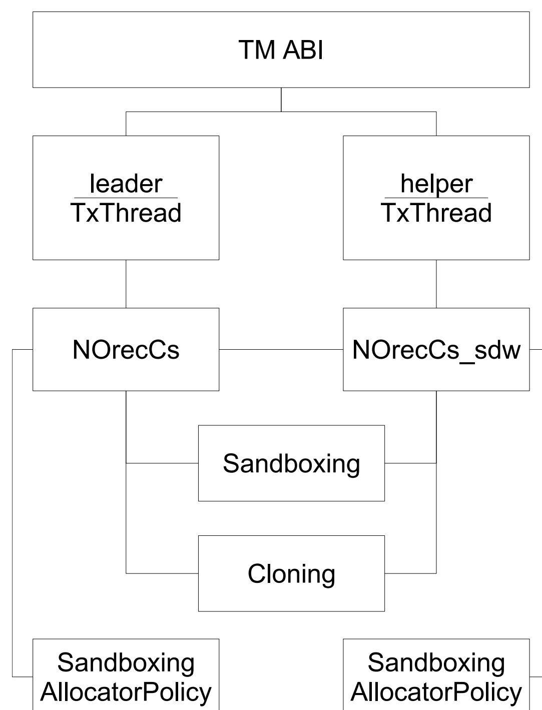

Figure 5.10.: NOrecCs component diagram

Synchronisation of the parallel transaction runs of leader and helper thread is achieved through the same member variables in `TxThread` as in the `NOrecCIV` module: `order` serves as a transaction counter and indicates whether a thread is in a transaction or not and `start_time` indicates on which state of the shared data the thread is currently working on.

order is updated on each transaction start and end. start_time gets sampled from sgsl at start and updated after each successful validation. If both variables have the same value in the TxThread instances of both threads the parallel execution is in synchronisation.

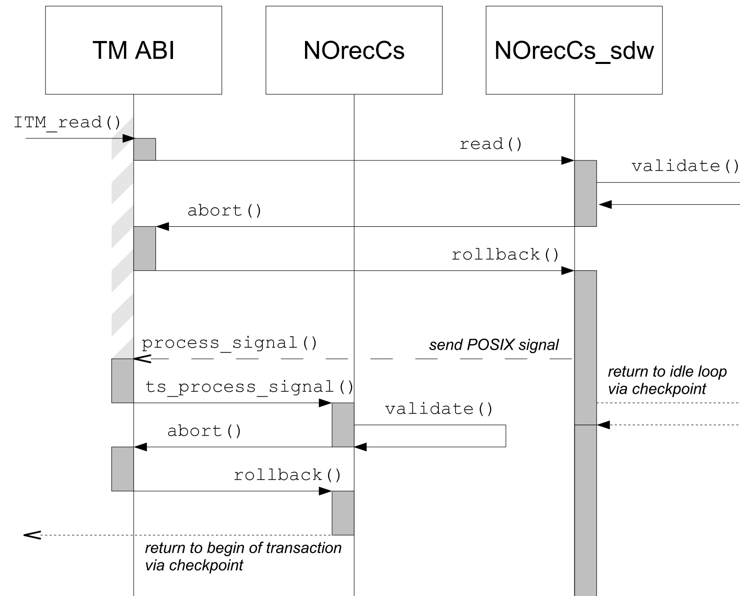

Figure 5.11.: NOrecCs sequence diagram: Helper aborts

The helper thread starts in an idle loop, observing the order variable of the leader. If the leader enters a transaction it tries to get a consistent copy of the start checkpoint by double checking the order variable after copying. During execution of the transaction it logs reads and writes and applies the usual eager validation during reads. If it detects an inconsistency it checks the state of the leaders order variable. Because the leader does lazy validation it is possible that it has already finished the run of the transaction. In this case the helper thread will return to its idle loop. Otherwise it assumes that the leader is facing the same inconsistency and reports it to him through a POSIX signal (cf. Figure 5.11). The leader receives the signal and performs a validation according to the periodic validation scheme. The leader might have done a restart already, which will be detected by a comparison of sgsl and start_time and it bypasses the validation, otherwise it will perform the abort and restart on the new state. The helper thread will perform an ordinary abort anyway and start a new run for the transaction to synchronise with the leader again.

Because commits are mutually exclusive, either the leader or the helper thread will commit the transaction first. The case where the leader commits first was explained above. If the helper thread commits first (cf. Figure 5.12), which is possible if it was able to fix the read inconsistency, it will create an end checkpoint using the cloning facility. It will also increment

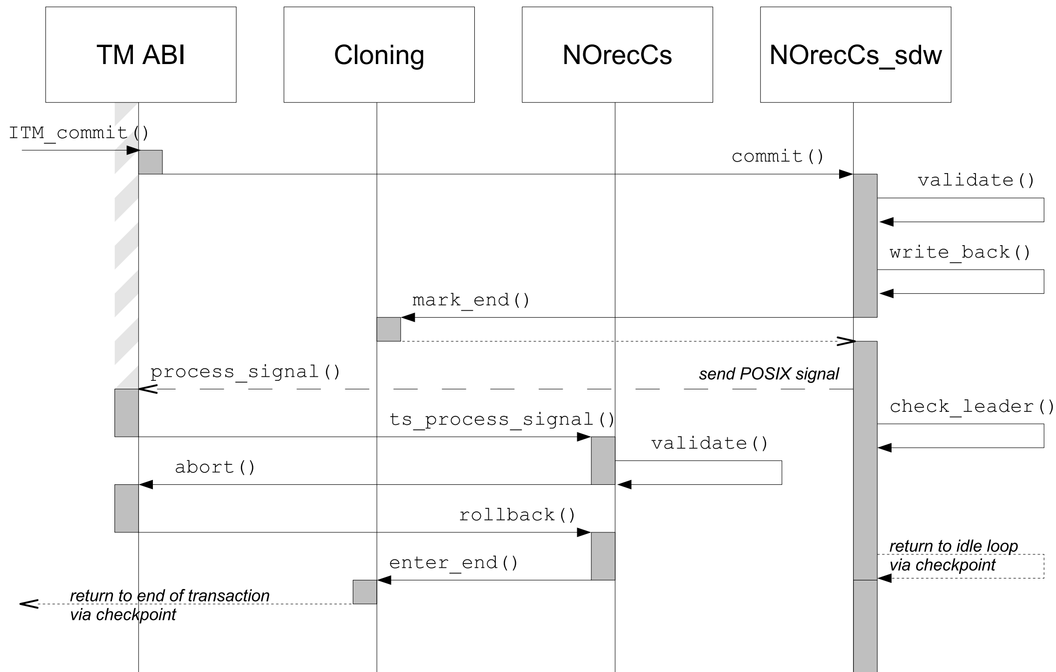

Figure 5.12.: NOrecCs sequence diagram: Helper commits before leader

sgsl and its order variable, which implicitly indicates its commit to the leader thread. If the leader tries to commit concurrently it will be forced to wait due to the mutual exclusion pattern. When the leader finally gets access to commit its transaction, it will perform its final validation. During the validation it also compares the order variables of leader and helper and thereby detects the commit of the helper thread. Consequently, it uses the cloning facility to enter the end checkpoint created by the helper, to get into the control path after the transactional section. After the helper thread has committed its run it will also send a POSIX signal to force the leader to validate its state again. This prevents the leader form getting stuck in endless loops and reduces the time to commit for the leader.

The leader has to perform a rollback in two cases: In case of a regular abort and if the helper has performed a commit. In the latter case the rollback is required to reset the state of the TxThread instance e.g. to clear the logs such as read-set and write-set (cf. Figure 5.12). Therefore, both cases are handled via the abort path. The validation of the leader is modified to report an inconsistency if the helper has already committed. In most cases the commit of the helper will cause an inconsistency of the leader anyways, because both work on exactly the same shared data. During rollback the leader checks whether a commit of the helper thread is available and enters it accordingly.

5.8. NOrecCa

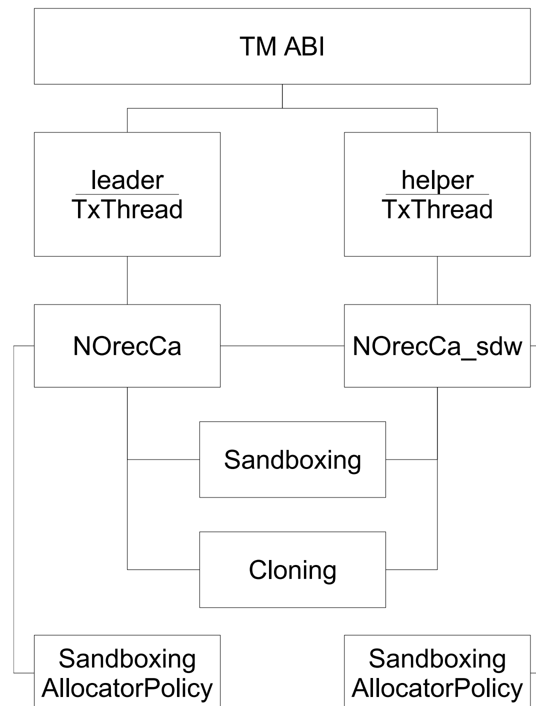

Figure 5.13.: NOrecCa component diagram

The NOrecCa module implements the approach introduced in Section 4.2.4. Leader and helper thread execute the transaction in parallel but not synchronised using different CCs: The leader uses an OCC with deferred update and lazy validation and the helper thread uses an OCC with deferred update too but eager incremental validation. In contrast to the previous approaches the leader thread gets the full set of sandboxing features including the timer-triggered periodic validation which causes the asynchronism. The helper thread is no longer notifying the leader about aborts (see Figure 5.14) and both will drift apart. The helper requires just limited sandboxing support to deal with segmentation faults of privatised and deleted memory areas. Both apply the functions of the cloning facility to realise cloning and control transfer in parallel execution of the transactions.

The design is almost the same as for the NOrecCs module (see Figure 5.13). The CC for leader and helper threads are implemented in separate components called NOrecCs and NOrecCs_sdw. Both have access to the sandboxing and cloning facilities and their functions are assigned to their respective representation of the TxThread data structure instance. Both need the StandbyAllocatorPolicy to properly support memory allocations and reclamations in transactions.

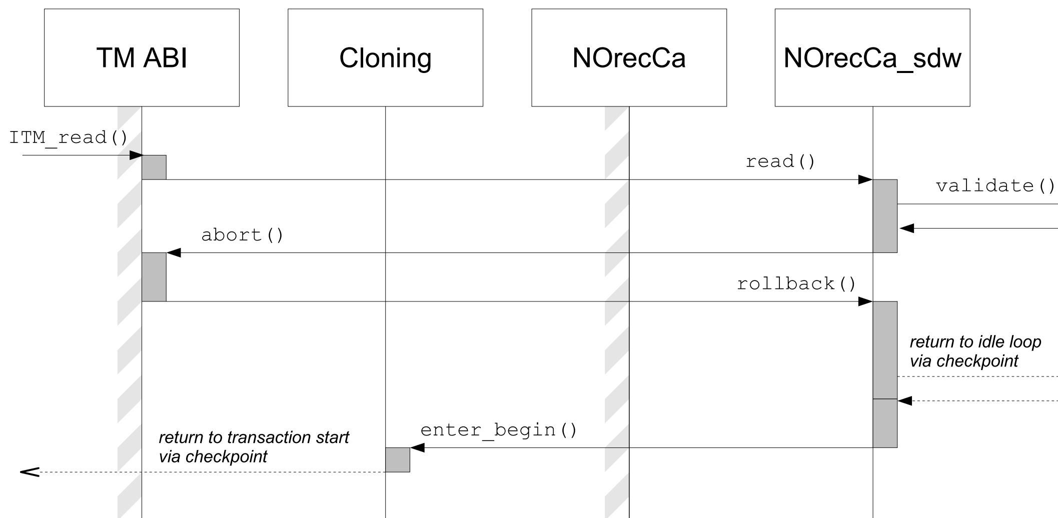

Figure 5.14.: NOrecCa sequence diagram: Helper aborts

Even if leader and helper thread do not need to execute the transaction on the same state of shared data, they have to execute the same transactional section at least. Thus, they still need to synchronise via the `order` variable, which is incremented on each transaction begin and end. This variable indicates whether a thread is inside a transaction or not by the least significant bit. If the `order` variables of leader and helper thread differ, it indicates that one of them is ahead of the other in terms of alternating sections of transactional and non-transactional code. A diff implicitly indicates a commit of one thread too, thus both react on it with an abort and rollback of their own activities to reset the internal data structures. The helper thread will enter its idle loop again to observe the leader again and enter the next transaction started. The leader thread will enter the end checkpoint of the committed transaction prepared by the helper thread.

Leader and helper thread act almost the same way as in the `NOrecCa` module. The helper starts in an idle loop, observing the leaders `order` value to detect the start of a transaction. On start of a transaction it obtains a clean copy of the machine state (the start checkpoint) by double checking the `order` variable and enters it. During validation both threads compare their `order` variable with the `order` variable of its sibling to detect whether one of them has already committed. As commits are mutually exclusive there will be only one commit at a time. Because every commit performs a validation as its first step, both threads will detect a commit of their counterpart guaranteed, before they try to write back their updates and switch to their respective abort path as described above. The helper thread will additionally send a POSIX signal after it has committed to reduce the time to finish the transaction in the leader thread (see Figure 5.15).

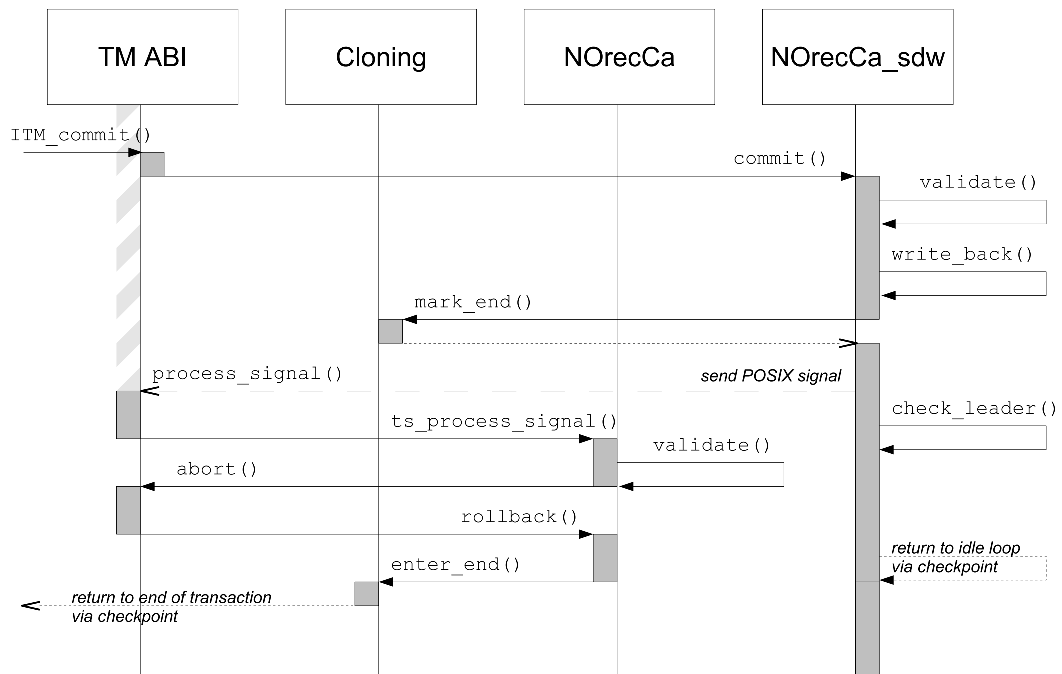

Figure 5.15.: NOrecCa sequence diagram: Helper commits

Due to the asynchronous behaviour, the `NOrecCa` approach is expected to have a disadvantage if leader and helper thread share the same caches. When the threads drift apart, the intersection of their memory accesses is expected to get smaller, which results in an increase of required cache space. Consequently, the cache miss rate should rise. Thus, leader and helper threads will be rather placed on cores that do not share the lower level caches, but the higher level caches. On the targeted AMD Opteron architecture those will be cores on the same CPU but with separate level 2 caches.

Chapter 6

Evaluation

To study the effects of parallelised transaction execution the response time of all CC algorithms was evaluated against the STAMP benchmarks [MCKO08]. This is a common method applied by most researchers in the field. To allow an interpretation of the results the code of the benchmarks has been reviewed to provide more detail in respect to their degree of concurrency and use of transactions. As the following evaluation will show, this knowledge is still not enough to reliably explain certain characteristics of the CCs analysed.

It is a known general issue in the evaluation of highly concurrent applications that their behaviour is very sensitive in regard to performance instrumentations. The inherent problem is that a meaningful instrumentation generates too much interference with the system under test and the altered application behaviour does no longer reflect its actual performance. Experiments with a source code instrumentation of the CCs did show exactly this issue in our case, too. The most significant influence induced by the instrumentation was the retrieval of timestamps for observed events. Actually, the performance countes of today's x86 CPUs provide adequate granularity, but they seem to scale disadvantageously in parallel access. Also, instrumentation of the STM increases effort spent in transactions. Even though the conflict potential of concurrent transaction remain the same, the time ratio between transactional and non-transactional code significantly changes. This in turn effects the conflict probability and the measured statistics no longer reflect the addressed use case, which means that they are consequently wrong. Furthermore, the per thread cache miss counters were inaccurate, which could have been a meaningful source of information, too. We came to the conclusion that the current implementation of the performance counter management supported by the Linux kernel has still some compatibility issues with our test setup.

