Towards Software Transactional Memory in Hard Real-Time Java Systems

9th Term Software Engineering Project

Department:

Database and Programming Technologies

Authors:
Marcus Calverley
Anders Christian Sørensen

Department of Computer Science
Aalborg University

Selma Lagerlöfs Vej 300
DK-9220 Aalborg Øst
http://www.cs.aau.dk

Title

Towards Software Transactional Memory in Hard Real-Time Java Systems

Department

Database and Programming Technologies

Project term

Fall 2011

Project group

sw903e11

Supervisors

Lone Leth Thomsen

Attachments

CD-ROM with STM source code and binaries and PDF-version of this report.

Abstract

This report documents our approach towards applying a pure software transactional memory (STM) in hard real-time systems (RTSs). We first refresh the concepts of RTS and schedulability analysis. Second, we describe transactional memory (TM) and present a set of design choices characterising TM systems. Based on these, we consider existing TM systems and describe them using these design choices. Finally, we discuss how concepts from Preemptible Atomic Regions (PAR) and Real-Time Transactional Memory (RTTM) can be combined and provide an analysable and predictable STM for hard real-time systems.

Participants

Marcus Calverley and Anders Christian Sørensen

Chapter 1

Preface

This report assumes the reader already has knowledge pertaining to programming languages Java and C. In addition, we expect familiarity with real-time systems theory and notation, namely regarding tasks, types of tasks, scheduling, WCET and response time analysis, although we do provide a brief recap of these. We also consider tools which rely on the UPPAAL model checker [1], so basic knowledge of UPPAAL is also expected of the reader. There is no prerequisite of familiarity with transactional memory as we cover this topic extensively.

Whenever we refer to a *programmer* in this report, we mean the person who uses the programming tools described, e.g. transactional memory.

We would like to thank our supervisor Lone Leth Thomsen for her in-depth feedback during this project, and Bent Thomsen for taking his time to discuss the topic of transactional memory and concurrency in general with us. Kasper Søe Luckow was also helpful with the Hardware-near Virtual Machine and the inner workings of TetaJ, which we consider briefly during this project.

Contents

Chapter 2

Introduction

Embedded real-time software (RTS) is a class of software which drives an increasingly large amount of electronic devices, such as the anti-lock brakes in motor vehicles, pacemakers, and anti-collision detection systems in airplanes. *Real-time* refers to the notion of computations with timing constraints, i.e. computations are required to finish within a given timeframe. Fulfilling the timing requirements set up is crucial to the system and failure to do so could have catastrophic consequences, such as the failure of the brakes on a car causing it to crash.

At the core of an embedded real-time system is the processor. Processors today are becoming increasingly parallel in that the industry is moving towards chip multi-processors (CMPs), i.e. processors with several processor cores that can perform parallel computations [2]. This leads to an increasing need for concurrent systems that can take advantage of this new generation of processors.

Using several concurrent threads of execution that can each run on different processor cores can lead to increased concurrency for the system, but many applications may require these threads to communicate. Shared memory is a communication method that allows threads to communicate by reading from and writing to memory that is accessible to all threads. However, one is often interested in working with a consistent snapshot of the parts of the shared memory that are needed. To this end, lock-based mechanisms are a common way to ensure mutually exclusive access to resources where needed [3, 4]. Our experience and literature will tell us lock-based mechanisms are inherently error-prone since they are manually put in place by the programmers, they will not scale with the number of cores if not implemented carefully, they are difficult or impossible to combine, and a frequent cause for deadlocks and priority inversion [5, 4].

Recently, transactional memory (TM) has gained interest as an abstraction for concurrency control in shared memory [4, 5, 6, 7, 8]. This allows the programmer to focus on the domain of interest rather than where to place locks to get the correct result of the computation but still achieve high concurrency. Thus TM can eliminate deadlocks, priority inversion, and other common problems that might arise when using lock-based concurrency. Transactional memory provides a simpler interface for programmers to access the current context, while letting the TM system take care of concurrency issues like scaling with the number of cores. However, the breadth of the current implementations of TM that we have looked at [4, 6, 8, 9, 10, 11, 12, 13] do not provide any timing guarantees that are necessary for RTS.

The first few steps towards bringing TM to RTS have already been taken [14, 15]. In this project, we research the concepts of RTS and TM to be able to analyse the existing work and determine under which conditions they have achieved their results. We then build on this to decide where, if possible, we can improve upon the state-of-the-art in real-time TM. More generally: we consider the schemes and techniques for handling transaction management in order to identity appropriate ones for achieving correct and high concurrency, and avoiding the pitfalls often encountered in concurrent computing, while examining which parts of the transaction management make it difficult or impossible to give timing guarantees.

2.1 Problem Statement

Locks are difficult to reason about and implement correctly [5]. Even so, lock-based mechanisms are the common way of achieving concurrency control in embedded real-time systems [16, 17]. In this context, can we use transactional memory (TM) as an alternative to locks? What have others done to achieve this and under which circumstances?

2.2 Subsidiary Goals

To answer the questions posed in our problem statement, we have devised a number of subsidiary goals that we have chosen to guide our way through the problem domain:

Real-Time Systems

- Define the characteristics of real-time systems and the properties required of the underlying technologies to understand what is necessary to support real-time systems.
- Choose a programming language to use as a focus for this project.
- Look at the tools available for developing and analysing real-time systems in the chosen programming language.

Transactional Memory

- Define the available options when designing a TM to delimit the different parts of a TM and understand their inner workings.
- Use the available options to analyse existing TM systems to show what choices their designers have made.
- Implement a few TM algorithms to better understand the impact of the available options when designing a TM ourselves.

Real-Time Transactional Memory

- Use the collected knowledge about real-time systems and transactional memory to analyse existing work on real-time transactional memory.
- Formulate an idea for a real-time transactional memory that would build upon the existing work to improve upon the state-of-the-art.

2.3 Report Structure

The report is structured as follows: Chapter 3 reiterates the most relevant information on real-time systems that we will use later in the report to ensure that we establish a common ground for understanding what is required of real-time enabled transactional memory. After this we look at tools that can support development of real-time applications in Chapter 4. In Chapter 5, we look at the basic concepts of transactional memory and what is required to build a transactional memory system, followed by Chapter 6 where we analyse two existing systems and our own implementations using the knowledge gained. We then use all our knowledge on real-time systems and transactional memory to look at how we can design a real-time transactional memory in Chapter 7, followed by an evaluation of our project period in Chapter 8 and the conclusion of the report in Chapter 9.

