Université de Neuchâtel
Faculté des Sciences
Institut d'Informatique

Software Transactional Memory for Distributed and Event-based Systems

par

Heiko Sturzrehm

Thèse

présentée à la Faculté des Sciences
pour l'obtention du grade de Docteur ès Sciences

Acceptée sur proposition du jury:

Prof. Pascal Felber, directeur de thèse
Université de Neuchâtel, Suisse

Prof. Christof Fetzer, rapporteur
Technische Universität Dresden, Germany

Prof. Rachid Guerraoui, rapporteur
Ecole Polytechnique Fédérale de Lausanne, Suisse

Prof. Peter Kropf, rapporteur
Université de Neuchâtel, Suisse

Soutenue le 09 Juin 2010

Université de Neuchâtel unine

FACULTE DES SCIENCES
Secrétariat-Décanat de la faculté
- Rue Emile-Argand 11
- CP 158
- CH-2009 Neuchâtel

IMPRIMATUR POUR LA THESE

Software Transactional Memory for Distributed and Event-based Systems

Heiko STURZREHM

UNIVERSITE DE NEUCHATEL

FACULTE DES SCIENCES

La Faculté des Sciences de l'Université de Neuchâtel,
sur le rapport des membres du jury

MM. P. Felber (directeur de thèse), P. Kropf,
C. Fetzer (Technische Universität Dresden, D)
et R. Guerraoui (EPF Lausanne)

autorise l'impression de la présente thèse.

Neuchâtel, le 8 juillet 2010

Le doyen :
F. Kessler

UNIVERSITE DE NEUCHATEL
FACULTE DES SCIENCES
Secrétariat - décanat de la faculté
Rue Emile-Argand 11 - CP 158
CH-2009 Neuchâtel
felix Kessler

"If I have seen further than others, it is by standing upon the shoulders of giants."

Isaac Newton

Abstract

Keywords: Software Transactional Memory, Design Patterns, Event Stream Processing, Parallel Processing, Distributed Systems

With the availability of end-user multi-core systems, new concurrency control systems have been developed. One of them, software transactional memory (STM) uses optimistic execution of code within transactions. In this thesis, we propose the application of the STM technology with event stream processing (ESP). In ESP applications, events flow through a network of components that perform various types of operations, e.g., filtering, aggregation, transformation.

Our approach, simplifies the parallelization of stateful components by reducing the complex and error-prone parallel programing to a minimum. Additionally, we present a directed distributed STM, which can extend the transactions from one component to the attached downstream components. To underline the advantages of our approaches and developments, we depict several use cases as well as a basic guideline for them.

Furthermore, we present our work in building a lightweight STM support for the unmanaged programming language `C` based on LLVM. The resulting framework can produce highly efficient STM-based applications without letting the programmer deal with too much parallelization aspects. The system is very flexible and can be used with a large group of `C`-based STMs.

In the third aspect of this thesis we analyze the STM approach for solving concurrency problems and propose a "Transactional Object" design pattern. With this proposal we want to support application designers and STM programmers with a blueprint to simplify their work.

Résumé

Mots-Clés: Software Transactional Memory, Design Patterns, Event Stream Processing, Parallel Processing, Distributed Systems

Avec la disponibilité de systèmes multi-cœur, de nouveaux mécanismes de contrôle de concurrence ont été développés. Un de ces mécanismes, la mémoire transactionnelle logicielle (STM), utilise une approche optimise pour l'exécution du fragment de code à l'intérieur d'une transaction. Dans cette thèse, nous proposons l'application de la technologie STM conjointement avec les traitements de flux d'événements (ESP). Dans des applications ESP, les événements fluent à travers un réseau de composantes qui exécutent différents types d'opérations sur les transactions, comme par exemple du filtrage, de l'agrégation ou des transformations.

Notre approche simplifie la parallélisation des composantes avec état, en réduisant la complexité ainsi que la sensibilité aux erreurs de la programmation parallèle. En plus, nous proposons une STM distribuée directionnelle qui permet de passer des transactions d'une composante aux composantes en aval attachées. Pour souligner les avantages de nos approches et développements, nous présentons plusieurs scénarios d'utilisation. En outre, nous présentons notre travail de recherche, visant à construire un support STM peu pesant pour le langage de programmation C en utilisant LLVM. Le framework finalement obtenu permet de produire des applications extrêmement efficientes basées sur des STM, sans que le développeur doive s'occuper trop des aspects de la programmation parallèle. Le système est très flexible et peut être appliqué à un grand nombre d'STMs basées sur C.

Finalement, dans le troisième aspect de cette thèse, nous analysons l'approche des STMs pour résoudre des problèmes de concurrence et nous proposons le motif de conception « Transactional Object ». Le but de cette proposition est de présenter aux designers d'applications ainsi qu'aux programmeurs un plan détaillé pour développer et travailler avec des STMs.

Kurzfassung

Schlüsselwörter: Software Transactional Memory, Design Patterns, Event Stream Processing, Parallel Processing, Distributed Systems

Die breite Verfügbarkeit von Mehrkernprozessoren führte zur Entwicklung neuer Nebenläufigkeitskontrollen. Eine dieser neuen Techniken ist Software Transactional Memory (STM). Diese erlaubt es Programmteile in atomarer Form auszuführen. In dieser Arbeit, schlagen wir vor diese Technologie im Event-Stream-Processing (ESP) einzusetzen. ESP-Anwendungen werden benutzt um auf Ereignisströmen verschiedene Operationen auszuführen, z.B. sortieren, zusammenfassen oder verändern. Die Verarbeitung kann dabei in einem Netzwerk von Komponenten durchgeführt werden.

Unser Ansatz vereinfacht die Parallelisierung von zustandsbehafteten Komponenten durch die Reduzierung der Komplexität und die Vereinfachung der parallelen Programmierung. Außerdem präsentieren wir einen verteilten STM-Ansatz, bei dem die Gültigkeit einer Transaktion von einer Verarbeitungskomponente auf die Folgenden ausgedehnt werden kann. Um die Vorteile unserer Ansätze zu unterstreichen, zeigen wir weiterhin mögliche Anwendungsfälle und einen Leitfaden zur Benutzung.

In dieser Arbeit stellen wir zudem ein System zur vereinfachten Benutzung von STM in der Programmiersprache C vor. Es ermöglicht eine einfache und effiziente Nutzung von STMs ohne das der Programmierer zu viel Arbeit in die Parallelisierung seiner Anwendung stecken muss. Dadurch dass unser System modular aufgebaut ist, kann es durch eine Vielzahl von unterschiedlichen C-basierten STMs genutzt werden.

Im dritten Aspekt dieser Arbeit vergleichen wir unterschiedliche STM-Umsetzungen, aus denen wir das Entwurfsmuster "Transactional Object" abgeleitet haben. Dieser Vorschlag soll sowohl die Erstellung von weiteren STMs durch eine Blaupause vereinfachen als auch das Verständnis der Nutzer für diese Technologie verbessern.

Acknowledgements

This thesis would not be possible without the support of a large number of people who have helped me both in big ways and little.

In particular, I would like to thank my adviser, Prof. Pascal Felber, for his support, patience and respect. By being demanding, understanding, and allowing me the freedom to explore my interests, he has driven me to succeed. Furthermore, I would like to acknowledge and thank the *Swiss National Science Foundation* [Swi, MIC] which supported my work under grant #5005-67322 (NCCR-MICS).

My gratitude goes also to Prof. Christof Fetzer (*Technische Universität Dresden*), Prof. Rachid Guerraoui (*Ecole Polytechnique Fédérale de Lausanne*), and Prof. Peter Kropf (*Université de Neuchâtel*) for accepting to examine this thesis.

I would like to thank the whole Computer Science Department of the *Université de Neuchâtel* and the System Engineering Group at the *Technische Universität Dresden* for their inspiring discussions and productive collaborations.

Special thanks go to my friend and colleague Claire Fautsch. She has been an invaluable sounding board for ideas and a welcoming ear to occasional frustrations. Furthermore, I would like to thank Ljiljana Dolamic and Patrick Marlier for the nice time we have spent together during work and beyond.

Last but not least, I want to thank my family for their invaluable support and unconditional love.

Contents

List of Figures

List of Algorithms

List of Tables

Abbreviations

CAS	**C**ompare **A**nd **S**wap
CEDR	**C**omplex **E**vent **D**etection and **R**esponse
CH	**C**onflict **H**andler
CM	**C**ontension **M**anager
CPU	**C**entral **P**rocessing **U**nit
CSP	**C**omplex **S**tream **P**rocessing
distTX	**DIST**ributed **T**ransaction
DSTM	**D**ynamic **S**oftware **T**ransactional **M**emory
ESP	**E**vent **S**tream **P**rocessing
gcc	**G**NU **C**ompiler **C**ollection
GPS	**G**lobal **P**ositioning **S**ystem
GSDM	**G**RID **S**tream **D**ata **M**anager
HTM	**H**ardware **T**ransactional **M**emory
IC	**I**ntegrated **C**ircuit
LLVM	**L**ow **L**evel **V**irtual **M**achine
LL/SC	**L**oad-**L**ink/**S**tore-**C**onditional
MICS	**M**obile **I**nformation and **C**ommunication **S**ystems
NCCR	**N**ational **C**enter of **C**ompetence in **R**esearch
POSA	**P**attern-**O**riented **S**oftware **A**rchitecture
PU	**P**rocessing **U**nit
RFID	**R**adio-**F**requency **ID**entification
SEFAGE	**SE**elbst**FA**hrer**GE**nossenschaft
SOH	**S**hared **O**bject **H**andler
STAMP	**S**tanford **T**ransactional **A**pplications for **M**ulti-**P**rocessing
STM	**S**oftware **T**ransactional **M**emory

TARIFA	**T**rans**A**ctions by assemble**R** **I**nstrumentation **F**r**A**mework
TCP	**T**ransfer **C**ontrol **P**rotocol
TH	**T**ransaction **H**andler
TM	**T**ransactional **M**emory
TO	**T**ransactional **O**bject
UML	**U**nified **M**odeling **L**anguage
WSN	**W**ireless **S**ensor **N**etwork
XML	**E**xtensible **M**arkup **L**anguage

Dedicated to my parents, Gunhild and Henry

and

in memory of my grandparents.

Chapter 1

Introduction

In 1965 Gordon E. Moore [Moo65] observed the trend that every two years the number of transistors (and with them their processing power) doubles. Nowadays, his observation is famous under the name "Moore's Law" (Fig. 1.1). In the first decade after his prediction, it was very easy to realize this forecast, since the used integrated circuits were quite large (around 10 μm).

In the last decade however, the size of the integrated circuits (IC) reached a size (around 40 nm) where they could not become much smaller, due to photolithography. This is a procedure used to print the design of an integrated circuit on a silicon wafer. It works similar to a photo camera. The minimal size of a structure to be printed is defined by the wavelength of the used radiation. This limit was lately reached with the deployment of x-rays for the printing process. With the downsizing of the ICs it was also possible to increase the clock frequency of the ICs. However, since a few years this frequency is not raising anymore due to thermal issues. The ICs in a processor produce to much heat when running at maximal clock frequency, resulting in irreparable damage. In order to keep the processor power increasing at the forecasted level, the industry (*e. g.,* Intel or AMD) started to parallelize their processors.

Since the first release of processors with the x86 architecture (Intel 8086), their layout stayed basically the same. With the introduction of 80386 design the basic concept was extended with caches. This is a fast addressable memory located in the chip. Later on each processor had several caches and one core, as shown in Figure 1.2(a). The core is the part of the processor where the actual processing happens. With the introduction of the first multi-core processors this internal layout changed. As the name implies each processor contains now several cores, which can process data independently from each other. Furthermore, each core has its own L1 cache which is only accessed by itself.

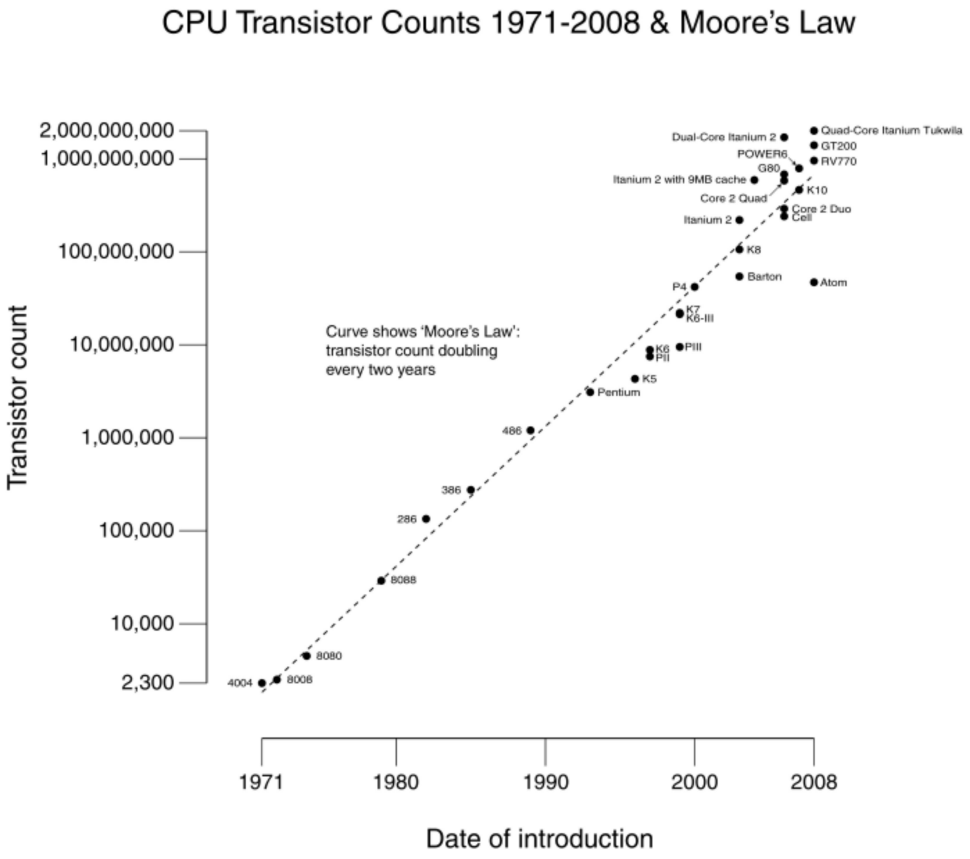
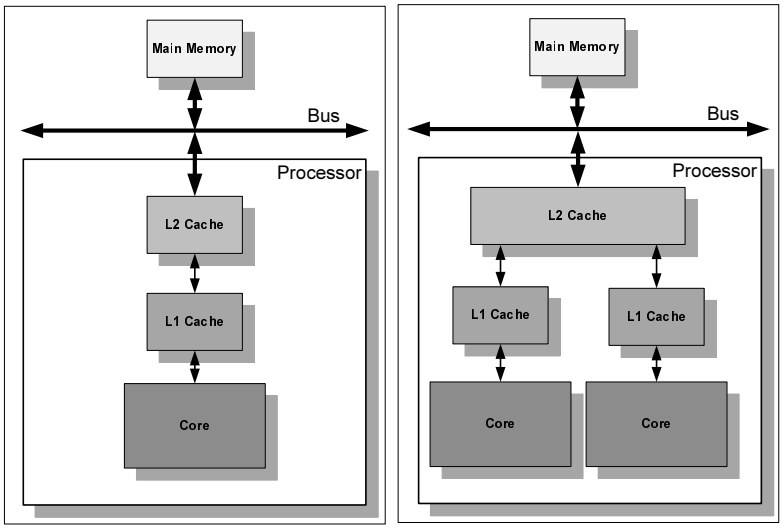


FIGURE 1.1: Moore's Law (This picture is distributed under the creative commons license and is available at http://en.wikipedia.org/wiki/Moore%27s_law)

Normally the L2 cache is shared by all cores on this die. Figure 1.2(b) shows such a layout for a dual-core processor.

Additionally to the development of multiple cores per processor, nowadays most cores can execute at least two hardware threads in parallel. This feature is called hardware multi-threading, and not to be mistaken for multi-threading of software. Furthermore, Intel introduced in 2002 its first hyper-threading processor. The main feature of this technology is the duplication of sections which store architectural states of hardware threads in the processor. This approach reduces the maintenance cycles to save and restore thread-local registers during the change between different threads.

These new technologies, helped to be still able to fulfill Moore's Law, at least regarding the processing power of end-user computers. Before the introduction of multi-core systems, all threads shared small time slices on the core, to create a quasi-parallelism, also known as multitasking. Unfortunately, due to the introduction of thread-level parallelism to end-user computers problems arose. Some of them were already known from large parallel computers, *e. g.,* the Cray supercomputers [CRA]. (i) Single-threaded programs won't speed-up, since they are not parallel and the processor frequency stays



(a) Single-Core Processor. (b) Dual-Core Processor.

FIGURE 1.2: Development of Intel-compatible processor design.

stable. (ii) Multi-threaded programs will slow down, due to interaction with each other, *e. g.,* conflicts when accessing the same shared data.

One way of solving this challenge is the application of transactional memory (TM). This is an optimistic approach based on the seminal work of Knight [Kni86] and Herlihy *et al.* [HM93]. The idea behind a TM is to speculatively execute parts of code (transactions), that must execute atomically and in isolation without worrying about how synchronization is implemented. The transaction can either be ended successfully or be aborted. No intermediate states will be visible to other threads. It also assures that all operations are handled in an atomic way, consistent and isolated. In contradiction to database transactions the durability has not to be guaranteed.

Depending on the implementation, TM can be classified into software transactional memory (STM) and hardware transactional memory (HTM). STMs are software libraries which are easy to develop and use, firstly proposed by Shavit *et al.* [ST95]. Its disadvantage is the software overhead. On the other hand HTMs are still not commercially available due to the great efforts to implement it in hardware. Nonetheless, it has been investigated by several research groups [AAK+05, HWC+04, HM93, MBM+06, MHW05]. When HTM will be available it is an accepted opinion that it will outrun STM, since hardware solutions are normally faster. Additionally to this two major types, a mixture of both approaches, the so-called hybrid TM [DFL+06a, MTC+07], was examined as well. In this thesis we will focus on software schemes. For a broader discussions of TM we suggest Larus *et al.* [LR06] and the "Transactional Memory Online" web page [Tra]. A more detailed view on the STM functioning can be found in the rest of this section and in Chapter 3.

Another field of interest is the event-driven architecture. It is dealing with the creation, modification, and reaction to events. It can be used for the data processing of large sensor networks so-called wireless sensor network (WSN). Typical applications for WSN is the environmental monitoring [HM06], *e. g.,* monitoring of trees, volcanoes, and glaciers. Each sensor in a WSN produces data, referred to as events, which have to be processed. Events produced by the sensors are streamed through different components, *e. g.,* filter or summing unit, and are finally collected in a data store. This type of applications is called event stream processing (ESP). It includes event visualization, event databases, event-driven middleware, event processing languages, and complex event processing (CEP).

However, the event stream processing is not limited to the processing of WSN events, but it can also be applied to any other event-driven information system. Usually these systems consist of event sources, processing components, and event sinks. All parts of such a system are connected with directed streams. Sources for events are as already mentioned any kind of sensors or event sinks of other processing systems. Each processing component can be either stateless or stateful. The processing on a stateless component depends only on the actual state of the processed event, *e. g.,* deleting events if their value is below a certain limit. In contrast to this, the stateful processing is based additionally to the event state on the internal state of the component. During the processing it is possible that both states are changed, *e. g.,* computing the average of the last ten events. The sink of an event-driven information system can be a data storage, visualization or other processing systems.

In this thesis we will consider both evolutions and eventually try to apply our research to both domains.

1.1 Motivation and Objectives

In 2006, it was estimated that the multi-core design would also be applied to small, energy-constrained devices, *e. g.,* cell phones, PDA's or wireless sensors. Under this assumption, we participated in the "Mobile Information & Communication Systems" project (MICS) [MIC] promoted by the National Center of Competence in Research [NCC]. The agenda of the MICS is wide-ranging from the study of fundamental principles to the development of platforms, and their deployment in applications.

The goal of our research was to develop an STM for sensor networks and/or energy-constrained devices with multi-core architecture. Additionally, we had the plan to extend this STM to a distributed one, *i. e.,* one STM runs on several devices, which share the processing of data and/or the data itself between each other. As a result the event

stream processing (ESP) can be done directly on the constrained devices without further backend processing. One challenge in ESP is the parallelization of stateful components, since the result of the processing is based on the state of the event as well on the internal state of the component. The concurrent access to this state is a non-trivial problem if one wants to use several parallel threads for the processing. Especially pessimistic concurrency approaches are either slowing the processing down to sequential processing or demanding high programming skills.

Nowadays, the software transactional memory (STM) community agreed more or less on a minimal API for STM libraries. However, until 2006, only proprietary compiler solutions existed. They had the disadvantage that each STM-enabled compiler supported only a few STMs at best. Thus, none of them was introduced into the development branch of standard compilers, *e. g.,* GNU Compiler Collection (gcc). Since most of the smaller devices and low level applications are programmed in `C`, one of our objectives was the development of a support for this unmanaged programming language, which can be easily used by application developers without specific STM knowledge. Additionally, we wanted to create a universal interface for STMs to improve their application without the special compilers for them.

1.2 Approaches

For the anatomy of the STM we need to compare different STMs. Therefore, we re-engineered several existing implementations. In our work, we concentrate on STMs written in object oriented programming languages, *e. g.,* `Java`, `C++` and `C#`, since we can easily obtain a visualization of the code in UML. For this task we use *Bouml*, a freely available software. This tool is also used for the creation of the UML diagrams in the design pattern.

The contributions presented in Chapter 4 uses the LLVM framework. We implemented a pass for the LLVM to transactify `C`-code. The pass itself is written in `C++` and has to detect and modify marked areas in the code which should be transformed. To be able to analyze our work we use TinySTM as backend STM and one of the micro benchmarks included in the TinySTM package.

For the STM-enabled ESP components we developed our own event streaming chain in `C` and in `Java` which we use for the evaluation of the approaches presented in Chapter 5 and 6. As STM support we have chosen for the `C` implementation again the TinySTM framework, since we were already familiar with it. For the version written in `Java` we use an STM called DSTM2. For our evaluation we assume mainly a mixed order of the

incoming events. Therefore, we used one of the components in the streaming chain to delay each event for a random amount of time, which can occur during the travel of the event in a network.

1.3 Contributions

This thesis has several foci of contribution. Nonetheless, they have all in common their connection to STM. One part of our work (Chapters 3 and 4) deals with improvement of the development and application of STMs in a general way. The second part (Chapters 5 to 7) is about the application of the STMs to event stream processing.

1.3.1 Improvement of the Development and Application of STMs

One contribution of this thesis is the concurrent design pattern called "Transactional Object". The design pattern allows programmers an easier application and integration of STMs to their software solutions, since the pattern summarizes its characteristic properties.

With the development of the TANGER framework we enable a compiler independent possibility to transactify `C` source code in a semi-automatic way. Our approach is based on the LLVM compiler framework and the first universal STM interface for the programming language `C`.

1.3.2 Application of the STMs to Event Stream Processing

In ESP it is a challenge to parallelize stateful components, due to the concurrent access to shared internal states. Therefore we applied the STM technology to event stream processing. Our two implementations, for `C` and `Java`, showed a good scalability of the stateful components in our experiments. Furthermore, we reduced the complexity of parallel programming nearly to the one of sequential programming, by hiding the parallelism to the user. The issue of event ordering is automatically solved by the underlying STM. However, it is limited to unique and discrete timestamps.

Additionally, we are contributing the feature of distributed speculations, by allowing events of not yet committed transactions to be processed speculatively by the following components. Our evaluation of this components shows that with this improvement we can reduce the idle times of the system with only a minimal overhead. Furthermore,

we could prove that for certain input orders, our system is able to outperform even the sequential processing when using just a single processing thread.

The last contribution of this thesis is the development of different use cases which could be improved with the application of our ESP approaches. We present exemplarily four use cases which are using WSN to create event streams which have to be analyzed to control the system.

1.4 Organization of this Thesis

The remainder of this thesis is organized as follows. First of all we will present the work from other researchers related to ours in the following chapter. In Chapter 3 we will pinpoint the anatomy of a software transactional memory. The objective is to provide a good foundation for the reading of this thesis. The following chapter introduces an approach for STM compiler support for the programming language `C`, which is able to transactify code semi-transparently. Furthermore, our system can be easily adapted to support other `C`-based STMs. In Chapter 5 we show the application of STM to event-based systems. The presented streaming engines for `C` and `Java` illustrate the strength of transactional memory in the area of stateful event processing. Furthermore, we introduce a distributed speculative approach for these systems in Chapter 6. Based on Chapters 5 and 6 we present in Chapter 7 several use cases for our STM enhanced event-based system. And finally Chapter 8 concludes this thesis by summarizing our work and giving an outlook on possible future work.

Chapter 2

Background and State of the Art

In this chapter we will present work which is related to our research outlined in this thesis. Furthermore, basic definitions and explanations are given, which are needed for a better understanding of the following chapters.

2.1 Experimental Methodology

To be able to compare the theoretical improvement of parallelization with the practical one, we have to measure the speed-up (S). Therefore, the processing speed of a single threaded version (T_s) is compared with the one of a parallel version ($T(p)$) under the same conditions and thus just depending on the amount of used processors (p). The processing speed can be quantified in two ways, either by counting the operations executed during a certain time period or by measuring the time for a certain amount of operations.

$$\textbf{Speed-up: } S(p) = \frac{T_s}{T(p)} \tag{2.1}$$

Furthermore it is possible to compute the efficiency (E) by relating the speed-up ($S(p)$) with the amount of used processors (p)

$$\textbf{Efficiency: } E(p) = \frac{T(p)}{p} \tag{2.2}$$

In 1967 Gene Amdahl formulated his theory (Eq.: 2.3) on parallel computer performance, nowadays known as "Amdahl's Law". It specifies the maximum theoretical parallel speed-up (S_p) of a program. To calculate the speed-up of parallel programs, the percentage of

sequential (T_1) and parallel (T_p) code, as well the number of used processors (p) are set in relation to each other. The sum of T_1 and T_p must be 100%.

$$\textbf{Amdahl's Law: } S(p) = \frac{1}{T_1 + \frac{T_p}{p}} \tag{2.3}$$

The original "Amdahl's Law" is representing the theoretical maximal speed-up. Taking also account of the overhead (H(p)) of the parallelization results in Equation 2.4. On a good parallel machine, the overhead is not necessarily linear or static, since it covers conflict detection and solution as well as thread scheduling.

$$\textbf{Extended Amdahl's Law: } S(p) = \frac{1}{T_1 + \frac{T_p}{p} + H(p)} \tag{2.4}$$

Amdahl's Law is leading to an interesting question. "Why are we still using sequential programs?" First of all, before the introduction of thread-level parallelism, it was most of the time not necessary to develop parallel programs. Nonetheless some applications already had to, *e. g.,* most applications with an user interface where one thread controlled the user interface and another one was responsible for the computation. Secondly, not everything can be parallelized, *e. g.,* access to the same location on hard disk or in memory. And last but not least, parallel programming is has a higher complexity than sequential programming, since one has to figure out all possible conflicts between the different parallel threads. Thus several concurrency control strategies were developed over the last decades. Some of them are presented in the following chapter, for instance software transactional memory (STM). This is a optimistic concurrency control approach analogous to database transactions, which executes a region of code in an atomic way and solves occurring memory access conflicts.

2.2 Concurrency Control

When two or more threads are working in parallel on the same set of data, it can happen that some of them are accessing the same data simultaneously. There are two types of accesses (read and write). Let's consider two threads working in parallel without any concurrency control and thus we have to face 3 concurrency cases.

(i) **READ-READ.** When both threads are reading the same value, no conflict will occur, since the data is not altered. When this case appears in an application, no concurrency control is needed. An Example could be a constant which always has the same value.

(ii) **READ-WRITE.** In this case one thread reads and the other writes at the same time. It can then happen that the reading thread is receiving an inconsistent value. This especially happens when both threads are accessing the several address locations in different order. In the end the reading thread loaded some old and some new values, instead of a complete new or old set.

(iii) **WRITE-WRITE.** In the last case both threads are writing at the same time to the same address. This can lead to a lost write, *i. e.,* the write of one thread will never become visible and thus can lead to inconsistency.