Unfortunately, secondary measurements such as the overall abort rate, do not provide any further information. For example, a higher abort rate can be positive and negative at the same time. The abort frequency does naturally rise if many concurrent transactions access the same data. The high amount of concurrent accesses generates more cache misses too. A transaction which aborts more frequently and thereby earlier, consequently reduces the amount of memory locations accessed by the transactions and the amount of unnecessary cache misses at the same time. On the other hand the higher frequency of aborts also increases the frequency of accesses to the same memory locations, e.g. the `sgsl` in this case. Consequently, the transactions will more frequently try to access the same memory location simultaneously, which increases the pressure on the cache line resulting in even higher access latency, as we will show later in the next chapter, too. Hence, the only reliable way to explain observed performance characteristics is to analyse the time spent in certain phases of the transaction such as abort, commit, validation, read/write access and so on. As explained above, the test environment lacks instruments to achieve this level of detail and the evaluation has to rely on measured response times of the benchmarks to the most part. To provide an explanation of general characteristics observed with the benchmarks, the following chapter investigates the hardware capabilities in respect to the read/write access latency in concurrent applications instead.

This chapter will start with the detailed description of the test environment including software and hardware. It is followed by the detailed description of the STAMP benchmarks. In the last section the evaluation method and the results will be presented and discussed.

6.1. Build and Test Environment

This section describes hardware and software of the test platform.

The hardware provides four AMD Opteron™6282 SE processors at 3GHz clock rate with 8 cores each and 128 GB DDR3 1600 RAM main memory. Cores are bundled in so-called Bulldozer™modules with 2 cores each, which share certain components such as instruction decoder, instruction fetch unit, floating point unit, level 2 and 3 caches (see below) and more. We consider the two threads of the two cores of a Bulldozer module to be more separated as threads in hyper threading but less than threads of independent cores.

The architecture has a 3 level cache hierarchy with the following capacities:

L1 Cache 48 KB per core

L2 Cache: 2 MB per 2 cores (2 MB per bulldozer module)

L3 Cache: 16 MB per processor

The system runs a Debian x86_64 GNU/Linux operating system with Kernel version 3.2.57-3. Benchmarks and the STM system have been built with the GNU C++ compiler version 4.7.2 applying optimisation at level 3 (compiler flag -O3).

6.2. Benchmarks

The STAMP (Stanford Transactional Applications for Multi-Processing) benchmark suite has been especially developed for the evaluation of transactional memory implementations and is commonly accept by researchers of the domain. The CC algorithms for parallel execution of transactions developed in this thesis require proper instrumentation of transactional sections and just limited use of so-called pure functions or waivered sections (cf. Section 4.3.2). Based on the modification effort and the variety of application behaviour to be tested a few benchmarks have not been adapted for the evaluation. The following sections provide a detailed description of the selected benchmarks to allow a proper interpretation of the measurement results.

6.2.1. Genome

The genome benchmark implements the analysis of a genome to determine its sequence of nucleotides. Because it is not possible to just read the atomic structure of a genome molecule, several alternative methods have been proposed. The first one was the Sanger method, which is just applicable for short DNA sequences. The method implemented by the benchmark is called the shotgun method, which allows decoding of larger DNA sequences based on the Sanger method using the following procedure: It starts with cloning of a given DNA multiple times until a huge set of clones is available. Those clones will be cut into pieces at random positions in their sequence. The shorter subsequences will be decoded by the Sanger method. By the identification of overlapping ends of subsequences the shotgun method then reconstructs the DNA sequence of the original genome by combining the fragments. The latter task is simulated by the benchmark.

During initialisation the genome benchmark creates the input to the sequencing algorithm, which is a large set of decoded subsequences (fragments) of the clones of the given genome. Fragments are represented by byte sequences with a maximum length of n elements. The set of fragments is partitioned into equally large subsets and each subset is provided to a separate thread for parallel processing. The task to be fulfilled by the threads for each fragment in its particular subset consists of the following three phases:

1. Deletion of exact duplicates of fragments by adding them to a globally shared table with unique entries.

2. Participation in the creation of globally shared lookup tables for starts and ends of fragments using either the first or last $[1..n]$ elements of a fragment as its key.

3. Combining of fragments by the following steps for each of the considered lengths of fragment starts:

   a) Identification of continuous fragments by the lookup of matching ends for given starts.

   b) Creation of new entries in globally shared lookup tables for a virtually composite fragment. The entries will just contain instructions for the composition of both fragments.

All threads get synchronised at a barrier after each phase. Applied transactions in this benchmark are rather short and cover just single accesses to any of the globally shared lookup tables in all phases. The most complex transaction is the insertion of additional composition instructions, which combines lookup of the entry and extension of the linked list of instructions.

At the end, the main thread uses the gathered information to assemble the original sequence.

6.2.2. Intruder

The intruder benchmark implements an intrusion detection system, which analyses incoming messages based on a set of known signatures of malicious messages. The system fulfils two tasks: In a decoder step the system combines message fragments of the input stream to actual messages. The input stream resembles the input of multiple applications (ports) such as on TCP level in the network stack of an operating system. In a subsequent detection step the system compares the messages to the known signatures to detect attack patterns. The task is delegated to multiple threads in the following manner:

- **Decode:** All threads participate in this step concurrently. Each thread receives single message fragments and adds them to a shared list of received fragments of the same message. The fragment lists are maintained in a lookup table.

- **Detect:** This step comprises the conversion of messages into an internal representation and the comparison to the known malicious signatures. Each thread takes one of the combined messages to apply this procedure to it and counts the detected attacks.

Each thread repeats these two steps until all messages have been analysed.

The benchmark processes a fixed amount of message fragments with configurable amount of threads. All threads run the same code except of the initialisation and shutdown performed by the main thread. Transactional code is limited to the decode step. Messages are inserted in

a queue when fully decoded and each thread removes one available message from the queue in each iteration for detection. The detection just uses read only access to the static set of known message signatures. Consequently, there is no concurrency control applied during the detection step.

6.2.3. Kmeans

The kmeans benchmark implements the K-Means clustering method. It provides clustering of a set of vectors into clusters of vectors which are similar to each other according to a given clustering criterion. The method tries to find the best matching amount of clusters considering the similarity of assigned vectors in the resulting clusters.

The clustering criterion applied by the benchmark is the euclidian distance of a vector to the centre of a cluster of vectors. The centre of the cluster is defined as the arithmetic mean of all vectors of the cluster. The average distance of all vectors to their assigned cluster centre represents the quality of a given clustering result. Starting at a random distribution of vectors to a given amount of clusters the algorithm has to perform the following steps:

- After the initial assignment of vectors to clusters the cluster centres have to be calculated.
- Subsequently, the initial clusters get optimised by repeating the following procedure on each vector:
  - First the distance of the given vector to each cluster centre is compared to find the closest cluster.
  - If a closer cluster centre was found the vector is removed from its current cluster and inserted in the closer cluster.
  - For each modified cluster, the previous and the new one, the cluster centres have to be recalculated.

Obviously, the algorithm itself is not guaranteed to terminate. Hence, the maximum number of iterations is predefined. The benchmark repeats the clustering algorithm for various amounts of clusters and determines the best amount of clusters based on the average distance of all vectors to their respective cluster centres.

The parallelism is applied to the lowest level. Threads concurrently grab the next vector from a shared list of not yet analysed vectors and apply the above steps to it. This procedure is repeated until all vectors have been processed and one pass of the algorithm for a given amount of clusters has been finished. Each pass of the algorithm with a different amount of clusters requires a preliminary synchronisation of all threads at a barrier.

Use of transactions is limited to the modifications of data structures again: The retrieval of a vector to be processed next, the assignment of the new cluster centres after recalculation and the update of the global quality measure for every intermediate clustering result after the analysis of each vector. Because a thread has exclusive access to the vector it is processing the change of membership does not require concurrency control. But membership is maintained in an array for all vectors and each modification will cause contention in a cache line, which contains membership information for multiple other vectors too. Threads grab vectors almost sequentially alternating and thus a modification of the membership in one thread will most probably cause a cache miss in another thread too, although there is no actual conflict. A similar behaviour is to be expected for the method which updates the centres. Even though a movement of a vector requires just two centres to be altered every thread updates all centres at once, overwriting most of the centres with the same value it already has. Whether this will cause cache misses too, depends on the hardware.

6.2.4. Labyrinth

The labyrinth benchmark implements Lee's algorithm [Lee61] to find paths through a three dimensional maze. Multiple threads apply this algorithm concurrently to find multiple paths with different start and end points. The maze is a three dimensional shared grid data structure used concurrently by all threads to find paths and annotate the grid nodes accordingly. The procedure to find a single path runs entirely in a transaction. Each thread repeatedly grabs a start and end and applies the algorithm until all nodes at the borders of the grid have been processed. To reduce the contention the procedure starts by copying the current state of the maze. The algorithm is applied to the copy, and the path found is validated against the original maze at the end. If the validation fails, the transaction is aborted on application level, otherwise it writes back the new path. This procedure has a disadvantageous impact on TM. Because every data of the grid is read at the start, any modification will cause a conflict in all concurrent transactions, even though the tasks may not actually need the data.

6.2.5. SSCA2

The SSCA2 (Scalable Synthetic Compact Applications 2) benchmark provides a set of algorithms which operate on a large shared directed, weighted multi-graph.[1] The graph is represented in an adjacency array and associated auxiliary arrays. During an initialisation phase the main thread creates a set of nodes and assigns them to cliques.[2] The SSCA2 algorithm used here for benchmarking implements a parallelised reconstruction of the adjacency

[1] A multi-graph allows the existence of multiple edges between two nodes.

[2] Each node of a clique has at least one edge to another node of the clique.

array based on the clique membership of nodes. The set of nodes is then partitioned in disjoint subsets. Each thread gets one subset of the nodes and adds the nodes based on its clique membership to the shared adjacency array. Access to the adjacency array is protected by transactions. Because threads are likely to work on different parts of the graph, the contention probability is low in comparison to other benchmarks.

6.2.6. Vacation

The vacation benchmark emulates a vacation planning system, which allows acquiring of various resources such as an apartment a flight and so on. All reservation actions for one client are covered in a large transaction and each thread processes a set of reservations for its own set of clients. Thus, contentions occur on resources and the duration of a transaction is long in comparison. The amount of resources is rather limited compared to other benchmarks. Thus, the probability of conflicts is high, but compared to the labyrinth benchmark a transaction will more likely have some non-conflicting resources too.

6.2.7. Transactional Characteristics

Benchmark	Tx Length	R/W Set	Tx Time	Contention
genome	Medium	Large	High	Low
intruder	Short	Medium	Medium	High
kmeans	Short	Medium	Low	Low
labyrinth	Long	Large	High	High
ssca2	Short	Small	Low	Low
vacation	Medium	Large	High	Low/Medium

Table 6.1.: Characteristics of the STAMP benchmarks

Minh et al. [MCKO08] provided a table (see Table 6.1) which contains rough classifications of several characteristics of the benchmarks related to their transactional behaviour. The qualitative measures given for a benchmark in the table are considered in comparison to all other benchmarks. A modification of the benchmark parameters, for example the CC algorithm or the number of threads to be applied, have of course significant influence on the runtime properties of the benchmark such as the estimated contention.

The runtime properties given in the table have the following meaning:

- **Tx Length:** This property provides a rough estimation of the amount of instructions to be processed inside a transactional section on application level.

- **R/W Set:** This property gives an estimation of the average length of the read and write set of a transaction, which roughly reflects the amount of read and write operations.
- **Tx Time:** This property gives an estimation of the percentage of execution time spent in transactions.
- **Contention:** This property reflects the probability of conflicts between threads derived from the average number of aborts per commit.

Due to the modifications made to the instrumentation of the benchmarks the properties slightly changed and **R/W Set** and **Tx Time** have been reevaluated. The now updated classification is based on the ranges given in Table 6.2.

Property	Short/Low	Medium	Long/High
R/W Set [*#items*]	$[0 - 10]$	$[10 - 100]$	$[100 - 1000]$
Tx Time [%]	[0-35]	[35-75]	[75-100]

Table 6.2.: Characteristics of the STAMP benchmarks

6.3. Results

The evaluation focuses on the response time of transactions, which is reflected in the overall execution time of a benchmark. The graphs depicted in Figures 6.2 to 6.8 contain the average response times (total execution time for a single benchmark run) in seconds (vertical axis) with sets of $1 - 32$ application threads (horizontal axis). Helper threads are not considered as application threads in this regard. Measurement runs have been repeated multiple times for each test configuration until the size of the 90% confidence interval for the calculated average fell below 5% of the standard deviation.

The following STM implementations have been considered in the measurements:

- `CGL`, a single global lock STM available in the RSTM framework.
- `NOrec`, the original implementation of the NOrec STM.
- `NOrecSb`, a NOrec STM with sandboxing and lazy validation using timer-driven periodic validation in the same thread.
- `NOrecHt`, the modified NOrec implementation with sandboxing and delegated periodic validation in a helper thread described in Section 5.5.

- NOrecCIV, the modified NOrec implementation with delegated incremental validation in the helper thread according to Section 5.6.
- NOrecCs, the modified NOrec implementation with parallel synchrounous execution of transactions described in Section 5.7.
- NOrecCa, the modified NOrec implementation with parallel asynchrounous execution of transactions described in Section 5.8.

The CGL STM has been added to the set of tested CCs to allow an assessment of the relative gain or loss in response time of certain STMs based on the total difference to concurrency with mutual exclusion. Besides, the results will show a significant advantage of CGL over STM implementations especially in benchmarks with longer transactions and high contention probability. Reasons are side effects such as the overhead induced by the GCC generated instrumentation, the required ABI layer and hardware characteristics such as cache sizes and CPU interconnections. However, the comparison to CGL here is not intended for an evaluation of STM in general. The main purpose of the evaluation is to compare the new CC algorithms with each other and to the original NOrec algorithm. Therefore, the CC algorithms are compared in the same implementation environment and CGL just serves for the classification of the value range.

The NOrecSb variant resembles the behaviour of the sandboxing implementation proposed by Dalessandro and Scott [DS12] with the advantages of the reduced sandboxing overhead provided by the implementation of our TM layer. It has been included to evaluate delegated periodic validation implemented in the NOrecHt and NOrecCIV variants.

The main goals are to evaluate the changes of the response time of the developed helper thread aided CCs addressing the following two aspects:

1. The detection of conflicts in sandboxing STMs through periodic validation applied by a helper thread (NOrecCIV and NOrecHt) in comparison to the timer-driven periodic validation in the same thread (NOrecSb).
2. The preparation and execution of alternative serialisation orders by parallelised CCs (NOrecCa and NOrecCs) in comparison to non-parallelised execution, i.e. the original NOrec implementation.

Expectations for those two evaluation goals have been very clear in regards to the algorithmic properties of the CCs:

1. evaluation goal

a) NOrecHt should perform better or equal to NOrecSb because the frequency of validations is similar but the NOrecHt approach does its first validation right after the initialisation of the helper thread. NOrecHt should have a slight advantage in longer transactions with low contention because the leader thread runs without the interruptions occurring through the timer-driven validation in NOrecSb.

b) The response times of NOrecCIV should range between those of NOrecSb and the original NOrec because the leader runs uninterrupted and the eager validation of the helper should have the same reaction latency as the eager validation in the original NOrec implementation.

2. evaluation goal

a) NOrecCs should perform almost similar to NOrecCIV because both apply eager validation in the helper thread, helper and leader run synchronously and the probability that the helper of NOrecCs actually commits a transaction earlier than the leader, should be very rare.

b) NOrecCa should perform better or at least equal to NOrecCs because both use almost the same source code except of the synchronisation between leader and helper thread.

c) NOrecCa should perform almost similar to NOrecCIV in cases of low contention, when the leader predominantly commits without a conflict. In this case there is no alternative serialisation order, which could provide a performance gain.

d) With rising contention probability the helper thread should provide an alternative serialisation order more frequently which should result in lower response times of the NOrecCa approach compared to NOrecCIV and the original NOrec.

However, the measurement results revealed some unexpected properties of the implementations, which where not only contrary to our expectations but also unusual in general. After careful verification of the implementation and the evaluation framework, we finally accepted the results and analysed them to find explanations.