Chapter 3

Real-Time Software

Real-Time Software (RTS) is the class of computer software, which is required to provide a *response* to an *event* within a given timeframe. In this context, an event can be the occurence of an action in the surrounding environment, such as a proximity sensor detecting an object closing in. It can also be a message from an internal clock, signalling an *interval* in time. The latter can be used to trigger an operation at specific intervals. A response is the result from a computation performed based on an event.

Examples of RTS applications are anti-lock brakes, hearing aid devices, and pacemakers. Each provides a response based on an event in the surrounding environment, e.g. anti-lock brakes allow wheels to interact tractively with the road surface while braking; hearing aid captures sounds, amplifies them, and replays them through an ear-piece; pacemakers emit small electrical shocks to stimulate the heart rate. One can then imagine how *timing* is equally as important as *functional* correctness in RTS.

We provide this chapter in order to refresh basic real-time terminology and response time analysis. Outside this report, we will use the theory when reading and evaluating scientific papers on the subject. We present the knowledge in the report as well for three reasons: (a) to cover the basic principles of real-time systems, (b) to elaborate on concurrency mechanisms in real-time systems to provide a basis for tying together real-time systems and transactional memory, and (c) to give an in-depth description of response time analysis of real-time systems, since we will present derivations found in the literature when discussing real-time systems in conjunction with transactional memory in Chapter 7.

Section 3.1 provides the general definition of *real-time system* alongside the concept of tasks. Scheduling and schedulability tests in real-time systems are explained in Section 3.2, and finally worst-case execution time and response time is refreshed in Section 3.3.

3.1 Definition

While "real-time systems" in general share the notion of response time, it is worth providing a more precise definition. We will be using the definition given in [18]:

> *A real-time system is a system that is required to react to stimuli from the environment (including the passage of physical time) within time intervals dictated by the environment.*
>
> —Randell *et al.*, 1995

It is not to be confused with the use of real-time with respect to simulations or media processing. Here, the term refers to the fact that the simulation or video preview is generated on-the-fly without perceivable delay.

In the context of this project, the term is more stringent in its meaning. Computations in real-time systems are bounded in their execution by a deadline, meaning it is important they finish within a defined timeframe. RTS can be divided into two categories: *soft* and *hard*, determined by the critical nature of the system. Distinguishing the two is straight forward:

Soft Deadlines can be missed occasionally without rendering the system incorrect.

Hard Deadlines cannot be missed at any point in time. If it happens, the system is operating in a faulty state.

In addition to having deadlines, computations are also triggered either by time or an event. Running a computation at a given interval requires a timing mechanism to trigger, and running a computation when an external event occurs requires awareness with respect to the surrounding environment. A computation running at a given interval is referred to as *periodic*, while an event-triggered one is referred to as *aperiodic*. If an aperiodic task has a minimum inter-arrival time, i.e. events can only occur with a specific interval, then it is called *sporadic*.

Logically coherent pieces of computations are grouped into *tasks*. The term task serves as an abstraction to encapsulate different operations, typically named according to the application domain. As an example, an aircraft collision detection system might have a task *Read-Aircraft-Spatial-Properties* responsible for gathering altitude and direction vector for a given airplane.

Tasks are defined differently depending on the programming language, operating system or hardware platform. For example, operating systems following the OSEK specification for real-time operating systems [19] dictates the use of a C dialect. It defines a meta language for associating C functions with task names. The Real-Time Specification for Java, Real-Time Java, and Safety Critical Java use native Java annotations and class semantics to define tasks. The programming language Ada [20] has a `task` construct for defining tasks, and Real-Time POSIX [17], which allows real-time programming in C, defines functions which map C functions to threads, much like POSIX Threads [21].

Notation	**Description**
B	Worst-case blocking time
C	Worst-case execution time
D	Deadline
I	Maximum interference
J	Jitter
P	Priority
R	Worst-case response time
T	Interval between release
U	Utilization of the task ($\frac{C}{T}$)

Table 3.1: Properties of the general task model.

Conceptually, tasks are defined by a series of computations and a set of properties defined by the task model, where the aforementioned deadline property is one of them. The properties are described in Table 3.1 and centre around temporal aspects of the lifetime of a task. Their meanings are explained throughout the rest of this chapter.

To simplify analysis of real-time systems, a simple task model is used to introduce the concepts. It relies on the same properties as found in Table 3.1, but several limitations are imposed and listed in [22]. Below is an excerpt:

- All tasks are periodic.
- Tasks are completely independent of each other.
- System overheads such as context-switching [23] are ignored.
- All tasks have deadlines equal to their periods.
- All tasks execute on a single core.

However, in order to express more complex systems, these restrictions are relaxed in a more general task model:

- Arbitrary deadlines are allowed.
- Sporadic and aperiodic tasks are also supported.
- Task interactions are allowed, such as sharing memory.

Allowing tasks to share memory also causes the use of synchronization mechanisms to achieve exclusive access in critical sections. In this case, tasks can suddenly experience blocking and interference from other tasks, which must also be included into the response time analysis. This is demonstrated in Section 3.3. In the remainder of this report, we only consider the general task model unless otherwise stated.

3.1.1 Tasks, Threads, and Concurrency

Unifying tasks and threads is motivated by the wish to model real-world parallelism. Many real-time applications deal with responses to observations, or stimuli, in the environment which are observably parallel.

Temporal properties, such as deadlines, priorities, and periods, are defined at task level. Since the terms task and thread are unified, the temporal properties therefore also apply to the threads, and tasks are also concurrent. Consider the definition of a thread [23]:

> *A thread of execution is the smallest unit of computation that can be scheduled by an operating system.*

Then, because schedulers in real-time operating systems which support the notion of tasks actually consider tasks as the smallest unit of computation, the analogy is further enforced. Multi-threaded operating systems are able to context-switch between threads which allows for concurrent execution, and using multi-processors even has the potential to allow for parallelism. This provides a sound foundation for unifying tasks and threads. [22]

Figure 3.1 shows the general idea of how tasks can be defined programmatically without temporal properties. Each of the bodies, including other function calls, are then mapped onto threads.

```
task read_sensor {
    // computations...
}
```

```
void read_sensor() {
    // computations...
}

register_task(read_sensor);
```

Figure 3.1: Pseudo-code showing how tasks can generally be defined programmatically: by language construct or library.

3.2 Scheduling

In the previous section, we stated that certain real-time operating systems (RTOS) consider tasks as the unit of computation. Even though the underlying representation of a task is actually a thread, the real-time scheduler is able to consider the temporal properties defined.

Scheduling is an important aspect of multi-tasking operating systems. Since it is not necessary to specify the order of task execution, the interleaving at task level is non-deterministic. Although mechanisms can be used to synchronize the execution at points where necessary, e.g. in critical sections, the general observation remains non-deterministic.