2.2.1 Basic Atomic Operations

One easy way to solve concurrency at hardware level would be the atomic memory access. Atomic means that the operation is not interruptible. Nowadays most processors guarantee the atomic execution of single read and write accesses. Lately more advanced operations are also supported by the major manufacturers, such as Compare-and-swap (CAS) or Load-Link/Store-Conditional (LL/SC). These operations modify the content of an address only if the expected value is still there. This functionality can avoid write-write conflicts since it checks the consistency of the old value.

Unfortunately, it can happen that more than one operation should be executed atomically. In this case a whole section of code has to be executed atomically, called critical section. For instance, during a bank transaction money is moved from one account to another. Normally one would expect that this is one operation, but it is not. In a first step money is withdrawn from one account and in the second step the withdrawn amount has to be deposited on the other account. These two accesses are independently executed and by now no commercial solution is available supporting such independent operations in an atomic way on hardware level. Nonetheless, this problem can be solved on software level with pessimistic or optimistic approaches.

2.2.2 Pessimistic Approach

Concurrency control algorithms which are prohibiting any parallel access to the same data even if they are not conflicting are called pessimistic. Its solution is to sequentialize the accesses. This approach is also known as mutual exclusion (MUTEX) [SGG09, Tan09, Dij65]. Nowadays, all major programming languages are support different models of this approach, *e. g.,* in `Java` the synchronized block or the classes `ReentrantLock` and `Semaphore`.

One solution using this approach is using locks. Algorithm 1 shows a simple example of locks implemented in `Java`. When a thread is calling method `m()`, a lock is acquired (line 6) to protect the method body (line 8). Until this is successful the thread is blocking. After the lock has been obtained, the function does something under the protection of the lock and finally the lock is released (line 10). A more sophisticated version would be a read-write lock. This lock will not forbid the access until a thread wants to gain write access. At that point any possible conflict with reading threads has to be solved, *e. g.*, letting the writing thread wait until all current read locks are released.

Algorithm 1 Class using ReentrantLock in Java

```
1  class X {
2    private final ReentrantLock lock = new ReentrantLock();
3    // ...
4
5    public void m() {
6      lock.lock(); // block until condition holds
7      try {
8          // ... method body
9      } finally {
10         lock.unlock()
11     }
12   }
13 }
```

Another well known concept is the semaphore, introduced by Dijkstra [Dij65] in 1965. In contrast to the lock, in a semaphore one can set the numbers of threads working parallel on one resource. An example implemented in `Java` can be found in Algorithm 2. Line 2 is defining the maximum number of threads allowed in parallel. The rest of the code follows the same scheme as the lock in Algorithm 1.

Algorithm 2 Class using Semaphore in Java

```
1  class X {
2    private static final int MAX_AVAILABLE = 100;
3    private final Semaphore semaphore = new Semaphore(MAX_AVAILABLE, true);
4    // ...
5
6    public void m() {
7      semaphore.acquire(); // block until condition holds
8      try {
9          // ... method body
10     } finally {
11         semaphore.release()
12     }
13   }
14 }
```

With the two examples of pessimistic concurrency control, one can easily see the mutuality. First the unique access is gained and then the processing is done. In fact for single-core systems, this approach is efficient since anyway only one thread can execute at a time. Unfortunately for multi-cores this approach can slow down the system. It will force parts of the code to be processed sequentially and thus limit the speed-up as we can easily realize from Amdahl's Law (Eq.: 2.3). Another problem that can occur is that one thread successfully acquires all needed locks but than has to release them without any modification, because of an exception or an unfulfilled condition. Until the release of the locks all other threads, which want to access some of the data protected by these locks, have to wait idle and loose time.

2.2.3 Optimistic Approach

The idea of optimistically executing tasks that may have undetected dependencies has been around for a long time. Steffan and Mowry proposed Thread Level Data Speculation (TLDS) [SM98] as a way to benefit from multiprocessor computers when the programs are not designed for explicit parallelism. They suggested that compiler support could enable activities in a sequential program to be executed simultaneously and then committed from the less speculative activity to the most speculative one. If some dependencies are detected the speculative activity is terminated and restarted.

Another approach is the transactional memory (TM), firstly introduced by Knight [Kni86] and later popularized by Herlihy *et al.* [HM93]. It uses mechanisms analog to the well known database transactions to solve conflicts. On the one hand, TLDS creates parallel tasks from sequential code, and on the other hand, TMs are mainly used to help synchronizing already parallel code.

2.3 Properties of STM

Over the last years, many different STMs have been released, with different approaches to improve the efficiency. Nonetheless we can determine several properties to classify the STMs. This list of properties does not attempt to be complete, especially, since it is still under research.

The first one is the access granularity of the shared data. Two different approaches are known. One is called word-based access [DSS06, FFR08, DGK09], where the granularity of the conflict detection is the memory location. This design is a good for operating systems, low-level languages, and compiler integration, but it can also be used to build

object-based STMs. The other one is called object-based access [HLM06, MSH+06, RFF06a] and it is able to detect conflicts on object granularity. Therefore, additional meta-data is stored in the shared object or a proxy. This design needs language support and is mainly used in object-oriented programming languages.

The next property defines the modification of shared data. On the one hand, the write-back approach [MBM+06, FFR08] creates a thread-local copy of the accessed data. This copy can be modified during the transaction and the new value is copied back to the original location upon commit. This design is also known as redo-log and supports a fast abort of the transaction, since the original data was not altered. On the other hand the write-through design [FFR08] is known. In this approach the original value of the shared data is stored in a thread-local undo log and all modifications of the transactional code are directly done on the shared data location. In case of an abort the old values are copied back. This design enables a faster commit and enables compiler optimizations.

How the STM handles the read operation is part of the third property. Either the STM uses visible reads, *i. e.,* all read accesses to shared data are collected, which can be used to detect read/write conflicts [SMSS06, IR08]. Or, all reads are invisible to other threads and thus the consistency of the shared data has to be validated on commit [FFR08, SMSS06].

The fourth property describes how the parallel access to shared values is organized. The lock-free approach [Fra03, ST95] is using compare-and-swap or similar operations to avoid using locks. Therefore, it needs more complex algorithms to solve conflicts. Normally they are based on threads which help each other to commit and thus some thread always progresses. An opposite approach is the lock-based design [Enn06, DSS06, DGK09]. It is simple and efficient, but it does not guarantee any progress. The third alternative is called obstruction-free [HLMS03, HF03]. It is an approach, where the contention manager guarantees the progress of threads. Therefore, it is necessary that transactions can be committed and aborted in a atomic way.

The last listed property, deals with the different spots where an access to shared data can become visible to other threads. Either it happens when the data is accessed the first time (encounter-time locking [FFR08, SATH+06], which can help to detect conflicts early), or the acquisition becomes visible during the commit (commit-time locking [DSS06]), which may reduce in some cases unnecessary aborts. It is as well possible to apply both techniques in the same STM to detect different kinds of conflicts [DGK09].

2.4 Programming Support for Concurrency Control

In this section, we will outline the different programming support approaches for programming languages. We describe the possibilities for unmanaged languages, *e. g.,* `C` or `C++` and for modern managed languages such as `Java` or `C#`.

Compiler support for STMs was not a new relevation, by the time we started to work on TANGER, our transactional `C` compiler, but it was only known for programming languages with managed environments such as `Java` or `C#`. For instance Harris *et al.* [HF03] developed the support for atomic blocks in `Java`. They used an STM implemented in `C` and merged it with a `Java` Virtual Machine (JVM).

Another approach for a compiler support for transactions was presented by Adl-Tabatabai *et al.* [ATLM+06]. In their work they combined a just-in-time dynamic compiler for `Java` and `C#`, a virtual machine and an STM, called multi-core runtime STM [SATH+06] (McRT-STM). Again this approach was done for a managed environment.

Harris *et al.* [HPST06] presented a solution to decrease runtime overheads by optimizing for fast transaction support. They used a compiler for Common Intermediate Language programs. Other approaches are using `Java` byte code modifications, as presented by Herlihy *et al.* [HLM06] or Riegel *et al.* [RFF06b].

Wang *et al.* [WCW+07] presented transaction support for unmanaged environments. They proposed several optimizations for a `C` compiler, *e. g.,* moving transactional loads out of loops. An altered `C` compiler for transactional support was presented by Damron *et al.* [DFL+06b].

Parallel to our development of TANGER, TARIFA [FFM+07] was developed. Sometimes transactifying the source code is not enough, especially when library functions are called from within atomic blocks. The problem is that not all libraries are available as source code, and even if they are available, it is often not known how they were initially configured and compiled. In their approach the binaries are disassembled into x86 assembly code and then all their memory accesses are transformed into calls to the underlying STM.

Shortly after we presented TANGER [FFM+07] at the TRANSACT workshop in 2007, Intel released their first `C++` STM compiler. Until now (March 2010) this prototype advanced to its 3rd major release. Furthermore the GCC community is finally developing an STM support for GCC. Unfortunately both compiler based implementations are still under development and until they are available in a stable version the TANGER framework will help to simplify the application of STMs.

Creating parallel code is sometimes harder than just programming an application with one thread. The main reason for this is the possible interaction between the threads especially by means of memory accesses. Modern high-level programming languages support normally simple mutual exclusion constructs with specialized keywords like `synchronized` (Alg. 3) or `lock` (Alg. 4). Those keywords can form blocks of code which are then protected from parallel accesses. Unfortunately the unmanaged low-level language `C` does not support these constructs automatically, but it is still widely used because of its resource saving properties. For instance the kernel of Linux is programmed in `C`.

Algorithm 3 Synchronize block in Java

```
1  class Sync {
2      public void do(object state) {
3          synchronize (this) {
4              ... //Do something
5          }
6      }
7
8      public synchronized void do() {
9          ... //Do something
10     }
11 }
```

Algorithm 4 Lock block in C#.

```
1  class Sync {
2      public void do(object state) {
3          lock (this) {
4              ... //Do something
5          }
6      }
7
8      [MethodImpl(MethodImplOptions.Synchronized)]
9      public void do(object state) {
10         ... //Do something
11     }
12 }
```

2.5 Low Level Virtual Machine

The "Low Level Virtual Machine" (LLVM) [LA04] is a compiler infrastructure providing language and target independent components, which is an essential part of our TANGER framework. Lately it became quite popular even beyond academia, *e. g.,* Apple, Adobe or NVIDIA are using it[1]. LLVM provides the user with its own intermediate representation (IR). It is a low-level representation, but with high-level type information. It merges

[1] A complete list of LLVM Users can be found at: `http://llvm.org/Users.html`, 5-Aug-2009

the best of both worlds. On the one hand we have the compact representation and wide variety of available transformations, and on the other hand all the information needed for an aggressive inter-procedural optimization to minimize the resulting binary.

The LLVM IR is based on a load/store architecture, *i. e.,* the operations to read or write a variable are represented explicitly with either a load or a store operation. Also the local variables are not held, as for instance in `C`, in a stack, but they are stored in registers which are accessed directly, without using the load or the store command. The result of this division is a clear determination between local variables and global ones which are potentially shared between the threads. This feature of the LLVM enables us to minimize false protection of non-shared variables and hence decrease the runtime overhead.

If the transaction covers the whole body of a function (*i. e.,* transactions have function granularity) no further additional glue code is needed. Otherwise our instrumentation needs to be careful with the rollback of local variables before retrying or aborting. The LLVM enables programmers to apply so-called "passes" on the IR, which have to be written in `C++`. These passes can alter the IR, by iterating with different granularity over it.

Another feature of the LLVM is the possibility to transfer the IR back to source code like `C++`. On the first look this ability doesn't sound so important, but the resulting code is altered and describes the way how the IR code would be created. Thus one can add something in the IR by hand and gain from the related source code the operations which have to be done by the pass to create these insertions. In fact this feature was a great help to develop TANGER.

For the first prototype of TANGER we used the LLVM release 1.9 which was available from November 2006. Since then all aspects of the LLVM have been improved resulting in a incompatibility between the release series 1.x and 2.x.

2.6 Design Patterns

Design patterns became an interesting research topic thanks to the seminal work of Gamma *et al.* [GHJV95] in 1995. They developed the first design patterns and encouraged others to do the same.

Some concurrency related work on patterns has already been published in books on "Pattern-Oriented Software Architecture" (POSA) [SSRB00, KJ04, BHS07], *e. g.,* monitor object or double-checked locking. Da Silva [dS99] also proposed several concurrent patterns in his Ph.D. thesis.

In the area of databases the work of Fowler [Fow02] needs to be mentioned. He introduced optimistic and pessimistic locks for databases, but unfortunately his work is very much bound to databases. Another interesting work in the area of databases by Grand [Gra99] presented several transaction patterns.

None of them, however, proposed an optimistic concurrency control pattern for transactional memory access as presented in Chapter 3.

2.7 Event Stream Processing

In recent years, several works have addressed the scalability of event stream processing, but all of them take an approach different from ours. For example, Koparanova and Risch [KR04] have developed GSDM (GRID Stream Data Manager) to handle processing of data produced in real time by a large amounts of sensors that receive signals from space. Their approach is to use data partitioning to split the events between several replicated components and then merge the results. This approach is, however, applicable only for computations in which the processing can be split into several units that do not require any synchronization (*e. g.,* in their case, Fast Fourier-Transforms). In addition, it needs components to split and merge the data and these components need to consider the semantics of the computation. Other works, like Borealis [AAB+05] or more recently StreamFlex [SPGV07], use the assumption that components are normally stateless and scalability can then be easily achieved through simple replication.

Another research direction focuses on the problem of efficient correlation of events. Correlation is strongly dependent on state and therefore difficult to parallelize. SASE [WDR06] is an example of an event stream processing system that focuses on matching event patterns efficiently.

With respect to out-of-order processing of events, one classical approach is to buffer an event until it is known that no prior events may arrive. CEDR (Complex Event Detection and Response) [BGAH07] tries to balance insensitivity to event arrival and system performance. It uses a temporal model that deals with out-of-order events by allowing results to be output and later be retracted and revised in case a relevant event arrives. The main motivation is that in some contexts an early, but conceivably wrong, result is more valuable than having buffers and delays to cope with late events. Similarly, Li et al. [LLD+07] propose an extension for SASE that allows a pattern to be tardily matched when some late event arrives.

Römer *et al.* [RM04] discussed the application of event-based systems for detecting event patterns. In their work they point out that automatic detection of them, so-called composite events, are less suited for real-world states produced by sensor networks.

2.8 Summary

In this chapter, we presented research and knowledge, which is the basis for our work, presented in this thesis. We introduced some formulas needed for the evaluation of our experiments. Then we shortly summarized concurrency control systems and had a closer look on STMs. Furthermore, we presented known programming language support for concurrent processing. In the following section, we introduced the LLVM framework which we are using for TANGER as described in Chapter 4. In Section 2.6, we shortly summarized the basics of design patterns, which are a good foundation for Chapter 3. And last but no least, we presented related work in the area of event stream processing, which is linked to our research presented in the Chapters 5 – 7.

Chapter 3

Anatomy of a Software Transactional Memory

In the area of software engineering, design patterns are abstract solutions to recurrent software problems. They are describing the problem and show how it could be solved in a general way, *e. g.,* an iterator which allows a sequential access to a set of objects. These blueprints are not a finished design and thus cannot be translated directly into source code.

Design patterns are describing only one specific problem and are belonging to the group of software patterns. At a higher level there are architectural patterns which are usually describing entire systems, *e. g.,* peer-to-peer or service oriented architectures. Other software patterns are algorithms. In contradiction to design patterns, algorithms are solutions to computational problems.

Since design patterns are just a formal description, they can be applied to all programming languages. Parallel to the design patterns the unified modeling language (UML) was developed. It is a descriptive language including graphical notations which can create an abstract model of software-intensive systems. Furthermore it is possible to translate UML diagrams into source code fragments and vica versa. These code fragments are describing the structure of the solution, but not the behavior. The UML design is closely related to object oriented programming, because it naturally supports the object oriented constructs, like classes and interfaces.

The work presented in this chapter was partly presented at the *First International Conferences on Pervasive Patterns and Applications* in 2009. Section 3.1 describes how we developed the STM design pattern, while in Section 3.2 our resulting pattern is presented. Section 3.3 summarizes our work.

3.1 Developing an STM design pattern

Creating a design pattern out of several implementations solving the same problem needs a good understanding of both topics, STMs and design patterns.

The well developed design pattern can be best described with a quote from *Antoine de Saint-Exupéry*:

Perfection is achieved,
not when there is nothing more to add,
but when there is nothing left to take away.

When starting to develop an STM design pattern, we first looked at other patterns to learn more about the used structure. Finally we decided to use the template proposed in [GHJV95]. Compared to other templates, *e. g.,* Fowler *et al.* [Fow02] or Grand [Gra99], it was more comprehensive and thus better suited for our initial work.

In a second step we collected several well known STM implementations whose source code was freely available. We had chosen implementations from several object oriented programming languages, *e. g.,* `Java`, `C++` and `C#`, to extend our analysis to a wider variety. For our comparison we analyzed the source code of the following STMs: DEUCE [KSF09, DEU], DSTM2 [HLM06, Her], JSTM [Noe08, JST], RSTM [MSH+06, RST] and SXM [Her05, Her].

For the comparison we used a freely available software called *Bouml* [Bou]. This program provides the possibility to translate source code into UML code for several languages, namely `C++`, `Java` and `PHP`. Another feature of this tool is the UML class diagram generator. It can dynamically create a diagram containing the classes and their relationships, which was a great help for analysing the design of the STMs.

When analysing the class diagrams, we realized that the classes can be partitioned into three handler groups as shown in Figure 3.1. The Transaction Handler (TH) group is responsible for all transaction related classes, *e. g.,* the transaction or its states. In the Conflict Handler (CH) group all classes dealing with there solution of conflicts are included. And finally the Shared Object Handler (SOH) group consists of the classes covering the creation and manipulation of the shared objects. Additionally to the three handler groups, Figure 3.1 shows the relation to the threads. Depending on the STMs, some are using extended thread classes while others just use the standard implementation, thus we decided to add them to this diagram, but more in the means of the access or usage of the STM.

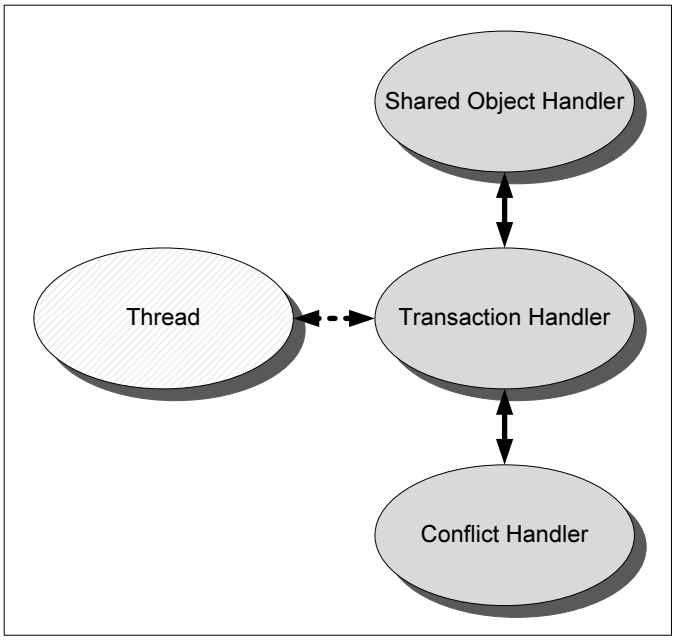


FIGURE 3.1: Rough structure of an STM

Of course this structure would be too simple for a design pattern, since one could not gather enough information about the dependencies within the groups. Nonetheless it supported us with a good foundation for separating the classes and their relationships.

After partitioning the STM structure as described above we defined the properties of each group and tried to find already existing design patterns which could be reused. For instance in the CH group it was obvious to reuse the strategy pattern, since usually an STM is using a contention manager to solve problems. By now several different strategies are available, but only one can be used during runtime. Another example for recycling other design patterns is the internal state of the transaction. The state can take several values and thus behaves like an enumeration pattern.

In the next step we merged the three parts together and added their relationships. After our first proposal we evolved the whole structure in many discussions within our department and with other researchers in this domain. Finally, we framed our work into the design pattern template as it is presented in the following section.

3.2 Transactional Object

In this section we present our developed STM design pattern. It is a first approach for an optimistic concurrency solution. The structure mainly follows the template used by Gamma *et al.* [GHJV95].

3.2.1 Pattern Name and Classification

As name for this design pattern we simply propose "Transactional Object" (TO). It indicates intuitively that the object is accessed via a transaction. Since it is dealing with concurrent access it should be, of course, classified as a concurrent pattern.

3.2.2 Intent

The goal of the Transactional Object is to provide a speculative concurrent access. Threads can modify the same shared object independently, protected by a transaction. Only when the transaction commits these changes become visible. In case of conflicts one of the concurrent transactions has to roll back and retry its processing.

3.2.3 Also Known As

The Transactional Object is known as (Software) Transactional Memory, which is the commonly used name for this technology. It can be defined as speculative or optimistic concurrency algorithm.

3.2.4 Motivation (Forces)

A well known problem in concurrent systems is the dining philosophers problem. In short, five philosophers are sitting at a table and are either thinking or eating. Unfortunately the forks are placed between them and each of them needs the left and the right fork to eat. Using a pessimistic approach (*e. g.,* mutual exclusion), all of them would acquire first the right fork and then the left one, or vice versa. This can lead to a deadlock as one can easily imagine and thus none of them will ever start eating.

When using an optimistic approach, the forks would be atomic objects, *i. e.,* when a philosopher eats he would use local copies of the forks which are protected by a transaction. When he finishes eating, the transaction would check, if the used forks were also used by someone else. If so, all conflicting philosopher's except from those ones which can work in parallel without conflicts are forced to rollback and re-start eating.

With an STM, one can consequently prevent deadlocks and process everything in a speculative way. Especially when it could happen that an exception occurs, *e. g.,* one philosopher gets sick while eating and he would have to re-start. In this case, the other philosophers wouldn't have to wait like in a lock-based approach and this possible conflict would be solved without intervention.

3.2.5 Applicability

The TO design pattern should be used to solve concurrent access problems. Its strength shows up when only little contention will occur. For example using a linked list where several threads are adding and removing elements in parallel. When using the well known mutual exclusion, one could use it as coarse grained lock, *i. e.,* only one thread at a time can access the whole list. Of course it would slow down the access dramatically.

Another option would be a fine grained lock where each thread is locking two successive elements (the actual one and its predecessor). This is needed to guarantee correctness; otherwise the actual element could be removed by another thread. In a scenario with approximately 50 elements and 5 parallel threads, fine grained locking would be quite costly since the probability of conflicts is quite low, nonetheless it is needed. When on the same problem an optimistic approach is applied, all threads could achieve a better performance.

This gain is the result of reacting only in case of a conflict. Of course it has to be assured that the TO keeps track of all modifications, which will cause some administration overhead. When a thread wants to commit a transaction, the TO has to check its internal read/write list for conflicts. If a conflict occurred, the TO has to contact the contention manager, which solves the conflict by aborting one of the conflicting threads. In our example with the linked list this conflicting case should be very unlikely and the TO will easily outperform the pessimistic approach. Of course the rate of conflicts has certain dependencies, *e. g.,* update rate or distribution of the modified values in the list.

When working with STMs we developed the decision tree shown in Figure 3.2. The purpose of this diagram is the support of developers to decide whether their application needs an STM or not in a simple way.

3.2.6 Structure

The general structure for the TO is shown in Figure 3.3. As one can see the center of our proposal is the `Transaction` class. All other classes are linked to it, either directly or indirectly.

3.2.7 Participants

In this subsection all classes used in the pattern are presented and their role in the design pattern is explained.

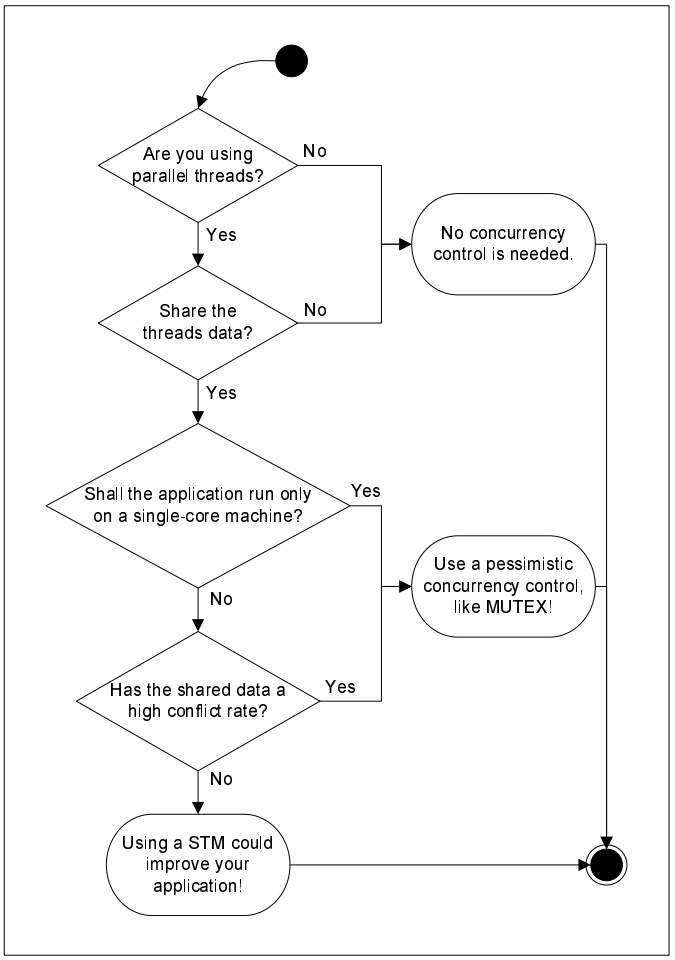


FIGURE 3.2: Decision tree for the application of an STM

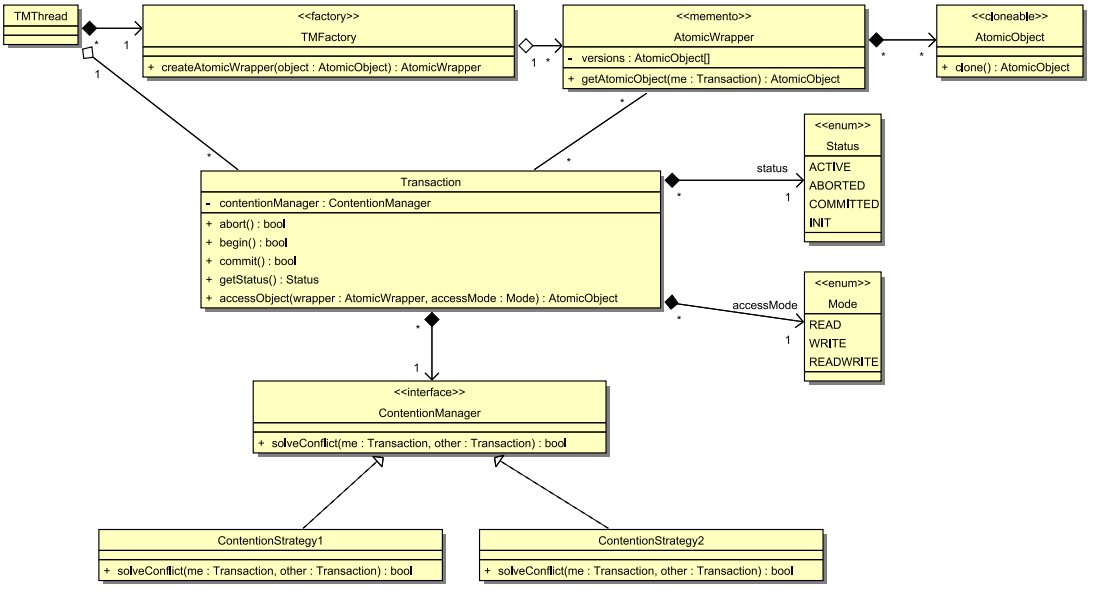


FIGURE 3.3: UML class diagram of the Transactional Object

- `AtomicObject` — is an object that stores the shared data. It may be cloned by the `AtomicWrapper` to obtain speculative versions which can be modified within `Transaction`s.
- `AtomicWrapper` — protects the shared data against any direct access. It has the responsibility to make only successfully committed versions of an `AtomicObject` visible to the system. Furthermore it has to manage the transaction-local versions of an `AtomicObject`.
- `ContentionManager` — solves conflicts when two `Transaction`s access the same `AtomicObject`. It has to decide which of them has to abort and which can proceeded.
- `Mode` — contains the different access modes in which a `Transaction` can access an `AtomicObject`.
- `Status` — represents the different states a `Transaction` can have.
- `TMFactory` — creates new `AtomicWrapper`s, which will protect an `AtomicObject`.
- `TMThread` — abstraction of a thread. Its instances can protect access to `AtomicObject` with transactions.
- `Transaction` — defines with its methods the bounds of the transactional protection.