Single-Threaded

Before starting a discussion of the results in respect to the evaluation goals, a common inconsistency has to be mentioned first. The results of the benchmarks in single-threaded operation should actually show the differences in the overhead induced by the instrumentation. Conflicts do not occur, which means that no abort will happen and the helper threads have almost no business. Thus, the overhead is caused by the begin and commit functions and

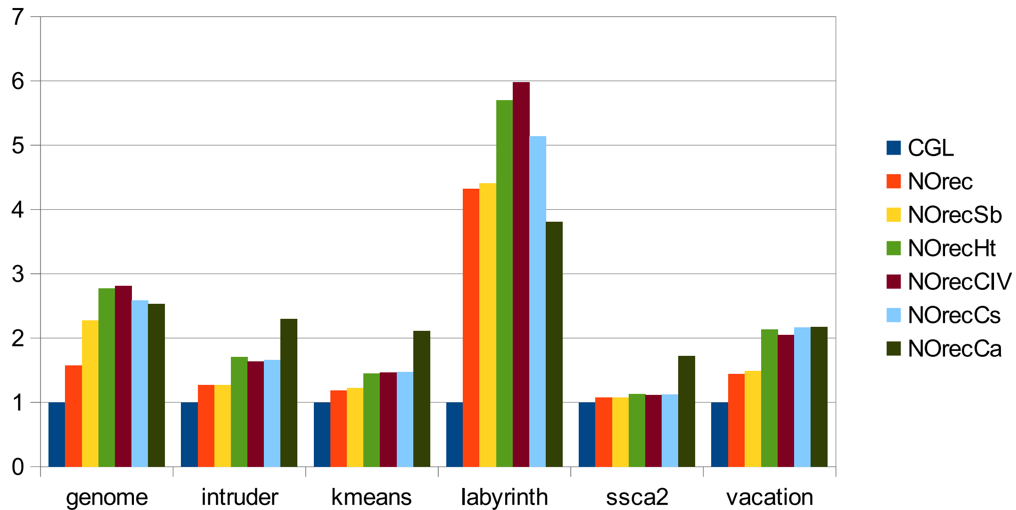

Figure 6.1.: Factor of overhead in single-threaded operation compared to CGL

the read and write functions executed by the leader. Due to the lack of conflicts, the eager validation performed by NOrec just compares its start_time value with the sgsl, which is always equal in this case, and returns without even touching the read-set. Because the sgsl is read very frequently it should be available in the cache most of the time and cause no cache miss. Hence, the response times of all CCs should be very close to each other in case of single-threaded operation neglecting the existence of the helper threads for now.

The chart in Figure 6.1 shows the relative response times of all benchmarks with all CCs in single-threaded operation, normalised against CGL (raw results can be found in Appendix C). CGL has the lowest overhead in this case, as expected.

NOrec and NOrecSb have almost the same overhead in most cases, which was expected, too. The only exception was found with the genome benchmark. According to the characteristics in Table 6.1, the genome benchmark belongs to the class of benchmarks with large R/W Set and high Tx Time shared with vacation and labyrinth. It has a medium Tx Length as vacation, while labyrinth has a long Tx Length. The contention is no concern in the single-threaded scenario. Thus, the benchmark ranges between labyrinth and the shorter/smaller benchmarks and anything observed in this benchmark should either be reflected in the labyrinth benchmark or the vacation benchmark. But neither does show a comparable significant difference between NOrec and NOrecSb.

All CCs with helper threads, namely NorecHt, NorecCIV, NorecCs and NorecCa,

show a significant offset to CCs without helper threads in most of the benchmarks. This offset is larger in benchmarks with larger R/W Set, i.e. with increased frequency of calls to the `read` and `write` functions. This was some expected effect caused by the necessary synchronisation between leader and helper thread. The decreasing overhead in benchmarks with smaller R/W Set also shows that the cloning overhead generated when the helper joins the execution of a transaction is rather small compared to the overhead in `read` and `write` functions.

However, the `NOrecCa` implementation shows some unexpected behaviour especially in comparison to `NOrecCs` which executes almost the same source code in the single-threaded case. In fact the source code executed by both CCs in this scenario differs in just two lines. Additionally the `NOrecCa` implementation beats the original `NOrec` in the benchmark with the largest R/W Set, the labyrinth benchmark. Further investigations revealed that `NOrecCa` in benchmarks intruder, kmeans and ssca2 for some reason commits more often in the helper thread and `NOrecCs` not. The opposite happens in the labyrinth benchmark. This is possible because the response time of lazy validation and eager validation do not differ too much in low contention scenarios. The effect vanishes with increased contention, which is also caused by increased concurrency respective an increasing number of application threads as shown in all benchmarks.

But still this does not explain why `NOrecCa` can be faster than `NOrecSb` in the labyrinth benchmark or why `NOrecCs` can be so much faster than the very similar `NOrecCIV` algorithm in the labyrinth benchmark, too. The only reasonable explanation for this behaviour are differences in the code generated by the compiler. Similar effects have been observed during the implementation and testing of different states already. Considering the layout in memory even a single additional line moves all subsequent code in the same binary on a different location. For example, a function which fits a cache line first, can span two cache lines afterwards if a single line had been added in the previous code. Due to the increased demand in cache space it causes more memory to be transferred and more cache misses in the text cache.

Multi-Threaded

A direct comparison of the `NOrecHt` and `NOrecCIV` variants to the sandboxing approach of Dalessandro and Scott implemented in `NOrecSb` and the original `NOrec` implementation shows the advantages and disadvantages of delegated periodic validation.

The `NOrecSb` variant demonstrates the general advantage over eager validation (original `NOrec`) in most cases. Considering just the algorithmic behaviour and thereby ignoring the hardware influences, `NOrecHt` should perform similar to `NOrecSb`. But obviously, it cannot compete with any of the other CC implementations in most cases. The observed effect gets

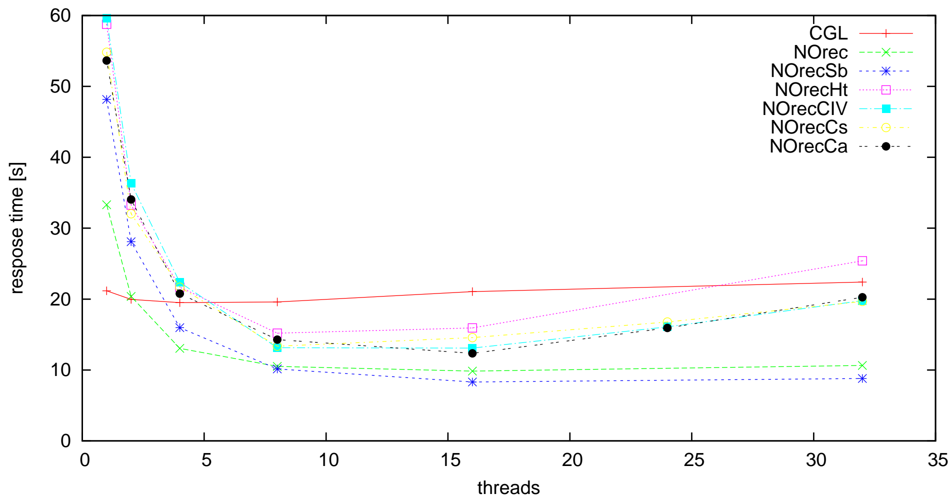

Figure 6.2.: Results of all STM algorithms with the genome benchmark

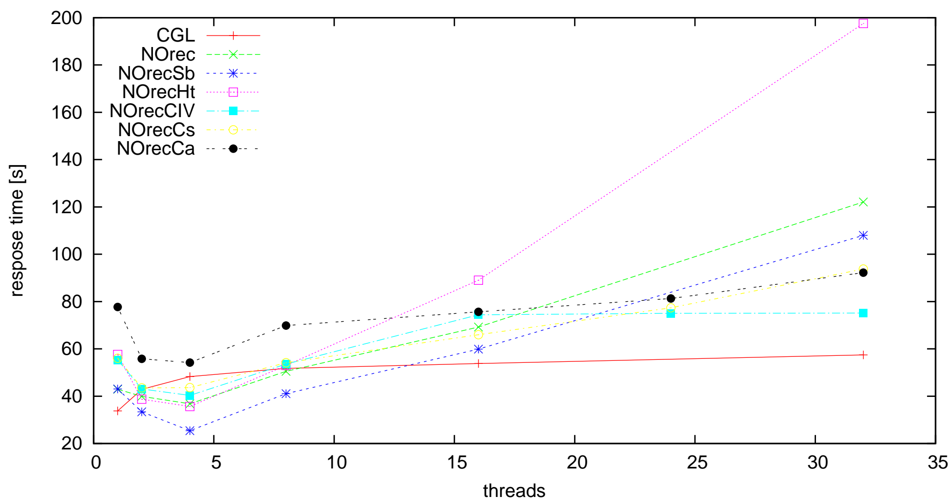

Figure 6.3.: Results of all STM algorithms with the intruder benchmark

more distinct with increased number of threads, which raises the frequency of concurrent commits and thereby the amount of validations performed by the helper thread. Thus, the disadvantage is generated by the delegated validation.

The frequency in which the helper thread validates the read-set of the leader is reduced to the frequency also used by the timer-driven validation in the `NOrecSb` implementation. Thus, the amount of reads on data shared by all transactions is almost the same in `NOrecHt` and `NOrecSb`. But there is additional contention between the leader and the helper thread by

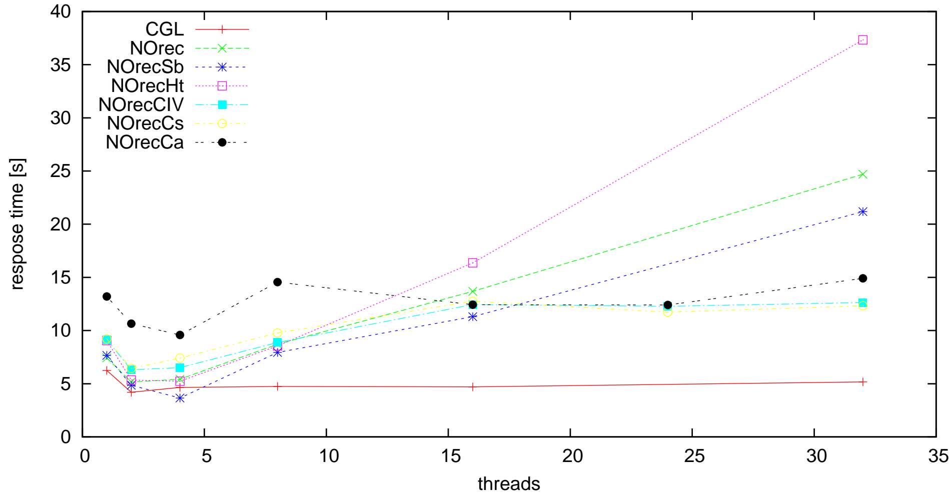

Figure 6.4.: Results of all STM algorithms with the kmeans benchmark

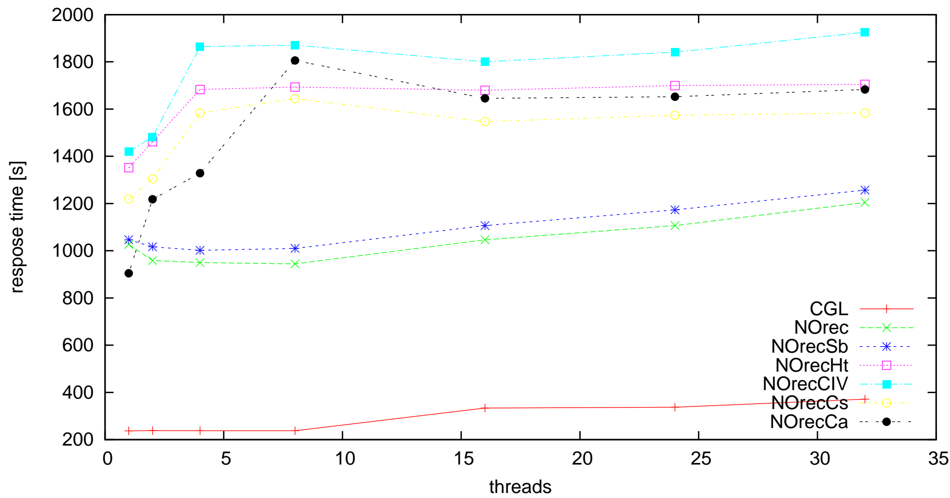

Figure 6.5.: Results of all STM algorithms with the labyrinth benchmark

sharing the read-set of the leader and some synchronisation to guarantee the consistency of the read-set observed by the helper. Also the application data shared between the application threads is now accessed by twice the amount of actual threads, which causes additional contention on hardware layer. Together, these are the reasons why the `NOrecHt` variant performs worse than `NOrecSb`.

`NOrecCIV` was developed especially to solve the issues with the increased cache contention experienced with the `NOrecHt` approach. This approach was very enlightening regarding

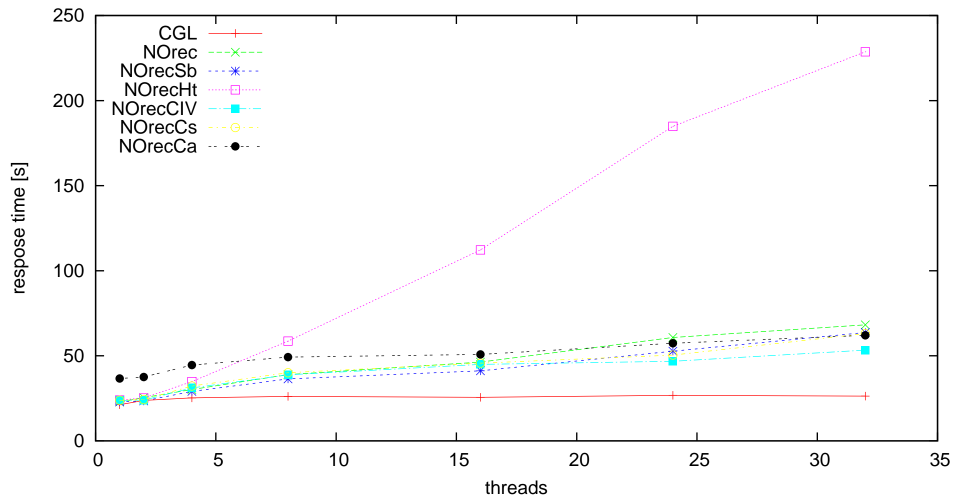

Figure 6.6.: Results of all STM algorithms with the SSCA2 Benchmark

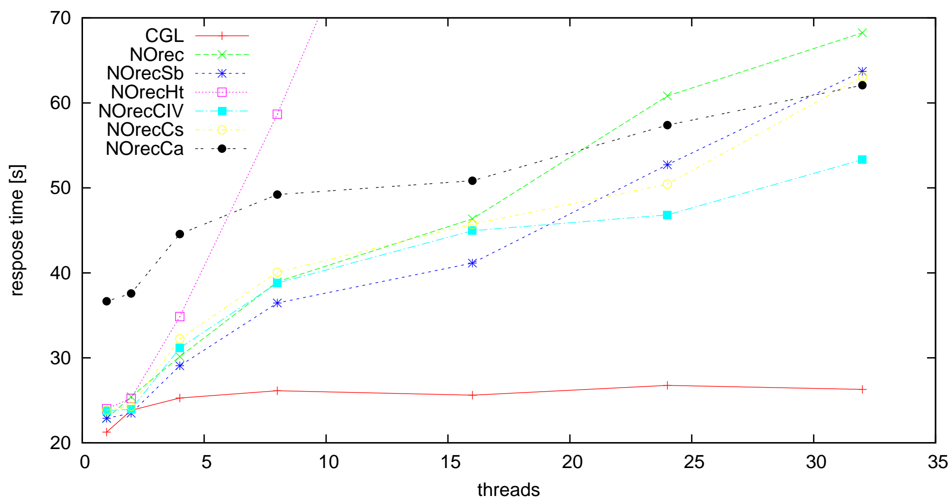

Figure 6.7.: Results with the SSCA2 Benchmark zoomed in

the evaluation of helper thread usage in transactions in general. The negative effect of the `NOrecHt` approach was dramatically reduced, which has improved the response time significantly. Additionally, the response time of `NOrecCIV` is in most cases close to the original `NOrec` implementation. The best results have been scored by `NOrecCIV` with more than 16 threads in the benchmarks with short transactions and small to medium sized read and write sets, namely intruder (see Figure 6.3), kmeans (see Figure 6.4) and SSCA2 (see Figures 6.6 and 6.7). The advantage over `NOrecSb` seems to be odd in the first place

but the timer-driven validation is configured to occur once per transaction run and thereby optimises non-conflicting runs. Consequently, timer-driven validation is less frequent as the eager validation of the helper thread and the `NOrecCIV` variant is supposed to be faster in higher concurrency scenarios.

The worst case situation for `NOrecCIV` was found with the labyrinth benchmark (see Figure 6.5), which runs the longest transactions with the largest read and write sets. A possible explanation for this effect is the additional cache space required by the helper thread. In fact, the required amount of cache lines is doubled in case of two threads working on the same transaction. The lack of space in the cache hierarchy leads to additional cache misses, which can be the explanation for the bad performance in this case. This hypothesis is backed by the fact that `NOrecHt` performs better in this benchmark. `NOrecHt` uses no own read-set and requires less space in caches consequently. The general disadvantage of increased footprint in terms of cache lines is shared by all CCs which apply a helper thread as shown in the labyrinth benchmark.