For example, in preemptive scheduling, a finite set of tasks can interleave in infinitely many ways. Non-preemptive scheduling provides an upper bound on the amount of possible interleavings, which is $n!$, where n is the number of tasks in the task set. The challenge is to let the tasks interleave in a way such that none miss their deadlines. To achieve this, the real-time system needs to limit the non-determinism exhibited by concurrent systems. This is the activity known as scheduling. [22]

3.2.1 Scheduling Schemes

The decision-making inside a scheduler is defined by *scheduling schemes*. We will consider two scheduling schemes for real-time systems, namely *earliest deadline first (EDF)* and *fixed-priority scheduling (FPS)*. Although many other techniques exist, we will focus on these two as they are commonly used in practice [22] and described in the related work. In addition, our main goal is not to cover scheduling schemes but to investigate whether or not transactional memory has merit in real-time systems. Using EDF and FPS as a basis, our findings will thus have the potential to affect a large share of real-time application domains.

Earliest Deadline First (EDF) The order of release is determined by the absolute deadline; the task with the nearest deadline is released first.

Fixed-Priority Scheduling (FPS) Straight-forward and most widely used. Every task is assigned a fixed priority, which is decided by the programmer, or at least, pre-run-time. Tasks are released in the order dictated by their priorities.

3.2.2 Schedulability Tests

In order to determine whether or not a task set is schedulable according to either EDF or FPS, two techniques can be employed: utilization-based testing and response time analysis.

Utilization-based testing is the simplest of the two. Recall that $U = \frac{C}{T}$ from Section 3.1, which is what utilization-based tests consider. The general idea is that each test provides a bound on utilization. If the bound holds, the task set is schedulable according to the given scheme. If the bound does not hold in the EDF test, the task set is not schedulable. In both the case for EDF and FPS, the schedulability tests given do not take task interaction or arbitrary deadlines into account. As such, they only adhere to the simple task model.

Response time analysis applies to the more general task model, which supports arbitrary deadlines, task interactions and aperiodic and sporadic tasks. It does so by considering the fact that resources can be shared among threads, and the completion of a task assumes it has exclusive access to the given resource(s). Response time analysis is described separately in Section 3.3, but only for the case of FPS. Information on how response times can be calcuated for EDF systems can be found in [24].

EDF

Liu and Layland [24] provided a *sufficient and necessary* utilization-based test for task sets under EDF. It is sufficient in the sense that if the task set passes, it is indeed schedulable, and necessary meaning if the task set does not pass, it is not schedulable. Furthermore, it assumes the simple task model, in particular $D = T$ and no task interaction. Also, recall the notation from Table 3.1. It is given by:

$$\sum_{i=n}^{N} \left(\frac{C_i}{T_i}\right) \leq 1$$

Work has, however, been performed towards incorporating arbitrary deadlines in a simple test, where [25, 26] are approximations and [27] is sufficient but not necessary. However, [28] provided novel work on supporting arbitrary deadlines and still allows for a simple schedulability test.

FPS

Liu and Layland [24] also provided a utilization-based test for task sets under FPS. It is sufficient in the sense that if the task set passes, it is indeed schedulable, but not necessary meaning if the task set does not pass, it might still be schedulable. Furthermore, it assumes the simple task model, in particular $D = T$ and no task interaction. Also, recall the notation from Table 3.1. It is given by

$$\sum_{i=1}^{N} \left(\frac{C_i}{T_i}\right) \leq N\left(N^{1/N} - 1\right)$$

Improvements over the original test has been developed since. One is described in [29], and correctly passes more cases. To support the general task model, we describe worst-case response time analysis in the next section.

3.3 Worst-Case Execution Time and Response Time

This section describes the differences and relation between worst-case execution time (WCET) and response time, and provides a formal definition on how to calculate the response times of a set of tasks in a fixed-priority scheduling (FPS) setting.

In order to decide whether or not a set of tasks can run successfully, i.e. within their deadlines, a worst-case execution time (WCET) analysis is performed. By knowing the WCET of each task in the task set, meaning its period, deadline, and priority, it is possible to express its response time, and, subsequently, determine the WCET of the entire system. Response time considers not only the WCET of a task, but also the fact that other tasks can block or interfere. To help understand this, consider a set S of tasks $t_0, t_1, \ldots, t_n$ each with a WCET of w_i. The WCETs are independent, meaning that each only describes the worst thinkable amount of time each *one* task takes to complete, not considering the fact that other tasks can run before or preempt the task in question.

When a task is about to run, the worst-case scenario is that every other task is of higher priority and also scheduled to run at the same point in time. In other words, all tasks are released at the same time—also referred to as a *critical instant*. A task prevented from running by a higher-priority task is being *interfered*. A task prevented from running by a lower-priority task is being *blocked*. Blocking happens when a lower-priority task holds a resource, which a higher-priority task attempts to acquire. In other words, the higher-priority task is waiting for the lower-priority task to complete, or at least give up the resource. This phenomenon is referred to as *priority inversion*.

As such, the WCET of a task is rarely the response time of a task when considering the entire system; it can be much worse. For example, if task t_0 is released but blocked or interfered by t_1 and t_2 which both run until completion, the observed execution time of t_0 can be as high as the running time of all three: $w_0 + w_1 + w_2$. Response time is used to express this phenomenon. It requires the WCETs for each task, the worst-case blocking time and amount of interference each can experience to be known.

3.3.1 Formal Definitions

We discuss literature regarding WCET analysis of real-time TM applications in Section 7.1. The concepts and ideas revolve around Equation 3.1, which have been extended throughout the literature to incorporate phenomena such as blocking and interference [22]. In the setting of introducing TM into real-time systems, attempts at extending this formula even further to include the notions of transactions have also been made. We provide a basic understanding of the formula in this section to provide a foundation for discussion in Section 7.1.

Using the notation from Table 3.1 in Section 3.1, the response time for a task using the simple task model t_i is expressed by [30]:

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{3.1}$$

where $hp(i)$ is the set of tasks with higher priority than task i, and $\sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$ is an expression for the worst-case interference task i can experience. However, it introduces R_i on the right-hand side as well, and the equation becomes a fixed-point equation. Solving the equation is not straight-forward because of the ceiling function, but it is indeed solvable. A fixed-point equation will potentially have many different values which satisfy it. The solutions for Equation 3.1 are expressed by the recurrence relationship [31]:

$$w_i^{n+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j \tag{3.2}$$

Using FPS defined in Section 3.2.1, it is logical to assume the task t_{hp} with the highest priority is not interfered by any other task. Following from that, the worst-case response time (which includes interference) is equal to the WCET of the task, that is $R = C$. The remaining tasks, which all have lower priorities than t_{hp}, can potentially suffer from interference, which is captured in response time analysis. An example of using the recurrence to calculate the response time for a set of tasks is shown in Appendix A.