3.2.8 Collaboration

The structure of the TO design pattern was shown in the class diagram (Fig. 3.3). In this section the interaction of those classes is described. To do this, we structured the interaction into five parts, representing the major activities. First the way how transactions are handled is shown, followed by the handling of objects.

Begin of a Transaction. When some code should be protected with a transaction in `TMThread`, first a new instance of `Transaction` has to be created via the standard constructor. In the second step the transaction has to be notified that the following code has to be protected. This is done by the `begin()` message, as shown in Figure 3.4. Among other STM algorithm specific operations, the `Status` will be changed from `INIT` to `ACTIVE`. A transaction in the `ACTIVE` state can be aborted at any time, for instance by the contention manager to solve a conflict.

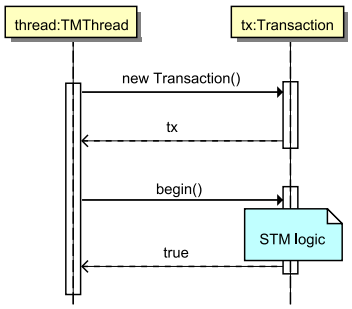


FIGURE 3.4: UML sequential diagram for beginning of a Transaction

Commit of a Transaction. When committing a transaction the changes done in the transactional protected area shall become visible to the system. Normally there are two possibilities, either the transaction can successfully commit (Fig. 3.5), because no contention occurred or because the contention manager solved the conflict and aborted the competitors. Then the `Status` of the transaction will be atomically altered from `ACTIVE` to `COMMITTED`. Otherwise the transaction will be aborted (Fig. 3.6) and rolls back to its initial state. Until the transaction is restarted, it will remain `ABORTED`.

Creation of an AtomicObject. The task of the TO pattern is to protect an object from being accessed concurrently from outside a transaction. Nonetheless it needs to be created outside a transaction. To enable this the factory class `TMFactory` has to be used. As shown in Figure 3.7 it will return an `AtomicWrapper` to protect the `AtomicObject`. This `AtomicWrapper` can be shared between the threads. However the implementation of the `AtomicWrapper` strongly depends on the used STM algorithm.

Access of an AtomicObject. The access (Fig. 3.8) to the `AtomicObject` will only be granted by the responsible `AtomicWrapper` from within a `Transaction`. The accessing transaction has to be in the `Status ACTIVE`. Then depending on the STM algorithm, usually a clone of the last committed version of the `AtomicObject` will be created and returned.

3.2.9 Consequences

When using the TO pattern the following side effects can occur: (i) The transaction encapsulates all changes done within its responsible area of code. It will also prohibit unknown states, lost updates or corrupted data. (ii) No deadlocks or livelocks will occur since the STM ensures progress for the transactional area. (iii) There are hidden costs, for the collision detection. Each time a transaction commits it has to check all

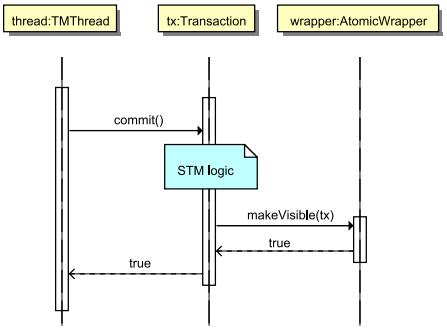


FIGURE 3.5: UML sequential diagram for committing a Transaction

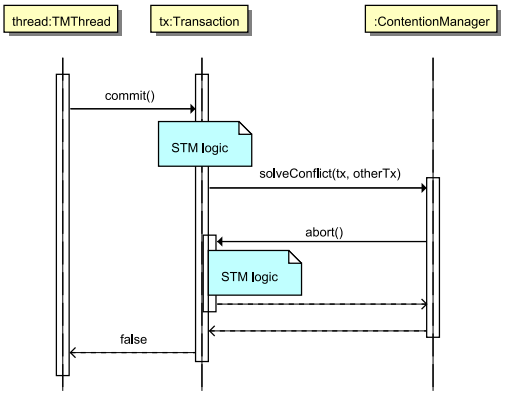


FIGURE 3.6: UML sequential diagram for aborting of a Transaction

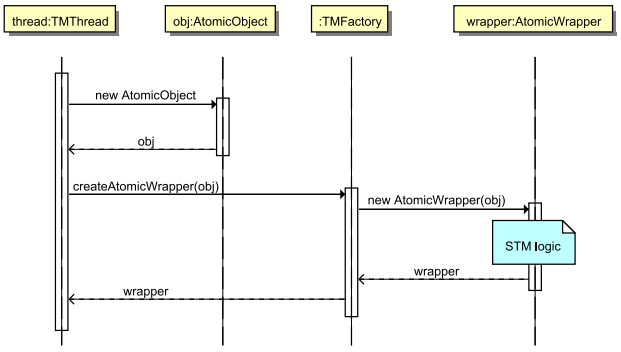


FIGURE 3.7: UML sequential diagram for creating an AtomicObject

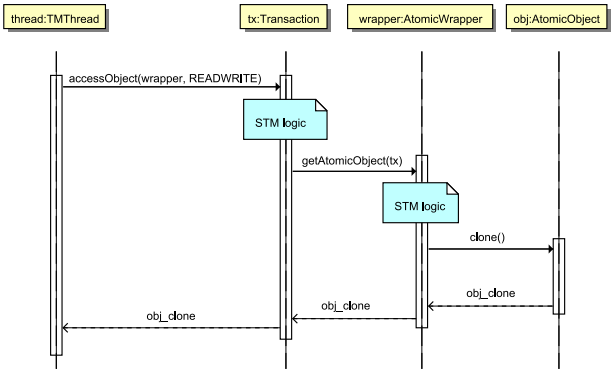


FIGURE 3.8: UML sequential diagram for accessing an AtomicObject

other transactions accessing the same shared object. Too much contention between the transactions can lead to an increase of the STM overhead. In this case a pessimistic mutual exclusion could be a better solution. (iv) All actions done within a transaction need the ability to roll back, *i. e.,* the original state must be restored. This can become impossible when a library function is called or an irrevocable command is executed, *e. g.,* launching a rocket.

3.2.10 Implementation

Currently many different STM algorithms are available. When implementing one of those or developing a new STM, one has to think first about the major issues presented in this section.

One issue deals with the contention manager. Scherer III *et al.* [SS04, SS05a] firstly proposed them to deal with conflict between transactions. Since then different contention managers were developed with different properties, *e. g.,* back-off, greedy, karma or kindergarten. While some of them are working with the presented interface, others need more information about the transactions to decide which one has to abort. So it might be useful to add additional methods to the contention manager or to store this information in the transaction, where it can be accessed from the manager.

For designing an STM one also needs to know how critical operations within the STM will be handled, either using a lock-based or a lock-free approach. The lock-free version can be realized with basic atomic operations, *e. g.,* compare-and-swap or load-link/store-conditional.

Last but not least one has to decide what would be the best language support. (i) The user has to do all calls explicitly as shown in the code fragment (Alg. 5). The `try-catch`-block is needed to avoid jumps outside the atomic area, due to an exception. (ii) With the

help of a language pre-processor or aspect oriented programming (AOP) the functions could be protected with a transaction, based on annotations (Alg. 6) or blocks (Alg. 7) added by the programmer. (iii) Another option is byte code modification. In this case the user would also mark a function with annotations (Alg. 6) and the code will be transactified during the compilation process.

Algorithm 5 Explicit implementation

```
1  ...
2  public void do() {
3          Transaction tx = new Transaction();
4          do{
5                  tx.begin();
6                  try{
7                          ... //Do something transactional
8                  } catch(Exception e){
9                          tx.abort();
10                         ... //Further exception handling
11                 }
12         }while(!tx.commit())
13 }
14 ...
```

Algorithm 6 Annotation-based implementation

```
1  ...
2  @atomic
3  public void do() {
4          ... //Do something transactional
5  }
6  ...
```

Algorithm 7 Block-based implementation

```
1  ...
2  public void do() {
3          atomic{
4                  ... //Do something transactional
5          }
6  }
7  ...
```

3.2.11 Sample Code

In the code fragment (Alg. 8) the major functions of the class `Transaction` are presented. The `begin()` function (Line 2) is needed to alter the state of the transaction to `ACTIVE` as one can see no CAS operation is needed at this point since there is no one else accessing this property in parallel.

When the code protected by the transaction was executed the `commit()` function (Line 8) is called. Before committing it has to check for conflicts with other transactions. If none occurred, the `Status` will be atomically changed from `ACTIVE` to `COMMITTED`.

The 3rd function shown is the `abort()` function (Line 20), which can be called from the contention manager (Alg. 9) to solve a conflict by aborting a transaction.

For the interested reader we would suggest to have a further look at DSTM [HLMS03] and DSTM2 [HLM06, HK08].

Algorithm 8 Fragment from a Transaction class

```
1  ...
2  public boolean begin() {
3          status = Status.ACTIVE; // switch to active
4          startTime = System.currentTimeMillis();
5          return true;
6  }
7
8  public boolean commit() {
9          validate (); //checks all accessed objects and solves conflicts
10         while (status == Status.ACTIVE) {
11                 if (statusUpdater.compareAndSet(this,
12                                 Status.ACTIVE,
13                                 Status.COMMITTED)) {
14                         return true;
15                 }
16         }
17         return false;
18 }
19
20 public boolean abort() {
21         while (status == Status.ACTIVE) {
22                 if (statusUpdater.compareAndSet(this,
23                                 Status.ACTIVE,
24                                 Status.ABORTED)) {
25                         return true;
26                 }
27         }
28         return status == Status.ABORTED;
29 }
30 ...
```

3.2.12 Known Uses

The first implementation of a software transactional memory was published by Shavit *et al.* [ST95]. Since then many different STMs have been developed, such as Deuce [DEU], LSA[LSA], RSTM [RST] and SXM [Her].

Algorithm 9 Fragment from a Contention Manager class

```
1  ...
2  public boolean resolveConflict(Transaction me, Transaction other) {
3          if (me.equals(other)) {
4                  return true; // same transaction
5          }
6          if (me.startTime < other.startTime) {
7                  return other.abort(); // me started earlier, other is aborted
8          }
9          return me.abort(); // give the priority to other
10 }
11 ...
```

The main problem for TO were the missing use cases. In the early days, the benchmarks were limited to simple applications, *e. g.,* linked list or bank accounts. Lately more and more "real" applications are available. One of them is "Atomic Quake" [ZGU+09], which is a modified version of the game "Quake". The game consists of a client and a server. Both parts were successfully altered to have an increased performance on multi/core systems. Another field of application is event stream processing [BFSF08, SFF09], which is presented in Chapter 5 and 6. Other fields of application are shown in the latest STAMP benchmark [CMCKO08, STA], *e. g.,* gene sequencing, network intrusion detection or client/server travel reservation system.

3.2.13 Related Patterns

In the TO design pattern we are using other well known design patterns [GHJV95, Gra99] as one can see in Figure 3.3 and 3.9, *e. g.,* the factory pattern (`TMFactory` class), the singleton pattern (`TMFactory` class), the memento pattern (`AtomicWrapper` class), the strategy pattern (`ContentionManager` class), the enumeration pattern (`Status` and `Mode` classes) and the transaction pattern (`Transaction` class).

There are several design patterns dealing as well with concurrent access, *e. g.,* lock, mutual exclusion, monitor object, double checked locking or read write lock. The difference to the TO is the pessimistic approach they are using. They are first allocating the resource instead of dealing with problems when they are occurring.

Furthermore the pessimistic concurrency control systems have normally poorer scalability compared to the STM. Scaling will become a major issue in the near future, since more and more threads can work in parallel on modern multi-core processors.

The transaction pattern is known to the community, but unlike to the use in this pattern, it is meant for access to databases and thus has other constraints.

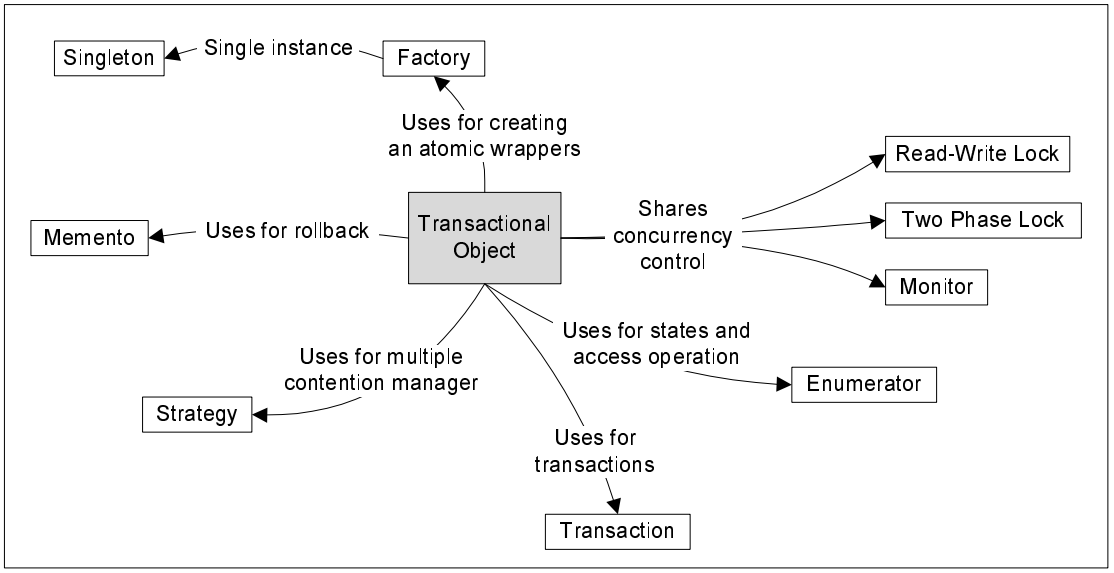


FIGURE 3.9: Relationship diagram of the Transactional Object

3.3 Summary

In this chapter we presented a proposal for a concurrent design pattern called "Transactional Object".

The used transactional memory is based on the concept of optimistic concurrency control. Due to its good scalability on multi-core systems, it has become an important research topic. Many implementations are already available and share the same basic design.

Design patterns can help to understand and solve problems. We contributed a new design pattern for concurrent access to shared objects using software transactional memory. With the help of class and sequential diagrams, we showed the basic functionality of an STM.

Chapter 4

Lightweight STM Support for an Unmanaged Language

It is normally quite complex to write parallel applications since we are used to think in a sequential way. When developing software using multiple threads one has to imagine all the problems which can occur through parallelism. Especially when the threads are accessing the same shared data. In this case the programmer has to protect this data from concurrent accesses. The development of transactional memory can reduce this burden. Unfortunately STMs are just one piece of the puzzle since the STM has to be merged with the concurrent program. Of course you can implement explicit calls to the transactional memory, but this exposes the software developer to the use of transactions and having their own pitfalls. A better solution is a semi-transparent support, *i. e.,* letting the system "transactify" the application. Then the programmer has to use dedicated language constructs [HF03] or declarative mechanisms and aspect oriented programming [RFF06b].

Both solutions have advantages, but also disadvantages. For instance programs with explicit calls can be faster since an advanced coder knows which variables need to be protected. However one has to be very rigorous, otherwise the application will cause problems when accessing unprotected shared data. The semi-transparent solution handles all variables within a transaction as if they need to be protected and thus is a little bit slower than the explicit one. The advantage is that it is less error prone.

Another solution would be a compiler and runtime support for transactional memory, *i. e.,* the programmer just highlights areas of code which have to be protected and the compiler takes care of it. This approach ensures the simplicity of the user interface and reduces the possibility of programming errors. Furthermore the compiler is identifying

the shared variables on their own and thus can optimize the code in a transaction as good or even better as if explicit calls would be used.

Unfortunately the development of such a compiler support just for research is quite expensive. When we started our work in this area at the end of 2006, only a few proprietary compilers were available. Those compilers were not compatible with other STM implementations and thus comparable performance results or representative workloads were not available. Another problem was that the modification of an existing compiler like the GNU Compiler Collection [GCC] (gcc) is quite complex, technically and politically. Especially if the modifications shall be integrated in the release of the main branch.

However without an easy application of transactional memory the development is slowed down. Therefore, our goal was to find a workaround for a fast STM support. We concentrated our investigation not on the modification of existing monolithic compilers, but instead we tried to reuse components of an existing compiler framework (which provides parts such as front-ends, back-ends for different platforms, or link time optimizers). One advantage was the machine-independent intermediate representation, which is generated automatically and can be easily altered. Another one was the support for front-ends of several programming languages. Our approach was aiming at a support for unmanaged programming languages such as `C` or `C++`.

Our solution to this problem is the TANGER framework, which was firstly presented in 2007 at the *Second ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing* in the paper *"Transactifying Applications using an Open Compiler Framework"*. This publication consisted of two tools, namely the TARIFA and the TANGER. TARIFA [TAR] can instrument existing libraries that export functions called from within atomic blocks. It was developed at the TU Dresden [TUD] and is not a contribution of this thesis. The ongoing development of TANGER after this publication was accomplished by our colleagues at the TU Dresden [TAN, Sys], led by Torvald Riegel. In this chapter we will describe therefore only the development of TANGER as presented in the given paper.

The following sections of this chapter are structured as follows. First, we will introduce the TANGER framework and evaluate its performance in the following Section 4.2. Finally, Section 4.3 concludes this chapter.

4.1 Tanger

Our TANGER framework is able to compile and transactify the source code of parallel `C`-applications. It can be used to create STM using applications or libraries as shown

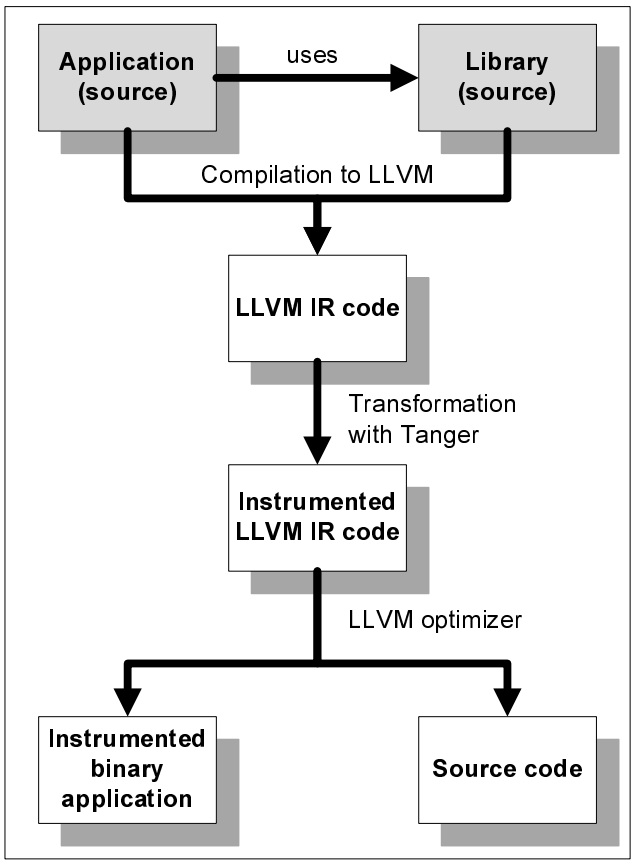


FIGURE 4.1: Architecture of TANGER to instruments machine-independent LLVM code.

in Figure 4.1. As one can see the normal STM free source code is compiled into the LLVM IR and the normal general-purpose optimization is done. Then the TANGER pass is applied to the IR and finally the LLVM back-ends produce native binary code or other supported source code.

TANGER consists, besides a LLVM pass, of an external header file providing marker functions to the programmers and an internal header file declaring the STM calls. In the rest of this section we are (i) introducing the marker functions, (ii) highlighting challenges with function calls, (iii) presenting our handling of nested transactions, and (iv) explaining our transformation approach.

4.1.1 Marker Functions

Since we wanted to create a possibility for transactifying `C` code without introducing new keywords, due to the complexity of altering the compiler, we had to find other solutions to highlight the parts of the source code which have to be handled within a transaction.

The first option was a do/while-loop as shown in Algorithm 10. However, this would be again explicit programming, since the programmer has to know technical details of the STM he wants to use. Furthermore if the programmer has only minimum knowledge on STM, he could miss-interpret the loop. The second option, which we use in fact, was to insert method calls like `startTANGER()` (Alg. 11 line 3) and `endTANGER()` (Alg. 11 line 6 and 10). Any code that has to be processed within a transaction has to be between those two calls. The methods are defined in a header file but are not implemented. Within the transformation process of the LLVM framework both calls are handled as calls to external libraries and though they are still visible in the IR as shown in the transformation section below. Additionally to this function calls a preprocessor command was introduced (Alg. 12 line 2). On the first look it seemed to work correctly, but later on problems occurred with branches leaving the atomic area earlier as shown in Algorithm 12 which should be equal to the code presented in Alg. 11. So this command was removed in later versions.

Additionally to the marker of the atomic block, functions for initializing (`initTANGER()`) and finalizing (`shutdownTANGER()`) the STM are needed. The initialization has to be done before the first use of the STM and the finalization of course after the last use or before terminating the application. Last but not least we added two marker functions for

Algorithm 10 Transaction in C manually instrumented.

```
1  ...
2    int do() {
3      do {
4        transaction_start();
5         ... //Do something transactional
6      }while(!transaction_commit())
7       ...
8    }
9  ...
```

Algorithm 11 Transaction in C with TANGER keywords.

```
1  ...
2    int do() {
3      startTANGER();
4       ... //Do something transactional
5      if (...) { //branch within the transaction
6        endTANGER();
7        return ...;
8      }
9       ...
10     endTANGER();
11     return ...;
12   }
13 ...
```

Algorithm 12 Transaction in C with ATOMIC() preprocessor keyword.

```
1  ...
2  #define ATOMIC(x) startTANGER(); x; endTANGER();
3  ...
4    int do() {
5      ATOMIC(
6          ... //Do something transactional
7      );
8        ...
9    }
10 ...
```

thread use. These two functions (`initThreadTANGER` and `shutdownThreadTANGER`) can be used in the same way as the standard initializer and shutdown but on thread level.

4.1.2 Function Calls

Function calls need special handling. Some of them are calling functions in external libraries. For those we can not guarantee their support for STMs. One way to enable this is to compile and transactify the library with TANGER. In case the source code of the library is not available, one could use the TARIFA framework (see [FFM+07]). Unfortunately we cannot provide an automatic check if the needed libraries are STM enabled, since they are already binaries and will be linked to the application during its compile process. Normally this is done after the TANGER pass has been applied.

For internal functions, our approach is to create a "transactional clone" of the function. Algorithm 13 shows the two functions `do_atomic()` and `do()` which are calling the same function `do_too()`, once from within a transaction and once without a transaction. The TANGER pass will create first a clone of the function `do_too()` then called `do_too_TANGER()` and then replaces the calls from within a transaction to this "transactional clone", whereas the transaction-free function will still call the original function. For the "transactional clone" the address of the transaction has to be added as parameter. The programming language `C` supports unfortunately functions with an open end parameter list, *i. e.,* the function `void foo(int arr [static 10])` has to be called with "at least" ten integers. The array `arr` is pointing to the first integer and the other ones can be accessed subsequently. If we would add our transaction parameter at the end, the compiler would have problems to determine where the array stops and where the transaction starts, thus we introduce the reference to the transaction at the beginning of the parameter list.

Algorithm 13 Function calls from within a Transaction.

```
1  ...
2    void do_atomic() {
3      ATOMIC(
4          ... //Do something transactional
5        do_too();
6      );
7       ...
8    }
9
10   void do() {
11      ...
12     do_too();
13      ...
14   }
15
16   void do_too() {
17         ...
18   }
19    ...
```

4.1.3 Nested Transactions

While executing parallel code, it may occur that another transaction will start while one is already running. This phenomena is called a nested transaction. In Algorithm 14 the atomic block in function `do_nesting()` is a nested transaction, if it is called by the function `do_atomic()`. The detection of a nested transaction in the control flow is well known and easy to implement. Normally a counter is incremented whenever a transaction starts and decremented when it commits or aborts without a retry. The so-called nesting level is then indicated by the counter. If the counter value is larger than two, this transaction is surrounded by at least one other transaction and therefore is nested. However, the value of the counter depends on the control flow.

Algorithm 14 Nesting of a Transaction.

```
1  ...
2            int do_atomic() {
3                    ATOMIC(
4          ... //Do something transactional
5          result = do_nesting();
6      );
7            }
8
9    int do_nesting() {
10     ATOMIC( //a nested transaction
11           ...
12     );
13   }
14    ...
```

An easy solution to this problem would be to stop the transformation so the user has to deal with it, but since TANGER shall help the programmer to transactify his code; this was not an option for us. We therefore decided for the presented prototype that it is suitable to remove the inner transaction and leave only the outermost initial transaction in the code. We flattened the transactions by deleting all nested transactions. With the value of the counter it is easy to determine the bounds of the initial transaction. As a result all function calls to `stm_start` and `stm_commit` will be deleted if the value of the counter is higher than 1.

4.1.4 Rollback of Transactions

Another important issue which had to be solved was the introduction of a rollback facility. Therefor we use the paired functions `setjmp` and `longjmp`. They are defined in the `C` standard library and enable programmers to use "non-local jumps". In the first step the `setjmp` function is called to save the actual environment and then the `longjmp` function can be used to go back to this point. Normally this mechanism is used for exception handling, because the control flow can jump out of many levels of nested function calls without dealing with their flags variables. Indeed our application is quite similar to exceptions since the `setjmp` is called at the beginning of a transaction and the corresponding `longjmp` call is used when an abort of a transaction is needed. Another option would have been to check during the whole transaction if it aborted or not. This approach would result in a more complex and also slower code.

4.1.5 Pass chain

The transformation process, *i. e.,* applying the TANGER pass to the code, is done after the translation of the source code to the IR and after the initial optimization. In fact these two steps are processed together by calling the `llvm-gcc` compiler with the parameters `-emit-llvm` to create the IR and `-O3` for enabling the optimization.

This optimization is needed to reduce the number of loads and stores. The thread local loads and stores are reduced to virtual register accesses, thus leaving only shared variables as "normal" loads and stores. The result of this reduction is that only the shared memory accesses are still handled as loads and stores. In the IR of a source code all function calls, and thus also the marker calls, are still in the same order as before as one can see in Algorithm 15 and 16.