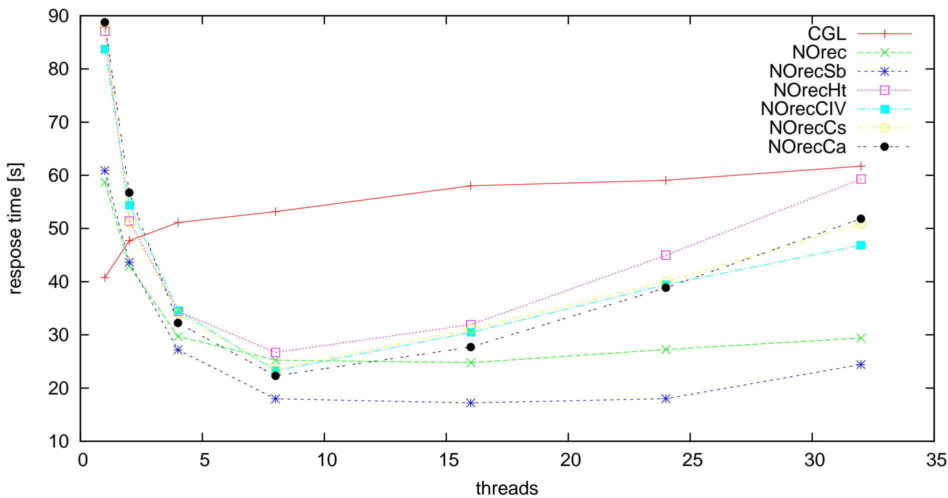

Figure 6.8.: Results of all STM algorithms with the vacation benchmark

The variant `NOrecCa`, which should actually improve the response time due to the reduction of the rollback effort performed worse than expected. Especially in the case of less than 16 threads the `NOrecCa` variant lost performance in comparison to `NOrecCIV` and `NOrecCs` (best shown in Figures 6.3, 6.4 and 6.7). Because the graphs progression of `NOrecCa` and `NOrecCs` is similar, the same effect observed in the single-threaded operation of the benchmarks is responsible for the interval up to 16 threads too. Afterwards the graphs of `NOrecCa`, `NOrecCs` and `NOrecCIV` converge with each other before the `NOrecCa` finally gets worse again. The similarity in the interval between 16 and 32 threads shows that

`NOrecCa` and `NOrecCs` benefit from the same effect experienced by the `NOrecCIV` variant. Thus, the `NOrecCs` variant has the same improved response time in conflicting cases due to the periodic validation performed by the helper thread. The helper thread of `NOrecCa` does not notify the leader about conflicts, which uses timer-driven periodic validation itself. Thus, the leader needs more time to detect a conflict and keeps touching more memory locations. The variants which synchronously abort helper and leader, consequently use less space in the cache and receive fewer cache misses the higher the concurrency gets. This explains the deviation of the `NOrecCa` algorithm starting with and above 24 threads.

Overall, in many cases the results are too blurry to give a reliable verdict. Build and test environment, such as the compiler generated code and hardware properties, had unexpectedly large influence on the evaluation results. Compared to the effort spent for the evaluation, just a few statements can be given with adequate certainty:

- Due to the observations, the most significant advantage can be achieved through the delegation of the eager validation to the helper thread, especially in high concurrency scenarios (above 16 application threads) and short to medium R/W set size.
- An advantage of the approach for the parallel execution of the same transaction following different serialisation orders was not confirmed based on the given implementation.

As the difference of most parallelised CC approaches to the `NOrecHt` variant shows, the generated latency can be astonishingly high. Earlier approaches based on locking mentioned in Section 4 had a much worse scalability. Even if the new approach for parallel execution of the same transaction does not show an improvement yet, it demonstrates a general concept to realise it without significant loss of scalability.

Chapter 7

Investigation of Hardware Influences

One result of the evaluation was that the influences of the test environment are more significant than assumed originally. The hardware influences in concurrent applications are very sophisticated. As already mentioned, the memory access latency is considered as the biggest influence in the response time of highly concurrent applications. This section is intended to analyse influences of the hardware which result in memory access latency, to explain behaviour observed in the evaluation and provide information for future developments aiming on parallelised transaction approaches in STM in terms of SSC.

As mentioned earlier, transaction processing aims at increased concurrency on shared data. Applications with a low degree of concurrency cannot benefit from the concept of transactions, which is also the reason why benchmarks for the evaluation of STMs usually belong to some class of highly concurrent applications. Highly concurrent applications generate high contention scenarios causing multiple and even concurrent cache misses.

This investigation especially focuses on high contention scenarios in shared memory, concurrent applications and analyses the impact of generated effects. The factors with the most significant influence in this case are the memory banks, the connections to the CPUs and between them and the cache coherency protocol (CCP) applied to keep data in caches consistent. This chapter therefore starts with a review of the hardware specifications of the test platform used for the evaluation, to show expected basic latencies in memory transfer. Afterwards, a section about the CCP applied by modern multi-core desktop PC architectures, provides information necessary to understand the effort spent by the hardware to maintain consistency of data. The following section then provides insight in the actual read and write

latency measured in selected high contention scenarios relevant for applications using STM. A final discussion establishes relationships between the results of the evaluation provided in Chapter 6 and the peculiarities found during the investigation of the hardware.

7.1. Shared Memory Related Hardware Specifications

To estimate the expected latency experienced in memory access certain hardware specifications of the given test system not yet mentioned in Section 6 have to be considered. All caches apply write-back caching using a variant of the MOESI cache coherency protocol family. One cache line is 64 bytes wide. There is one L1 data cache per core. Both cores of a Bulldozer module share one L2 cache of 2 MB, providing at least 1 MB for each core. The specification does not explain whether the cores will share cache lines in the L2 cache. All eight cores of a processor share one L3 cache which contains 2 MB space for each module. Again, there is no further information about cache line sharing available. Each processor has a memory controller (2 GHz clock speed), which provides access to the main memory while the processor runs at 3GHz frequency, which equals 1 cycle every 0.333 ns.

Main memory of the system is provided through 128 GB DDR3 1600 RAM with a nominal maximum throughput of 12.8 GB/s. Accordingly receiving one cache line requires at least 4.65 ns. This does not include the access and waiting times to activate the memory banks. Those times range between 16 and 20 of the 800MHz clock cycles (DDR3 1600) of the RAM module, which means about 20 to 25 ns to activate a memory address before fetching it. Because a cache line (64 byte) is read or written in a single burst the activation and waiting cycles count just one time, which results in 25 to 30 ns, the time of about 74 to 89 cycles. Access to memory (i.e. cache misses) are operated along an extended pipeline of the Opteron processors, which reduces the address activation and waiting cycles if the prefetching is successfully. Every processor has just one link to the main memory, which means that multiple memory accesses of the 16 cores of the processor have to be serialised. In the worst case the memory access latency could be 16 times higher (i.e. 394-474 ns), in theory.

Every processor has one direct HyperTransport^{TM}link to every other processor, which is used to operate the CCP and exchange cache lines. The HyperTransport links operate at 25.2 GB/s ($6.4GT/s \cdot 32bit$) throughput. The exchange of a single cache line between two processors would take at least 2.33 ns, accordingly. Because the 16 cores share the same link the latency to receive data can be up to 16 times higher in the worst case, as well. This means a cache line exchange can take up to 37 ns. This estimation does not include the time required to operate the CCP.

7.2. The MOESI Cache Coherency Protocol

Another major influence in memory access latency is the cache coherency protocol which controls consistency of cache lines in distributed caches. The cache coherency protocols (CCPs) applied by AMD and Intel in their most recent multi-core architectures belong to the class of the MOESI CCPs [SS86]. MOESI differentiates the following states of a cache line present in a cache:

- **Modified:** This actually stands for `exclusive modified` and means that the cache containing this cache line is the only one using it and was modified by it. Thus, the content in main memory can be incorrect.
- **Owned:** This stands for `shareable modified`, which means that the cache has modified the cache line, whose content is shared by other caches, and that the cache is responsible for the integrity of this data in all participating caches.
- **Exclusive:** This stands for `exclusive unowned` cache lines, which means that the cache is the only one using that data and thus does not need to take responsibility for the consistency of this data in other caches.
- **Shareable:** This stands for `shareable unowned`, which means that there are other caches using the same data, which has not been modified by the cache, but it might be modified by another cache, which would be the owner then.
- **Invalid:** This is simply the case if a cache line does not reflect the currently agreed state of the corresponding data. It does not matter whether the current state of this part of the memory image is currently in main memory or in another cache, which owns it. It also is the initial state of a cache line. Every other state of a cache line implies that the cache line is valid.

AMD specifies their interpretation of the states of a cache line and the transitions between them in a programmers' manual [Adv11]. According to this information, caches are operated in write-back mode, but every write miss requires fetching of the most recent state of the cache line in advance to its update, either from another cache or from main memory. This is probably required to serve other caches with the current data until the intended update has finally performed because a cache line in modified state is supposed to hold "*the most recent, correct copy of the data*" (see [Adv11], Page 169-170).

A memory access generally starts with an address cycle. The cache controller starts with acquiring the exclusive access for the bus. We consider the bus here as an abstract communication channel to be further specified later. The cache gets the bus-master and puts

the required address on the bus. All other caches notice this action by listening to the bus activity (snooping). When all caches acknowledged the address request, the master finishs the access cycle and, in the simplest case, the data of the respective location in memory can be transferred.

For the different cases in memory transactions six signals (messages) exist, to be exchanged between caches to maintain the consistency. The first three are sent by the cache who initiated the address cycle. The initiator signals what it is intended to do with the cache line:

CA The *Cache Master* signal indicates the intention of a cache to store the requested cache line. In contrast, others might request a cache line for a single-term use.

IM With *Intention to Modify* a cache indicates its intention to modify a cache line. Caches holding a copy of the cache line may have to invalidate the cache line. `IM` requires a previous `CA` signal.

BC The *Broadcast* signal indicates to other caches that intended modifications will be broadcasted actively to other caches afterwards. `BC` requires `IM` and `CA` to be sent beforehand.

This generally provides protocols with two possibilities to maintain consistency during updates. (1) By broadcasting the new content (`CA` followed by `IM` and `BC` at last), which allows the receivers to instantly update their copy or (2) by forcing them to invalidate their copy (`CA` followed by `IM`) and fetch it with their next access to the given cache line.

To deal with cases where multiple caches work on the same cache line, the following three signals exist, which are used as reply to the previous signals.

CH The *Cache Hit* signal is sent during the address cycle to indicate that the sender already has a copy of that cache line. This allows the initiator of the address cycle to derive the proper state for its cache line accordingly.

DI The *Data Intervention* signal indicates that the sender is the current owner of the cache line. It will either provide the data of the cache line instead of the main memory or update its copy in case of a modification.

SL The *Select Line* signal serves as reply to a `BC` signal, to show the master an interest in the broadcast.

Furthermore, MOESI specifies a *Busy* signal (`BS`), used to interrupt a memory transaction and allow an update of the main memory instead.

As mentioned, MOESI just specifies states and signals and supports different strategies that way. The developers of CCPs are free to chose whatever strategy fits best with their system

architecture. However, MOESI specifies for a write miss case that the cache line always has to be fetched, before it is modified. This is probably necessary, because with the modified state the cache gets the responsibility to serve requests of other caches for the cache line at the same time. These arrangements also provide compatibility with other protocols of the MOESI family. However, this rule appears to be too restrictive and a somehow interwoven use of different protocols appears to be too weak as an argument if this would prevent a more efficient protocol to be deployed.

The general structure of the CCP provides some insight in the behaviour of concurrent applications. For example, in shared memory multi-threading, a common memory access pattern is to read a set of shared locations and write a set of memory locations as result of a critical section. Thus, a critical section has inputs defined by the read-set and generates outputs defined by the write-set. A subset of read and write-sets will have an intersection in most cases and the writes will benefit from the fetches already performed by the reads in many cases but it is also not unusual to have multiple disjoint shared reads and writes. Thus, a majority of writes will suffer a write cache miss penalty before the actual write can be performed. Updates to shared data such as in direct update STMs or STMs using shared orecs suffer on these platforms because of the repeated write misses after each rollback.

It can be assumed that the communication effort linearly rises with the number of involved caches. After initiating a memory transaction all caches first have to send a reply signal. In today's architectures the bus has been replaced by a network of direct links between CPUs and between CPUs and memory controller hub (cf. Section 7.1). Additionally, cores have their own memory controller. Hence, replies do not need to be necessarily serialised (as on a bus), which could allow for more parallel processing in the CCP. Furthermore, it can be assumed, that the processing time of received signals is low. The generated overhead by a CCP should be almost constant even with an increasing number of involved cores/caches. Thus, the most influencing factor should be the transfer of data between main memory and caches and not the CCP.

From a logical point of view, a memory controller represents a queue with memory transactions to be processed serially. This is necessary, because main memory is usually reached via a bus of limited capacity.[1] An increasing number of cache misses in different cores will consequently hit the capacity of the memory bus. Considering a constant flow of incoming memory transactions, all cores will be equally influenced and the time to serve a memory transaction will rise proportionally with the number of cores.

[1] Some architectures provide even parallel access to distinct memory banks, but this requires memory transactions present in the memory controller, which actually address different memory banks, in other words, it depends on the application and the memory management.

7.3. Cache Miss Penalty

It is a rule of thumb that a cache miss causes a latency of about 100 to 150 instruction cycles. On the given architecture this would take about 33 to 50 ns. Observations during the evaluation lead to the assumption that the latency can be significantly higher, especially in higher contention scenarios. The calculations based on the hardware specifications in Section 7.1 already showed that this rule is not sufficient for highly concurrent applications. This section covers a set of measurements on read and write operations on shared data to get more appropriate real world estimations for the given test platform (see Section 6) used in the evaluation of the CCs.

The benchmark developed for this investigation is kept simple. All scenarios address the latency of read and write operations on a set of continuous locations in virtual memory space experienced by threads. The first scenarios iterate over all 64 bytes of a single cache line in a loop either in read or write mode or performing a certain access pattern such as write after read. The benchmarks are written in C and thus the read and write operations inside the loop are actually assignments from shared data to local variables (i.e. processor registers) or vice versa. To prevent the C compiler from optimising the reads or writes out, some arithmetic instruction, such as a summation (+) had to be added. The remaining scenarios use similar loops on larger memory areas. Each single access in the loop reads or writes 64 bit at once. To reduce the measurement overhead, this loop is repeated a reasonable amount of times in an outer loop, depending on the effort spent in the inner loop.

Each thread of the benchmark independently measures the total execution time of the outer loop and calculates the average execution time for a single iteration of the inner loop in the end. This execution time provides a rough estimate for an average of sustained access latency for the given thread, assuming the few instructions around to be insignificant in comparison. Each thread can be assigned with an individual task and provides an individual measure. A task comprises a function with a loop to be executed, data to be accessed and even the core to be used by the thread. This allows to run different scenarios by configuring some tasks to be executed on certain cores.

An external evaluation framework controls the execution of the benchmark. It calculates statistics for the results of performed runs and repeats executing the benchmark in a given scenario until a 90% confidence interval for every calculated mean is less than 5% of the standard deviation for each measure considered in the graphs. Considering the large amount of iterations of the outer loop, the accuracy of the calculated mean is actually much higher.

7.3.1. Single Cache Line

The scenarios presented in this section represent cases where a single cache line is accessed concurrently. Once a cache line was fetched from memory it should remain in the cache and one cache should get the ownership. Consequently, subsequent accesses does not involve a data transfer from main memory but from one of the caches while the CCP is applied to find a valid copy.

Read Only Scenario

The latency of read-only access (`RO` in Figure 7.1) of multiple threads to a single cache line without cache misses is around 1.7 ns, even though it shows a peak of 6.7 ns with 2 threads on the same CPU. If just one thread is running in the system, the latency is about 1.2 ns, which shows a possible overhead of the CCP between 0.5 ns ($\approx +20\%$) and 4.5 ns ($\approx +300\%$) in case of concurrent reads. Thus, the CCP causes about 3/4 of the read access latency in the worst case.

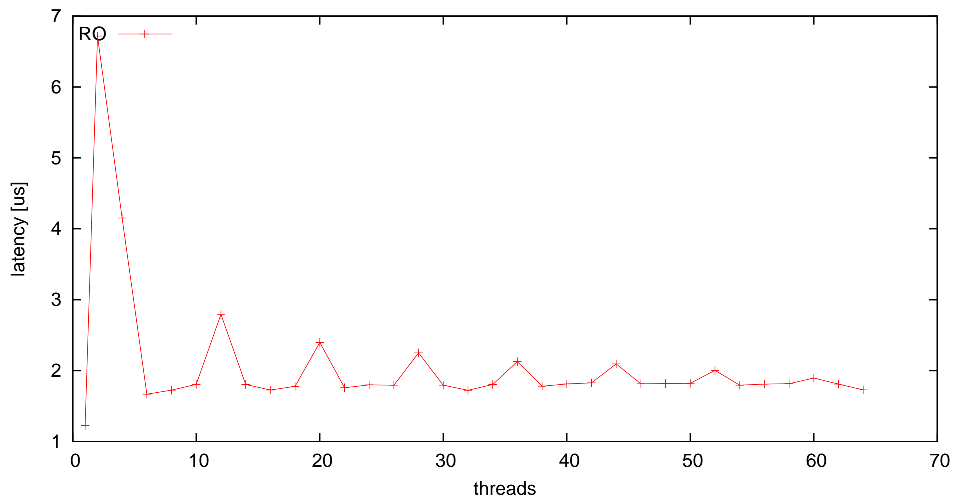

Figure 7.1.: Read only scenario

Most of the peaks observed in this scenario describe a repeated pattern and occur each time the third and fourth core of a not yet fully utilised CPU gets involved in the measurement run. But the peak at two threads is apart from this pattern. This behaviour is always observable if just two of the cores are active and read the same cache line at the same time as further results in the following sections will show.