Chapter 4

Real-Time Software Tools

In this chapter, we look at some of the tools that can be used to develop real-time software. For this project, we focus on real-time software made in the Java programming language, so the tools are related to making Java run on real-time embedded systems and analysing their worst-case execution time. Java was chosen because it is a popular programming language that has shown its worth in development for real-time systems [32]. Since we want to design a real-time TM to use for concurrency control in hard real-time Java applications, we will need to be able to analyse the implementation to verify that it is schedulable. Using such a tool to perform the analysis will automate the verification.

It should be noted, however, these tools are not put into practice in this project. We include them to provide a basis for deciding which hardware platform we will implement our approach on in future work.

We begin by looking at the Java Optimized Processor (JOP) in Section 4.1 and the tool SARTS to analyse the running time of Java applications on the JOP in Section 4.2. Afterwards, we look at the Hardware-near Virtual Machine (HVM) in Section 4.3 and the tool TetaJ in Section 4.4 that can analyse running times of applications running on the HVM.

4.1 Java Optimized Processor (JOP)

The Java Optimized Processor (JOP) [33] is a processor which executes Java bytecode in a time predictable fashion. Where modern processors include several forms of execution time enhancing features such as branch prediction and multiple levels of caching [34], the JOP does not and thus provides a predictable execution model. Predictability is preferred over performance in real-time systems to ensure the system is schedulable under all conditions (Section 3.1).

General Java virtual machines (JVMs) interpret the Java bytecode which is emitted by the Java compiler, and it is also able to *just-in-time* (JIT) compile it. Implementing JIT-compilation in embedded hardware is usually impossible due to the nature of embedded hardware; it is simply too resource demanding, and it imposes another level of unpredictability [35]. On top of that, the observed execution time suddenly includes both time consumed JIT-compiling and actual execution time.

4.1.1 Execution Model

General purpose processors have *machine code* as their native language, i.e. they execute machine code directly. The JOP is different in that it translates Java bytecode into *microcode* which is then

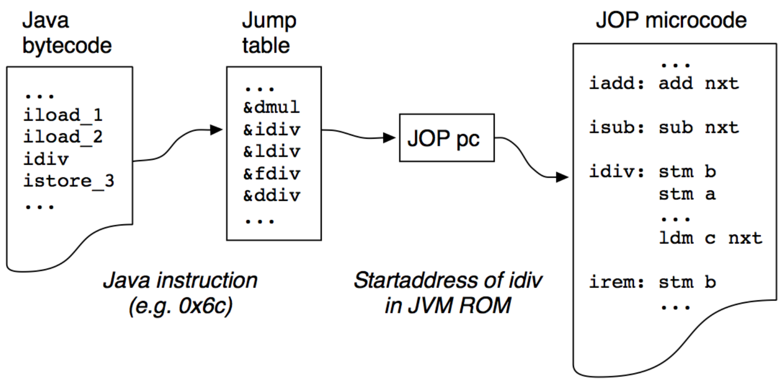

Figure 4.1: Flow inside the JOP: from Java bytecode instruction to microcode instruction. [33]

executed. This flow is illustrated in Figure 4.1. The Java bytecode instruction `idiv` is fetched, and then used as an index in the jump table to look up the microcode instructions equivalents.

4.1.2 Predictability

The JOP was developed solely for serving as a processor in real-time systems. As such, certain design choices were made towards this. General processors usually have features such as branch prediction and several layers of data cache, but due to the unpredictable behavior of these, they are difficult to reason about in WCET analysis and thus inappropriate for real-time systems.

Below we describe an excerpt of the design choices applied in the JOP. The purpose is to underline the difference between general purpose and real-time hardware platforms, and the fact that predictable execution times are preferred over performance.

Interrupts. In general purpose OSs, interrupts often arrive due to device drivers signalling something. They are considered more important than normal tasks, and will thus interrupt anything currently running. The JOP defines interrupts at the same level as tasks, meaning interrupts are also schedulable. This allows for interrupt handler costs to be included in the schedulability analysis.

Branch prediction. To increase the throughput on general purpose CPUs, some attempt to guess which branch a program will take, e.g. in an `if-then-else` statement. A correct guess will save cycles throughout the pipeline, while an incorrect guess will impose execution time equal to as if branch prediction did not exist. *Why does the JOP not support branch prediction then, if the upper bound on execution time known anyway, and cycles can be saved?* First, the pipeline of the JOP is only four stages (four cycles). The risk of branch misprediction will incur a delay in addition to the four cycles, which is bound to happen at some point. Second, predictability is preferred over performance in hard real-time systems.

Details on how the above is achieved and other design choices can be found in the original work on the JOP [35, sec. 5.7].

4.1.3 Java and the JOP

Java is not suitable for real-time systems as-is. However, *real-time profiles* for Java have been implemented to accommodate this. Real-time profiles address the concept of tasks, temporal properties on tasks such as periods, deadlines, and lastly scheduling and memory allocation patterns. The Real-Time Specification for Java (RTSJ) [16] is one attempt at this, but due to its complexity it is undesirable for the JOP: it takes up too much space, and most hard real-time applications can be implemented using a much simpler profile, which in turn is also easier to analyse [35, ch. 6]. Thus a hard real-time profile that is a subset of RTSJ was defined for use on the JOP.

Some Java features should be avoided to yield the most WCET-analysis friendly bytecode. An extensive list of constructs to avoid is provided in [36]. Below is an excerpt:

Static class initialization. If a class with a static constructor is either instantiated, or a static method/field is accessed or invoked, the static initialization will be run in advance. This is not transparent and can result in pessimistic WCET analysis.

Inheritance. Using Java interfaces and excessive class inheritance should be avoided to minimise dynamic dispatching of method invocations.

String concatenation. A string literal is the only type of string which is allowed to be used for concatenation, and only when done in ever-living memory scopes.

Reflection. Dynamically loading and injecting classes, as well as dynamically accessing objects, is disallowed.

Application Structure

Applications are divided into an *initialization* phase and a *mission* phase. Initialization covers thread initialization and memory allocation. Mission is the part of the application which constitutes the actual logic responsible for providing the wanted behavior.

Threads are assigned fixed priorities during initialization, and the number of threads are fixed, too. To summarise, the characteristics of the JOP in this context are:

- Initialization and mission phase.
- Fixed number of threads.
- Threads are allocated during initialization.
- Shared objects are allocated during initialization.