Algorithm 15 Original C code with atomic block for testing containment in an integer set.

```
1  int set_contains(intset_t *set, int val)
2  {
3    int result;
4    node_t *prev, *next;
5
6    startTANGER();
7    prev = set->head;
8    next = prev->next;
9    while (1) {
10     v = next->val;
11     if (v >= val)
12       break;
13     prev = next;
14     next = prev->next;
15   }
16   result = (v == val);
17   endTANGER();
18
19   return result;
20 }
```

Algorithm 16 LLVM bytecode generated from Algorithm 15.

```
1  int %set_contains(%struct.intset_t* %set, int %val) {
2  entry:
3    ...
4    tail call void %startTANGER( )
5    %tmp22 = getelementptr %struct.intset_t* %set, int 0, uint 0
6    %tmp23 = load %struct.node_t** %tmp22
7    %tmp25 = getelementptr %struct.node_t* %tmp23, int 0, uint 1
8    %next.2 = load %struct.node_t** %tmp25
9    %tmp29 = getelementptr %struct.node_t* %next.2, int 0, uint 0
10   %tmp30 = load int* %tmp29
11   %tmp33 = setlt int %tmp30, %val
12   br bool %tmp33, label %cond_next, label %bb39
13   ...
14   tail call void %endTANGER( )
15   %result.0.in = seteq int %tmp30.1, %val
16   %result.0 = cast bool %result.0.in to int
17   ret int %result.0
18 }
```

Normally most STMs have several functions which have to be called to use them, *e. g.,* to initialize and dismiss the STM, to handle the transactions or to load and store variables within a transaction. The TANGER framework supports, as already mentioned, an internal header file. This file is the connector between TANGER and the used STM. For our prototype we were using an STM called TINYSTM [FFR08], but other STMs can be used as well by implementing the following functions:

- `int stm_load(stm_tx_t* tx, int *addr)` - loads something from an address
- `void stm_store(stm_tx_t* tx, int *addr, int value)` - stores something to an address
- `void stm_begin(stm_tx_t* tx)` - called at the begin of a transaction
- `void stm_abort(stm_tx_t* tx)` - called when a transaction is aborted
- `uint stm_commit(stm_tx_t* tx)` - called when a transaction committed
- `stm_tx_t* stm_get_tx()` - used to obtain the actual transaction of this thread
- `void stm_init()` - called to initialize the STM at system start
- `void stm_shutdown()` - called before the system shutdown
- `void stm_thread_init()` - called to initialize the STM after system start on thread level
- `void stm_thread_shutdown()` - called before the system shutdown on thread level

4.1.6 Transformation

The TANGER pass is altering the IR in several steps to create a transactified version. (i) As already described in Section 4.1.2, all functions are cloned resulting in a normal version and one that can be called from within a transaction. (ii) All transactional clones of functions will be transactified, *i. e.,* removing the marker functions and modifying loads, stores and calls. (iii) TANGER is searching for the marker function (see Sec. 4.1.1) by iterating over all original functions, which are highlighting the bounds of transactions. When the pass finds the start marker, it will alter the code until it finds the stop markers at the end of each control flow.

In the leftover part of this Section we will describe the replacement process in detail. First the exchange of the two marker functions is explained followed by the modifications of the loads and stores and concluding with the altering of function calls.

The marker `startTANGER()` has to be replaced by the pass with other code. First a reference to a new transaction is gained (Alg. 17, line 3). Then a new label is inserted as jump point if a transaction needs to retry (Alg. 17, line 4–6). The next line is preparing the rollback of the transaction (see Section 4.1.4) by getting a pointer to it. In line 8 we are inserting the `setjmp` call with the actual transaction as parameter.

Each `startTANGER()` marker can have several `endTANGER()` markers as already mentioned in section 4.1.1. Also one has to check if there are nested transaction and in our case delete both marker functions of them (see Section 4.1.3). The end marker is replaced by a `stm_commit` call, followed by a branch which proceeds if the commit was successful or jumps to the label inserted at the beginning of the transaction (line 6) in case a retry is needed (lines 25–27).

Between those two markers the pass has to replace all loads and stores with a function call to the corresponding functions of the STM (`stm_load` or `stm_store`). Additionally one has to add type casts if the data type accessed by the program does not match one of the native types managed by the STM implementation. These casts normally produce no extra instructions in the binary code (*e. g.,* casting between pointer types as on line 11, between unsigned and signed type as on line 21, or between memory addresses and unsigned value of the same size as on line 13).

All calls to an internal or external function from within an atomic block are handled by the system as if they were supporting transactions. For internal calls we have implemented a possibility to transactify these functions. When the pass has identifying an internal function call from within a transaction, it replaces this call with a call to the transactified version. The calls to external functions are not altered by the TANGER pass, but the pass will warn the user that the usage of this function can result in errors.

4.2 Evaluation

At the time we developed the first version of the TANGER it was quite difficult to compare our approach to other existing compile-time approaches and comparing state of the art systems with this early version would also be inappropriate. Thus we were comparing our approach with a manually instrumented and an optimized code compiled via LLVM, as well as with a standard gcc. The LLVM code was typically as good as the manually optimized code in our benchmarks.

For the tests we were using TINYSTM as backend STM. It is a lightweight word-based STM first described by Felber *et al.* [FFR08]. It uses encounter-time locking, similar to

Algorithm 17 Instrumented LLVM bytecode generated from Algorithm 16.

```
1  ...
2  cond_false:
3    %tmp = tail call %struct.stm_tx_t* (...)* %stm_get_tx( )
4    br label %cond_false.startTX
5
6  cond_false.startTX:
7    %txref21 = getelementptr %struct.stm_tx_t* %tmp, int 0, uint 6, int 0
8    %txref22 = tail call int %_setjmp( %struct.__jmp_buf_tag* %txref21 )
9    tail call void %stm_start( %struct.stm_tx_t* %tmp, int 0, int 1 )
10   %tmp22 = getelementptr %struct.intset_t* %set, int 0, uint 0
11   %castA23 = cast %struct.node_t** %tmp22 to uint*
12   %callB23 = tail call uint %stm_load( %struct.stm_tx_t* %tmp, uint* %castA23 )
13   %tmp231 = cast uint %callB23 to %struct.node_t*
14   %tmp25 = getelementptr %struct.node_t* %tmp231, int 0, uint 1
15   %castA24 = cast %struct.node_t** %tmp25 to uint*
16   %callB24 = tail call uint %stm_load( %struct.stm_tx_t* %tmp, uint* %castA24 )
17   %next.22 = cast uint %callB24 to %struct.node_t*
18   %tmp29 = getelementptr %struct.node_t* %next.22, int 0, uint 0
19   %castA25 = cast int* %tmp29 to uint*
20   %callB25 = tail call uint %stm_load( %struct.stm_tx_t* %tmp, uint* %castA25 )
21   %tmp303 = cast uint %callB25 to int
22   %tmp33 = setlt int %tmp303, %val
23   br bool %tmp33, label %cond_next, label %bb39
24   ...
25   %txref28 = tail call int %stm_commit( %struct.stm_tx_t* %tmp )
26   %txref29 = seteq int %txref28, 0
27   br bool %txref29, label %cond_false.startTX, label %bb39.commit30
28
29 bb39.commit30:
30   ...
```

the approach presented by Wang *et al.* [WCW+07], and guarantees consistent reads for every active transaction.

For our performance evaluation we used a "classical" *intset* micro-benchmark as it was used in [HLMS03] and by several other groups at that time. The benchmark is a set of integers implemented as a sorted linked list. The task of n parallel threads is to alter (add or remove) or look-up items. During each of these three operations a thread starts at the root of the linked list and digs through it until its operation is finished. The initial size of the list was 256 random elements. We also experimented with larger values and observed the same trends. We kept the size of the integer set almost constant by alternatively adding and removing an element. The rate of look-ups was set to 80%. Each experiment was run five times and we kept the median value.

We have compared three version of the benchmark: (i) a version that was optimized by hand, *i. e.,* we explicitly inserted the minimal number of transactional load and store operations, (ii) a version that was instrumented using TANGER, and (iii) a version that

was instrumented using TARIFA. We compiled the first version, using both gcc and the LLVM compiler chain. Experiments were run on a two-way machine with dual-core AMD Opteron processors at 2GHz, which makes 4 cores.

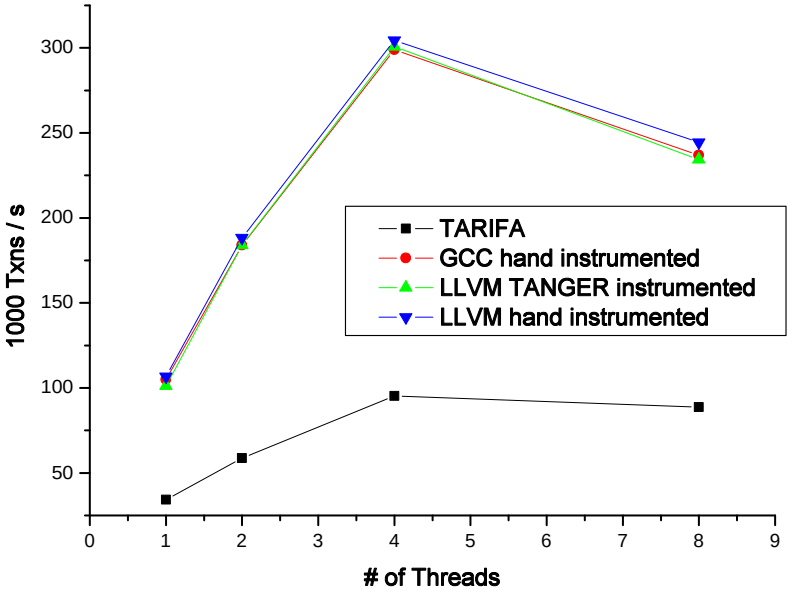


FIGURE 4.2: Performance of the TANGER, TARIFA, and hand-optimized benchmark versions.

Results are shown in Figure 4.2. One can observe that the version produced by TANGER performs as well as the hand-optimized version when scaling up the number of threads. The version instrumented using TARIFA runs slower due the additional accesses to the transactional memory (about twice as many), the management of the stack, and the suboptimal register allocation, but it still scales well. Interestingly, the LLVM compiler chain produced code that is as good as the code generated by the gcc.

The peak performance when using 4 threads is a result of using the two-way machine with dual-core CPUs. In this machine at most 4 threads can be processed in parallel and thus the best performance can be achieved when using 4 treads. If one uses more threads the performance will drop, due to the processor scheduling which has to deal with more threads then the machine can handle in parallel and a higher rate of aborts since the transactions will take longer to commit.

4.3 Summary

In this chapter we introduced the TANGER framework, which is able to transactify `C` source code using a compilation and modification facilities of the LLVM framework. TANGER is instrumenting the LLVM intermediate code in a semi-automatic way. Our tool considerably reduces the effort to use an STM, only some marker functions are added and TANGER is used for the compilation. This was a first step toward a easy application of STMs, even for programmers at beginner level.

Another feature of the TANGER framework is that it is STM independent, *i. e.,* one can use any appropriate STM by implementing the internal header file of TANGER. At the time we developed this system it was unique, all other available implementations could only handle one STM. The TANGER tool helped the community to save time in developing or adopting test suites for new STMs written in `C`, since TANGER can also use the STAMP [CMCKO08] benchmark suite.

The test results of TANGER compared to the manual implementation were really encouraging, since the overhead was very small. On the other hand possible errors are reduced by this automatism.

Chapter 5

STM for Event Stream Processing

The event-driven architecture is a pattern used in software architecture, dealing with the creation, modification, and reaction to events. In his paper, Chandy [Cha06] defines events as "a significant change in state", since their content influences the behavior of the system. For example, a telephone starts to ring when a call is incoming. In this case the incoming call would be the event produced by the caller. The event changes the state of the telephone from silent to ringing.

This event-driven architecture can be used for large sensor networks. Events produced by the sensors are streamed through different components, *e. g.,* filter or summing unit, and are finally collected in a data store. This type of applications is called event stream processing (ESP). It includes event visualization, event databases, event-driven middleware, event processing languages, and complex event processing (CEP).

Each component of an ESP system consists of input and output streams as connectors. If the processing done in the component depends only on the incoming event, the component is stateless, for example, removing all events with values below a certain border. Otherwise, when the result of the processing relies on an internal state, the component is called stateful. An example for a stateful component could be the mean average of the last ten events passing through the component.

Normally the stateless components contain a simpler logic than the stateful ones. Also it is easier to parallelize stateless components, since they only depend on the events. Typically parallelized stateless components share the same streams and just their access has to be managed. It is furthermore possible to de-multiplex the stream before a stateless component and multiplex it afterwards again. Depending on the implementation, it is possible that the order of the events has changed after the parallelized processing in the stateless component.

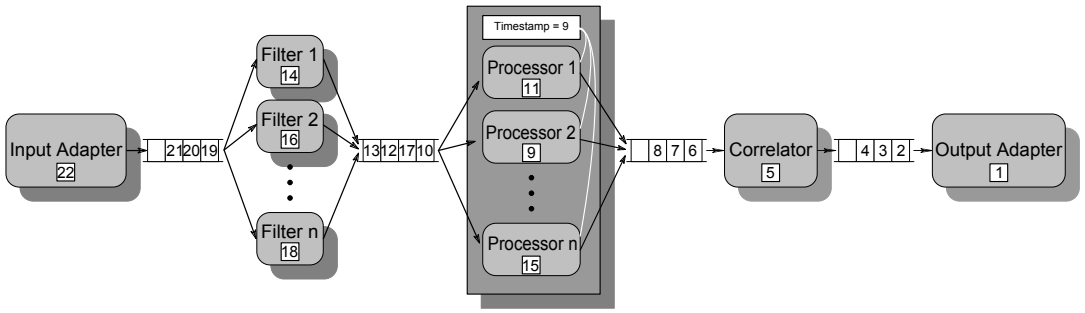


FIGURE 5.1: A simple typical ESP application network.

The parallelization of stateful components however is quite complicated, due to the maintenance of a shared state. These components share not only the streams, but also the internal state. Normally this is quite complicated and is used only for certain tasks. For example the Radix-2 algorithm [CT65], used for the fast Fourier transformation, can be parallelized using butterfly graphs where only a small amount of parallel components share the same state.

In this chapter, we consider ESP systems with components connected in a cascade (see Figure 5.1). We are interested in parallelizing the operations of the components by exploiting the processing capabilities of multi-core architectures. Some components are stateless and can be trivially parallelized, but in doing so they may reorder the events.

Other components are stateful and must typically (but not necessarily) process events in order. We assume that the order of events is determined when they enter the system, *e. g.,* by associating monotonically increasing logical timestamps with each of them.

Example 5.1. *Consider the simple ESP application network shown in Figure 5.1. The stream is first processed by an input adapter that transmits events in order to a filter component. As the filter is stateless (e. g., parsing XML events and converting them into a native format), one can have multiple instances of the component which processes events in parallel. The processing time might not be the same for each event and the outgoing stream might be "slightly" out of order. For instance, we observe in the figure that event* 17 *has overtaken event* 12*. The unordered stream enters a stateful "processor" component (e.g., stream query operator) that uses speculative execution to account for out-of-order executions. It allows events to be processed optimistically but does not output them until all preceding events have been completed. The stream then traverses a correlator component that forwards events in order to the output adapter.*

Since the parallelization of stateful components is not trivial we propose in this chapter to apply the concept of STM to those stateful components. Furthermore, we will present

and evaluate two implementations of this approach. The first one was realized in `C` together with Andrey Brito, a colleague from the TU Dresden [Str, Sys]. The second approach was implemented in `Java` and extended to a streaming library.

5.1 System Assumptions

Depending on certain conditions the environment might be structured in different ways and various components behave differently. For our development we make following assumptions on the components and the environment.

First, the events receive a logical timestamp as they enter the system. These timestamps need to be unique. They do not have to be strongly continuous but there must be a function to calculate the next timestamp and there has to be a total order. To keep that assumption valid throughout the system, each time an event is discarded (*e. g.,* a filter removing irrelevant event) a null event is inserted to carry the timestamp through the system. These null events are important especially if the components can be deployed in distributed computer nodes, as there is no way to distinguish a discarded event from a late one. We are considering the timestamps to be integer numbers starting from zero. Thus we are using the natural order.

Second, the algorithms written by the user to process the events should obey certain constraints. All user defined functions should guarantee progress (lock-free). In addition, the *process()* function cannot execute external actions, as these cannot be rolled back in case the transaction would abort, and it has to use the STM internal constructs for concurrent data structures.

Third, we assume that a node has sufficient memory to keep (out-of-order) events in memory until they can be processed and committed. In our experiments this case never occurred, nonetheless since the memory consumption strongly depends on the disarrangement of the events and their size, it could become a threat.

Last but not least, we assume that the connections between the components are reliable and events cannot be lost.

5.2 Enhancing Stateful Components

Each stateful component consists of input and output queues, internal states, and the processing logic. Usually these components have only one incoming and one outgoing queue, but it is also possible to use more queues, *e. g.,* when the component conjuncts

two streams. Each component is seen by the other components upstream or downstream as a black box. Thus they just know that there are other components from which they receive events or to which they are sending events.

5.2.1 Application Interface

In order to simplify the usage of our newly created STM support for stateful components, we decided to use a function based interface. Thus the user has to implement in several pre-defined methods the logic of the stateful component. This approach has the advantage that the user can concentrate on his needs and does not have to understand the underlying STM construct. In the `C` version we are using a header file and in `Java` an interface. Both approaches are supporting the same basic functions, while the `Java` version has a slightly more advanced interface, giving broader possibilities of interaction.

We separated the whole processing of an event in the component as follows: (i) The component takes an event out of the input queue. (ii) It processes the event. (iii) The processed event is put to the output queue. Based on these three steps we created the minimal interface for the component. Only step (ii) had to be split up in two parts: (iia) start the transaction and process the event in it, and (iib) try to commit the transaction. This breakdown is needed to process events out-of-order and/or in parallel and still be able to commit them in the correct logical order. Another advantage is that in step (iib) a sequential execution is guaranteed, which is sometimes needed for external access.

In the `C` version we named these functions *getEvent()*, *process()*, *onCommit()*, and *postCommit()*. In the `Java` version we used the following names *getEvent()*, *doProcess()*, *onCommit()*, and *putEvent()* since they corresponded better to the available functions of the underlying DSTM2.

Figure 5.2 shows the processing flow chart of all usable user-defined methods in the `Java` version. The gray shaded methods with an asterisk actually represent two methods: one ending with Once that will be executed only once (*e. g., onBeginOnce()*), and the other one ending with Local that is specific to each transaction (*e. g., onBeginLocal()*).

5.2.2 Predictors

With the use of STM in the stateful components, we are introducing optimistic speculation to increase the parallelization of the event processing. Unfortunately this optimistic approach can become, if overly applied, counterproductive and result in a high

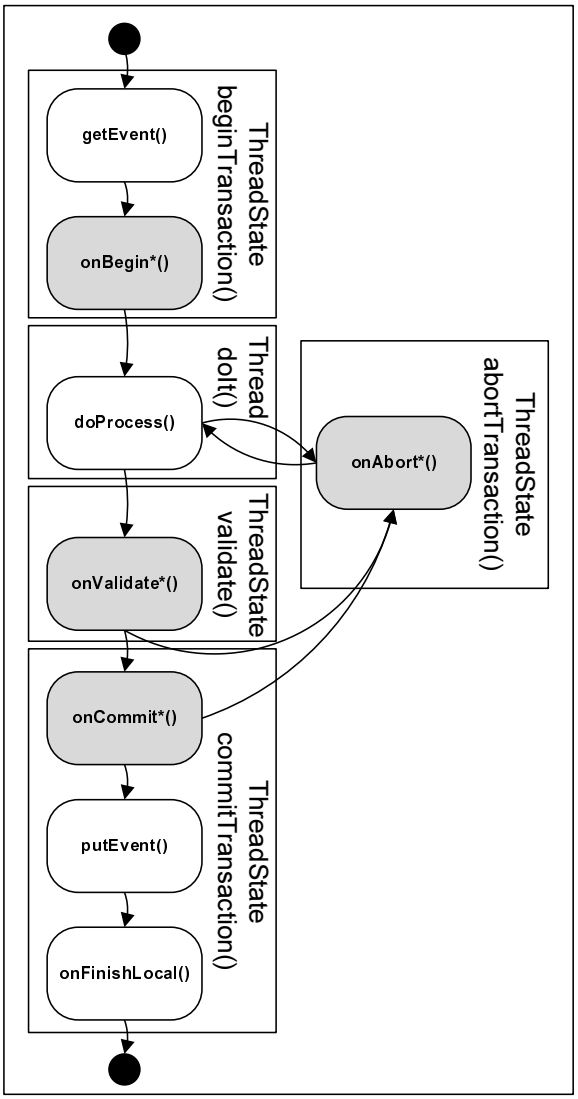


FIGURE 5.2: Flow chart of the user-defined Java methods

abort/retry rate. And finally the performance of the component can be lower than with the sequential version of the component.

In order to reduce such overly optimistic processing, we propose *conflict predictors* (or for short, predictors). Their task is to determine the likelihood of an abort and retry. We created several simple predictors and enabled an interface for users to develop their own ones.

One simple predictor we implemented limits the processing of events to only 10 timestamps ahead of the last committed timestamp. This limitation is already reducing the abort/retry rate efficiently, since the probability of success is inversely proportional to the difference between the current timestamp and the timestamp of the event.

Static analysis on the processing rules can also be used to generate predictors. For example, static analysis might be used to generate a predictor that conservatively predicts the possible conflicts.

Finally, dynamic predictors would operate based on statistics collected by the system at runtime. Such a predictor could be based on estimations on the density of dependencies, and thus collision, to estimate a recommendation value for the optimistic processing of an event. Another example of a dynamic predictor could be one that increments or decrements the number of logical time ticks in the future that are acceptable for speculation based on the current rate of aborts. Such a predictor is useful with operators that build sketches of data, like a histogram. In the case of a simple histogram, for example, the probability of a conflict depends only on the ratio between the number of speculations and the number of buckets. With the dynamic predictor, the speculation horizon would converge to this value. In Section 5.4, we evaluate the effect of simple predictors on the performance of our speculative worker.

5.2.3 Contention Manager

For solving conflicts between transactions, STMs normally use contention managers (CM). If two transactions are accessing the same shared data, a contention manager has to decide which transaction can be processed further and eventually be committed and which has to be aborted and has to restart its processing. Usually, when one transaction encounters a conflict it calls the CM with the competitor. This decision is sometime non-trivial and leads to several different approaches [GHP05, GC08, SS05b]. For instance for the aggressive CM the calling transaction always succeeds over the conflicting one, whereas the karma CM increases the priority of the transaction each time it acquires a new shared location and in case of a conflict the calling transaction makes a number of attempts equal to the priority difference.

For our STM-enhanced component we modified the timestamp CM to take the ESP environment into consideration. Our demand is that the events are leaving the component in the correct order of their unique event timestamps and not as originally the timestamp of the transaction creation. Our CM is still aborting the transaction with the highest timestamp as Algorithm 18 shows.

Algorithm 18 TimestampManager resolveConflict($Transactionme, Transactionother$)
Require: Valid transactions $me, other$.
Ensure: The abort of the transaction with the largest timestamp.

```
1: if me.getTimestamp() ≤ other.getTimestamp() then
2:     other.abort()
3: else
4:     me.abort()
5: end if
```

5.2.4 STM Modifications

In this subsection we describe the necessary changes to be done on an STM. As an example we will explain the modifications we did on the DSTM2.1.[1]

Algorithm 19 Original DSTM2.1 doIt(*xaction*)
Require: A valid callable *xaction*.
Ensure: The processing done by the callable will be done atomically.

```
1: loop
2:     beginTransaction()
3:     xaction.call() {Calling the user defined method}
4:     if commitTransaction() then
5:         return
6:     end if
7:     abortTransaction()
8: end loop
```

Usually, DSTM2 executes a user-defined runnable class in a transaction, as shown in Alg. 19. Upon conflict, the processing is aborted and restarted automatically until the transaction successfully commits. Additionally, it is possible to inject code at certain execution points, *e. g.,* before the transaction starts, at the commit of a transaction or in case of an abort. The sphere of action of these functions can be defined to be `local` or `global`, so either they affect only this thread or all threads. There is another possible option which can be used to inject code at the mentioned positions. With the addition of `once` the code in this function would be called once for the whole execution of this transaction. For example the `onAbortOnce` function is executed when the first abort occurs, but never on the following aborts.

In the modified version of the DSTM2 used in *TM-Stream*, the user still has to provide a class implementing the actions to be done on the events covered by a transaction (Alg. 20). In our case, however, we have an infinite loop since the flow of incoming events is endless. In every execution, the thread has to check first if there are events

[1] Beta version from June 2008, available from *http://www.cs.brown.edu/~mph/*.

Algorithm 20 TM-Stream doIt()

```
1: loop
2:     checkPredictorAndWaitingTX()
3:     if getEventAndBeginTransaction() then
4:         if predict() then
5:             processEvent() {Calling the user defined method}
6:             tryToCommitTransaction()
7:         end if
8:     end if
9: end loop
```

ready for processing or for committing. If there are no further events in the predictor or in the waiting queue, the thread tries to get a new event covered in a new transaction. In the next step the event is evaluated by the predictor, if it can be processed or should wait for it. In case the prediction was successful, the *processEvent()* method is called and afterwards the transaction tries to commit. If the event can not commit, the transaction is hibernated and stored in the waiting queue for later commit or abort. As a result we do not need the explicit abort in the *doIt* function.

Algorithm 21 TM-Stream checkPredictorAndWaitingTX()

```
1: flag ← TRUE
2: while flag do
3:     if checkPredictor(clock.getTime()) then
4:         processEvent() {Calling the user defined method}
5:         tryToCommitTransaction()
6:     else
7:         flag ← FALSE
8:     end if
9:     if checkWaitingTX(clock.getTime()) then
10:        tryToCommitTransaction()
11:        flag ← TRUE
12:    end if
13: end while
14: return
```

Internally, the commit procedure was also changed. In the original DSTM2 code, a transaction commits if it executes until the end without conflict. In *TM-Stream*, since the events must commit in the order defined by their timestamps, we use an internal clock to determine when transactions can commit: a transaction can only commit when the clock reaches the value of the timestamp. If a transaction wants to commit but its timestamp is still higher than the internal clock it will be delayed and the thread can process another event. Since it is likely that some events may encounter high contention during their processing, it is useful to delay their processing by a so-called predictor.

Transactions paused by the predictor are never aborted since they did not start the execution of *processEvent()* and thus never accessed transactional data. When incrementing the internal clock it is necessary to check if delayed transactions (waiting to commit or paused by the predictor) can be continued. This check (Alg. 21) is done by each running thread at the beginning of the loop (Alg. 20, line 2).

Algorithm 22 Original DSTM2.1 commitTransaction()

```
1: if (validate() and transaction.commit()) then
2:     return TRUE
3: end if
4: abortTransaction()
5: return FALSE
```

Algorithm 23 TM-Stream tryToCommitTransaction()

```
1: if transaction.getTimestamp()=clock.getTime() then
2:     if (validate() and transaction.commit()) then
3:         clock.incrementTime()
4:     else
5:         abortTransaction()
6:         delayTransaction()
7:     end if
8: else
9:     delayTransaction()
10: end if
11: return
```

Compared to the original committing function (Alg. 22), transactions can only commit if their timestamp has the same value as the internal ordering clock. This clock is incremented when a transaction has committed (Alg. 23 line 3). To support certain use cases, we allow overriding the behavior of the clock (e.g., one could branch the streams and distribute even- and odd-numbered events to different components with internal clocks that would take the appropriate values). Additionally, we extended the DSTM2 with a new contention manager that aborts, upon conflict, the transaction with the higher timestamp (even if it is fully processed and only waiting to commit).

Overall, *TM-Stream* still supports most of the additional user-defined methods of DSTM2 (*e. g., onBeginLocal()*, *onCommitOnce()*, ...) with the restriction that global functions should not be used (e.g., *onBegin()*, *onCommit()*, ...). It is still possible, however, to process non-concurrent operations outside transactions with the remaining functions. Figure 5.2 shows the processing flow chart of the usable user-defined methods. The gray shaded methods with an asterisk actually represent two methods: one ending with *Once* that will be executed only once (*e. g., onBeginOnce()*), and the other one ending with *Local* that is specific to each transaction (*e. g., onBeginLocal()*).

5.3 TM-Stream Framework for Java

We have developed a set of simple components to evaluate our `Java`-based event streaming framework. All components are connected via streams or queues. They are described in Table 5.1. The components all have in common that they can be started, stopped and reset. We categorized our developed components into the following classes, depending on their task. (i) Processors can modify events either in a stateful or stateless way. (ii) Creators and Terminators are providing artificial sources and sinks for test runs. (iii) Adapters contain components for an advanced relay of streams, *e. g.,* via network or joining and splitting streams. (iv) Queues, which are the basic connectors between all other components.

5.4 Evaluation

In this section, we present performance results of our speculative execution engines. We will evaluate the different implementations first and finally compare both. Most of the presented results were published in "Speculative Out-of-order Event Processing with Software Transaction Memory" [BFSF08] and "TM-STREAM: an STM Framework for Distributed Event Stream Processing" [SFF09].

5.4.1 TinySTM-based Engine

We shall illustrate the operation of our ESP system and our speculative execution algorithm on the sample application network of Figure 5.3. This network consists of 5 components, with an event source (the input adapter), a stateless component (the filter), a stateful component (the processor), a correlator and an event sink (the output adapter). Events generated by the source have monotonically increasing logical timestamps with no gaps. Events are shuffled when traversing the parallelized filter component and are received out-of-order by the processor, which processes them speculatively and reorders them upon commit. In our evaluation, the output adapter also performs verifications on the order and values of the events processed by the previous modules, in particular it checks that the results generated by the processor component are correct. All tests were run on an 8-core Intel Xeon machine at 2 GHz running Linux 2.6.18-4 (64-bit).