Write Only Scenario

In a write only scenario (`WO` in Figure 7.2), the latency rises from 1.3 with one thread up to 235 ns with 64 threads. The increase is almost proportional to the number of threads. Thus, a concurrent write access on the same cache line causes an additional overhead of about 3.8 ns per thread, here.

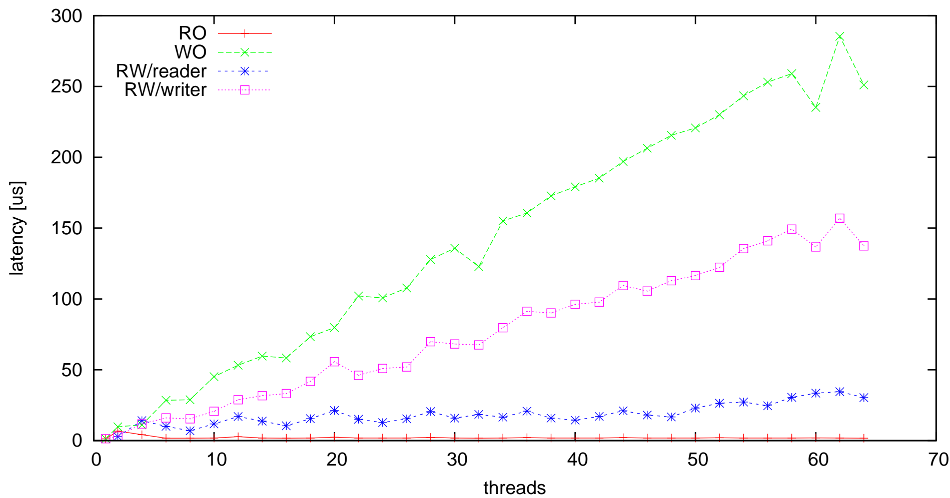

Figure 7.2.: Read only (`RO`), write only (`WO`) and mixed reader writer (`RW`) scenarios

Mixed Reader Writer Scenarios

In the mixed reader writer scenario (`RW` in Figure 7.2 above) readers are placed on the first half of the set of cores applied in the given measurement run and writers run on the remaining half of the set of cores. This scenario shows a linear dependency between the amount of running threads and the latency too. The read latency (`RW/reader`) rises with every additional thread by 0.4 ns and the write (`RW/writer`) latency rises by about 2.7 ns.

Comparing this scenario with the write only (`WO`) scenario, we have to consider that the amount of writing threads in the WO scenario is doubled, because in the mixed scenario (`RW`) just half of the applied threads does write. Thus, the per writer thread increase in write latency here is actually at about 5.4 ns while it was 3.8 ns in the `WO` scenario. This means, that the presence of a reader significantly adds about 1.6 ns latency per thread to the write latency in this scenario. According to the same calculation, the readers suffer 0.8 ns in latency per concurrent write thread here.

Assuming shared cache usage of cores, the access latency should lower if the data has already been fetched to the cache by a neighbouring core. In the scenario presented in the

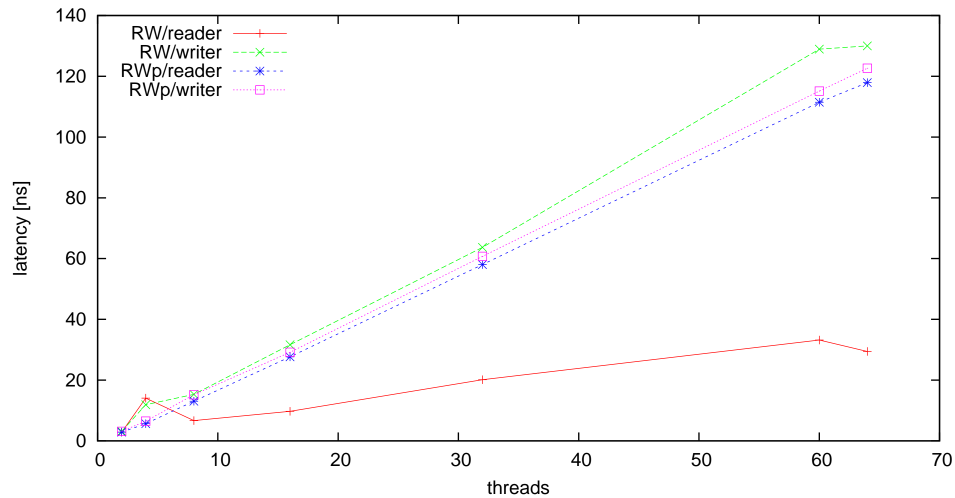

Figure 7.3.: Mixed reader writer scenarios unpaired (`RW`) and paired (`RW/p`)

graphs with the `RW/p/` prefix in Figure 7.3, pairs consisting of one reader and one writer have been placed on neighbouring cores of the same Bulldozer module. Compared to the separated readers (`RW/reader`) and writers (`RW/writer`), the read latency (`RW/p/reader`) here is significantly increased and surprisingly similar to the write latency (`RW/p/writer`). This means, that writers have a dramatic impact on readers addressing the same highly contended cache line on the neighbouring core - the opposite of our expectation.

Single Reader and Single Writer Scenario

The following two scenarios serve to investigate more precisely how much read access influences latency of write access and vice versa. The first scenario (`SW` in Figure 7.4) has just one writer while the amount of reading threads was iteratively increased. The second scenario (see `SR` in Figure 7.5) implements just the opposite situation with a single reader and multiple writers. Writing threads are always placed after the reading threads, which causes the single writer to move through all CPUs while iterating over the different amounts of threads during the measurement runs.

The results show, that the single writer latency (`SW/writer`) is influenced very little by increased number of readers and almost asymptotically approaches 7 ns besides the outliers close to 10 ns, which have already been observed in the read only scenario. In contrast, the read latency (`SW/reader`) tends to decrease after it has reached its maximum with 3 readers (4 threads in total).

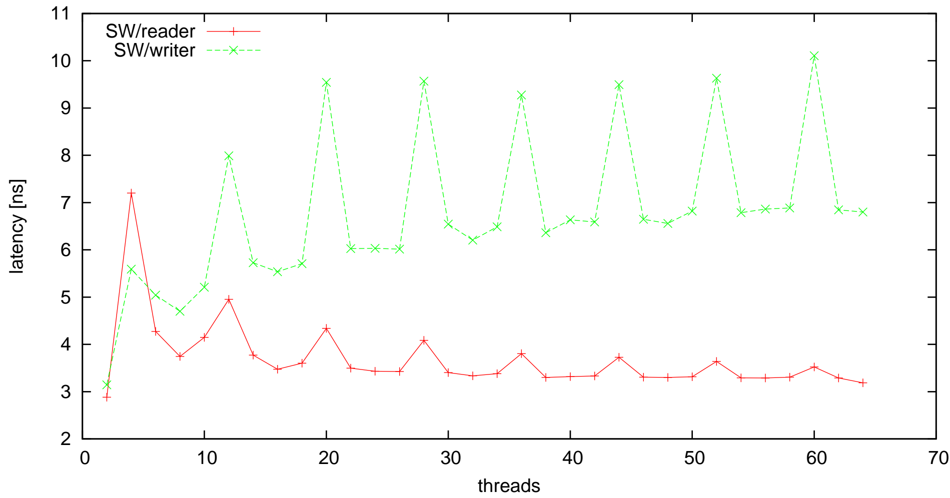

Figure 7.4.: Single writer scenario (`SW`)

In contrast to the single writer latency, the single reader latency (`SR/reader` in Figure 7.5) constantly rises with the amount of concurrent writers by about 1.76 ns per thread.

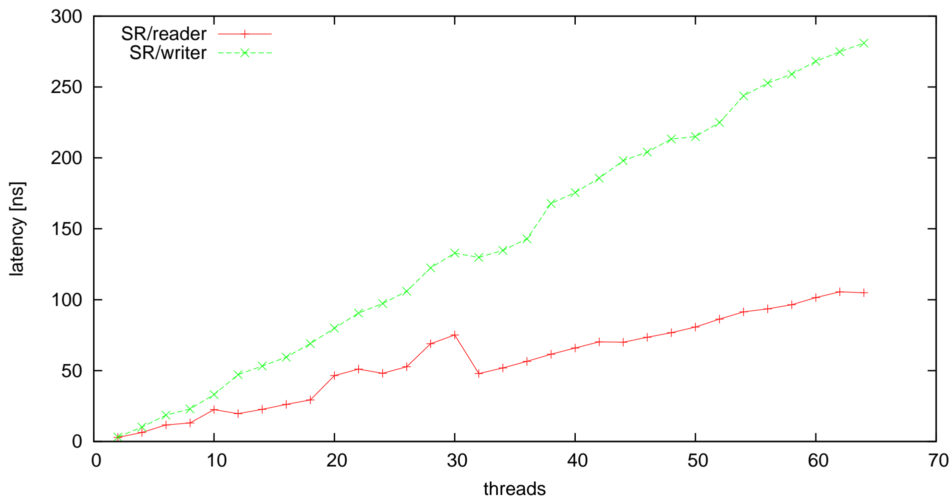

Figure 7.5.: Single reader scenario (`SR`)

The almost linear increase of the read latency in proportion to the amount of writers was unexpected. Because reads represent actually dirty reads, which do not have any guarantee regarding their consistency in terms of a memory model, the cache miss penalty for a single read should be approximately constant and almost independent of the amount of cache misses experienced by multiple concurrent writers. According to previous observations the reader

thread has a higher access frequency than writer threads, which results in a lower read miss probability with fewer concurrent writers, consequently. With increasing amounts of writer threads, the read miss probability should rise until every read suffers a read miss. Accordingly, (1) the read latency should *asymptotically approach* a given maximum of something about 3.5 ns (cf. `SW/reader` in Figure 7.4) and (2) it should be far lower than the measured maximum of 128 ns (see `SR/reader` in Figure 7.5).

The almost linear increasing latency seems to indicate some serialisation of concurrent cache line updates caused by the CCP, which influences the readers in the same way. The bottleneck at the HyperTransport links between CPUs is no explanation. If so, the latency should rise significantly each time another CPU gets involved in the measurement runs, which would be the case with 17, 33 and 49 threads, which is not reflected in the results.

Influence of Disjunct Address Cache Misses

A remaining question is, whether the proportional latency gain with increasing amount of writers will be observable if concurrent cache misses occur on disjoint cache lines (addresses) too. That means, if concurrent cache misses generally have to be negotiated in a serialised manner and cannot be parallelised instead.

This required a scenario where pairs of one reader and one writer access a shared cache line which is disjoint to the cache lines shared by other pairs of readers and writers. The results shown in Figure 7.6 disprove the above hypothesis. Reader and writer latency approach asymptotically some maximum with an increased number of threads, which is nearly reached at about 16 threads already. This has also been proven in another scenario where reader and writer pairs where placed on separate CPUs (not shown here). The latency reached is roughly the same as in the single writer scenario (see Figure 7.4). Thus, the concurrent cache misses on disjoint cache lines can be negotiated in parallelised manner.

Write After Read

Another interesting situation in concurrent applications is a write after read sequence. This is a very common sequence for example when incrementing a variable, acquiring a lock or in privatisation schemes. We have chosen an increment to evaluate this case. The results in Figure 7.7 show that the latency is exactly the same as in the write only scenario. This implicitly proves, that the CCP always fetches a cache line before it is updated. Otherwise the write without former read had to be much faster.

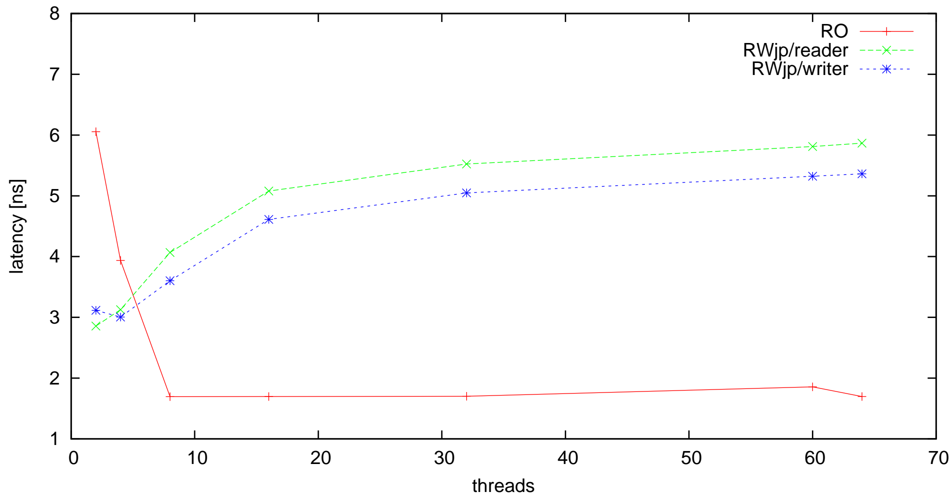

Figure 7.6.: Reader writer pairs on sibling cores working on data, disjoint from other pairs

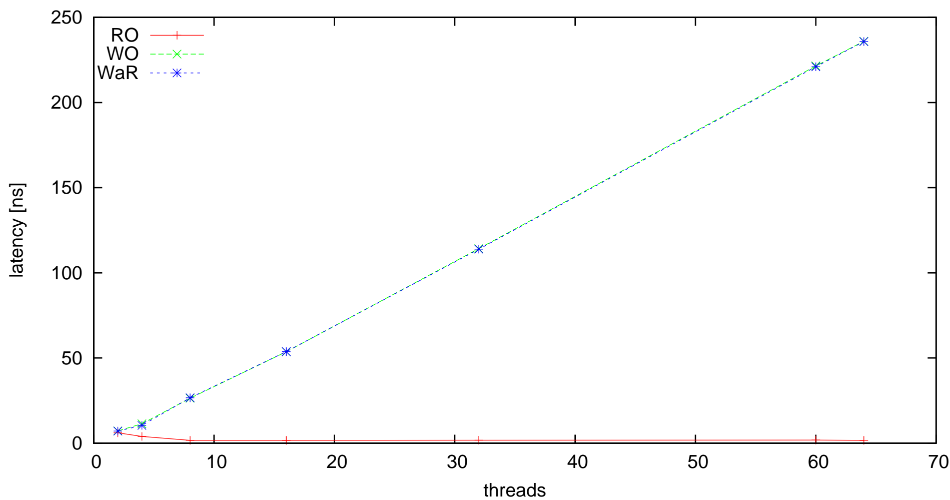

Figure 7.7.: Write after read (`WaR`) compared to read-only and write-only access

7.3.2. Atomic Instructions

An important performance factor in STMs and concurrent applications in general are atomic instructions used to implement synchronisation mechanisms (locks, mutexes, etc.) and non-blocking concurrency, too.

The scalability of atomic instructions is significantly worse compared to non-atomic instructions according to observations made in the domain of spin-locks (see e.g. [ME08]). We wondered whether this is a general problem of atomic instructions and how they perform in

comparison to non-atomic instructions which fulfil the same task on the given platform.

Atomic Increment

An atomic increment of a shared variable is the most basic case we can think of. It also renders the same memory transactions occurring in the non-atomic write after read scenario for comparison.

In assembler an atomic increment is a simple `ADD` instruction with a `LOCK` prefix. The prefix indicates the atomic execution of the following instruction. Atomicity is realised differently, depending on the architecture used. Modern multiprocessor systems, such as AMD Opteron architectures, have two alternative mechanisms: Either the system bus will be locked (*bus lock*) or a certain cache line will be locked (*cache lock*) for the time the instruction is executed. The latter variant can be applied if accessed data fits into a single cache line.

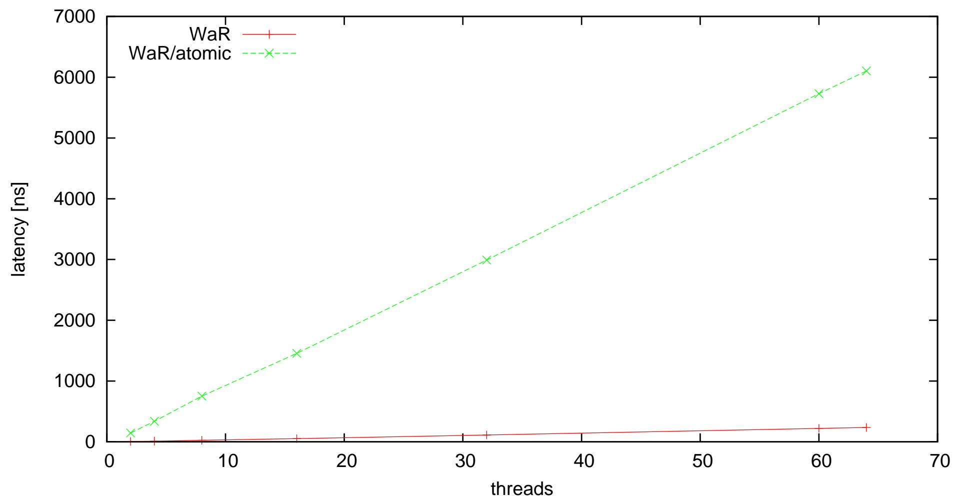

Figure 7.8.: Atomic increment (`WaR/atomic`) compared to non-atomic increment (`WaR`)

The results of the measurement runs (`WaR/atomic` in Figure 7.8) show the immense costs for a cache lock in high contention scenarios. Another measurement (not shown here) using bus locks provided exactly the same results: The latency of non-atomic access (`reader`/`writer`) was similar to scenarios without concurrently executed atomic instructions (e.g. `WR/reader` and `WR/writer` in Figure 7.2).