4.2 Schedulability Analysis Abstractions for Safety-Critical Java (SARTS)

Schedulability Analysis Abstractions for Safety-Critical Java (SARTS) [37] is an automated model-generation tool for real-time Java applications written for the JOP in a modified Safety-Critical Java (SCJ) profile called SCJ2. According to the authors of SCJ [38], it was developed for the main reason that standard Java and RTSJ were inadequate for some safety-critical applications. Acknowledging a simpler profile would also result in simpler schedulability analysis.

SARTS analyses the real-time system source code and generates a UPPAAL model representing the system. The model represents each of the tasks and their temporal properties, and is able to respond to the question whether or not the system is schedulable.

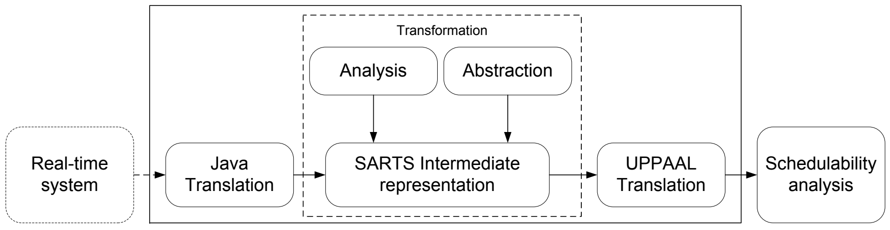

Figure 4.2: Overall architecture of SARTS. [37]

SARTS generates a corresponding UPPAAL model from a real-time system written in SCJ2. The architecture of SARTS is illustrated in Figure 4.2 which shows the steps involved in transforming source code to a UPPAAL model.

Detailed descriptions of each step is found in the original work on SARTS [37].

4.2.1 Safety-Critical Java 2

SCJ2 was developed by the authors of SARTS. It is based on SCJ, and its purpose is to be a *"simple, but sufficient, profile targeted hard real-time systems"* [37]. Applications have three states as depicted in Figure 4.3. This is similar to that of the SCJ profile.

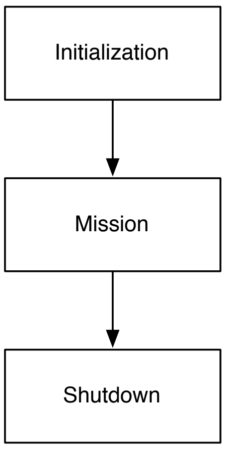

Figure 4.3: Possible states of SCJ2 applications. [37]

To keep SCJ2 simple, but still sufficient for many types of applications, it only supports periodic and sporadic tasks. Tasks are setup in the initialization phase, running according to their temporal specifications in the mission phase, and ensured a proper shutdown in the shutdown phase. This is similar to the application structure for the JOP and the original profile, as described in Section 4.1.

Memory Model

SCJ allocates memory during the initialization phase: one area shared between the tasks, and areas local to each thread. Allocation of additional memory during other than the initialization phase is not allowed, and thus the need for a garbage collector is alleviated. A real-time garbage collector has been developed for the JOP [39], but it is still a research project.

It is also discussed in [37] that a more dynamic memory model, i.e. one which allows for allocation during the mission phase, is desirable. After all, one of the main purposes of introducing transactional memory as concurrency control in real-time systems is to lessen the burden of the programmer. Demanding memory to be allocated during initialization phase is, in our opinion, a fair limitation in real-time systems and is also practiced in other real-time settings [22].

4.3 Hardware-near Virtual Machine (HVM)

The Hardware-near Virtual Machine (HVM) is a Java VM intended for embedded systems. It has several interesting features which make it suitable for embedded platforms, such as:

Intelligent Linking Only the parts of the Java framework used are linked with the application.

SDK Independence It does not rely on the application being written for a specific Java SDK version.

OS Independence The HVM itself is linked along with the application, and as such an OS on the device is not required.

Storage Awareness Read-only data is placed in read-only data areas of the memory where possible, which effectively minimises the RAM usage.

Optional Ahead-of-Time Compilation Java methods can be marked to be compiled ahead-of-time (AOT). Doing this, they will be compiled to C code directly, and thus saving the interpreter in the HVM of doing the work. AOT takes up more ROM, but in turn the methods run up to four times faster.

Applications for the HVM are written in Java, and the workflow is as follows:

1. Compile the Java code using a Java compiler.
2. Determine the dependency extent of the application, i.e. which classes are used, methods and so on.
3. Compile the dependency extent into C code, which is essentially a set of bytecode instructions represented in C rather than Java *class* files.
4. Compile the generated C code using GCC or any other C compiler. The resulting object files are uploaded to the device, and the HVM will execute the bytecodes generated.

HVM development takes place in the integrated development environment (IDE) Eclipse[1]. The tools are integrated into Eclipse as a plug-in and are available when in Java mode. A second project—in C mode—must be open in the Eclipse workspace as well, which will function as the *target* project for the generated C code. This C code can then be compiled for the target processor that the real-time system will run on.

The *HVM analyzer* is a bundled application capable of determining if the resulting application will take up too much space.

[1]`http://www.eclipse.org`

Native methods, i.e. methods specific to the hardware platform, are also supported. These will not be implemented in the generated C code, but stubs for them will. Every native function is passed a pointer to the Java stack `sp`. If the user-implemented native functions return a value, this must be left in `sp[0]`, and parameters passed to the native function can be read from `sp[0]` and up. Upon success, native functions should return -1. Missing native function implementations will result in compile errors, and these must of course be corrected before flashing the device.

The idea of mapping bytecode instructions to machinecode is not novel (see Section 4.1), but the size of the HVM and almost hardware-agnostic approach is. Since the HVM has this mapping, the execution of Java bytecodes is also time-predictable. This calls for tools for generating WCET analysis automatically by static analysis.

This information is based on what we found on the HVM website[2]. As of this writing, there is no actual documentation other than what is described in this section.

4.4 TetaJ

Like SARTS, TetaJ [40] is tool that can be used to construct a UPPAAL model for an application to use for WCET-analysis. The model is generated from the Java source code of the application and captures control flow and which bytecode instructions are executed. Combined with a model of the underlying JVM and hardware platform, it can output the WCET of the application. The source code is required to be annotated with loop bounds, which is crucial in order to keep an upper bound on the execution time.

Contrary to SARTS, TetaJ is able to perform this analysis of any annotated Java program, provided a model for the underlying JVM and hardware platform is defined. In the original work on TetaJ [40], the analysis is performed based on the HVM described in Section 4.3.