We consider 6 scenarios for our performance analysis. The non-speculative scenarios use a sequential processing component. Thus, an event can only be processed after the event with the immediately preceding logical timestamp is committed. The speculative scenarios use the STM-equipped processor with 4 worker threads. Thus, up to four events

	Component	Function	Input	Output	Processing
Processor	Stateful				
	TM-Stream	Processes the events in order with the support of an STM.	1 Queue	1 Queue	n Threads
	Sequential	Processes the events in order. If they arrive unordered they are stored and ordered.	1 Queue	1 Queue	1 Thread
	Stateless				
	Mixer	Mixes up the incoming events by random processing time.	1 Queue	1 Queue	n Threads
	Correlator	Checks if the processing of the stateful component was successful.	1 Queue	1 Queue	1 Thread
Creator and Terminator	Source				
	Source with Event<long>	Creates events with an integer value as payload.	1 Queue (optional)	1 Queue	1 Thread
	Sink				
	Sink with Event<long>	Checks that events are in order and gather statistics. Afterwards it destroys or recycles them.	1 Queue	1 Queue (optional)	1 Thread
Adapter	Network				
	QueueToTcp	Retrieves events from a queue and sends them via TCP.	1 Queue	TCP Connection	1 Thread
	TcpToQueue	Receives events from TCP and puts them into a queue.	TCP Connection	1 Queue	1 Thread
	Join and Distribute				
	Demultiplexer	Distributes events to several queues by a user defined algorithm.	1 Queue	n Queues	1 Thread
	Multiplexer	Joins events from several queues into one using a round-robin strategy.	n Queues	1 Queue	1 Thread
	Parallel Multiplexer	Joins events from several queues into one using parallel threads.	n Queues	1 Queue	n Threads
Queues	Blocking	Blocks when a queue becomes too large or when it is empty.			
	Ordering Queue	Orders the incoming events by their timestamp.	-	-	-
	Not Ordering Queue	FIFO queue.	-	-	-
	Not Blocking	Busy waiting until an event has arrived.			
	Ordering Queue	Orders the incoming events by their timestamp.	-	-	-
	Not Ordering Queue	FIFO queue.	-	-	-

TABLE 5.1: TM-Stream Components

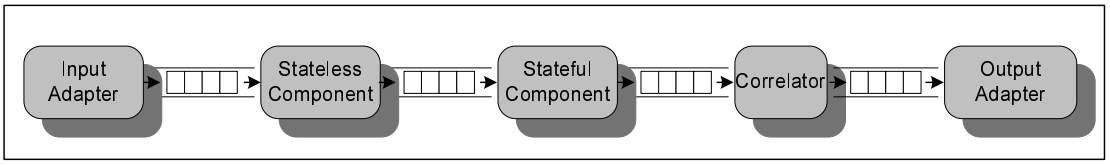


FIGURE 5.3: A simple typical ESP application network.

can be processed at a time. In these experiments, with exception of the one with ordered event input, we used a predictor that allowed an event to be speculatively processed only if the difference between its timestamp and the last committed timestamp was less then 10. For the speculative execution with sorted input, we set this value to 500 to evaluate the maximum level of parallelism achieved.

The ordering of events depends on the number of filter worker threads. If there is a single filter thread, no order inversion occurs. The unordered version uses 4 filter threads. As the threads concurrently take events from the common input queue and insert them in the output queue, the order will be changed.

Algorithms 24 and 25 illustrate the two kinds of processing functions we are using in the different scenarios. These algorithms resemble sketching operators, like histograms. Algorithm 24 will cause an event e_{t+20} to always conflict with event e_t. Algorithm 25 may cause additional conflicts with events in the form e_t and e_{t+5} with e_{t+1} and e_{t+6}, respectively. A busy waiting loop is used to ensures that the processing has a predefined minimum duration.

Algorithm 24 ProcessWithConflict1(*Event e*)

```
1: startTime ← getTime()
2: pos ← e.ts % STATE_SIZE
3: updateState(state[pos])
4: while getTime() < startTime + TASK_SIZE do
5:                                   {Ensure a minimum duration for the task.}
6: end while
```

Algorithm 25 ProcessWithConflict2(*Event e*)

```
1: startTime ← getTime()
2: pos ← e.ts % STATE_SIZE
3: updateState(state[pos])
4: if (CONFLICTS = 1 ∧ e.ts % 10 = 0) ∨ (CONFLICTS = 2 ∧ e.ts % 5 = 0) then
5:                                   {Also update next state entry (create conflicts).}
6:     updateState(state[(1 + pos) % STATE_SIZE])
7: end if
8: while getTime() < startTime + TASK_SIZE do
9:                                   {Ensure a minimum duration for the task.}
10: end while
```

The 6 scenarios were defined as follows.

Ord. Seq. Ordered event input, with sequential processor: in this case, there is only one filter component (and thus, no order inversion) and single thread processing component. The processing component task is illustrated in Algorithm 24. The `STATE_SIZE` value has no effect on the computation.

Unord. Seq. Unordered event input, with sequential processor: there are multiple filters and the processing component is still single threaded. The processing component task is illustrated in Algorithm 24. Again the `STATE_SIZE` value has no effect.

Ord. Spec. Ordered event input, with speculation: only one filter and the processing component try to optimistically parallelize the event to be processed. The processing component task is illustrated in Algorithm 24. The `STATE_SIZE` value, which determines the frequency of conflicts, is set to 1000. In this case, there are no conflicts between events being processed but they are processed in parallel and committed in order.

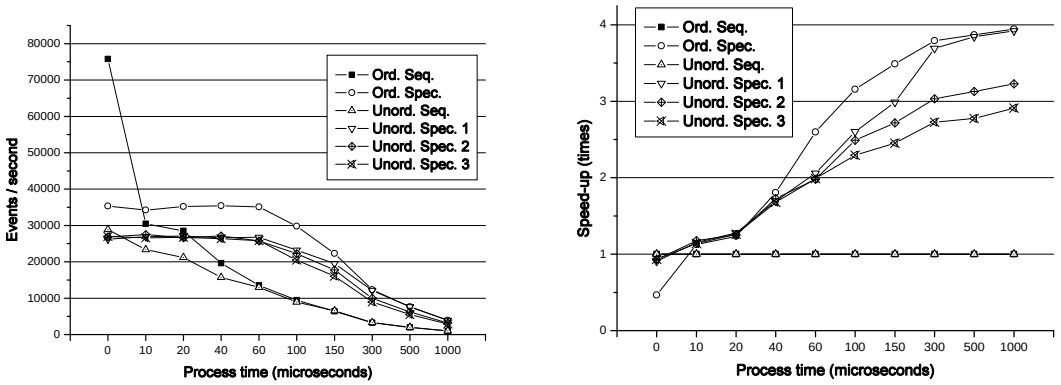
Unord. Spec. 1 Unordered event input, with speculation and 0% conflicts: there are multiple filters and the processing component tries to optimistically parallelize events and may process them out-of-order. The processing component task is illustrated in Algorithm 24, with the `STATE_SIZE` value set to 20, which results in no conflicts in the horizon allowed by our default predictor.

Unord. Spec. 2 Unordered event input, with speculation and 10% conflicts: there are multiple filters, with optimistic parallelization and out-of-order processing. The processing component task is illustrated in Algorithm 25, with the `STATE_SIZE` value set to 20 and `CONFLICTS` set to 1.

Unord. Spec. 3 Unordered event input, with speculation and 20% conflicts: there are multiple filters, with optimistic parallelization and out-of-order processing. The processing component task is illustrated in Algorithm 25, with the `STATE_SIZE` value set to 20 and `CONFLICTS` set to 2.

The results of the experiments are illustrated in Figures 5.4(a), 5.4(b) and 5.5. For ordered executions, one can observe in Figure 5.4(a) that, even though the speculative version has no conflicts and a very far horizon for speculation, the throughput of the non-speculative version is initially much higher. This is due to the overhead of the STM and the synchronization costs of the ordered commit. However, this overhead becomes much less significant as the task size grows. Figure 5.4(b) also shows that the overhead of parallelization becomes negligible with longer tasks and the speedup of the parallel processor improves almost linearly with the number of processor workers.

For unordered executions, the performance of the non-speculative version is not better than the speculative ones even with shorter task durations. A deciding factor in this case is that the non-speculative version must wait until the proper event arrives. This is not required by the speculative version. As the size of the tasks grows, the ordered and



(a) Throughput.

(b) Speed-up.

FIGURE 5.4: Comparison of the various configurations.

unordered non-speculative versions tend to perform similarly. This can be explained by the fact that if the processing of the tasks is long enough, by the end of the processing of the i^{th} event, the $(i+1)^{th}$ event will have already arrived and no waiting will be necessary.

The difference between the three unordered speculative experiments can be seen more clearly in Figure 5.4(b). With longer tasks the various synchronization costs and STM overheads tend to be negligible and the difference in the amount of useful work appears clearly: the performance of the 20%-conflict version is about 26% lower than the speculative version with no conflicts; the performance of the 10%-conflict is about 18% lower than the no-conflict one. Finally, the 0% version (*Unord. Spec. 1*) has a performance similar to the ordered one with a further speculation horizon (*Ord. Spec.*), because with large process times events are naturally ordered by the interaction of the queue and the processing threads. In this case, the scheduling variations have less impact and thus there are rarely two events with conflicting timestamps (*e. g.,* e_t and e_{t+21}) being processed by different threads at the same time.

Another metric that can be very significant in event stream processing is latency. The end-to-end latency, from the input adapter to the output adapter, is depicted in Figure 5.5. These graphs show that the latency for the non-speculative version is far lower for ordered events. But this advantage quickly disappears when the processing length grows and the parallelization becomes profitable. For the same reason, the right-hand side of the graph shows a considerable difference in latency in the non-speculative executions in comparison to the speculative ones.

To illustrate the impact of predictors, we analyzed the performance of 5 very simple predictors in two scenarios that could exhibit conflicts in optimistic executions. The first

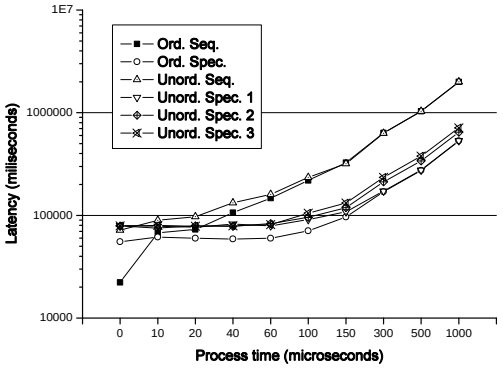


FIGURE 5.5: Average end-to-end latency of the various configurations.

scenario is similar to the fourth scenario in Section 5.4.1, with the conflicts defined by Algorithm 24 for `STATE_SIZE` set to 20. The second scenario defines conflicts similarly to the sixth scenario in Section 5.4.1, with `STATE_SIZE` set to 20 and `CONFLICTS` set to 2 in Algorithm 25. The 5 predictors were defined as follows.

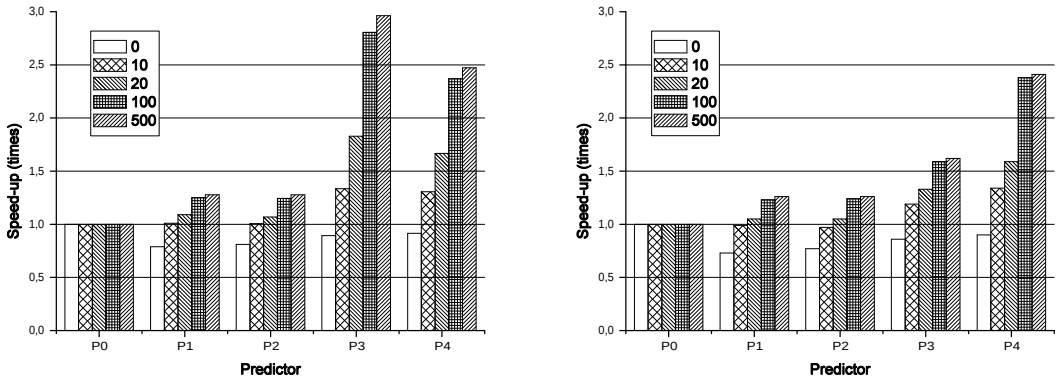
Predictor 0 returns *true* for every call, and thus won't delay any processing. This predictor is useful if conflicts are rare, otherwise, as discussed previously, the system may enter a steady state in which all events are processed sequentially.

Predictor 1 returns *true* only if the event is within 20 logical clock ticks from the timestamp of the last committed event. It predicts perfectly the conflicts in our first scenario, but limits its return to a boolean value indicating if a conflict will occur or not in case the evaluated event is processed immediately.

Predictor 2 returns *true* if the event will conflict according to the collisions specified in Algorithm 25, for `STATE_SIZE = 20` and `CONFLICTS = 2`. This predictor is able to perfectly predict the conflicts for our second scenario, but as with Predictor 1, limits its evaluation to a boolean value.

Predictor 3 works as predictor 1, but instead of returning $false$ if the event is likely to conflict, it returns the event that should be committed before the evaluated event can be processed, so that no conflicts would ever occur. For example, the evaluation of event e_{23} would return 3, indicating that event e_{23} is likely to conflict with event e_3, and thus, should wait for it to be committed before it is processed. This predictor returns *true* if the event will not generate a collision, *i. e.,* the events that could conflict were already committed.

Predictor 4 has the same knowledge as predictor 2 but, as predictor 3, returns a value indicating the event that should be committed before the event that is being evaluated can be processed without generating conflicts.



(a) Scenario 1.

(b) Scenario 2.

FIGURE 5.6: Speedup for different task sizes.

The speedup achieved by the predictors in the first scenario is depicted in Figure 5.6(a). In the figure, we have a bar for each of 5 possible event processing task lengths. For tasks with short processing times we observe a decrease in performance when using predictors. This is indeed expected, because with such settings the best results are obtained with sequential processing as the relative overhead of speculation is significant. The best approach would be not to do any speculation at all and use a predictor that returns *true* only if the event is the next to be committed. With such a pessimistic predictor the results would be similar to predictor 0, but with much less CPU utilization.

When increasing the duration of the tasks, we observe noticeable improvements for the more sophisticated predictors. As expected, predictor 3 has better performance as it is capable of telling when a conflict is going to occur and what should be done to avoid it without sacrificing any non-conflicting parallelism. The usage of predictor 4 leads to sub-optimal performance as it predicts more collisions than are actually happening.

Results for the second scenario are depicted in Figure 5.6(b). The same reasoning as for the first scenario applies for tasks with short processing times. For longer tasks, one can clearly see that predictor 4 produces higher speedups, as expected. To give an idea of the amount of useful work, with predictor 0 all events are aborted once (and then retried when their timestamp is reached); with predictor 3 there are 40% aborts; with the fourth predictor there are no aborts.

Although having the perfect predictors lead to the best results, one can observe in the graphs that sub-optimal predictors also enable impressive improvements. Predictor 3 in the second scenario is a good example: even though there are still 40% unforeseen conflicts, the overall speedup is still above 1.5. As a matter of fact, one cannot expect

perfect predictors to be available for most practical cases. Some predictions may depend not only on event parameters, but also on the current state of the system, which could change between the time the event is evaluated until it is processed. With perfect predictors there would be no need for speculation support. Thus, exactly because of the inability to perfectly predict conflicts, support from a speculation infrastructure is required. The infrastructure we developed can dynamically monitor the processing and reevaluate events when conflicts occur and, in spite of that, exploit as much parallelism as the predictor is able to identify.

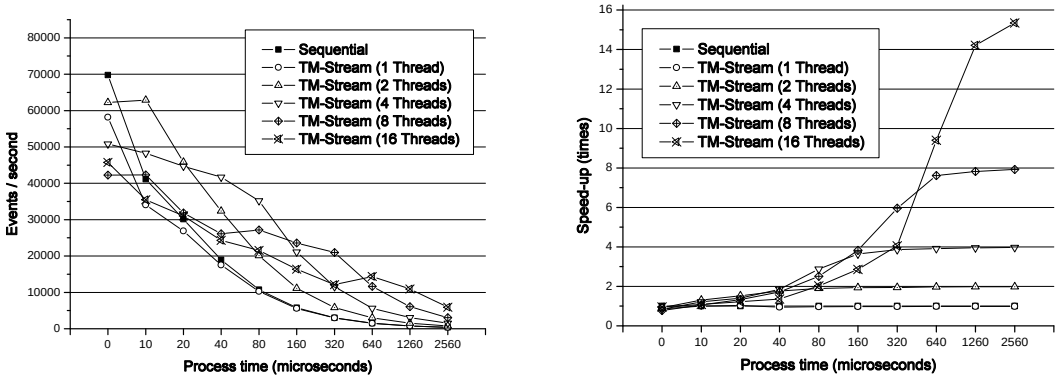
5.4.2 DSTM2-based Engine

In order to evaluate *TM-Stream*, we set up an experimental processing stream similar to the one used with the tinySTM-based engine. All tests were run on an 16-core machine with 4 AMD QuadCore processors at 2.20 GHz running Linux 2.6.25 (64-bit) and `Java` 1.6_10. The runtime of all experiments was 20 seconds with a warmup period of 1 second. Each point in the following graphs represents the average of three measurements.

The sample application network consists of 5 components as shown in Figure 5.3. Events are generated in the *Input Adapter* and processed in a *Stateless Component*, which can mix the order of the event as they are processed by several parallel threads with a random process time ($0 \leq$ process time $< 100\,ns$). Afterwards, the events are processed in a *Stateful Component*, which receives the events in the sequence given by the *Stateless Component* but have to process them in order. Then, the events are checked in the *Correlator* and, finally, they are collected in the *Output Adapter* that checks their order and produces the statistics.

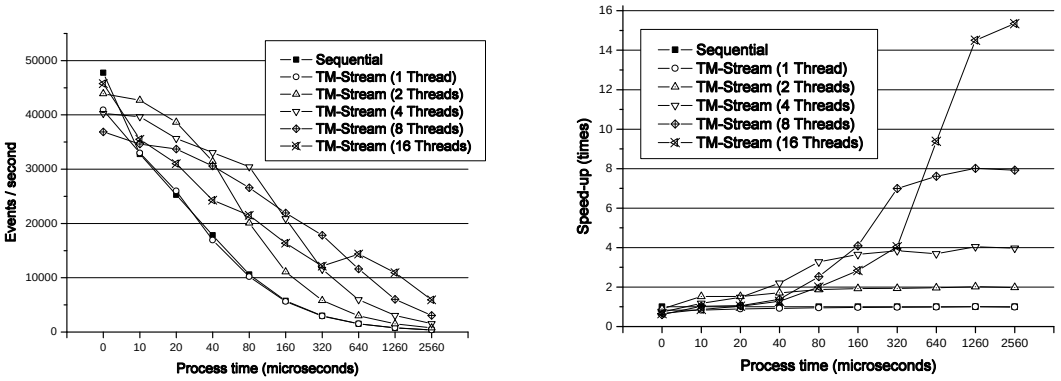
The stateful sequential component first sorts incoming events and then processes them in order. For the *TM-Stream* component, we consider a processing method where every 23rd event has a concurrent access to a local state, *i. e.,* event e_i must be committed before the events $e_{i+(n*23)} \forall n > 0$ can be processed. The prime number 23 was chosen to avoid possible side effects between numbers of threads and the events, since there wouldn't be a speed-up anymore if we have fewer local states than parallel threads. We are also applying a predictor (see Section 5.2.2) to delay the processing of those events to minimize aborts. In our case no event is processed if it is more than 22 time units in the future, which makes it a perfect predictor without any abort (except upon conflicting transactions).

The results of the experiments with ordered incoming events are shown in Figures 5.7(a) and 5.7(b). From both figures one can observe that, with increasing processing time per event, the speed-up of the *TM-Stream* component (with respect to the sequential



(a) Throughput. (b) Speed-up.

FIGURE 5.7: Events arriving in-order.



(a) Throughput. (b) Speed-up.

FIGURE 5.8: Events arriving out-of-order.

component) improves. Apparently, each configuration has an optimum that is related to the processing time and the number of parallel working threads. At that point, all threads are fully utilized. This behavior is also reflected by the corresponding speed-up figure in which the speculative components with more than 1 thread reach a maximum value and then remain flat. Figure 5.7(b) also gives a good indication of the scaling with multiple threads: the speed-up almost reaches the number of threads, which is a very good result.

For the next experiments, the stateless component consists of 8 parallel threads that can produce events out-of-order. The results are shown in Figure 5.8(a) and 5.8(b). Of course the throughput is lower than in the ordered case but this difference is caused by the required ordering effort. The conclusions that can be drawn from both figures are nonetheless similar to the previous experiment.

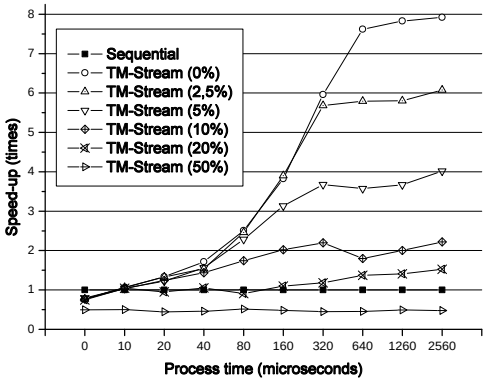


FIGURE 5.9: Speed-up for out-of-order event arrivals with contention.

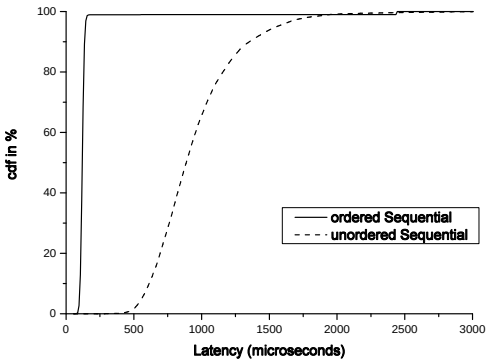


FIGURE 5.10: CDF for ordered and unordered Event arrival.

In another experiment we artificially generated contention, *i. e.,* a certain amount of events are also accessing the local state of their following events. The *TM-Stream* component is using 8 threads in parallel for this setup and the events arrive out-of-order. As a result the abort rate of the STM is increased and events have to be reprocessed, which limits the speed-up significantly. This becomes apparent in Figure 5.9 where the speed-up is reduced dramatically with higher contention. When reaching 20% contention, *TM-Stream* becomes slower than the sequential implementation.

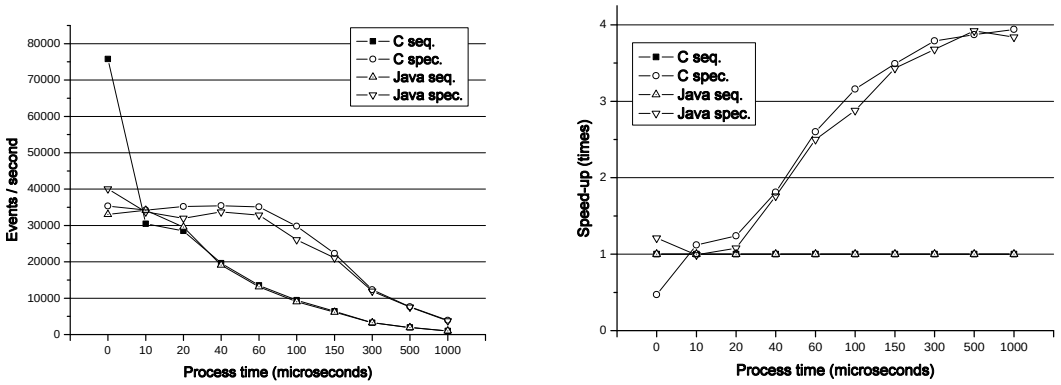
Figure 5.10 shows the distribution of the latencies for ordered and unordered sequential processing. As expected, one can observe that out-of-order events have a higher latency then ordered events.

5.4.3 Comparison of the Engines

In this last part of the evaluation, we will compare the TinySTM-based engine with the DSTM2-based approach. Therefore we are using the same set-up as described in Section 5.4.1. The tests were run again on the 8-core Intel Xeon machine at 2 GHz running Linux 2.6.18-4 (64-bit) and both STM-equipped processors were using 4 parallel threads for the processing.

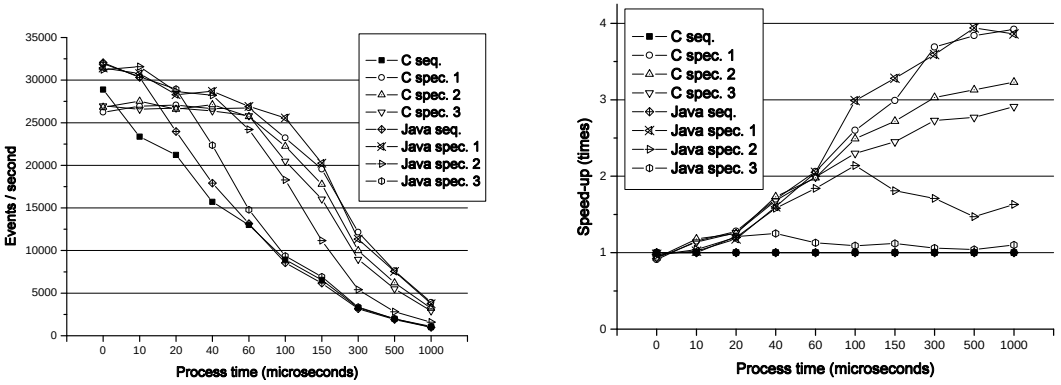
When the events are arriving in-order, both implementations behave similarly as one can see in Figure 5.11. The average deviation in the throughput of both versions is 3% and the DSTM2-based version is usually a little bit less performant.

In Figure 5.12 we compare both systems with events arriving out-of-order. As one can see, the conflict free-versions (Spec. 1) are performing equally good and are nearly scaling up to 4 times in terms of speed-up. When the systems have to deal with conflicts



(a) Throughput. (b) Speed-up.

FIGURE 5.11: Events arriving in-order.

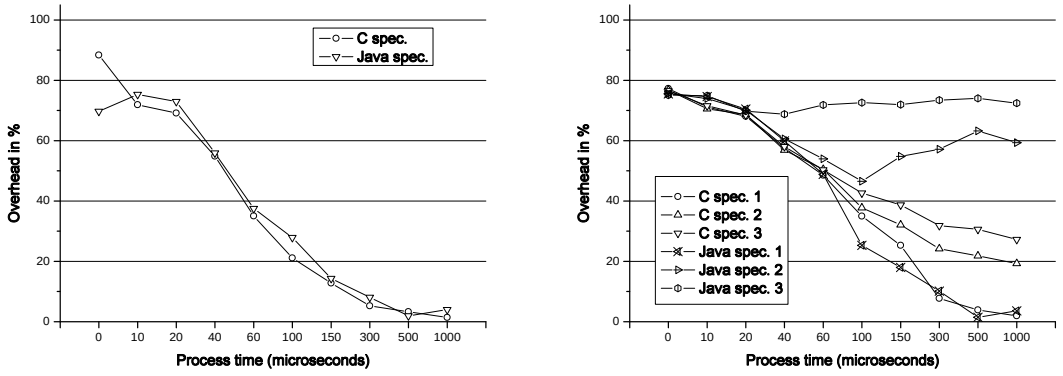


(a) Throughput. (b) Speed-up.

FIGURE 5.12: Events arriving out-of-order.

(Spec. 2 and Spec. 3) the TINYSTM-based version outperforms the DSTM2-based implementation. A reason for this discrepancy is likely in the STM algorithm that differs in both implementations.

Deviated from the speed-ups in Figure 5.11(b) and 5.12(b), we analyzed the overhead of both STM implementations compared to the maximal speed-up. As one can see in Figure 5.13, the overhead depends in both implementations on the length of the processing time. With increased processing time the likelihood of parallel access to incoming streams as well as parallel commit attempts is reduced and thus as well the overhead. When we introduced conflicts to both systems (Spec. 2 and Spec. 3 in Fig. 5.12(b)), the overhead is larger then without them, since the STM has to resolve the conflicts by aborting one of the conflicting transactions.