Bus or cache locks conflict with other atomic instructions only. Another measurement (Figure 7.9) with concurrently executed non-atomic read and write instructions has proven this statement.

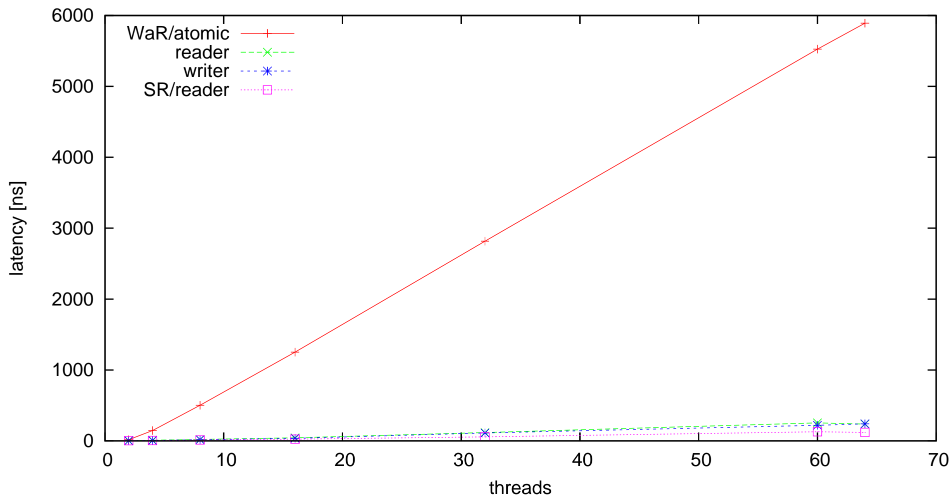

Figure 7.9.: Non-atomic reads (`reader`) and writes (`writer`) during concurrent atomic increments(`WaR/atomic`)

Atomic Compare and Set

A compare and set instruction (Assembler: `XCHG`) is another atomic instruction. It compares the current value of a location with a given value and changes it to another given value in case its equal. Thus, it has to read a cache line and compare and update the same cache line, which means the main difference to an atomic increment is the comparison. Every STM implementation known to us requires such an instruction or similar instructions at some point. It is usually applied in a loop, which is repeated until the instruction was successful, similar to a spin-lock. This in turn generates much more contention on the cache lock as locks which suspend threads. But the advantage of spin-locks and similar constructs is that a blocked thread can resume without a context switch if the lock is released.

To render some similar contention pattern as experienced by the `sgsl` in `NOrec`, the first scenario (`WaRe` in Figure 7.10) tested here contains a loop with the following instructions:

1. Read the current value of the shared variable into a register,
2. increment the register,
3. use CAS to check whether the shared variable still contains the original value and set the incremented value in this case, and leave the loop.
4. Otherwise retry the loop.

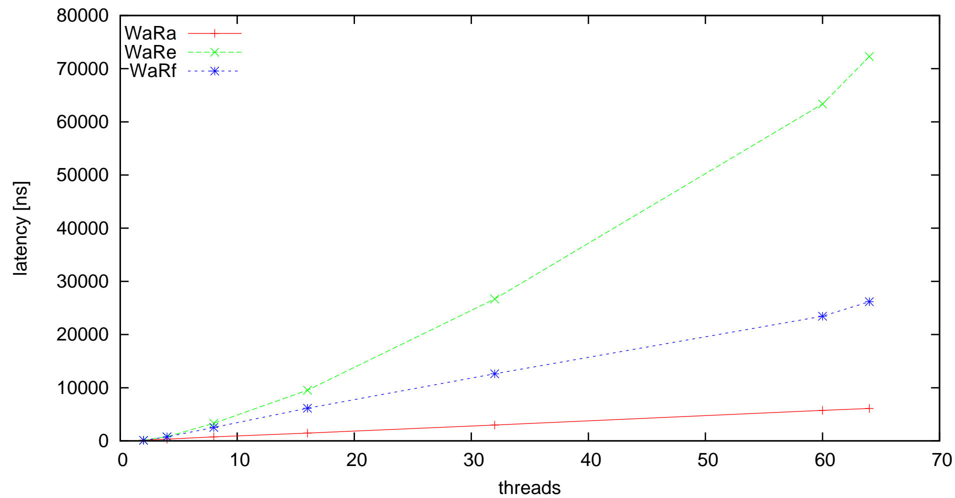

Figure 7.10.: Spin-Lock optimisation potential

Figure 7.10 depicts a comparison of the latency received with this improved loop (`WaRe`) compared to a simple atomic increment (`WaRa`). Thus, the loop is much more expensive although it just adds a few more instructions.

To demonstrate the optimisation potential for spin-locks on the given hardware, we have applied a simple tweak to our loop. As the results in Figure 7.9 showed, a concurrent read on a locked cache line is significantly less expensive than a CAS operation. Thus, a simple non-atomic comparison in front of the CAS operation can check first if the CAS operation will be unsuccessful anyway and bypass it by restarting the loop. This actually adds more control paths to the code, which can reduce the success rate of the branch prediction for the pipelines. But as the results of this modified loop `WaRf` in Figure 7.10 show, the latency is reduced by more than half the amount.

7.3.3. Main Memory Access

If the working set of a task exceeds the size of the associated caches, cache lines will be discarded and data has to be fetched from memory. The scenarios investigated here address the latency received in main memory access in certain special cases:

1. Linear (iterative) and non-linear (`x` in graph identifiers) access to a large block to evaluate streaming capabilities of the memory controller. In the non-linear pattern, the thread iterates through the data from both sides and toggles between begin and end with each access.

2. Pairs of a reader and a writer on neighbouring (p in graph identifiers) or separated cores, each pair working on a shared data block disjoint (j in graph identifiers) from others to evaluate advantages using the same caches.

Benchmark	Core Affinity	Memory Access Pattern
RWj2MB	separated	linear
RWjp2MB	neighbouring	linear
RWjx2MB	separated	non-linear
RWjpx2MB	neighbouring	non-linear

Table 7.1.: Memory access benchmarks

Four configurations for the benchmark have been developed to test all combinations of the above variations, all working on 2MB data sets for each pair (see Table 7.1). Recalling the specifics of the cache hierarchy of the given test platform each core is provided with 1 MB space in the different levels of the hierarchy. Thus, a working set of 2 MB will certainly exceed the available cache space by at least 100%. Additionally, all scenarios use the mixed reader writer pattern, having one half of the set of cores utilised with reading threads and remaining cores with writer threads (threads are placed according to the information given in Table 7.1).

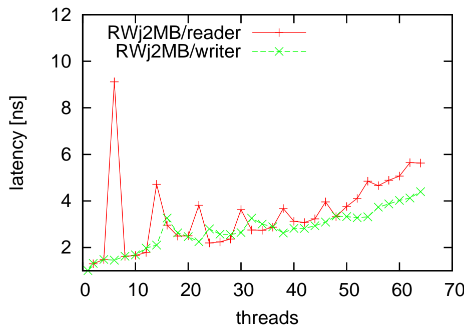

Figure 7.11.: RWj2MB

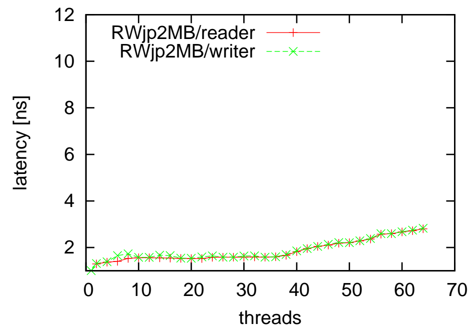

Figure 7.12.: RWjp2MB

A comparison of RWj2MB (Figure 7.11) and RWjx2MB (Figure 7.13) show a significant difference between linear and non-linear access. Non-linear access has comparably more latency even though the difference is low. This means, that the hardware is more optimised for iterative memory access. The average latency is astonishingly low, when comparing the results for cache misses on a single cache line (see e.g. Figure 7.6), which certainly demonstrates the efficiency of the pipeline extension of the Opteron processor to handle cache misses.

A comparison of the access latency for pairs on separated or neighbouring (p) cores, depicted in RWj2MB (Figure 7.11) and RWjp2MB (Figure 7.12), or RWjx2MB (Figure 7.13)

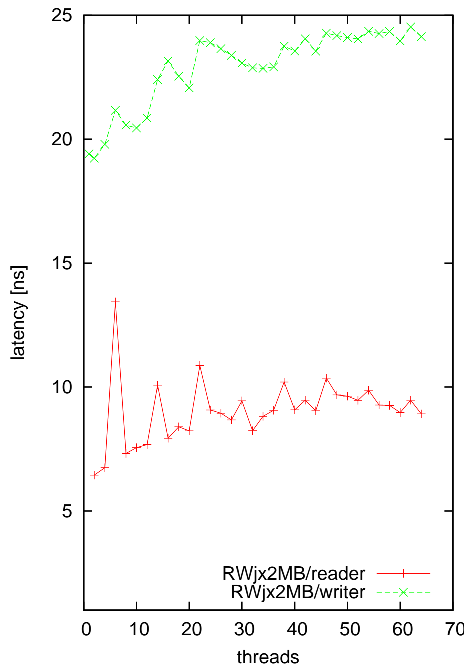

Figure 7.13.: RWjx2MB

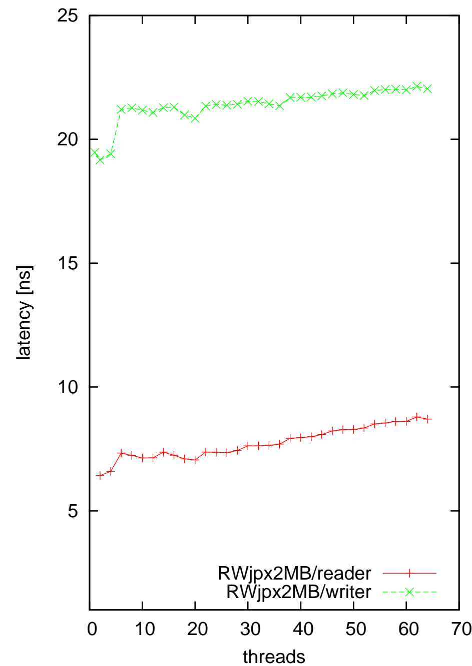

Figure 7.14.: RWjpx2MB

and `RWjpx2MB` (Figure 7.14) shows that neighbouring cores have a slight advantage over separated cores accessing the same data in these scenarios. The advantage is up to 3 ns (8 ns considering the outliers) per access and independent of the memory access pattern. It affects readers and writers in the same way and does not increase the read latency as experienced with multiple threads accessing the same single cache line (cf. Section 7.3.1).

Because the thread pairs work on disjoint data, cache misses in different cores will not address the same cache line (the main difference from the read write scenario in Section 7.3.1). Thus, the observed overhead is probably mainly caused by the memory busses and the memory controller. If reader and writer work on separate cores (i.e. Figures 7.11 and 7.13) and the amount of threads exceeds 16, which results in threads placed on the next CPU, the latency of the HyperTransport links adds in.

7.4. Discussion

The investigation generally shows that the access latency on cache in higher contention scenarios can be significantly higher than 100 to 150 instruction cycles in concurrency intensive applications, especially if atomic instructions are involved, which cause up to 100/1000 times more latency. Furthermore, it shows, that the CCP has significant influence on the latency, too.

Even though the influence of concurrent reads on unmodified cache lines is much lower than concurrent updates and limited by some hardware specific maximum, it is still significantly higher than in single threaded operation, especially with two threads only. This increased latency is probably caused by the CCP.

The significant increase of the read latency, especially if just two neighbouring cores of a CPU access the same cache line, is one of the reasons for the general overhead in all the helper thread aided CCs especially observed in the measurement runs with a single application thread. The cloned thread actually accesses the same data almost simultaneously if it executes the same transaction and thereby the same application code in parallel.

The single reader and single writer scenarios back the common expectation that cache misses are generally caused by updates. Additionally, the write after read scenario shows that even writers will suffer a cache miss after an update, which conforms to the CCP, too. The write latency rises proportionally with the amount of concurrent updates, which means that concurrent updates seem to be serialised somehow. The resulting increase in latency can be very high if multiple write misses occur concurrently. This is not intuitively understandable, because just one of the concurrent updates will be accepted in the end anyway.

The majority of those observed characteristics affect concurrent applications in general but the effect of concurrent reads on the write latency especially influences transaction commits in our case. The commit requires to write back the new state of shared data. An increased number of threads increases the probability for concurrent reads on these shared data and thereby slows down the commit. Because the amount of reading threads is even higher with helper threads, this effect gets even worse in parallelised transactions.

Furthermore, all cores reading a concurrently updated cache line, are apparently involved in the negotiation of the new cache line content and cannot proceed until all updates have been applied. This is an interesting effect but its impact on the CCs used here is very little, because commits, and thereby most of the updates of transactions, are mutually exclusive and not concurrent to each other. Thus, multiple concurrent updates cannot occur and the read latency will not rise that high.

The negative effects of concurrent reads and writes of two cooperating threads have been especially observed with the `NOrecHT` implementation where the leader thread updates its read-set, which is repeatedly read by the helper thread. Placing threads on neighbouring cores helps, but not enough. Even the improved helper threads, with decreased amount of data shared with the leader suffer from these effects. As the mixed reader writer scenarios demonstrated, paired access to the same cache lines unfortunately has a negative impact in addition, because the read latency is dramatically increased in comparison to unpaired access. This causes leader and helper threads placed on sibling cores to suffer. The `NOrecCa` implementation

uses non-sibling cores, what sometimes has a positive effect, but increases latency for data exclusively shared by helper and leader in-turn. Thus, finding a general rule will be difficult.

The issues with concurrent atomic instructions is another disadvantage, which affects transactional memory in general and parallelised transaction execution in particular. In the prototypes demonstrated here, the `sgsl` is affected by this issue. The pressure on the `sgsl` is even higher through the helper threads trying to commit concurrently. This disadvantage can be significantly reduced by proposed optimisations such as using a simple read to check the availability before executing the CAS instruction on the `sgsl`. But this will improve even the original `NOrec` implementation and not just our parallelised versions.

The investigation of the hardware influences have a common result: Highly concurrent applications have bad runtime scalability in respect to the number of threads on the given hardware. Consequently, highly concurrent multi-threaded applications using transactions will always suffer from this disadvantage more than applications using mutual exclusion instead. The latter implicitly reduces the level of contention by suspending threads that are blocked at a lock. This explains why the single global lock CC (CGL) provides better results in the evaluation as any of the STM approaches in most cases.

The generally bad scalability of highly concurrent scenarios is a significant disadvantage for any CC which applies additional helper threads and thereby further increase the degree of concurrency. The advantage of neighbouring cores is almost non-existent and thus it cannot compensate for the additional overhead caused by the hardware, due to the increased concurrency. This explains why the parallelised CCs could not compete with the single threaded CCs in the majority of all cases. It might be possible to optimise the implementations, considering the hardware properties but the increased contention will always be a major issue on architectures like this.

According to this investigation, the given architecture rather suites the concurrent operation of loosely coupled processes working on shared data in a mutual exclusive manner. The concurrent data access seems to be considered as a rare case as reflected by the separation in the cache hierarchy too. Also, the MOESI-based CCP seems to have issues with higher contention.

The inherent problem causing all this issues is the attempt to support a shared memory programming model on top of a distributed system consisting of multiple interconnected caches which have to exchange messages to keep data consistent. The development of efficient hardware, capable of dealing with this distributed problem properly, can be considered as a very complex task. Against this background, and considering that most of the software is still single-threaded anyway, current end-user multi-core architectures will perform similarly. The fact that Intel uses a MOESI CCP as well backs this assumption. But considering the time, science and industry spent in research and development of multi- and many-core architectures,

there might be more specialised concepts and implemented hardware already which provide a better suited platform for parallelised CCs and transactions in general.

Chapter 8

Conclusion

While transactional memory provides the opportunity to solve most of the common issues of traditional mutual exclusion in critical sections, transactions still do not fully exploit their potential in regards to increased concurrency. Conflicts between transactions are solved by aborts and retries and thereby loose progress in comparison to some ideal CC mechanism. Due to the increased availability of processing units in today's and future personal computer hardware it seems a viable approach to utilise additional processing units to parallelise the transaction itself and thereby improve its response time.

This thesis studies different approaches to apply helper threads in transactions. It considers the delegation of processing effort to a helper thread in general and the actual parallel execution of the same transaction in respect to different serialisations in relation to concurrently running transactions in particular. The latter goal was motivated by an approach of Bestavros for a concurrency control in real-time databases called speculative concurrency control. It was the first in applying a parallel execution of transactions following different serialisation orders.