Figure 4.4 illustrates the flow of TetaJ. The highlighted blocks in the figure denote the main components of TetaJ:

Model Generator Extracts a control flow graph (CFG) from the Java bytecode and generates a UPPAAL model from it. Besides the control flow, the CFG also captures bytecode instructions for each code block, loop bound annotations and instruction-to-source code mapping.

Model Combiner Combines the hardware, JVM and program model into one model. The purpose is not only to consider the program model and every instruction it consists of, but also taking into account the underlying platform. Having this information allows for determining a realistic WCET for the application on the given platform.

Model Processor Queries the combined model using UPPAAL to output the WCET for the application.

Since TetaJ is hardware and JVM agnostic, it does not impose restrictions on the choice of hardware or JVM.

[2]`http://0x4dd56dac.adsl.cybercity.dk/`

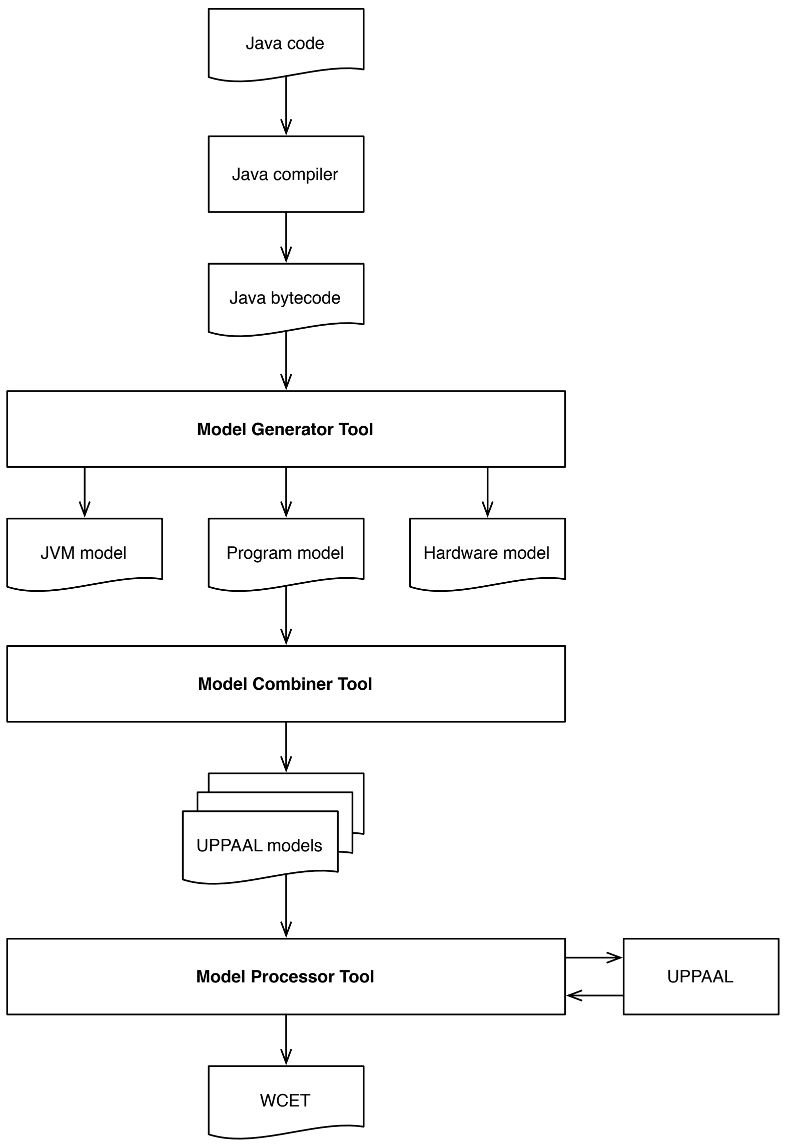

Figure 4.4: Overview of the activites in TetaJ. [41]

Chapter 5

Transactional Memory

A recent alternative to lock-based concurrency is *transactional memory* (TM) [6]. TM uses a different approach to protecting shared memory: instead of locking the memory, critical sections are run concurrently based on the assumption that they rarely conflict. In case a conflict does occur, the TM system monitors the critical sections to detect the conflict and then resolves it automatically. When using TM to protect critical sections the programmer wraps them in *transactions*.

In this chapter, we will look at the theoretical foundation of transactional memory and the design choices that are made during the design of a TM system. We begin by motivating TM as a concept and compare it to traditional lock-based concurrency in Section 5.1. We then define the concept of TM transactions in Section 5.2, before we introduce correctness in the setting of transactional memory in Section 5.3. After this, we give an overview of the different elements in a TM system and what choices must be made to successfully build a working TM solution for general purpose programming in Section 5.4 and the following sections.

5.1 Motivation

Concurrency is becoming increasingly important as Chip Multi-Processors (CMPs) are becoming wide-spread [42]. This means that applications must utilise concurrency to take full advantage of the available processing power.

A common way to introduce concurrency in applications today is by using threads, which are sequential bits of code that run in parallel [43]. In many cases parts of this code must communicate with other threads in order to proceed. One way to communicate is through shared memory that can be read and written to by all threads. However, some programs require that certain invariants hold in the shared memory. The current mainstream approach to maintain these invariants involves threads *locking* certain parts of the shared memory during *critical sections* of the execution of the threads. With lock-based concurrency the shared memory is thus accessible to only the thread that holds the lock and the thread can make its changes to the shared memory without other threads being able to access the inconsistent state. [22]

However, locking the shared memory (or parts thereof) means that other threads will have to wait their turn to continue executing, which hampers concurrency, while locking too little may make an inconsistent state accessible to other threads. The challenge of concurrency using shared memory is thus to improve the performance of the software while still ensuring its correctness. However, since locks require the programmer to manually add them at appropriate points and ensure that they guarantee the appropriate protection of the shared memory, they are inherently difficult to use.

A programmer can either choose coarse-grained locking, in which all (or at least an overly conservative amount) of the shared memory is locked to ensure the critical sections execute correctly, or fine-grained locking, in which extreme care is taken only to lock very specific parts of the shared memory only at the times they are needed. The problem with coarse-grained locking is that it concurrency is decreased because threads can be blocked more frequently waiting for locks that are held more than strictly required. Fine-grained locking solves this issue, but in its place adds increased complexity for the programmer. [8]

The current approach of lock-based concurrency has also been shown to be a significant hindrance to *composability* [5]. Composability is the ability for software components to be composed into larger software components. With lock-based concurrency, components that work correctly independently may not function correctly when composed into larger components. An example of this is found in [5], where bank account management software has functions that allow depositing and withdrawing money. The implementation uses fine-grained locks to allow concurrent access while avoiding race conditions when the account balance is updated. However, composing these functions into a function to transfer money from one account to another proves more difficult than just calling withdraw on one account and deposit in another if the state where the money exists in neither account may not be visible.