(a) Ordered.

(b) Unordered.

FIGURE 5.13: Measured overhead.

5.5 Summary

In this chapter, we presented two speculative execution environments for event stream processing components. We used STM to speculatively process events in parallel and delay the commit of the associated transaction until we have successfully processed preceding events.

The evaluations of our system confirms that good performance improvements can be achieved through speculation even if some computations may have to be disregarded and re-executed. Within a reasonable processing time ($\geq 100\,\mu$sec) our systems scale almost linearly with the number of threads for stateful components.

We have also proposed using application specific conflict predictors to drive the system towards more efficient executions. These predictors can be specified by the user or generated automatically by static or dynamic analysis. We showed that even very simple predictors (*e. g.,* limiting how far in the future the speculation should go) can improve the speedup and that they do not need to be always correct to be useful.

With the application of STMs to stateful components one can simplify the parallelization of these kinds of components. Of course, the distance between events accessing the same state has to be big enough; otherwise the processing is reduced to sequential throughput or below. A big advantage for the programmer is that deterministic and non-deterministic conflicts can be handled equally well with the same programming efforts.

Chapter 6

Distributed Speculations in Event-Based Systems

In the previous chapter we presented two implementations for the application of STM to event stream processing. However, these approaches are limited to execute speculative processing on a single component. In this chapter, we extend the speculative manipulation of events beyond the boundaries of single components. Our approach allows us to transmit events produced by transactions that are not yet committed, to remote processors. For distributed speculative execution, we use our ESP-enabled DSTM2 version, presented in Chapter 5.

6.1 Distributed Transactions

In the basic version of TM-Stream, events, speculatively processed within a transaction, had to wait until their commit timestamp was reached to be committed. This mechanism is needed to ensure that the events are processed in the correct order without conflicts. In case of a conflict, the event with the higher timestamp would need to rollback and be reprocessed. Unfortunately, it can then occur that the following ESP components remain idle, because of an event arriving late and thus causing a jam.

Therefore, it could be useful in our opinion to extend the boundaries of the speculation from one component to a cascade of components. The first speculative component would propagate events, even if the associated transaction is not yet committed. The downstream component can already process this event speculatively, but commit it only if the received event is finally committed on the prior component. This speculative behavior can reduce the latency in the system if there is little contention. Note that the

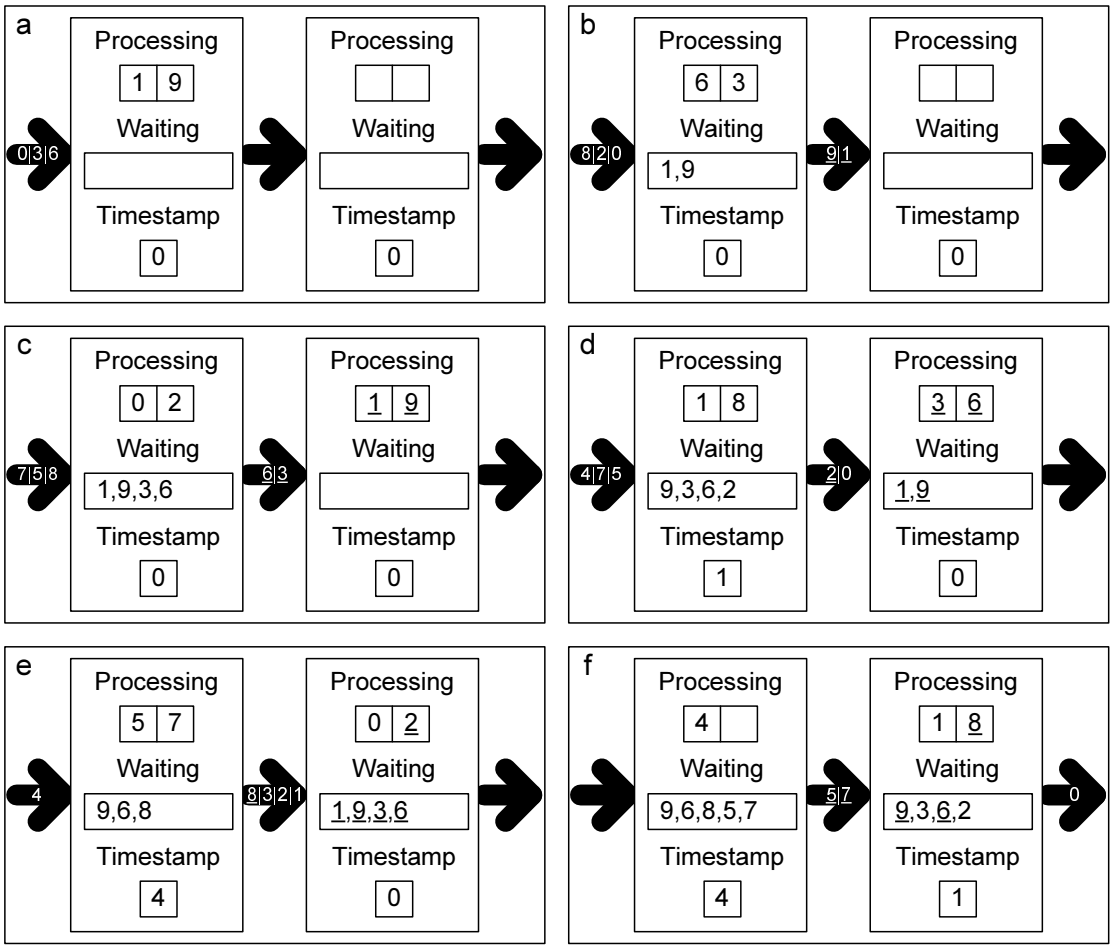


FIGURE 6.1: Processing with distributed transactions.

downstream components will become stateful, even though they only perform stateless operations, because their states depend on upstream components. Certainly, the chance for an abort is increasing with every additional component, but still for two or three components in a row the approach should be acceptable.

Example 6.1. *Let's assume we have a setup of two STM-enhanced stateful components, each with two parallel threads as shown in Figure 6.1. The first component is sending events while the second one is receiving them. For this example we neglect the use of a predictor to simplify matters. The events are represented by the number of their timestamp. If the number is underlined the event is a clone of a not yet committed event. Each frame in the figure represents one of six time steps marked with a to f.*

As one can see in Figure 6.1`a` *the first two events are processed and the first event which can commit needs the timestamp 0. In step* `b`*, the processing of the events 1 and 9 is finished, but none of them can commit. Both events remain in the waiting queue of the first component, while their clones are send to the second component. These clones can*

be processed on the downstream component as indicated by step `c`*. Finally, in step* `d` *the event 0 was successfully committed and raised the internal timestamp to 1. We assume that event 1 conflicted with event 0 and thus has to be reprocessed. Nonetheless, on the second component the processing of the clones of events 1 and 9 was finished and they are waiting for a commit notification of the upstream component. With step* `e` *the events 1, 2, and 3 could commit in a burst on the first component and are sent to the next one. The next event to be committed on the first component is now event 4. Since the event 1 had to be reprocessed on the first component, its already waiting clone on the second component has to be deleted in step* `f`*, since it was a clone of the first unsuccessful try. However the events 2 and 3 did not conflict and thus the processing of their clones is still valid and they will replace the original events.*

6.1.1 Event Tracking

With the distributed transaction mechanism, we need to clone events that are processed but not yet committed in order to send them to the following component. Since both components are only connected via the one-way event stream, we have to attach status information to the event, *e. g.,* if it was already aborted or if it was committed. Of course it would be possible to connect all components involved in distributed transactions with additional message streams, but this would increase the complexity and decrease the usability of the system. Therefore, we simply clone and send the events downstream every time they finish their processing. Whenever an abort and retry is executed, a new clone is created and sent downstream. The receiving component needs to track every received event and has to check if it already received an earlier version. If the component is processing an older version of the event, it aborts and retries with the newer version. Since the components are connected by streams, which provide a first-in-first-out semantic, we don't need to add a version counter to the events.

We also need to attach two flags to the events. First, the *COMMIT* flag, indicating the successful commit of this event at the previous component. Second, the *ABORT* flag, reporting that the processing was rolled back. The possible combinations of the two flags and their meanings are shown in Table 6.1.

Furthermore there are only certain transitions possible between the states of the two flags. These are shown in Figure 6.2. As can be seen, there is only one cycle, at the state *process and wait*, which does not prevent termination of the protocol. Indeed, the next event to be committed (with timestamp equal to the internal clock) has the lowest timestamp among those not yet committed. This event will be eventually received

		COMMIT	
		false	true
ABORT	false	event can be processed, but cannot commit	event can be processed if not done yet and can commit
	true	event has to be reprocessed and cannot commit	event has to be reprocessed and can commit

TABLE 6.1: Flag states of an event clone

and the associated transaction will commit, because the contention manager privileges transactions with lower timestamps.

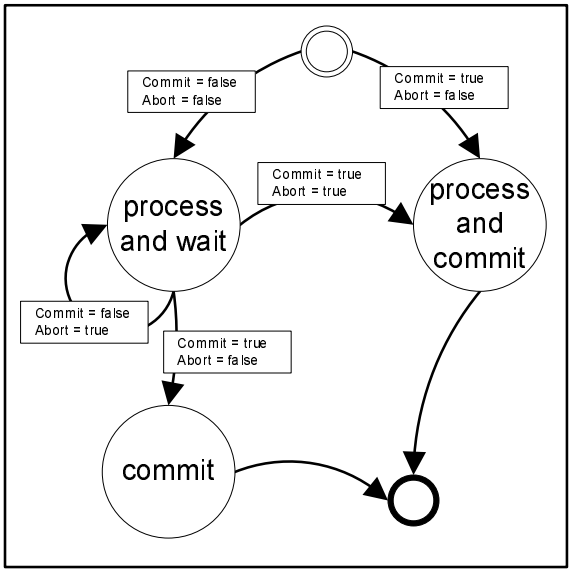


FIGURE 6.2: State diagram of the flag states

6.1.2 Modification of the Components

With the introduction of distributed transactions, all processing components, like filters or aggregators, using this feature are becoming stateful, even though normally stateless. This transformation is needed since the components are not just relying on the state of the event itself, but they also depend on the state of the previous components. Nonetheless, stream managing components like multiplexer, de-multiplexer, or network connectors are still stateless, since they do not modify the event. Their only purpose is to route the events through the streams.

The existing stateful components, as described in Chapter 5, have to be extended with two flags, set on initialization. These flags are enabling the distributed transaction

feature. They indicate if this component is able to receive and/or send events under the protection of a distributed transaction.

When the receiving flag is enabled, the component has to store each received event as long as it is not committed, as shown in Algorithm 26. When it finally commits, the event can be deleted from the local storage.

Algorithm 26 TM-Stream beginTransaction()-Modification()

```
1: event ← getEvent(){Receive a new event}
2: if Thread.receivesDistributedTX then
3:     if !event.isWasSent() then
4:         {Event arrives for the first time}
5:         transaction ← createNewTransaction(event)
6:         putToWatchlist(transaction)
7:     else
8:         if event.isCommitted() and !event.isAborted() then
9:             {Prior sent clone can commit now}
10:            otherTransaction ← getFromWatchlist(event.getTimestamp())
11:            otherTransaction.getEvent().setCommited()
12:            return
13:        else
14:            {Prior sent clone has to be deleted}
15:            otherTransaction ← removeFromWatchlist(event.getTimestamp())
16:            otherTransaction.dismiss()
17:            transaction ← createNewTransaction(event)
18:            putToWatchlist(transaction)
19:        end if
20:    end if
21: end if
```

If the component is sending events with distributed transactions downstream, it has to clone and send the event each time it is processed and moved to the waiting queue. In Algorithm 27 and 28 some of the modifications to enable the down streaming of cloned events are shown.

6.2 Evaluation

In order to evaluate our distributed speculative ESP system, we combined different kinds of distributed stateful components. We analyzed the performance of two chained stateful components, as shown in Figure 6.3. Compared to the setup of the experiments in Chapter 5, the *Correlator* has been replaced by a second, stateful component and TCP connectors have been added between both components. Each distributed component is running on a separate virtual machine on the same server (*i. e.,* TCP communication

Algorithm 27 TM-Stream commit()-Modification()

```
1: ...
2: if clock.getActualTime()! = transaction.getTimestamp() then
3:     {Transaction can only commit when the timestamp matches the actual time}
4:     delayTransaction()
5:     return FALSE
6: end if
7: ...
8: if Thread.receivesDistributedTX then
9:     {Only when receiving distributed transactions}
10:    if !transaction.getEvent().isCommitted() then
11:        {Event was not committed on the prior component}
12:        delayTransaction() {Wait for the prior component to commit}
13:        return FALSE
14:    end if
15: end if
16: ...
17: if Thread.sendsDistributedTX then
18:    {Only when sending distributed transactions}
19:    transaction.getEvent().setCommitted() {Set commit-flag}
20: end if
```

Algorithm 28 TM-Stream delayTransaction()()

```
1: if Thread.sendsDistributedTX then
2:     {Only when sending distributed transactions}
3:     transaction.sendDelayedEvent() {Send a clone}
4: end if
5: putToWaitingList(transaction) {Stores the delayed transaction}
```

remains local). Both stateful components always have the same processing time for this experiments.

The experiments were executed on an 16-core machine with 4 AMD QuadCore processors at 2.20 GHz running Linux 2.6.25 (64-bit) and `Java` 1.6_10. The runtime of all experiments was 20 seconds with a warm up period of 1 second. Each point in the following graphs represents the average of three measurements.

6.2.1 Comparison of Various Distributed Configurations

In our first test, we used five different setups. The first one uses two sequential components and represents the baseline. For the second setup, two TM-Stream components are used with distributed transaction support enabled (distTX). The third setup is identical to the previous one, but with distributed transaction support disabled. The last two setups are using a mixture of TM-Stream and a sequential component.

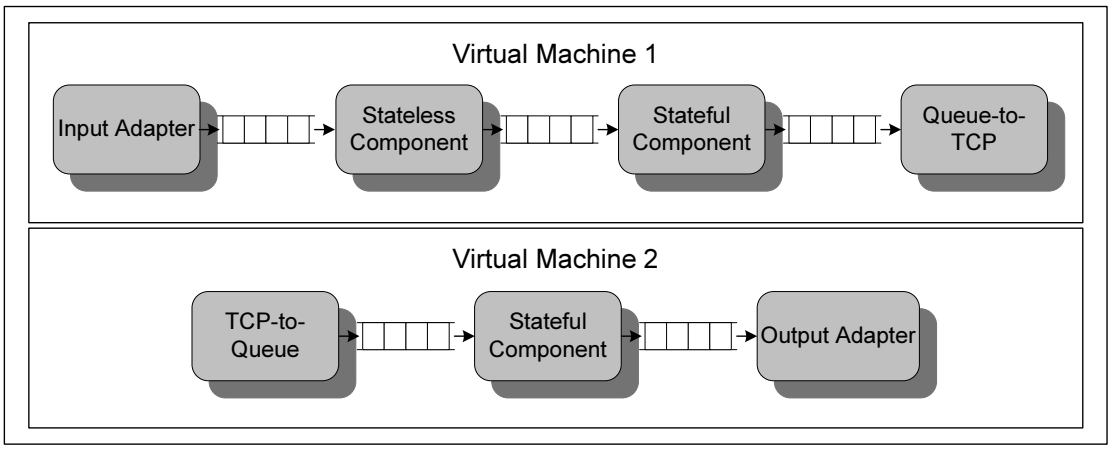
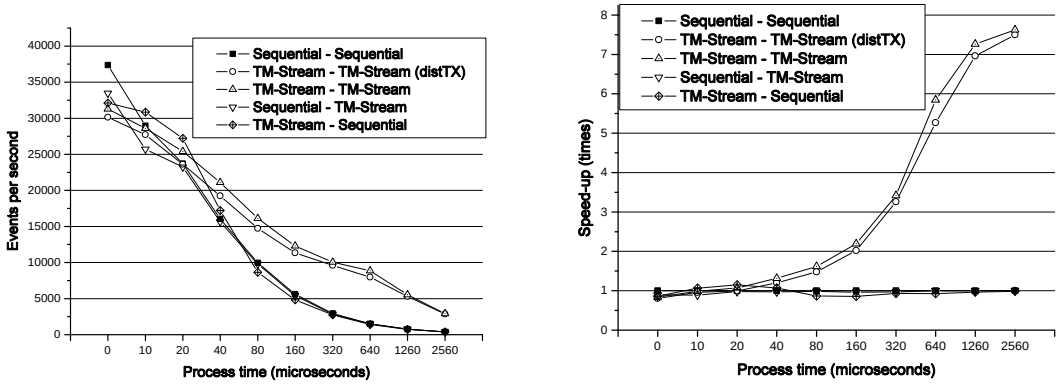


FIGURE 6.3: Setup for the distributed transaction measurements.



(a) Throughput.

(b) Speed-up.

FIGURE 6.4: Comparison of various distributed configurations.

The results from this evaluation are shown in Figures 6.4(a) and 6.4(b). As one can clearly see, the combination of a sequential component followed by TM-STREAM does not produce any gain, compared to the baseline, since the sequential component slows down the whole system. On the other hand, in the setup where the TM-STREAM component is upstream, we measure a small speed-up until processing times reaches 40 μsec. It seems that the TM-STREAM is doing well as a sorting component for certain parameters. Both configurations with only TM-STREAM components show a behavior consistent with the results presented in Chapter 5. One can observe a very small overhead when enabling distributed transactions.

Figure 6.5 shows the decrease in idle time on the downstream TM-STREAM component when the distributed transaction feature is enabled. For both setups we measured the average idle time, *i. e.,* each time a thread on the second TM-STREAM component had

to wait due to an empty input queue, wait-time was accounted. As one can see, the distributed transaction can reduce the idle times on downstream components. An explanation for the low values between 160 and 640 μs could be hard disk accesses needed to store the large amount of data during the experiment. Other possible error sources in the measurement could be the randomly chosen input or scheduling problems of the underlying operating system, *i. e.,* matching the threads to the optimal cores.

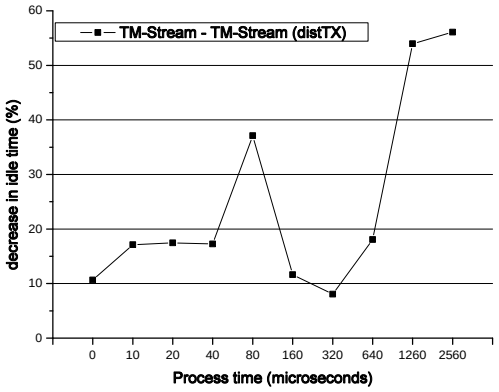


FIGURE 6.5: Improvement of the distributed Transaction in means of idle time.

6.2.2 Comparison of Distributed Transaction Chains

For the second experiment we extended the setup of the first experiment (see Figure 6.3) with three additional stateful components, each located in another virtual machine, which are inserted between the TCP connections. Figure 6.3 depicts one of these components. Each stateful component using the STM-based approach is using 2 threads to process the incoming event.

In this setup we are comparing the 5 sequential stateful components with 5 STM-based stateful components in various configurations. The configurations of the STM-based cases differ in the use of distributed transactions as shown in Figure 6.7. The black boxes around the components depict the range of the distributed transactions, *e. g.,* in

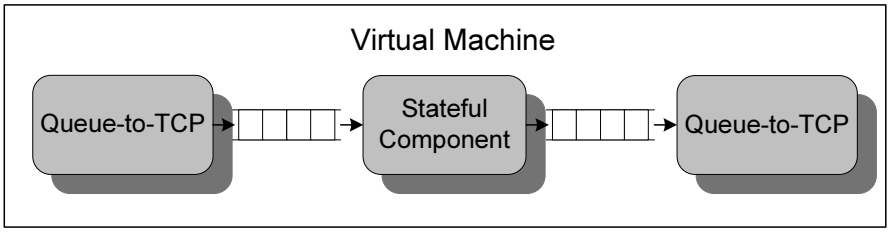


FIGURE 6.6: Additional stateful component.

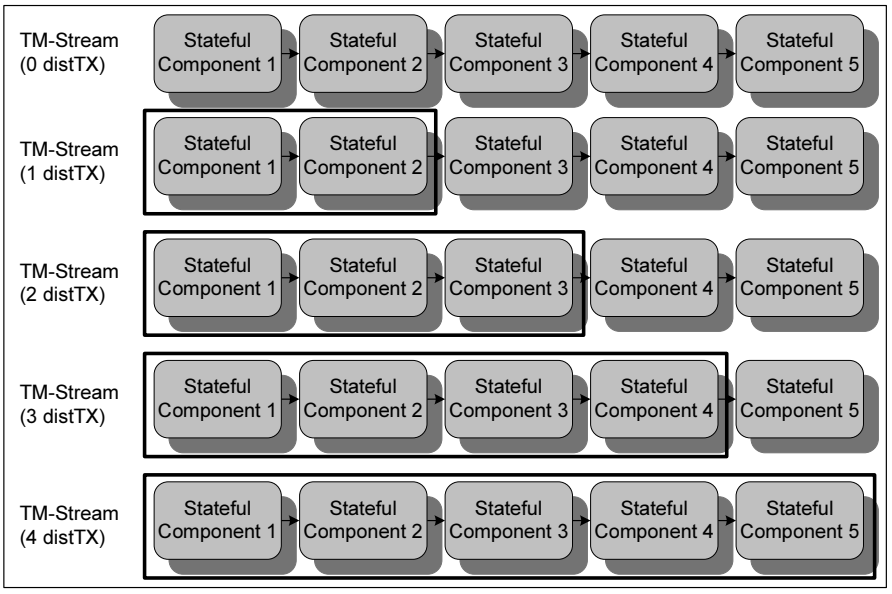
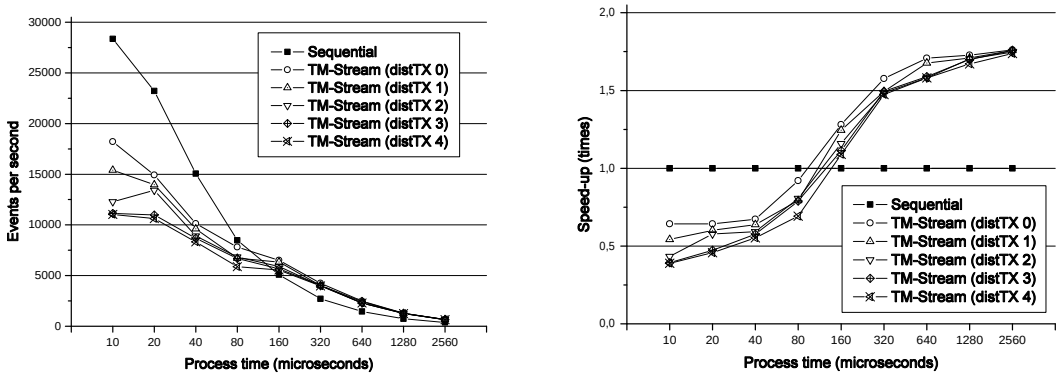


FIGURE 6.7: Different TM-STREAM setups.



(a) Throughput.

(b) Speed-up.

FIGURE 6.8: Comparison of 5 enqueued stateful components in various distributed configurations.

the first configuration no distributed transaction used is, in the second one the first two components are using it, etc.

The results of this experiment are shown in Figure 6.8. With increasing number of components using the distributed transaction, the throughput and thus the speed-up decreases slightly. As already pointed out in the prior experiments, the overhead influence of the STM is reduced with longer processing times, resulting in a speed-up of about 1.8 times.

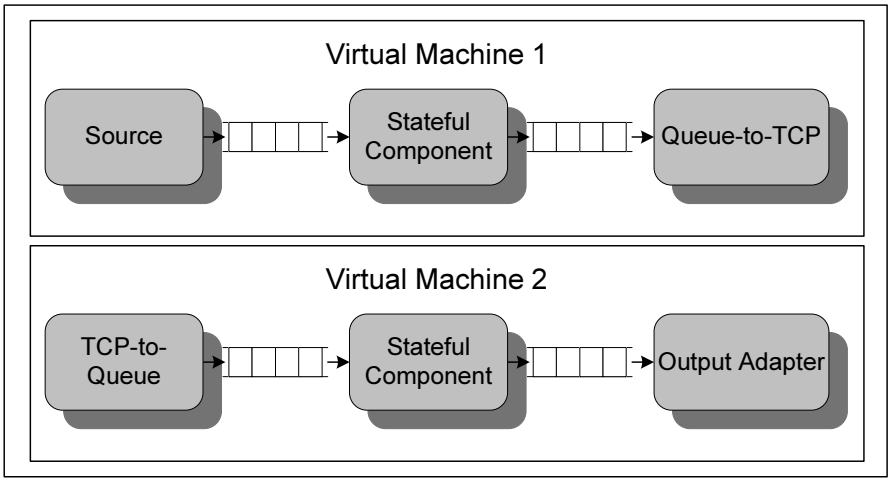


FIGURE 6.9: Setup for the measurements with events arriving in inverse order.

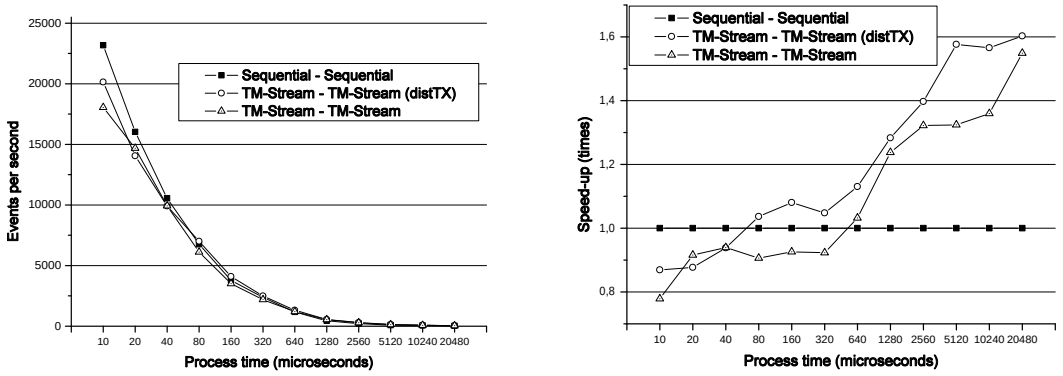
6.2.3 Measurements with Inverse Input Order

For the following experiment we have chosen a deterministic input order. The events are arriving in blocks of 20 events with an inverse order, *e. g.,* event with timestamp 19 arrives 1st and event with timestamp 0 as 20th. To keep this order until the stateful component, we removed the stateless component, which was mixing up the events in the prior experiments. The final setup is shown in Figure 6.9. Each STM-enhanced stateful component uses only one thread for processing the events.

The results of our single thread comparison are shown in Figure 6.10. In the throughput and the speed-up diagram one can clearly see that the setup with the distributed transaction performs better than the TM-STREAM setup without it. The reason for this behavior is based on the specific order of the events, which causes a high rate of processing not yet committed events on the second stateful component. Both TM-STREAM setups are improving their performance compared to the sequential version with longer processing time. This effect is also cause by the speculative behavior of the underlying STMs.

6.3 Summary

In this Chapter we presented an extension for our DSTM2-based event processor, described in Chapter 5. The system supports distributed speculations by allowing events of not yet committed transactions to be processed speculatively by the following components. Our experimental results have shown that our approach can help to reduce



(a) Throughput.

(b) Speed-up.

FIGURE 6.10: Events arriving in inversive order.

the idle time of downstream components. The overhead of the distributed transaction is quite small compared to our pure TM-STREAM setup without this feature.

Furthermore, we have shown that even for events arriving in a certain order, a single threaded STM-based stateful component can improve the processing speed, especially with the application of distributed transactions. Of course this gain could also be possible with conventional components, but only with a much higher complexity.

Chapter 7

Use Cases for STM-enabled Event Stream Processing

In Chapters 5 and 6 we discussed the merging of the STM technology with stateful event streaming processing. To encourage its application, we present in this chapter several use cases which could be improved using this approach. The described scenarios are all linked to sensor networks, widely used as event sources for ESP. Of course, the application of our improved stateful components is not limited to these use cases. They just show some possible applications. A basic guideline where our approaches could be used is described in the next section, followed by four use cases: (i) a local positioning system, (ii) a sensor network with unreliable sensors for environmental monitoring, (iii) a ride-sharing system, and (iv) an adaptive traffic control system.