Requirements on transactions in database and distributed systems differ from those of transactional memory. Tests with a CC similar to Bestavros approach implemented in the run-up to this thesis revealed pessimistic concurrency control, direct updates and increased synchronisation overhead between leader and helper threads as a main disadvantage. All approaches for the application of helper threads in transactions presented in this thesis consider these experiences. The reference concurrency control algorithm to be enhanced by the approaches developed in this thesis is NOrec, a cutting edge optimistic concurrency control for transactional memory developed by the University of Rochester. It uses no additional shared metadata on shared data objects such as ownership records. And it applies deferred updates, reducing the rollback effort, and eager incremental validation. This concurrency control has

been modified in different ways to achieve the variations of parallelised transactions proposed here. In all approaches the communication between leader and helper threads is kept as low as possible.

In respect to the goals of this work the analysis focuses on a set of helper-thread aided concurrency control approaches covering two variations: delegation and parallel execution. The first two approaches delegate the validation to a helper thread. While the first approach just constantly validates the read-set of the leader the second approach actually executes the transaction in leader and helper thread concurrently using different validation schemes. While the leader performs its validation in its commit only, the helper applies eager validation with every read. Because both threads start executing on the same state of shared data, all conflicts experienced by the helper thread apply to the leader thread as well. That way, the helper thread can validate the state of the leader without touching its read-set, which results in a reduced contention between both threads. The remaining two approaches address the parallelised execution of the transaction in respect to different serialisation orders. Both approaches use lazy validation in the leader thread and eager validation in the helper thread, much as the previous approach, but here the helper is allowed to commit. The first approach has a tight coupling between leader and helper thread by aborting both threads in case of a conflict detected by the helper, keeping both synchronously executing each run of a transaction in parallel with each other. This way it is actually very unlikely that the helper thread will follow another serialisation order than the leader. In case a commit occurs concurrently to a read in the helper thread, it is able to instantly switch to the new valid serialisation order and potentially commit while the leader will most likely abort. The main reason for the implementation of this variant was to get some comparable concurrency control for the evaluation of the next approach during evaluation. The asynchronous cloned execution approach does not control the progress of helper or leader during transaction execution, which allows them to abort independently. In case of multiple consecutive aborts of the same transaction, both threads will drift apart and potentially follow different serialisation orders. As mentioned, the lack of progress control between leader and helper thread was intended to reduce the synchronisation overhead.

The analysis of those approaches helped to identify constraints for the instrumentation of transactions and required sub-systems to operate helper-thread aided transactions. Especially the use of non-transactional (so-called transaction unsafe) sections are considered to be critical. Because usually transactions are executed in a single thread, most instrumentation environments provide constructs to allow access to thread-private data to be non-transactional. With parallelised execution of a transaction even thread-private data is considered to be accessed concurrently by multiple threads and instrumented accordingly. Data on the execution stack, such as local variables, is considered to be copied in the stack of the helper in the course

of cloning of the transaction start.

A result, helping in future developments in this field of research, is the set of sub-systems required by parallelised transactions:

- A sandboxing facility provides functionalities required to realise sandboxing of a transaction.
- A cloning facility provides functions to realise cloning of a transaction start into the helper thread and control transfer from a transaction committed by a helper thread to its leader.
- Different allocators for the memory management inside transactions have been proposed for certain use cases such as the parallel execution of transactions with and without commit and sandboxing in general.

The presented sandboxing approach is for the most part an enhancement of the existing approach of Dalessandro and Scott, which improves the protection and further reduces the validation overhead, too. While the existing approach still needs to validate in-place stores to protect the stack, our approach considers cases where pointers address critical parts inside stack frames of the transaction as indicator for an invalidity and immediately abort.

The challenge in cloning transactions was mainly to keep the synchronisation overhead between leader and helper thread low while the helper thread needs a valid copy of the current stack frame of the leader. Fortunately, the GCC compiler provided an opportunity by instrumenting each transaction start with code which generates a copy of the local variables required inside the transaction. This allows us to create another copy for the clone while the leader already starts to execute the transaction and probably modifies its own copies of local variables.

A positive side-effect of the analysis was the development of an alternative solution for the transparency of memory management in transactional memory, supporting privatisation. This spin-off solution has lower management overhead and less memory utilisation than previous approaches.

The analysed and designed extensions for the approaches and required sub-systems have been integrated in the framework of the Rochester University which contains the original implementation of NOrec as well. The helper-thread aided CCs have been evaluated against the NOrec implementation and a simplified sandboxing variant on a modern 64 bit multi-core architecture using the benchmarks of the Stanford Transactional Applications for Multi-processing (STAMP). The measurement results have proven that the parallel execution of a transaction for the purpose of a delegated periodic validation can be beneficial. In almost all tested scenarios, the cloned incremental validation outperformed the repeated validation of

the leader's read-set in the helper thread, due to the reduced contention. In comparison to single-threaded approaches it competes and outperforms them in higher contention scenarios with small and medium sized read-sets only. In contrast, the parallel execution with commit in the helper thread cannot provide any further improvements over the cloned incremental validation, whether in synchronous or asynchronous mode.

The general disadvantage of the helper-thread aided transactions observed in the evaluation led to the assumption that additional concurrency has much higher impact on the response time than originally expected. Unfortunately, it was not possible to increase the granularity of the measurements to get more detailed information about the reasons. Instead, the capabilities of the given test platform in regards to data access latency in highly concurrent applications have been evaluated. This investigation revealed some interesting behaviour. According to the results, the multi-core architecture has serious issues with concurrent access to shared data. The cache coherency protocol seems to serialise access to shared cache lines at some points, which results in an access latency linearly rising with the number of concurrently accessing cores. Atomic instructions perform even worse. Even though atomic instructions are known to cause scalability issues, the effect is excessively high in concurrency intensive scenarios. Further investigation showed that the cache hierarchy does not consider physically sharing of cache lines in shared caches of neighbouring cores. This was an expected feature, especially on the given hardware, which considers two neighbouring cores of a so-called Bulldozer module as cooperating units.

According to the observed behaviour and the knowledge gained about the cache coherency protocol the given architecture is more suitable to run loosely coupled processes, not sharing too much data, rather than highly concurrent multi-threaded shared memory applications. Because the realisation of an efficient cache coherency protocol for high concurrency is very difficult and the fact that other processor vendors trust in the same cache coherency protocol concept (MOESI), we assume that this behaviour will be observed on other modern multi- and many-core architectures, too.

In comparison to the traditional mutual exclusion pattern, transactional memory especially aims to increase concurrency. Consequently, current multi-core architectures are in conflict with the use of transactional memory and parallelised transactions even more. This is also demonstrated in the evaluation by comparison to a simple common global lock CC which outperforms all the other concurrency control mechanisms in the majority of all cases. Consequently, it needs special hardware to run advanced concurrency control mechanisms in respect to software transactional memory to accelerate beyond disjoint parallel access as practised through mutual exclusion.

The approaches developed in this thesis have to be considered as intermediate stages towards a concurrency control using parallelised transactions in terms of a speculative concurrency

control to be applied in STM. The results of this thesis in regards to constraints on the instrumentation of transactional sections and required sub-systems build a first basis for further research. The degree of speculative execution in respect to different serialisation orders in these approaches is still low. Concurrency control schemes using direct updates even combined with multi-versioning allow more progressive speculation because future serialisation orders are visible to other transactions during their execution and long before commit time. This allows a parallelised transaction to utilise a helper thread with the execution of a transaction in respect to a concurrent commit, which is expected to occur in the future. Thereby it provides an alternative result for its own transaction, which could immediately commit in case the leader is in conflict with exactly this transaction. Of course, direct updates and multi-versioning require the introduction of shared metadata (ownership records), which are known to reduce the scalability, but there might be a point of break even between advantage through progressive speculative execution and additional contention on metadata.

We can also imagine a different approach derived from our current approaches, which allows progressive speculative execution with less concurrency on shared metadata. Instead of globally sharing the versions of shared data, helper threads can be controlled to follow a particular concurrently running transaction which is assumed to generate a future conflict. The write-set of a transaction reflects the future state of shared data after the commit of the transaction. Thus, the helper thread can execute a transaction considering this future state in its reads and during validation and thereby potentially provide a commit, which does not conflict with the foreign transaction. The contention induced is lower in this case because the data is just shared between the helper thread and the foreign transaction.

The lack of appropriate hardware is an issue to be considered solved in the future. As already mentioned, the complexity of software will keep rising and the demand of processing power will further increase, too. Unless some astonishing advances in the field of computing will take place (e.g. quantum computing), parallelisation with increased concurrency will be the answer to the current dilemma. Certainly, concepts or even implementations for highly concurrent multi-processing architectures already exist, but the demand in the end-user domain is still too low to allow for a profitable production. However, research has to step ahead and prepare solutions for future demand, and the potential of parallelised transactions is too promising to give up researching on, unless it was finally proven to fail in the general case.

Bibliography

[Adv11] Advanced Micro Devices Inc. *AMD64 Technology - AMD64 Architecture Programmer's Manual - Volume 2: System Programming*. Revision 3.23. Advanced Micro Devices Inc., May 2011.

[ATSG12] Ali-Reza Adl-Tabatabai, Tatiana Shpeisman, and Justin Gottschlich. Draft Specification of Transactional Language Constructs for C++, Version 1.1, Feb. 2012.

[BB93] Azer Bestavros and Spyridon Braoudakis. SCC-nS: A family of Speculative Concurrency Control Algorithms for Real-Time Databases. In *Proceedings of the 3rd Intl. Workshop on Responsive Computer Systems*, 1993.

[BBKO10] Woongki Baek, Nathan Bronson, Christos Kozyrakis, and Kunle Olukotun. Implementing and Evaluating Nested Parallel Transactions in Software Transactional Memory. In *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA 2010, pages 253–262, Thira, Santorini, Greece, New York, NY, USA, 2010. ACM.

[Bes93] Azer Bestavros. Speculative Concurrency Control. In *Proceedings of the 21st VLDB*, pages 122–133, 1993.

[BW93] Azer Bestavros and Biao Wang. Multi-version Speculative Concurrency Control with Delayed Commit. Technical report, Boston University Computer Science Department, 1993.

[CYD$^+$10] Jaewoong Chung, Luke Yen, Stephan Diestelhorst, Martin Pohlack, Michael Hohmuth, David Christie, and Dan Grossman. ASF: AMD64 Extension for Lock-Free Data Structures and Transactional Memory. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 39–50, Washington, DC, USA, 2010. IEEE Computer Society.

[DLMN09] Dave Dice, Yossi Lev, Mark Moir, and Daniel Nussbaum. Early Experience with a Commercial Hardware Transactional Memory Implementation. *SIGARCH Comput. Archit. News*, 37(1):157–168, March 2009.

[DS06] Dave Dice and Nir Shavit. What Really Makes Transactions Faster? In *TRANSACT Workshop*, 2006.

[DS12] Luke Dalessandro and Michael L. Scott. Sandboxing Transactional Memory. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT 2012, pages 171–180, Minneapolis, Minnesota, USA, New York, NY, USA, 2012. ACM.

[DSS06] Dave Dice, Ori Shalev, and Nir Shavit. Transactional Locking II. In *Distributed Computing*, volume 4167 of *Lecture Notes in Computer Science*, pages 194–208. Springer Berlin Heidelberg, 2006.

[DSS10] Luke Dalessandro, Michael F. Spear, and Michael L. Scott. NOrec: Streamlining STM by Abolishing Ownership Records. *SIGPLAN Not.*, 45(5):67–78, January 2010.

[FC11] Sérgio Miguel Fernandes and João Cachopo. Lock-free and Scalable Multi-version Software Transactional Memory. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 179–188, San Antonio, TX, USA, New York, NY, USA, 2011. ACM.

[Fra03] Keir Fraser. *Practical Lock-Freedom*. PhD thesis, PhD thesis, University of Cambridge Computer Laboratory, 2003.

[GK08] Rachid Guerraoui and Michal Kapalka. On the Correctness of Transactional Memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 175–184, Salt Lake City, UT, USA, New York, NY, USA, 2008. ACM.

[Gra78] J.N. Gray. Notes on Data Base Operating Systems. In R. Bayer, R.M. Graham, and G. Seegmüller, editors, *Operating Systems*, volume 60 of *Lecture Notes in Computer Science*, pages 393–481. Springer Berlin Heidelberg, 1978.

[Gra81] Jim Gray. The Transaction Concept: Virtues and Limitations (Invited Paper). In *Proceedings of the Seventh International Conference on Very Large Data Bases - Volume 7*, VLDB '81, pages 144–154, Cannes, France. VLDB Endowment, 1981.

[HLMS03] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer. Software Transactional Memory for Dynamic-sized Data Structures. In *Proceedings of the Twenty-second Annual Symposium on Principles of Distributed Computing*, PODC '03, pages 92–101, Boston, Massachusetts, New York, NY, USA, 2003. ACM.

[HLR10] Tim Harris, James Larus, and Ravi Rajwar. *Transactional Memory*. Morgan and Claypool Publishers, 2nd edition, 2010.

[HM93] Maurice Herlihy and J. Eliot B. Moss. Transactional Memory: Architectural Support for Lock-free Data Structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, May 1993.

[HR83] Theo Haerder and Andreas Reuter. Principles of Transaction-oriented Database Recovery. *ACM Comput. Surv.*, 15(4):287–317, December 1983.

[HS12] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Elsevier, 2012.

[HSATH06] Richard L. Hudson, Bratin Saha, Ali-Reza Adl-Tabatabai, and Benjamin C. Hertzberg. McRT-Malloc: A Scalable Transactional Memory Allocator. In *Proceedings of the 5th International Symposium on Memory Management*, ISMM '06, pages 74–83, Ottawa, Ontario, Canada, New York, NY, USA, 2006. ACM.

[HT11] Ruud Haring and B Team. The Blue Gene/Q Compute Chip. In *The 23rd Symposium on High Performance Chips (Hot Chips)*, volume 4, pages 125–180, 2011.

[HW90] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[Int08] Intel®Corporation. Intel®Transactional Memory Compiler and Runtime Application Binary Interface. Revision: 1.0.1, November 2008.

[KCH+06] Sanjeev Kumar, Michael Chu, Christopher J. Hughes, Partha Kundu, and Anthony Nguyen. Hybrid Transactional Memory. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, pages 209–220, New York, New York, USA, New York, NY, USA, 2006. ACM.

[KGH+11] Gokcen Kestor, Roberto Gioiosa, Tim Harris, Osman S. Unsal, Adrian Cristal Cristal, Ibrahim Hur, and Mateo Valero. STM2: A Parallel STM for High Performance Simultaneous Multithreading Systems. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 221–231, Oct 2011.

[Lam79] Leslie Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *Computers, IEEE Transactions on*, C-28(9):690–691, Sept 1979.

[Lau09] Georg Lausen. Two-Phase Locking. In *Encyclopedia of Database Systems*, pages 3214–3218. Springer, 2009.

[Lee61] C. Y. Lee. An Algorithm for Path Connections and Its Applications. *Electronic Computers, IRE Transactions on*, EC-10(3):346–365, Sept 1961.

[LLM+09] Yossi Lev, Victor Luchangco, Virendra Marathe, Mark Moir, Dan Nussbaum, and Marek Olszewski. Anatomy of a Scalable Software Transactional Memory. In *Proc. 4th ACM SIGPLAN Workshop on Transactional Computing*, 2009.

[MCKO08] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 35–46. IEEE, 2008.

[ME08] Jan Christian Meyer and Anne C Elster. Latency Impact on Spin-Lock Algorithms for Modern Shared Memory Multiprocessors. In *Complex, Intelligent and Software Intensive Systems, 2008. CISIS 2008. International Conference on*, pages 786–791. IEEE, 2008.

[MHJM13] M. Matz, J. Hubicka, A. Jaeger, and A. Mitchell. System V Application Binary Interface – AMD64 Architecture Processor Supplement. Draft Version 0.99.6, 2013.

[Mos81] J. Eliot B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. PhD thesis, Massachusetts Institute of Technology, April 1981.

[MS98] Maged M. Michael and Michael L. Scott. Nonblocking Algorithms and Preemption-Safe Locking on Multiprogrammed Shared Memory Multiprocessors. *Journal of Parallel and Distributed Computing*, 51(1):1 – 26, 1998.

[MSS05] Virendra J. Marathe, William N. Scherer, and Michael L. Scott. Adaptive Software Transactional Memory. In Pierre Fraigniaud, editor, *Distributed Computing*, volume 3724 of *Lecture Notes in Computer Science*, pages 354–368. Springer Berlin Heidelberg, 2005.

[MT13] Holger Machens and Volker Turau. Opacity of Memory Management in Software Transactional Memory. Technical Report No. arXiv:1308.2881, arXiv.org e-Print archive, Cornell University, August 2013.

[NMAT+07] Yang Ni, Vijay S. Menon, Ali-Reza Adl-Tabatabai, Antony L. Hosking, Richard L. Hudson, J. Eliot B. Moss, Bratin Saha, and Tatiana Shpeisman. Open Nesting in Software Transactional Memory. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '07, pages 68–78, San Jose, California, USA, New York, NY, USA, 2007. ACM.

[Pap79] Christos H. Papadimitriou. The Serializability of Concurrent Database Updates. *J. ACM*, 26(4):631–653, October 1979.

[SMSS06] Michael F. Spear, Virendra J. Marathe, William N. Scherer, and Michael L. Scott. Conflict Detection and Validation Strategies for Software Transactional Memory. In Shlomi Dolev, editor, *Distributed Computing*, volume 4167 of *Lecture Notes in Computer Science*, pages 179–193. Springer Berlin Heidelberg, 2006.