Transactional memory promises to alleviate these problems, and in the rest of this chapter we will look at what it takes to implement transactional memory for use in general purpose programming.

5.2 Transactions

The term transaction comes from the database world, where transactions are a set of database queries that run as if they are the only ones accessing the database. The database is free to run other transactions concurrently as long as transactions do not interfere with each other. This is accomplished by ensuring the *ACID* properties: atomicity, consistency, isolation and durability. [11]

In the database world the ACID properties have the following meaning [11]:

Atomicity ensures that either the entire transaction commits or all of the changes made are rolled back in case of failure.

Consistency means that the invariants that must hold for the database to be in a consistent state are maintained by each transaction if it commits.

Isolation requires that running transactions appear to have the database entirely to themselves, meaning that no other transaction may interfere with a running transaction.

Durability ensures that the results of a committed transaction must be durable, i.e. even in the event of a system crash, the database must retain the results of the transaction.

By maintaining these properties for each transaction, it is possible for the user of the database to write queries without worrying about concurrency issues like race conditions and deadlocks.

The idea of transactional memory is thus to take this concept of transactions from the database world and apply it in a general purpose programming context. However, there are some fundamental differences from database transactions that must be considered before the concept can be applied in this setting [11]:

Non-Transactional Access In databases every query is a transaction, so it is not possible to access data without using transactions. In general purpose programming this may not be the case; access to shared memory could be possible without it being part of a transaction. How transactions react to this case must be considered.

No Durability Concerns Transactional memory abstracts access to volatile memory, which inherently is not a durable storage. Also, durability in a memory context relies solely on the lifetime of the surrounding process: when a process is terminated, the memory it had allocated is no longer durable.

Stricter Correctness Criteria Where serializiability is the most important correctness criteria for database transactions [44], it does not suffice in transactional memory: it does not capture the requirement that every transaction must only access consistent state. *Opacity* is introduced as a new correctness criteria for transactional memory systems, and is described in Section 5.3.

5.3 Correctness

When using a TM to develop an application, the programmer may have certain expectations of how transactions will work. In addition to the transactional properties defined in Section 5.2, the programmer may expect reads to always return the latest consistent value of a shared memory location. If not, the code in the transaction may loop indefinitely or crash the application. This leads us to the fact that TM systems require stronger correctness criteria than database transactions because the transactions may contain arbitrary code. [11]

In this section we look at what expectations there are of TM systems and how they should behave to meet them by defining the correctness properties that a TM system should ensure. Using these correctness properties transactions will have the same semantics when they are run concurrently as when they run sequentially, which is the foundation of TM. [11]

5.3.1 Expectations

The characteristics of transactions in TM may seem intuitive, and people using TM often have unspoken expectations towards how transactions behave. However, since transactions can be used to handle all concurrency in a system, the need to express TM transactions characteristics correctly becomes increasingly important. This applies to both the users, but also authors of TM systems.

Atomicity Committed transactions must appear as if they executed instantaneously, or *atomically*. That is, observably, nothing else but a given committed transaction was running during its lifetime. In the case of several committed transactions, an example transaction history[1] H is given in Figure 5.1. If each committed transaction is observably executed atomically, then there must be an *equivalent* and *sequential* transaction history H' of H. Equivalent in this context implies that the sequential transaction history H' must (1) contain the same transactions as H, and (2) the operations in H' read/write the same values as in H. The correct sequential history of H is when the transactions run in the following order: T_3, T_1, T_2.

[1]A transaction history is an overview of how transactions interleave and modify/read data, where $x \leftarrow 1$ indicates that 1 is written to x, and $x \rightarrow 1$ indicates that 1 is read from x. Variables are initially 0. A solid dot at the end of a transaction indicates that it has committed, while an outlined dot indicates an abort. The y-axis is time.

Preserving Real-Time Ordering A transaction must not read an outdated state of the system. As an example, transaction T_1 modifies x and commits, and after this transaction T_2 starts and reads x, then T_2 must read the value written by T_1.

Preventing Inconsistent Views If transaction A access inconsistent data, the committed work by A will in most cases also be inconsistent. As an example, two transactions A and B accesses shared objects $x = 6$ and $y = 6$. Let x have the invariant such that $x \geq 6$, and y the invariant such that $y = x$. If A then executes $x \leftarrow 10$ followed by $y \leftarrow x$ and commits, and B concurrently reads the *new* value of x (10) and the *old* value of y (6), then B has read inconsistent data, and will possibly crash, enter an infinite loop, or continue to work with the inconsistent data.

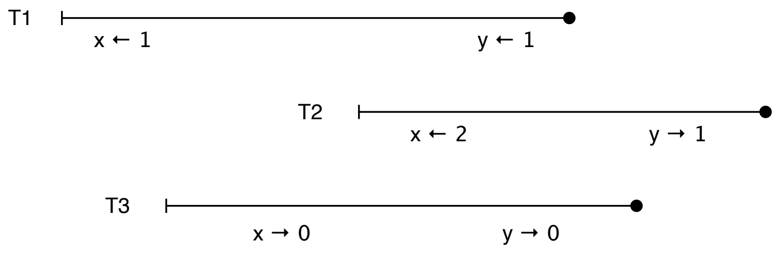

Figure 5.1: Transaction history H. [10]

The observable behavior of a series of transactions manipulating the same set of shared object will be as if they each executed sequentially. However, one of the key ideas of TM is abstract away from what can and should be run concurrently, and let the run-time handle it, and utilising potential multiple cores.

5.3.2 Motivation

The need for a more strict correctness criterion in TM is based on the fact that linearizability [45] and serializability [46] on their own do not suffice. They are both an effort to capture the idea of sequential ordering mentioned in the introducing text in this section, which is also referred to as *sequential consistency* [47]. More specifically, they do not describe the requirement that every transaction only accesses consistent state [48].

To reiterate, the issues with linearizability and serializability in a TM context are listed below.

Linearizability Every transaction should appear as if it occurred at an indivisible, unique point in time [45]. However, since transactions should be able to run concurrently, it is unclear how linearizability can be accommodated: if a transaction appears to run at an indivisible, unique point in time, then transactions should not be able to conflict, as a transaction should not be able to discern any intermediate part of another transaction. [11]

Serializability Considering a series of transactions executing (concurrently), then the result thereof should be equal to the result of all *committed* transactions run sequentially. Not only from an external point of view, but every read and write operation, too. This does not suffice for TM transactions since (a) it assumes every read on transactional variable x returns the last-written value, (b) it allows only read and write operations, and (c) it does not consider the state of the accessed memory by transactions (including aborted ones).