7.1 Requirements

For the successful application of our STM enhanced stateful components it is fundamental to meet certain basic properties. First of all, the incoming events shall not all access the same state during their processing. There must be a certain number of different states to access, *e. g.,* one state for each sensor. Otherwise, the whole processing would be serialized by the STM. Nonetheless, serialization can be accepted for minor states like the timestamp counter, but their access should be limited to the commit process. If there are certain event groups accessing the same states, one has to guarantee that the events are not direct successors, because it would again result in a sequential processing.

As can be realized from the evaluations in the prior chapters, it is irrelevant if the events enter the component ordered or unordered, but all events have to leave the component in

a predefined order. Furthermore we have shown that under certain circumstances (order of the incoming events), our enhanced component, running only with a single thread, can even outperform the sequential approach. We would recommend a multi-core system for the ESP, but sometimes a classical single-core system might also be suitable for the processing.

The best improvement using our component can be reached in a real time or a time constrained system where it would take too long to process the events with the support of a database or to store them for later processing.

7.1.1 Special Requirements for Sensor Networks

Sensor networks are usually constrained, compared to desktops or servers, in the amount of available energy and/or processing power. Therefore special requirements are needed for them. For instance, one has to decide to either process the information at least partly on the sensor nodes and reduce the energy-consuming (wireless) communication or send the complete information to a stronger processing and/or storage unit.

If an STM shall be used already on the sensor nodes, one has to ensure that there is an STM available for the programming language for the node. Furthermore, one has to decide if the distributed transaction feature is enabled, since it would lead to a increased power consumption due to the increased amount of transmitted messages.

7.2 Local Positioning System

The first use case for the application of our STM-enhanced ESP is dealing with the localization of badges in a sensor node array. The resolution of the position is obtained by the radio range of the sensors. For a usable result at least 3 sensor nodes need to detect the badge, if it moves only in a two-dimensional space. If the badge shall also be detected in 3 dimensions, the minimum number of nodes needed is 4. The badge itself could be for instance a radio-frequency identification (RFID) tag. This use case can be applied to any kind of logistics where goods are moved and their location has to be detected. Another possibility is the localization of animals in a certain area, where each beast has an attached badge.

Each sensor node used, is a source for events, which are released within a certain time range. Each event contains the information of its origin (a sensor node identifier or its location), the time of creation, the badges detected by the sensor, and their detection

time. It could also be useful to detect the distance to the badges either via the signal-response-time or the signal strength.

The events can be transmitted from the sensor nodes to the multi-core processing unit (PU) either via wired or wireless communication. Due to the different distances between sensors and the PU, the events can arrive in a different order than they were sent. The sensor nodes could also send their events in a tree-like network to reduce the transfer load by aggregating event messages. It is also possible to introduce a time-shift between the transmissions from the sensor nodes to minimize the conflict rate. This is especially useful in wireless environments.

The PU determines the position of the badges within the sensor network and therefore needs to synchronize (combine) the sensor information with the map and badges. Since a badge can be detected by multiple sensors, the processing is stateful and thus we would encourage the use of our STM-enhanced ESP. The conflict rate would be low since normally up to 4 sensors would update the location of a badge. With the knowledge of the current and former positions one could also determine the current movement of the badge as well as the speculative future movement.

Additionally, our system would be able to release early results of the position of the badges. As soon as late events arrive and modify the current position of the badge, it could be updated. This could be an advantage for instance for a mobile tracing device, which presents the locations of the badges to a user in real time. Normally, this device does not need the 100% correct position, since it can be interpolated with the last position and its movement. Another sink for the processed information of the PU could be a database. For this persistent storage device, the processing time is not important, but the correctness of the received information is. Thus the database is not interested in intermediate results and would ignore them. A basic setup of such a system is shown in Figure 7.1.

Example 7.1. *Consider a badge moving through a sensor array as shown in Figure 7.1 and the PU using our STM-enhanced ESP system to process the location of the badge.*

In time step one (small gray-shaded circle with a "1" inside), the badge would be in the range of sensor S1. When the PU receives the message of S1 it would update the location of the badge. Since S1 is the only sensor updating the position at this time step, this operation is conflict free. Nonetheless, the accuracy of the badge location depends in this case highly on the equipment of the sensor, e. g., sector antennas to determine the direction, signal-response-time or signal strength to interpolate the distance. Without this advanced equipment the PU can only ensure that the badge is close to S1.

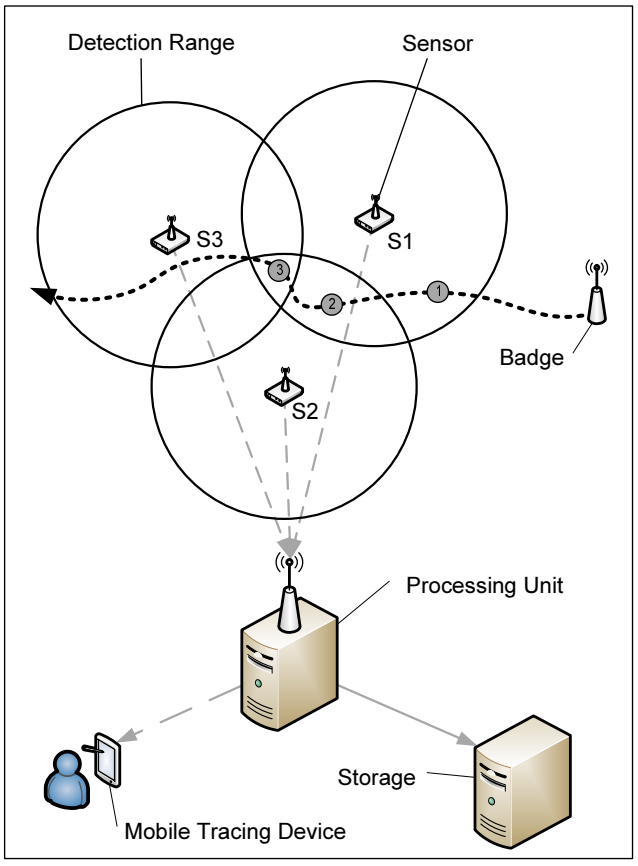


FIGURE 7.1: Sensor-based Localization Infrastructure

The badge is moving in the second time step to a position where it can be detected by the sensors S1 and S2. Due to the parallel architecture of the PU, it is possible that both messages are processed at the same time. This could result in a conflict when both want to update the position of the badge at the same time. With the support of the STM this conflict would be detected and solved automatically. With two detecting sensors it is still complicated to receive a high accuracy for the location, since the badge could be anywhere in the intersection of both sensor ranges.

When the badge arrives at the position in time step three, where it is detected by all three sensors, the conflict probability during the processing is raised again. However, at this location the badge can be detected with a high precision and thus it would be worth pursuing, that it is always in the range of at least 3 sensors. In a conventional setup for the PU the programmer would have to deal with this shared access manually, but with the application of the STM this task is simplified.

In a larger environment one would need to develop a more elaborated version, possibly with different levels of clustering. Such a clustering is exemplarily shown in Figure 7.2. The example contains one processing unit (PU) which is connected to two collectors.

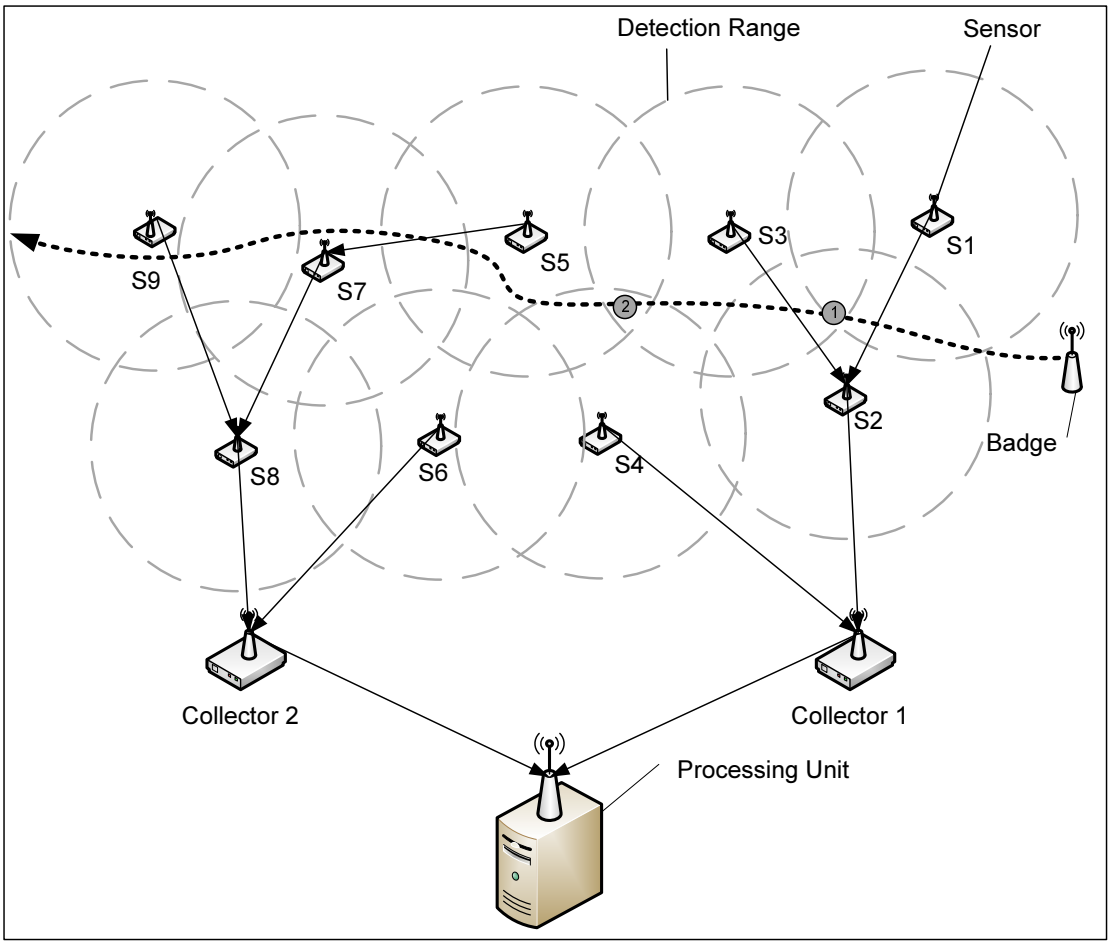


FIGURE 7.2: Clustered Localization Infrastructure

They are aggregating and preprocessing the location of the badges based on the information received from their sensors. Collector 1 is connected to the sensors S1 to S4 and Collector 2 to S5 to S9. The collectors themselves could be as well sensor nodes or special purpose nodes with fewer limitations, *e. g.,* more energy or higher processing power.

If the collectors receive enough information to localize the badge they could already finish processing and would only relay the coordinates. Therefore it could be useful to apply our STM-based approach which would automatically start the processing and abort it, if too few sensor data has been received within a given time period. Parallel to its own processing the collector could either delay until a processing aborts or directly relay the sensor information to the PU. The decision about which approach would be useful should depend on the earlier processing results.

Example 7.2. *For instance in time step one (small gray-shaded circle with a "1" inside) Collector 1 would receive 3 datasets from the sensors S1 to S3 which is sufficient for the localization. Whereas, at time step 2 the 3rd sensor data from sensor S5 is missing for the processing at the collector level. As a result Collector 1 would need a relay to the PU.*

Our STM-based ESP component could also be used for error detection in the system. For instance if in time step 1 the badge would be falsely detected as well by sensor S9, the PU could abort this processing, due to the successful processing on Collector 1 and its topological knowledge of the sensor clusters.

7.3 Environmental Monitoring

Monitoring changes in the environment is one of the most important applications for wireless sensor networks (WSN). Hart *et al.* [HM06] described several typical examples, *e. g.,* monitoring of trees, volcanoes, and glaciers. The sensor nodes are normally distributed in a certain metric over the area of interest and are equipped with several sensors, *e. g.,* temperature, pressure, or humidity. On a regular base these sensors are sending their measurements to a processing unit or storage. Depending on the goal of the application, there are quite different approaches. For instance the PermaSense project [Per, THGT07] is mainly interested in long term observations of high alpine environment. On the other hand the Hydromon project [NSBC09] deals in real time with water pollutions.

This use case proposes a real time monitoring of an active volcanic area. Each sensor node could be equipped with temperature and gas detectors. The sensor information has to be processed in real time at the base station. The result shall be presented on a digital map of the region. This map should visualize areas of interest, *e. g.,* locations with unusual volcanic activity. With the support of our STM-enhanced event processing it would be possible to interpolate sensor information if the sensor node was disabled by the local conditions or if the information arrives late. The interpolation should consist of the last data of the sensor node and the current information of the nodes nearby.

Example 7.3. *Let's consider a network of wireless sensors located at the flanks of a volcano. The sensors create a self-governed wireless network and send their measurements (temperature and pressure) to the processing unit at the base of the mountain as shown in Figure 7.3. Let's assume that all sensors are working and established transfer routes for the messages (indicated by the arrows with a 1 in the figure). Maybe the sensor nodes are aggregating incoming messages of other sensors to reduce the number of outgoing messages, e. g., S4 receives the messages of S1 and S2 and sends their measurements together with its own as one message to S7.*

In the case of a volcano eruption it is possible that some sensors are disabled by lava or slag. Nonetheless it is important that the messages of the remaining sensors are still reaching the base station with its processing unit. In such a case the network would modify its routes. For instance, if sensor S7 is disabled the network routes could change

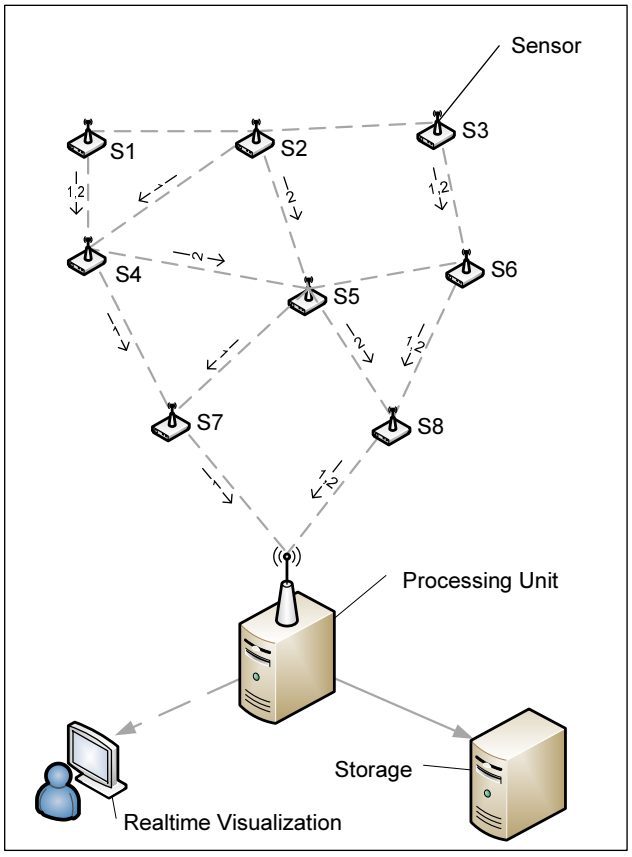


FIGURE 7.3: WSN Infrastructure for environmental monitoring

as shown in the figure by the arrows with a 2. During this self-healing of the system some messages can arrive late at the PU. Nonetheless they should still be processed for later non-real-time analysis. For the real-time visualization the PU could use speculative data consisting of last measured values and the values of sensors near by. As soon as the PU receives the missing information it could re-process it.

7.4 Ride-sharing System

Sharing cars between neighbors or colleagues to reduce the private traffic and/or needed cars in cities is a well known concept of environmental protection in large cities. The first known service was the Swiss *Selbstfahrergenossenschaft* (SEFAGE) in Zürich in 1948. Nowadays, mainly the car-sharing approach [Moba, DB , Gre] is used, *i. e.,* several cars are distributed over the city and can be used by every participant. Another approach is ride-sharing, where several persons share a single car during travel. Such a service was tested by the Carlos GmbH [Car] between 2002 and 2004.

With the recent simplification of technology, nearly every cell phone has an internal GPS detector and can thus determine its position. This technology could be easily used in a

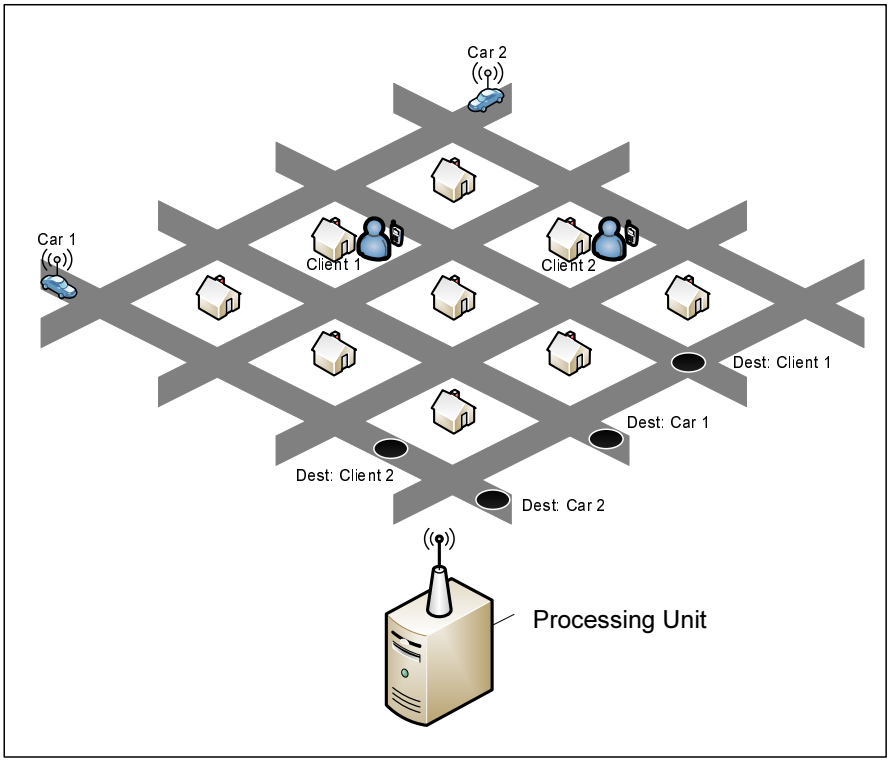


FIGURE 7.4: Infrastructure for ride-sharing

ride-sharing system, as Gidofalvi *et al.* presented in [GP08, GPRZ08]. In their approach, they propose a cab-sharing system, where the route of the cab can be altered on the fly to collect other passengers with nearly the same destination. Such a ride-sharing system could also be improved with the application of an STM-based event processor. The routes of the cabs are speculative and with the support of an STM, they could be altered for best matching. Also it could reduce the usage of databases, since the durability of the system is only needed for accounting, but not necessary for the routing.

Example 7.4. *Consider a scenario as shown in Figure 7.4. Cars 1 and 2 are participating in the ride-sharing system and clients 1 and 2 want to get a ride from their current location to their destination. They use their cell or smart phones to connect to the processing unit, indicating their location and point of destination to the system. The PU is in constant contact with all available cars in the city, to request their position and their destination. We assume that the data accessed concurrently is the number of empty seats in a car and its route. Each transaction would be a request from a client, who wants to acquire an empty place in a car and tries to redirect its route. Let's consider that the transactions of both clients allocated them a place in car 2. Even though there is still enough space in the car, both want to modify the route, resulting in a discrepancy of the routes and thus a conflict. The STM conflict manager could solve this problem in either aborting one of the clients transactions and supporting the aborted one with further information for the re-try, e. g., proposing client 1 to acquire car 1. Or to join both route to a*

single one. As soon as a transaction was successful the PU sends the route modifications to the regarding car and an acknowledgment to the client.

7.5 Adaptive Traffic Control System

Controlling and altering the traffic flow in cities is worldwide a challenging task. Nowadays, intelligent traffic lights and signs are used to minimize the traffic congestion and avoid traffic jams. For example, emergency vehicles [U.S06] are able to interact with traffic lights to guarantee a fast advance in case of emergencies by stopping the interfering traffic. Another approach is to forecast the traffic based on daily statistics and modify the duration of the traffic lights according to the main traffic or in longer terms modifying the roadways [FHB06, PL08].

With the support of modern GPS navigation systems and event streaming systems in combination with centrally controlled traffic lights and signs, we think a real time adaptive traffic system could improve the traffic flow in the major cities.

Figure 7.5 shows an extract of a street map including traffic lights. They are globally managed and controlled by a central processing unit (PU). All cars can interact globally with the traffic control system for route optimization and locally with the traffic lights ahead of the car. They are equipped with a GPS navigation system and a possibility to create a bidirectional connection with the processing unit (*e. g.,* WLAN or GSM) and with upcoming traffic lights (*e. g.,* WLAN or other short range radio communication).

The processing unit has the ability to re-route the cars to an optimal route to avoid for instance traffic jams. For the re-routing the PU takes additionally into account the location of all cars and the possible waiting time at the traffic lights.

Each car can also interact with the traffic lights ahead of its way. In the traffic light system each approaching car has its own control thread and the modification of the signal of the traffic light is handled in STM transactions. If two transactions want to modify the signal of a traffic light in a conflicting way one of them has to abort and maybe ask the car for a change of its route. It is also possible to use depending transactions to allocate already the traffic light after next. The contention manager used to solve conflicts could abort randomly one of the conflicting threads or the one with the lowest abort rate. Also it could be possible to create car clouds consisting of cars traveling in the same direction. The cars in the cloud could interact with each other via WLAN and could have a higher priority for signal modifications at the traffic lights.

Example 7.5. *Let's assume the scenario shown in Figuree 7.5. Two cars want to reach their destination. Both can allocate a free passage for two traffic lights in advance.*

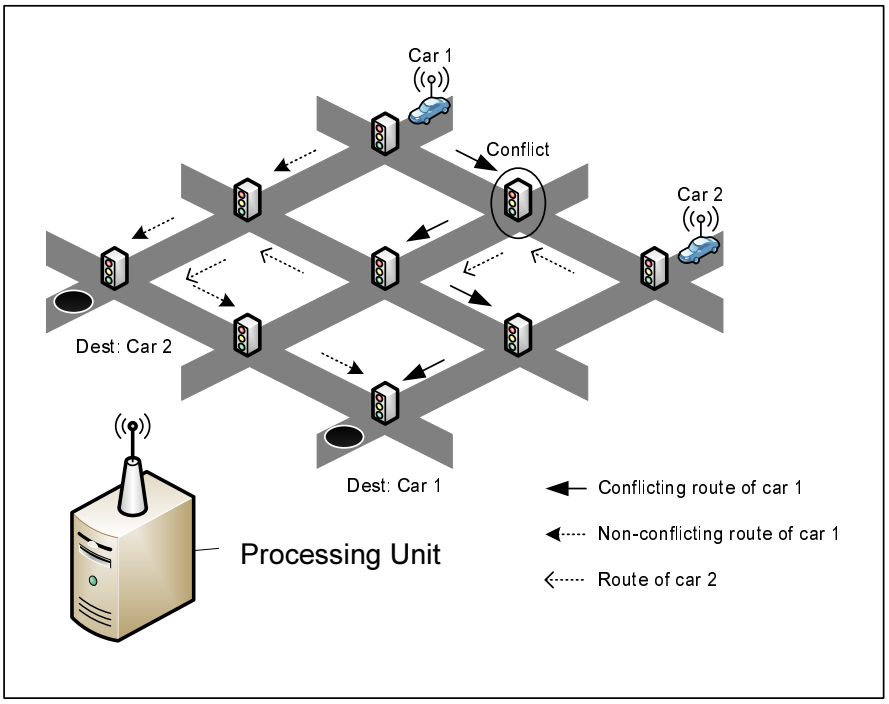
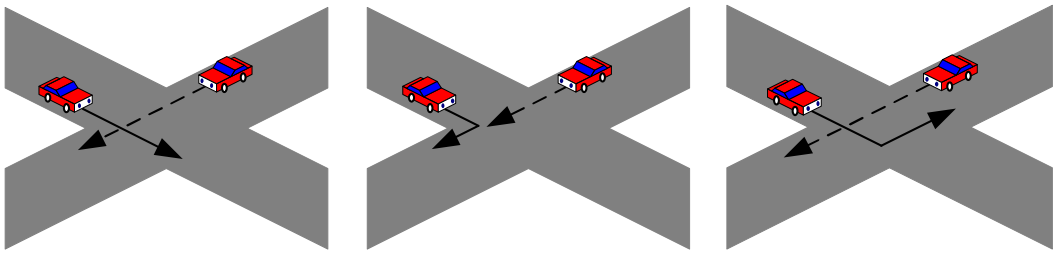


FIGURE 7.5: Infrastructure for adaptive traffic control

Unfortunately, some of their routes create conflicts, because both cars want to have the permission to drive at the same traffic light simultaneously. For instance if car 1 is taking the conflicting route it creates a conflict at the second traffic light marked in the figure. This conflict can be solved in a global or a local way. In the global conflict solution car 1 would be re-routed by the PU to the non-conflicting route and thus neither car 1 nor car 2 would have to wait in this scenario. The local conflict solution would be handled by the contention manager in the traffic light deciding which of the cars has to stop and which can go on.

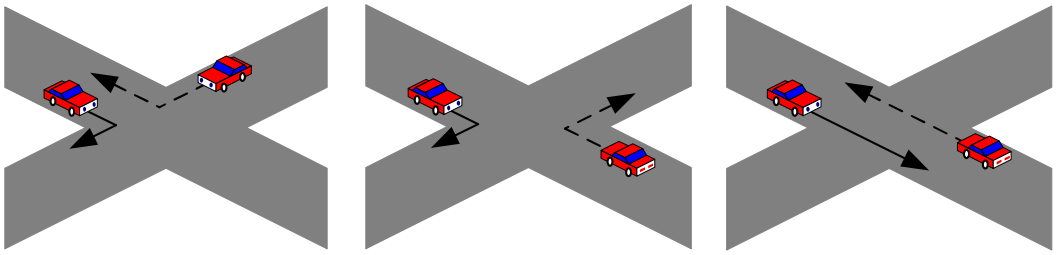
Crossroads with traffic lights have usually different lanes for the different directions a car can turn. Normally, there is a lane for turning left, one for turning right, and one for going straight. Sometimes the lanes for turning right and going straight ahead are combined, but for simplicity we use the 3-way approach. Figure 7.6 depicts different cases where both cars would raise a conflict in the traffic light system. Therefore, one of the cars has to stop and wait until the crossroad is clear again. Whereas in Figure 7.7 both cars can pass the crossroad without interference, even though they both change the traffic lights at the same time. As one can see, it is a complex procedure to enable all non-conflicting routes at an optimal crossroad.

It should be noted that we neglected all issues of security for this use case. For instance, one would need to prevent any misuse of the signal modification by using appropriate protection measures.



(a) (b) (c)

FIGURE 7.6: Conflict situations at a crossroad.



(a) (b) (c)

FIGURE 7.7: Conflict-free situations at a crossroad.

7.6 Implementation Approaches

Since our use cases are not yet implemented we will present in this section some approaches on how to build them. We would recommend simulating the use cases before the application in real environments. For WSN there exist several simulators which could be used for first prototypes. Some of them are already included in the operating system package for some specific sensor nodes, *e. g.,* COOJA [sDE+06] which is part of the Contiki [DGV04, Con] operating system or TOSSIM [LLWC03] as part of the TinyOS [Mat08, tin]. However, there are also platform independent simulators available, e.g., J-Sim [Tya02, J-S], Freemote Simulator [MKKW08, Fre], Castalia [Bou07, Cas], or Mobility Framework for OMNeT++ [DSH+03, Mobb]. The choice of the simulator used mainly depends on which type of real sensor node the application will be deployed.

For the last two use cases which are using transport-related scenarios we propose the use of traffic simulators [Lin, VIS, Upp]. They are able to generate different types of traffic in a cartographic environment. These data can then be used as input for the processing described in the use cases.

7.7 Summary

In this chapter, we have shown different use cases for our STM-enhanced stateful event processing component. Furthermore we developed a guideline for use cases which could benefit from our system. Of course, it is non-trivial to decide if a system really gains from our approach, since there are many conditions influencing the gain. The major one is the conflict rate of the shared accesses. Only in a system where conflicts are rare (below 10%, see Chapter 5 and 6), the STM-enhancing is useful since it can reduce the programming effort considerably and increase the processingrate of events in such cases.