[SS86] Paul Sweazey and Alan Jay Smith. A Class of Compatible Cache Consistency Protocols and their Support by the IEEE Futurebus. In *ACM SIGARCH Computer Architecture News*, volume 14, pages 414–423. IEEE Computer Society Press, 1986.

[SS05] William N. Scherer and Michael L. Scott. Advanced Contention Management for Dynamic Software Transactional Memory. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Principles of Distributed Computing*, PODC '05, pages 240–248, Las Vegas, NV, USA, New York, NY, USA, 2005. ACM.

[ST97] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

[Tan84] Andrew S. Tanenbaum. *Structured Computer Organization*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 1984.

[Tan92] Andrew S. Tanenbaum. *Modern operating systems*, volume 2. Prentice hall Englewood Cliffs, 1992.

[TRS12] Alexandru Turcu, Binoy Ravindran, and Mohamed M Saad. On Closed Nesting in Distributed Transactional Memory. In *Seventh ACM SIGPLAN workshop on Transactional Computing*, 2012.

[WCW+07] Cheng Wang, Wei-Yu Chen, Youfeng Wu, B. Saha, and Ali-Reza Adl-Tabatabai. Code Generation and Optimization for Transactional Memory Constructs in an Unmanaged Language. In *International Symposium on Code Generation and Optimization, 2007. CGO '07.*, pages 34–48, 2007.

[WGW+12] Amy Wang, Matthew Gaudet, Peng Wu, José Nelson Amaral, Martin Ohmacht, Christopher Barton, Raul Silvera, and Maged Michael. Evaluation of Blue Gene/Q Hardware Support for Transactional Memories. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 127–136. ACM, 2012.

[WRFF10] Jons-Tobias Wamhoff, Torvald Riegel, Christof Fetzer, and Pascal Felber. RobuSTM: A Robust Software Transactional Memory. In *Proceedings of the 12th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS'10, pages 388–404, NewYork, NY, USA, Berlin, Heidelberg, 2010. Springer-Verlag.

[WS08] M.M. Waliullah and Per Stenström. Intermediate Checkpointing with Conflicting Access Prediction in Transactional Memory Systems. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–11, April 2008.

[YHLR13] Richard M. Yoo, Christopher J. Hughes, Konrad Lai, and Ravi Rajwar. Performance Evaluation of Intel®Transactional Synchronization Extensions for High-performance Computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 19:1–19:11, Denver, Colorado, New York, NY, USA, 2013. ACM.

Index

Symbols

2PL 7

A

ABA-problem 23
ABI 9
abort 14
application binary interface 9
atomic 11, 14

B

base pointer 8
BP 8
bus lock 11, 111
byte-based 24

C

cache lines 9
cache lock 11, 111
cache-coherency protocol 10
CAS 11
cascading rollbacks 7
CC 6
CGL 22
checkpoint 26
child transaction 26
closed nesting 26
commit 14
common global lock 22
compare and set 11
compensating action 26
compiler barriers 10
compiler fences 10
concurrency control 6
concurrent 5
concurrent objects 15
consistency model 10
consistent 13
contention management 24
context switch 8
critical sections 6

D

deferred update 7
direct update 7
durable 14

E

eager validation 23

F

flat nesting 26

G

global version clock 23

H

hardware transactional memory 16
HTM 16
hybrid transactional memory 17
HyTM 17

I
incremental validation 23
instruction pointer 8
inter-process communication 5
IPC 5
irrevocable 15
isolated 14

L
lazy validation 23
linear nesting 26
linearisability 15
load fence 10
lock stealing 22
locking 22

M
machine state 8
memory fence 10
memory management unit 9
memory-ordering model 10
message passing 5
MMU 9
multiversioning 22

N
nesting 25

O
object-based 23
OCC 7
opacity 23
open nesting 26
optimistic concurrency control 7
orec 22
ownership records 22

P
page table 9
parallel 5
parent transaction 26
PCC 7
pessimistic concurrency control 7
privatisation 46
program counter 8
publication 46

Q
quiescence 52

R
read-set 14
rollback 7, 14

S
sandboxing 23
SCC 31
sequential consistency 10
serialisability 15
serialisation order 28
shared memory 6
single global lock 22
software transactional memory 17
SP 8
speculative concurrency control 31
stack frames 8
stack pointer 8
static transactions 22
STM 17
store fence 10

T
task 5
thread-level speculation 28

thread-local data 9
thread-specific data 9
timer-based 22
timer-triggered validation 23
TLB 9
TM 13
transaction queuing 22
transactional memory 13
translation lookaside buffer 9
two phase locking 7

V

validate 22
validation 15

W

waivered code 49
word-based 24
write-set 14

Appendix A

C++ Transactional Memory Application Example

This appendix provides a full application example for the transactional language constructs for C++ introduced in Section 3.4.1. The application implements a solution to the dining philosopher problem. It describes a group of philosophers sitting around a round table while thinking and eating, alternately. Each philosopher has his own plate and needs two forks to eat, one from the left side and the other from the right side of his plate. Unfortunately the forks are shared with the left and the right neighbour of each philosopher respectively. Thus, if the left or the right neighbour is already eating, the philosopher has time to think instead. Each philosopher follows a simple algorithm:

1. Think until the left fork is available and grab it.
2. Resume thinking until the right fork is available and grab it as well.
3. Eat.
4. Put both forks back on the table.
5. Restart from the beginning.

The forks represent shared resources in this classic example for a possible deadlock condition. It is very likely that the philosophers run in a condition where every philosopher has a fork in his left hand and keeps thinking forever. The Listing A.1 implements a solution for ten philosophers with the following difference: A philosopher thinks until *both forks* are available, then he eats, puts the forks back and finally starts over from the beginning.

```
/* Create a thread which calls 'thread_entry' with given 'argument' */
void start_thread(void(*thread_entry)(int argument));

const int FORK_AVAILABLE = 0;
const int NUM_PHILOSOPHERS = 10
int forks[NUM_PHILOSOPHERS];

[[transaction_safe]] void grab(int fork, int philosopher) {
   forks[fork] = philosopher;
}
[[transaction_safe]] void put(int fork) {
   forks[fork] = FORK_AVAILABLE;
}
[[transaction_safe]] int available(int fork) {
   return (forks[fork] == FORK_AVAILABLE);
}

void philosopher_do (int philosopher)
   int left_fork = philosopher % NUM_PHILOSOPHERS;
   int right_fork = (philosopher+1) % NUM_PHILOSOPHERS;
   int can_eat;

   for (int i = 0; i < NUM_ITERATIONS; i++) {
      can_eat = false;
      while (!can_eat) {
         think();
         __transaction_atomic {
            if (available(left_fork) && available(right_fork)) {
               can_eat = true;
               grab(left_fork, philosopher);
               grab(right_fork, philosopher);
            }
         }
      }
      eat();
      __transaction_atomic {
         put(right_fork);
         put(left_fork);
      }
   }
}

int main (int argc, char** argv) {
   memset(forks, FORK_AVAILABLE, sizeof(forks));
   for (int philosopher = 1; philosopher <= NUM_PHILOSOPHERS; philosopher++) {
      start_thread(philosopher_do, philosopher);
   }
}
```

Listing A.1: Dining Philosophers using transactions

Obviously, mutual exclusion on both critical sections (lines 27 to 33 and 36 to 39) will allow the philosophers to think and eat alternately without starving. The philosopher grabs the forks if both are available and no one can interfere until he puts them back. Using transactions (as we do here) multiple threads may access the first critical section at the same time and try to grab the forks next to them. But as soon as one of the transactions is committed, all transactions that tried to grab one of the forks now occupied will get aware of the conflict and abort. All modification they did, such as the note to themselves to be ready to eat, will be reverted. They will automatically restart from the beginning of the critical section (line 27) and the condition in line 28 will no longer hold, which forces them to resume thinking.

To showcase some pitfalls of transactions we will discuss another transactional variant, shown in Listing A.2. Here the entire procedure of eating with getting and returning of the forks is combined in one large critical section. With mutual exclusion all philosophers are forced to wait blocking and unable to think, while one philosopher is eating. With transactions (considering direct updates) at least the philosophers that do not need one of the forks occupied by the eating philosopher, will be able to eat concurrently. But the conflicting transactions will still not be able to think concurrently. Assuming that we have intended such behaviour, it implicitly improves concurrency of the application.

```
1  [[transaction_safe]] void eat();
2
3  void philosopher_do (int philosopher)
4     int left_fork = philosopher % NUM_PHILOSOPHERS;
5     int right_fork = (philosopher+1) % NUM_PHILOSOPHERS;
6     int eaten;
7
8     for (int i = 0; i < NUM_ITERATIONS; i++) {
9        eaten = false;
10       while (!eaten) {
11          think();
12          __transaction_atomic {
13             if (available(left_fork) && available(right_fork)) {
14                grab(left_fork, philosopher);
15                grab(right_fork, philosopher);
16                eat();
17                eaten = true;
18                put(right_fork);
19                put(left_fork);
20             }
21          }
22       }
23    }
24 }
```

Listing A.2: Dining Philosophers using transactions

Unfortunately, the behaviour of this implementation depends on the type of CC in use, something that should be transparent to the developer. A CC with deferred update will never generate a conflict with other transactions in this case: Any fork occupied for eating, is returned to the table before the commit takes place. As a result, the forks appear to be available for the whole time and other philosophers can eat concurrently ignoring the virtually occupied forks. Considering the data output of the critical section, there will be no difference to mutual exclusion and thus the behaviour is actually legal in terms of the application code, but the serialisation on the forks is lost. However, a directly updating CC will not have those issues.

The actual difference shown by the application with either mutual exclusion or transactions using deferred updates is related to an intended temporal relationship, which is not respected by transactions. In the latter example transactions run in parallel while we actually wanted them to be serialised when using the same forks. Consequently, the philosophers would spend less time in thinking as they do with mutual exclusion, which can be a significant difference. The forks are used to control whether a philosopher has to think or can eat. This is hidden in the transformations inside the transactions. The constraints for the transaction whether it is serialisability or linearisability simply require producing an output that is similar to that with mutual exclusion. Temporal relationships are unknown to transactions and the intermediate states inside the transaction are considered to be irrelevant for the application.

But we can translate temporal dependencies into transaction visible relationships. For example every fork could get a usage counter to be incremented by the philosopher using it. This usage counter defines the use of a fork (for eating) as an application significant transformation. Transactions resulting in a modification of the counter are consequently in a causal relationship. In the current example the use appears to be insignificant because the modification is reverted before the transaction finishs.

Another solution is given in the first example: We identify significant intermediate states of the application, which are relevant elements of the intended runtime behaviour and make sure that those get visible to all threads by a commit of the transaction. In other words: transactions are not allowed to span across those important states, to force a commit before the transformation to the next important state takes place.

Appendix B

Average Conflict Position

This appendix provides a very rough approximation for the average location of the first conflict between k concurrent tasks executing transactions. The approximation is based on a very simplified model and not intended to be accurate because we just want a rough approximation of the behaviour of transactions in highly concurrent applications.

A task of a concurrent application alternately executes transactional and non-transactional sections of arbitrary length. Consequently, the code computed by an average task of an average concurrent application can be modelled as a loop over a section of n instructions which contains a transaction of average length followed by a non-transactional section of average length. All instructions inside the transaction can conflict with other transactions.

Considering one concurrent transaction, it may cause one possible conflict to occur on any instruction inside the transaction. This is a conservative consideration because one concurrent task may cause multiple conflicts too. Thus, the approximation developed here represents just a lower bound.

Because all instructions have the same probability the average position of a conflict will be in the middle of the regarded interval, which is the arithmetic mean of all positions. The transactions of both tasks may or may not overlap, which means the conflict may or may not occur. We model this case by considering even the instructions of the non-transactional section as possible positions for the conflict. This will be enough to approximate the behaviour of the first position of a conflict in high contention scenarios.

Considering a third concurrent task, it will cause another possible conflict which is randomly distributed over the whole interval again. The transaction has to recompute the section in forward direction beginning at least at the first conflict. Thus, the question is: Which one of

multiple randomly distributed conflicts which have been occurred in the interval, is the first one in forward direction.

To approximate the position of the first conflict in n instructions out of k randomly distributed conflicts caused by k transactions, we can apply the following transformation of the problem. Each position in the observed section may be represented by digits of the interval $[1..n]$. A sequence of k digits represents a set of conflicts caused by k concurrent tasks. This sequence of digits may be alternatively interpreted as a number of the positional numeral system with a base of n where the smallest digit represents the position of the first conflict.

To determine the expectation for the smallest digit we will first calculate the amount of numbers (variations), where $x \in [1..n]$ represents the smallest digit.

If the smallest digit x is located on the first position, the remaining $k-1$ digits must have values greater or equal to x. Thus, they may have one out of $n-x+1$ possible values.

Position:	1	2	...	$k-1$	k
Possible values:	1	$\cdot(n-x+1)$	...	$\cdot(n-x+1)$	$\cdot(n-x+1)$

The product of possible values for the digits equals the amount of variations, where x on the first position represents the smallest digit.

$$V_1(x,k,n) = 1 \cdot (n-x+1)^{(k-1)} \tag{B.1}$$

The determination of the amount of variations for the smallest digit x on the second position is analogously:

Position:	1	2	...	$k-1$	k
Possible values:	$(n-x)$	$\cdot 1$	...	$\cdot(n-x+1)$	$\cdot(n-x+1)$

Now, the position in front of the smallest digit (first position here) can have values greater or equal to x only, which equals $n-x$ possible values.

$$V_2(x,k,n) = (n-x)^{(2-1)} \cdot 1 \cdot (n-x+1)^{(k-2)} \tag{B.2}$$

The sum of those products for all positions produce the quantity of all variations which contain x as the smallest digit at arbitrary position.

$$V(x,k,n) = \sum_{i=1}^{k} (n-x)^{(i-1)} \cdot (n-x+1)^{(k-i)} \text{ for } x < n \tag{B.3}$$

This expressions is undefined for the case where $x = n$ and $i = 1$, because the first term results in 0^0. But we can easily see that just 1 variation exists for $x = n$.

The expression above can be transformed into:

$$V(x,k,n) = (n-x+1)^k - (n-x)^k \tag{B.4}$$

To get the expectation $E(k,n)$ for x, out of all possible variations for k digits out of n values, we calculate the sum of all $V(x,k,n)$ weighted by x and divide it by the quantity of all possible variations for all x.

$$E(k,n) = \frac{\sum_{x=1}^{n} x \cdot V(x,k,n)}{n^k} \tag{B.5}$$

$$E(k,n) = \frac{(\sum_{x=1}^{n-1} x(n-x+1)^k - x(n-x)^k) + 1 \cdot n}{n^k} \tag{B.6}$$

For the variation with $x = n$ we add $1 \cdot n$ to the sum.

An analysis showed that this expression can be approximated for $n \in [1, \infty]$ and $k \in [1, n]$ with the following simple expression:

$$E(k,n) \approx \frac{n}{k+1} \quad ; \quad \epsilon \approx -(0,085/n + 1/2) \tag{B.7}$$

This expression roughly approximates the expectation for the upper bound of the first conflicting position in a transaction for k tasks considering alternating transactional and non-transactional sections with a combined average length of n instructions. The error of $\epsilon \approx -(0,085/n + 1/2)$ represents a deviation of less than one instruction.

The approximation clearly shows that the first of multiple simultaneously occurring conflicts moves very fast towards the begin of the transaction when the number of concurrent tasks is increased.

Appendix C

Raw Measurement Results

The raw measurement results of the evaluation and the hardware investigation can be downloaded from `http://www.ti5.tu-harburg.de/research/pmca/ct/` (see Section D). Only the response times for the single-threaded operation of the STAMP benchmarks have been added here, summarised for comparison in Table C.1.

Benchmark	CGL	NOrec	NOrecSb	NOrecHt	NOrecCIV	NOrecCs	NOrecCa
genome	21170	33301	48164	58770	59594	54821	53654
intruder	33764	42942	43070	57609	55346	56048	77679
kmeans	6242	7430	7667	9060	9140	9221	13199
labyrinth	237253	1026480	1046840	1351986	1419542	1219589	904390
ssca2	21272	22859	22897	24012	23800	23861	36663
vacation	40770	58697	60885	87123	83738	88326	88775

Table C.1.: Response times in single-threaded operation

Appendix D

Source Code and Raw Measurement Results

All results of this thesis are available as a package that can be downloaded from the server of the Hamburg University of Technology at this location: `http://www.ti5.tu-harburg.de/research/pmca/ct/`. The package contains source code for all implemented STM CCs, benchmarks, evaluation frameworks, build and evaluation scripts, evaluation results and the thesis. The folders in the package contain README files with further information on the given content and how to use it.

The package content is roughly sorted by the following top-level folders:

- **`src`**: All sources of software used in this thesis.
- **`eval`**: All results from the evaluation of the STM CCs (Section 6) and the investigation of the hardware influences (Section 7) as well as the evaluation scripts or environments used to measure them.
- **`thesis`**: The source code of the thesis and a digital copy (PDF).

Please note that all README files use the UTF-8 character encoding. On Windows operating systems you can use the `wordpad.exe` application for proper visualisation (in contrast `notepad.exe` cannot show its content properly).