Chapter 8

Conclusion

With the development of multi-core processors and their availability to end-users, new algorithms for concurrency control were developed. Software transactional memory is one of them, as described before. It tries to solve conflicts in an optimistic way, which enables a high rate of parallelism with the penalty of the possible re-execution of conflicting code.

In the final chapter of this thesis, we summarize our contributions for the development and application of software transactional memory in event-based systems, evaluate if the previously described objectives have been met, and give an outlook on possible future work.

8.1 Contributions

In Chapter 3 we presented the complex characteristics of STMs by the means of design patterns. We compared the design of different STM implementations. From their similarities we derived the concurrent design pattern called "Transactional Object". With this new design pattern, we want to improve the development of new STM algorithms. Furthermore, our design pattern allows programmers an easier application and integration of STMs to their software solutions, since the pattern summarizes its characteristic properties. Our approach was published in November 2009 and therefore we cannot prove the impact of our development.

We also described STM programming support for the unmanaged programming language `C` in Section 4. With the introduction of TANGER, our LLVM-based compiling framework, we were able to transactify `C` source code semi-automatically with the usage of special marker functions. The resulting binaries are nearly as efficient as manually modified ones. The advantage of our solution is that the programmer needs only a minimal knowledge

of parallel programming and has to add only a few marker functions instead of several STM functions including each access to shared variables. TANGER was the first universal STM interface for this programming language.

In Chapter 5 we applied the STM technology to event stream processing. The STM ability of speculative processing within a transaction leads to our newly created STM-enhanced stateful ESP component. We implemented our approach in two programming languages (`C` and `Java`) and showed the good scalability of the component. The advantage of our system is a simplified possibility to program a parallel stateful component by hiding the parallelism. The programmer delivers the processing task of the component in a method which is executed in a transaction. All shared variables he uses are protected by the underlying STM and thus he can program as for a sequential programs.

Furthermore, the issue of temporal ordering of events is solved automatically by our approach. Therefore, each events needs to be tagged globally with a unique timestamp. In our approach events can be pre-processed out-of-order, but may have to delay their completion (*i. e.,* their commit) when ordering is required. All dependencies between events are detected dynamically by the STM and will result in sequential processing if necessary (possibly delaying, or aborting and restarting some transactions).

The extension of the STM-enhanced stateful ESP component to support distributed transactions was presented in Chapter 6. Our DSTM2-based implementation can handle distributed speculations by allowing events of not yet committed transactions to be processed speculatively by the following components. Our evaluation of this component shows that with this improvement we can reduce the idle times of the system with only a minimal overhead. Furthermore, we could prove that for certain input orders, our system is able to outperform even the sequential processing when using just a single processing thread.

Finally, in Chapter 7 we presented several use cases in which our ESP components could be used. Therefore, we defined first some major requirements for possible applications. Our use cases include examples from local positioning, environmental monitoring, ride-sharing, to traffic control systems. We have shown with these examples the wide range of possible applications.

The topic of this thesis mainly deals with software transactional memory, beyond that the areas of design patterns, programming language support, event stream processing, and sensor networks have also been touched in our research. We have developed new approaches to apply and use the STM technology in unusual ways. For instance the TANGER is transactifying and compiling `C` code based upon the LLVM framework or TM-STREAM is using an STM for parallelizing a stateful ESP component.

Except for the content of Chapter 7, our research has been published in major international peer-reviewed conferences. Overall the feedback we received for our approaches was positive and led to interesting discussions with people from the TM community and other area of expertise. Appendix A contains a list of all published papers. This list contains papers related to the work of this thesis as well as publications in other fields.

8.2 Evaluation of the Objectives

With the work presented in this thesis, we partially fulfilled the objectives set at the beginning. We created a support for the programming language `C`, which is easy to use, and the resulting binaries have nearly same performance as a manually instrumented STM applications. We focused on the programming language `C` since modern managed programming languages like `C#` or `Java` have already supports for STMs. Nonetheless, `C` is mainly used in constrained environments. It is used for system programing, *e. g.,* programming the kernel of operating systems or in the area of embedded systems. Sensor nodes and smaller devices, use subsets of `C` or completely different languages, which limits the application of TANGER. Nonetheless, it is possible to create back-end modules for LLVM which would support these special subsets of `C`. Thus, TANGER is a flexible tool, which is very useful for prototypical use of `C`-based STMs.

However, the estimations for the development of small and tiny devices using a multi-core design were not correct and until now commercial devices are not widely available. This is why our research in this direction was not pushed any further. To be still able to fulfill our objectives at least partly, we created two STM-based systems to improve the processing of sensor data in an event streaming system. Both systems, one written in `C` and the other one in `Java`, can improve the processing speed especially in stateful processing components with minimal shared data conflicts. We also extended the `Java` version to a distributed STM component. In contrast to the distributed STM approaches of Kotselidis *et al.* [KAJ+08] or Noel [Noe08], we implemented distributed transactions which can be extended to the attached downstream components and thus creating a directed distributed STM. Furthermore, we show with our experiments, that the STM-enhanced ESP component can improve the throughput of certain input orders, even if only one thread is used for processing. As a result, the application to single-core systems could be used for certain input orders.

Another objective, set at the beginning, was the investigation of possible applications using the STM technology. We presented several use cases and a guideline to find others. Additionally, to our objectives we developed a blueprint for STMs. This “Transactional Object” design pattern can help to simplify the application and development of STMs.

8.3 Future Work

In this final section, we conclude this thesis with a brief outlook on possible future work.

As we have seen in the chapters dealing with the ESP, our framework was bound to discrete timestamps. They were used for the global ordering of the events and prioritization of conflicting transactions. We think it could be interesting to extend our approach from natural numbered timestamps to real numbered ones. The STM would have to deal in this case with a window or interval of acceptable timestamps. Our approach to distributed transactions could also be used to guarantee the overall order correctness of the processed events.

In a real ESP system, the frequency of incoming events could vary, thus we think it could be interesting to examine an STM-based stateful component with a dynamic adaption of parallel working threads. One could even think of switching to a sequential version if the event rate is low, to remove the overhead of the STM. For this improvement a detector system would be needed to measure the throughput and latency of the events.

Since our TANGER framework is based on LLVM, it could be a worthwhile contribution to extend the LLVM to generate code for non-C-based sensor nodes or to create sensor node which can run LLVM code natively. Another approach could be the application our STM-based ESP component to sensor node which are using high level programming languages.

A worthwhile challenge would also be to analyze if the STM could be used for testing in ESP environments similar to the approaches of Bobba *et al.* [BXY+09] for web server.

Last but not least, an interesting angle to investigate could be the implementation of bigger use cases or the ones we exemplified. These cases could be used as show cases to highlight the STM technology or as benchmarks to evaluate STMs.

Appendix A

List of Publications

A.1 Peer-reviewed Papers

2006

- H. Sturzrehm and H. Unger.
  Silkworm: A semi structured Peer-to-Peer System.
  In *Proceedings of the 2nd Malaysian Software Engineering Conference* (MySEC'06), 2006.

2007

- T. Riegel, C. Fetzer, H. Sturzrehm and P. Felber.
  Brief Announcement: **From Causal to z-Linearizable Transactional Memory.**
  In *Proceedings of the 26th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing* (PODC'07), 2007.

- P. Felber, C. Fetzer, U. Müller, T. Riegel, M. Süsskraut and H. Sturzrehm.
  Transactifying Applications using an Open Compiler Framework.[1]
  In *Proceedings of the 2nd ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing* (TRANSACT'07), 2007.

[1] The extended content is presented in Chapter 4

2008

- H. Sturzrehm, F. Aubert, P. Kropf and R. Corfu.
  NeOS: Neuchâtel Online System
  In *Proceedings of the 2nd International Workshop on E-Learning and Virtual and Remote Laboratories* (VLAB'08), 2008.

- A. Brito, C. Fetzer, H. Sturzrehm and P. Felber.
  Speculative Out-of-order Event Processing with Software Transaction Memory.[2]
  In *Proceedings of the International Conference on Distributed Event-Based Systems* (DEBS'08), 2008.

2009

- H. Sturzrehm, P. Felber and C. Fetzer.
  TM-STREAM: an STM Framework for Distributed Event Stream Processing.[3]
  In *Proceedings of the IEEE 23rd International Parallel & Distributed Processing Symposium* (IPDPS'09), 2009.

- H. Sturzrehm, P. Felber and C. Fetzer.
  Approach to a Transactional Memory Design Pattern.[4]
  In *Proceedings of the Computation World: Future Computing, Service Computation, Cognitive, Content, Patterns 2009* (ComputationWorld'09), 2009.

2010

- H. Sturzrehm, C. Fautsch and P. Kropf.
  Field Report on Five Years of eLearning. Observations and Inspirations.
  In *Proceedings of the 2nd International Conferences on Computer Supported Education* (CSEDU'10), 2010.

[2] The extended content is presented in Chapter 5
[3] The extended content is presented in Chapter 5 and 6
[4] The extended content is presented in Chapter 3

A.2 Books and Journals

2008

- H. Sturzrehm, F. Aubert, P. Kropf and R. Corfu.
  NeOS: Neuchâtel Online System
  In *International Journal of Online Engineering* (iJOE), Volume 4 Number 2, 2008.

- **H. Sturzrehm.**
  Silkworm: Ein hybrides Peer-to-Peer System.
  Vdm Verlag Dr. Müller, October 2008, ISBN-10: 3-63909-166-3.

A.3 Papers without Peer-review

2008

- H. Sturzrehm.
  Applicating Software Transactional Memory on Parallel Event Processing.
  In *Technischer Report IAM-08-003*, 2008.

2009

- H. Sturzrehm.
  Software Transactional Memory meets Event Stream Processing.
  In *Technischer Report IAM-09-006*, 2009.

Bibliography

[AAB+05] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag S Maskey, Alexander Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stan Zdonik. The design of the borealis stream processing engine. In *Proceedings of the 2nd Biennial Conference on Innovative Data Systems Research (CIDR'05)*, Asilomar, CA, January 2005.

[AAK+05] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded transactional memory. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA'05)*. Feb 2005.

[ATLM+06] Ali-Reza Adl-Tabatabai, Brian T. Lewis, Vijay Menon, Brian R. Murphy, Bratin Saha, and Tatiana Shpeisman. Compiler and runtime support for efficient software transactional memory. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation (PLDI'06)*, Jun 2006.

[BFSF08] Andrey Brito, Christof Fetzer, Heiko Sturzrehm, and Pascal Felber. Speculative out-of-order event processing with software transaction memory. In *Proceedings of the 2nd annual International Conference on Distributed Event-based Systems (DEBS'08)*, New York, NY, USA, 2008. ACM.

[BGAH07] Roger S. Barga, Jonathan Goldstein, Mohamed H. Ali, and Mingsheng Hong. Consistent streaming through time: a vision for event stream processing. In *Proceedings of the 3rd Biennial Conference on Innovative Data Systems Research (CIDR'07)*, Asilomar, USA, January 2007.

[BHS07] Frank Buschmann, Kevlin Henney, and Douglas C. Schmidt. *Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing*. Wiley, May 2007.

[Bou] Bouml. http://bouml.free.fr/, 5-Mar-2010.

[Bou07] Athanassios Boulis. Castalia: revealing pitfalls in designing distributed algorithms in wsn. In *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems (SenSys'07)*, New York, NY, USA, 2007. ACM.

[BXY+09] Jayaram Bobba, Weiwei Xiong, Luke Yen, Mark D. Hill, and David A. Wood. Stealthtest: Low overhead online software testing using transactional memory. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques (PACT'09)*, Washington, DC, USA, 2009. IEEE Computer Society.

[Car] Carlos GmbH. http://www.carlos.ch/, 5-Mar-2010.

[Cas] Castalia. http://castalia.npc.nicta.com.au/, 5-Mar-2010.

[Cha06] Mani K. Chandy. Event-driven applications: Costs, benefits and design approaches. In *Gartner Application Integration and Web Services Summit 2006*, San Diego, CA, USA, 2006. California Institute of Technology.

[CMCKO08] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *Proceedings of The IEEE International Symposium on Workload Characterization (IISWC'08)*, September 2008.

[Con] Contiki. http://www.sics.se/contiki/, 5-Mar-2010.

[CRA] CRAY Inc. http://www.cray.com/, 5-Mar-2010.

[CT65] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19(90), 1965.

[DB] DB Carsharing. http://www.dbcarsharing.de, 5-Mar-2010.

[DEU] Deuce. http://sites.google.com/site/deucestm/, 5-Mar-2010.

[DFL+06a] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Dan Nussbaum. Hybrid transactional memory. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'06)*, 2006.

[DFL+06b] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid transactional memory. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'06)*, New York, NY, USA, 2006. ACM Press.

[DGK09] Aleksandar Dragojevic, Rachid Guerraoui, and Michal Kapalka. Stretching transactional memory. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI'09)*, jun 2009.

[DGV04] Adam Dunkels, Björn Grönvall, and Thiemo Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the 1st IEEE Workshop on Embedded Networked Sensors (EmNets'04)*, Tampa, Florida, USA, November 2004.

[Dij65] Edsger Wybe Dijkstra. Cooperating sequential processes. Technical Report EWD–123, Technological University, Eindhoven, The Netherlands, 1965.

[dS99] Antonio Manuel Ferreira Rito da Silva. *Concurrent Object-Oriented Programming: Separation and Composition of Concerns using Design Patterns, Pattern Languages, and Object-Oriented Frameworks.* PhD thesis, Universidade Technica de Lisboa, 1999.

[DSH+03] Witold Drytkiewicz, Steffen Sroka, Vlado Handziski, Vlado H, Andreas Köpke, Holger Karl, and Technische Universität Berlin. A mobility framework for omnet++, 2003.

[DSS06] Dave Dice, Ori Shalev, and Nir Shavit. Transactional Locking II. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC'06)*, September 2006.

[Enn06] Robert Ennals. Software transactional memory should not be obstruction-free. Technical Report IRC-TR-06-052, Intel Research Cambridge Tech Report, Jan 2006.

[FFM+07] Pascal Felber, Christof Fetzer, Ulrich Müller, Torvald Riegel, Martin Süßkraut, and Heiko Sturzrehm. Transactifying applications using an open compiler framework. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT'07)*, August 2007.

[FFR08] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th annual ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'08)*, New York, NY, USA, 2008. ACM.

[FHB06] Bent Flyvbjerg, Mette K. Skamris Holm, and Søren L. Buhl. Inaccuracy in traffic forecasts. In *Transport Reviews*, Jan 2006.

[Fow02] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[Fra03] Keir Fraser. *Practical lock freedom*. PhD thesis, Cambridge University Computer Laboratory, 2003. Also available as Technical Report UCAM-CL-TR-579.

[Fre] Freemote. http://www.assembla.com/wiki/show/freemote, 5-Mar-2010.

[GC08] Justin Gottschlich and Daniel A. Connors. Extending contention managers for user-defined priority-based transactions. In *Proceedings of the 2008 Workshop on Exploiting Parallelism with Transactional Memory and other Hardware Assisted Methods (EPHAM'08)*. Apr 2008.

[GCC] GNU Compiler Collection. http://gcc.gnu.org/, 5-Mar-2010.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[GHP05] Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Polymorphic contention management. In *Proceedings of the 19th International Symposium on Distributed Computing (DISC'05)*. LNCS, Springer, Sep 2005.

[GP08] Gyozo Gidofalvi and Torben Bach Pedersen. Instant social ride-sharing. In *Procedings of the 15th World Congress on Intelligent Transport Systems (ITS'08)*. Intelligent Transportation Society of America, 2008.

[GPRZ08] Gyozo Gidofalvi, Torben Bach Pedersen, Tore Risch, and Erik Zeitler. Highly scalable trip grouping for large-scale collective transportation systems. In *Proceedings of the 11th International Conference on Extending database technology (EDBT'08)*, New York, NY, USA, 2008. ACM.

[Gra99] Mark Grand. Transaction patterns a collection of four transaction related patterns. In *Proceedings of the annual Pattern languages of programs (PLoP'99).*, 1999.

[Gre] Greenwheels. http://greenwheels.nl/, 5-Mar-2010.

[Her] Maurice Herlihy. http://www.cs.brown.edu/~mph/, 5-Mar-2010.

[Her05] Maurice Herlihy. Sxm1.1: Software transactional memory package for c#. Technical report, Brown University & Microsoft Research, May 2005.

[HF03] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *Proceedings of the 18th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'03)*, Oct 2003.

[HK08] Maurice Herlihy and Eric Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *Proceedings of the 13th annual ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'08).*, New York, NY, USA, 2008. ACM.

[HLM06] Maurice Herlihy, Victor Luchangco, and Mark Moir. A flexible framework for implementing software transactional memory. In *Proceedings of the 21st annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA'06)*, New York, NY, USA, 2006. ACM.

[HLMS03] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd annual symposium on Principles of distributed computing (PODC'03)*, New York, NY, USA, 2003. ACM.

[HM93] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th annual International Symposium on Computer Architecture (ISCA'93)*, New York, NY, USA, 1993. ACM.

[HM06] Jane K Hart and Kirk Martinez. Environmental sensor networks:a revolution in the earth system science? *Earth-Science Reviews*, 78, 2006.

[HPST06] Timothy Harris, Mark Plesko, Avraham Shinnar, and David Tarditi. Optimizing memory transactions. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation (PLDI'06)*, Jun 2006.

[HWC+04] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA'04)*. IEEE Computer Society, Jun 2004.

[IR08] Damien Imbs and Michel Raynal. A lock-based stm protocol that satisfies opacity and progressiveness. In *Proceedings of the 12th International Conference on Principles of Distributed Systems (OPODIS'08)*, Berlin, Heidelberg, 2008. Springer-Verlag.

[J-S] J-Sim. http://sites.google.com/site/jsimofficial/, 5-Mar-2010.

[JST] JSTM. http://www.xstm.net/, 5-Mar-2010.

[KAJ+08] Christos Kotselidis, Mohammad Ansari, Kim Jarvis, Mikel Luján, Chris Kirkham, and Ian Watson. Distm: A software transactional memory framework for clusters. In *Proceedings of the 2008 37th International Conference on Parallel Processing (ICPP'08)*, Washington, DC, USA, 2008. IEEE Computer Society.

[KJ04] Michael Kircher and Prashant Jain. *Pattern-Oriented Software Architecture Volume 3: Patterns for Resource Management*. Wiley, June 2004.

[Kni86] Tom Knight. An architecture for mostly functional languages. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming (LFP'86)*, New York, NY, USA, 1986. ACM.

[KR04] Milena Gateva Koparanova and Tore Risch. High-performance grid stream database manager for scientific data. In *European Across Grids Conference 2003*. Springer-Verlag Berlin Heidelberg, 2004.

[KSF09] Guy Korland, Nir Shavit, and Pascal Felber. Noninvasive Java concurrency with Deuce STM (poster). In *Proceedings of the 2009 Israeli Experimental Systems Conference (SYSTOR'09)*, may 2009.

[LA04] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[Lin] Linear Road Benchmark. http://www.cs.brandeis.edu/~linearroad/, 5-Mar-2010.

[LLD+07] Ming Li, Mo Liu, Luping Ding, Elke A. Rundensteiner, and Murali Mani. Event stream processing with out-of-order data arrival. In *Proceedings of the 27th International Conference on Distributed Computing Systems Workshops (ICDCSW'07)*, Washington, DC, USA, 2007. IEEE Computer Society.

[LLWC03] Philip Levis, Nelson Lee, Matt Welsh, and David Culler. Tossim: accurate and scalable simulation of entire tinyos applications. In *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems (SenSys'03)*, New York, NY, USA, 2003. ACM.

[LR06] James R. Larus and Ravi Rajwar. *Transactional Memory*. Morgan and Claypool, 2006.

[LSA] Lazy Snapshot Algorithm. http://www.tmware.org/lsastm, 5-Mar-2010.

[Mat08] Rainer Matischek. *A TinyOS-Based Ad Hoc Wireless Sensor Network: Introduction, Versatile Application Design, Implementation*. VDM Verlag, Saarbrücken, Germany, Germany, 2008.

[MBM+06] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. Logtm: Log-based transactional memory. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture (HPCA'06)*. Feb 2006.

[MHW05] Kevin E. Moore, Mark D. Hill, and David A. Wood. Thread-level transactional memory. *Technical Report: CS-TR-2005-1524, Dept. of Computer-Sciences, University of Wisconsin*, Mar 2005.

[MIC] MICS. http://www.mics.org/, 5-Mar-2010.

[MKKW08] Timothée Maret, Raphaël Kummer, Peter Kropf, and Jean-Frédéric Wagen. Freemote emulator: A lightweight and visual java emulator for wsn. In *Proceedings of the 6th International Conference on Wired/Wireless Internet Communications (WWIC'08)*, Tampere, Finland, May 2008. Springer.

[Moba] Mobility. http://www.mobility.ch/, 5-Mar-2010.

[Mobb] Mobility Framework for OMNeT++. http://mobility-fw.sourceforge.net/, 5-Mar-2010.

[Moo65] Gordon E. Moore. Cramming more components onto integrated circuits. In *Electronics Magazine*, 1965.

[MSH+06] Virendra J. Marathe, Michael F. Spear, Christopher Heriot, Athul Acharya, David Eisenstat, William N. Scherer III, and Michael L. Scott. Lowering the overhead of software transactional memory. In *Processdings of the 1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT'06)*, Jun 2006.

[MTC+07] Chi Cao Minh, Martin Trautmann, JaeWoong Chung, Austen McDonald, Nathan Grasso Bronson, Jared Casper, Christos Kozyrakis, and Kunle Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *34th International Symposium on Computer Architecture (ISCA'07)*, 2007.

[NCC] NCCR. http://www.snf.ch/E/targetedresearch/centres/, 5-Mar-2010.

[Noe08] Cyprien Noel. *XSTM Object Replication for Java, .NET & GWT*. Lulu Enterprises, Inc., 2008.

[NSBC09] Corrado Noseda, Alejandro Schnyder, Jürg Brand, and Edoardo Charbon. Hydromon: The first built-in on-line water quality monitoring system in a public supply network. In *Distribution Systems Symposium (DSS'09)*, Reno (CA), 2009.

[Per] PermaSense. http://www.permasense.ch/, 5-Mar-2010.

[PL08] Pavithra Parthasarathi and David Levinson. Post-construction evaluation of traffic forecast accuracy, 2008.

[RFF06a] Torvald Riegel, Pascal Felber, and Christof Fetzer. A lazy snapshot algorithm with eager validation. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC'06)*, volume 4167 of *Lecture Notes in Computer Science*. Springer, Sep 2006.

[RFF06b] Torvald Riegel, Christof Fetzer, and Pascal Felber. Snapshot isolation for software transactional memory. In *Processdings of the 1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT'06)*, Jun 2006.

[RM04] Kay Römer and Friedemann Mattern. Event-based systems for detecting real-world states with sensor networks: A critical analysis. In *Proceedings of the 2004 Intelligent Sensors, Sensor Networks and Information Processing Conference (ISSNIP'04)*, Melbourne, Australia, Dec 2004.

[RST] Rochester Software Transactional Memory. http://www.cs.rochester.edu/research/synchronization/rstm/, 5-Mar-2010.

[SATH+06] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. Mcrt-stm: a high performance software transactional memory system for a multi-core runtime. In *Proceedings of the*

ACM SIGPLAN 2006 Symposium on Principles and Practice of Parallel Programming (PPoPP'06), Jun 2006.

[sDE+06] Fredrik Österlind, Adam Dunkels, Joakim Eriksson, Niclas Finne, and Thiemo Voigt. Cross-level sensor network simulation with cooja. In *Proceedings of the 1st IEEE International Workshop on Practical Issues in Building Sensor Network Applications (SenseApp'06)*, Tampa, Florida, USA, November 2006.

[SFF09] Heiko Sturzrehm, Pascal Felber, and Christof Fetzer. TM-Stream: An STM framework for distributed event stream processing. In *Proceedings of the 23rd annual International Parallel and Distributed Processing Symposium (IPDPS'09)*, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[SGG09] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts.* Wiley, eighth edition, 2009.

[SM98] J. Steffan and T Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture (HPCA'98)*, Washington, DC, USA, 1998. IEEE Computer Society.

[SMSS06] Michael F. Spear, Virendra J. Marathe, William N. Scherer III, and Michael L. Scott. Conflict detection and validation strategies for software transactional memory. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC'06)*, 2006.

[SPGV07] Jesper H. Spring, Jean Privat, Rachid Guerraoui, and Jan Vitek. Streamflex: high-throughput stream programming in java. In *Proceedings of the 22nd annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA'07).* ACM Press, New York, NY, October 2007.

[SS04] William N. Scherer III and Michael L. Scott. Contention management in dynamic software transactional memory. In *Proceedings of the ACM PODC Workshop on Concurrency and Synchronization in Java Programs (PODC'04)*, St. John's, NL, Canada, Jul 2004.

[SS05a] William N. Scherer III and Michael L. Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of the 24th annual ACM Symposium on Principles of Distributed Computing (PODC'05)*, New York, NY, USA, 2005. ACM.

[SS05b] William N. Scherer III and Michael L. Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of the 24th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC'05)*, Las Vegas, NV, Jul 2005.

[SSRB00] Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschman. *Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects*. Wiley Series in Software Design Patterns. John Wiley & Sons, 2000.

[ST95] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the 12th annual ACM Symposium on Principles of distributed computing (PODC'95)*, New York, NY, USA, 1995. ACM.

[STA] Stanford Transactional Applications for Multi-Processing. http://stamp.stanford.edu/, 5-Mar-2010.

[Str] StreamMine. http://streammine.inf.tu-dresden.de/, 5-Mar-2010.

[Swi] Swiss National Science Foundation. http://www.snf.ch/, 5-Mar-2010.

[Sys] Systems Engineering Group at the TU Dresden. http://wwwse.inf.tu-dresden.de/, 5-Mar-2010.

[TAN] TANGER. http://tm.inf.tu-dresden.de/, 5-Mar-2010.

[Tan09] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice-Hall, second edition, 2009.

[TAR] TARIFA. http://www.hackshack.de/tarifa/, 5-Mar-2010.

[THGT07] Igor Talzi, Andreas Hasler, Stephan Gruber, and Christian Tschudin. Permasense: investigating permafrost with a wsn in the swiss alps. In *Proceedings of the 4th workshop on Embedded networked sensors (EmNets'07)*, New York, NY, USA, 2007. ACM.

[tin] tinyOS. http://www.tinyos.net/, 5-Mar-2010.

[Tra] Transactional Memory Online. http://www.cs.wisc.edu/trans-memory/, 5-Mar-2010.

[TUD] System Engineering Group at the TU Dresden. http://wwwse.inf.tu-dresden.de/, 5-Mar-2010.

[Tya02] Hung-Ying Tyan. *Design, realization and evaluation of a component-based compositional software architecture for network simulation*. PhD thesis, 2002.

[Upp] Uppsala DataBase Laboratory. http://user.it.uu.se/~udbl/, 5-Mar-2010.

[U.S06] U.S. Department of Transportation - ITS. *Traffic Signal Preemption for Emergency Vehicles. A CROSS-CUTTING STUDY*. U.S. Department of Transportation, 2006.

[VIS] VISSIM. http://www.vissim.de/, 5-Mar-2010.

[WCW+07] Cheng Wang, Wei-Yu Chen, Youfeng Wu, Bratin Saha, and Ali-Reza Adl-Tabatabai. Code Generation and Optimization for Transactional Memory Constructs in an Unmanaged Language. In *Proceedings of the 2007 International Symposium on Code Generation and Optimization (CGO'07)*, 2007.

[WDR06] Eugene Wu, Yanlei Diao, and Shariq Rizvi. High-performance complex event processing over streams. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data (SIGMOD'06)*, Chicago, USA, June 2006. ACM Press, New York, NY.

[ZGU+09] Ferad Zyulkyarov, Vladimir Gajinov, Osman S. Unsal, Adrián Cristal, Eduard Ayguadé, Tim Harris, and Mateo Valero. Atomic quake: using transactional memory in an interactive multiplayer game server. In *Proceedings of the 14th annual ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP'09)*, New York, NY, USA, 2009. ACM.