5.3.3 Opacity

We provide an informal description of opacitiy in this section, while [48] defines opacity and its justification formally.

Opacity captures the exact expectations given in Section 5.3.1 formally, that is (1) all operations performed by every *committed* transaction seem as if they were performed at a single, unique instant during its lifetime, (2) any operation performed by any *committed* transaction is never visible to other transactions, and (3) every transaction always sees consistent data.

(1) is captured by *global atomicity* [49], which states that given a transaction history H, then removing all non-committed transaction from H results in a history H' which in turn is equivalent to some *legal* sequential history S. Legal means that every transaction respects the sequential specification of all shared object. Additionally, S must also maintain the real-time ordering of transactions in H. Global atomicity does not, however, account for the data modified by *aborted transaction*, and thus is not strong enough to capture points (2) and (3), too.

(2) and (3) are addressed when considering *complete* histories only, i.e. histories containing non-live transactions only, every transaction (aborted or committed) sees the data as produced by preceeding transactions. In other words, there must exist a sequential history S equivalent to H, where equivalent is defined in Section 5.3.1, the real-time ordering is preserved, and S only contains legal transactions.

While it is easy to determine whether a sequential complete history is legal, i.e. when all transactions in it are legal with respect to each other, an incomplete history H first has to be transformed into a complete one H'. This is achieved by either committing or aborting every live transaction in H. However, a live transaction which is not commit-pending can only be aborted, as it shares the same semantics as those of an aborted transaction: none of the changes made to shared objects can be visible to other transactions.

5.4 Design and Implementation Choices

We now begin our general discussion of the components of a TM system. There are several interesting choices that must be made when implementing a TM system to support transactions in general purpose programming. We have already touched upon some issues, like how the behaviour of non-transactional and transactional concurrent access to shared memory must be considered. In this section, we give an overview of the issues that need considering and in the following sections we will look closer at them one by one. The list of issues being discussed has been derived from [10, 11, 7] and represents the choices that we have concluded to be essential to the design of a TM system.

Operational Structure To use a TM system the programmer must identify transactions in the code and communicate that to the TM system. The TM system can here either be a library that has certain functions which the programmer must call at specific times to use the TM system, or it can be a more integrated part of the programming language that provides new syntax to the programmer which will then handle the calls to the TM system automatically. In Section 5.5, we look at these possibilities and what they entail.

Conflict Detection When two transactions come in conflict over a part of the shared memory, the TM system must resolve this conflict to ensure that transactions only work with consistent data. The system can either detect it as the conflicting part of the shared memory is read or written, at commit time, or at some other point during the transaction. The merits and problems introduced with this choice are discussed in Section 5.6.

Direct or Deferred Update When a transaction makes changes to shared memory there are two options for where to store these changes: directly in the shared memory, or in a special log that can be used to write the changes to memory when committing the transaction. The former is called *eager versioning*, as the new version of the variable is stored immediately, while the latter is called *lazy versioning* as writes are saved in transaction-local storage until they are finally committed to memory. Each strategy comes with its own benefits and challenges that we will discuss in Section 5.7.

Isolation In addition to the isolation guarantees provided for transactional access to shared memory, the TM system may also give certain guarantees about non-transactional access. The behaviour of the TM system in this area can be a result of the other design choices, where e.g. eager versioning might lead to non-transactional access reading inconsistent data. The TM system may also transparently wrap non-transactional access to shared memory in transactions as to provide consistent access to shared data from anywhere. These ideas are elaborated in Section 5.8.

Nested Transactions Another implementation detail is how to handle nested transactions. The concept of nested transactions, where transactions are started from within another transaction, is a central part of the ability for TM systems to provide composability. There are several ways to handle nested transactions in a TM system which, depending on the amount of contention, may increase the concurrency of the TM system. The schemes for nested transactions and their effects are discussed in Section 5.9.

Hardware or Software When designing a TM system it is necessary to consider whether it will be implemented in hardware or software or if it will be a hybrid of the two. *Hardware Transactional Memory* (HTM) usually provides some performance benefits that can not be achieved in *Software Transactional Memory* (STM). However, STM is more flexible as it is possible to update the STM system without replacing the hardware platform. STM also allows programmers to use a TM system on off-the-shelf hardware that does not provide HTM. The last option, *Hybrid Transactional Memory* (HyTM), tries to combine the two in a reasonable way, thus trying to provide the speed benefits of a hardware implementation while leaving certain details to software that increases the flexibility of the TM. This is further elaborated in Section 5.10.

Granularity The TM system stores metadata about the individual locations of the shared memory that it is tracking. The granularity of the tracking of changes to shared memory affects the performance of the TM system, both in respect to how much metadata has to be stored and maintained, but also for the conflict detection process, which may be affected if the shared memory is tracked at too coarse a level. An example of granularity is a TM system that tracks changes to shared memory at the object level in an object oriented programming language: here changes to shared memory are detected when a transaction alters an object, and conflicts can arise when multiple transactions access the same object, even if the transactions are accessing distinct parts of the object. The strategies and issues of granularity are discussed in Section 5.11.

Static or Dynamic When defining a transaction scope programmatically, a *static* TM requires the accessed memory to be specified in advance. A *dynamic* TM detects accessed memory during run-time, and hence it does not impose this requirement. Consequences of choosing either one are described in Section 5.12.

Blocking or Non-Blocking Whether to use locks or not in the implementation of a TM system has a great impact on certain characteristics of the resulting system. When using locks in the implementation of a TM system it is categorised as a *blocking* system, whereas a TM system implemented using techniques such as hardware *compare-and-set* instructions can be used to create designs categorised as *non-blocking* that can allow transactions to preempt each other at any point in time. The specifics of how these two possibilities are implemented and how they affect the resulting TM system are given in Section 5.13.

Contention Management In order to avoid transactions from continually aborting each other and thereby creating a livelock, the TM system may need a *Contention Manager*. The contention manager is responsible for ensuring that each transaction is eventually allowed to commit, and can also provide fairness properties so that transactions are not kept from committing for too long by utilising a contention policy. The details of a contention manager and some examples of contention management strategies are given in Section 5.14.

5.5 Operational Structure

To use transactional memory in an application the programmer must inform the TM system that the code being executed belongs to a transaction. The basic structure of a transaction used by many TM systems [10] is shown in the following code listing:

```
// Begin transaction loop
do {
  startTx(); // Mark start of transaction
  // The body of the transaction starts here
  writeTx(&x, readTx(&x) + 1); // Increment x by 1
} while (!commitTx()); // Loop until the transaction commits
```

This example shows a transaction that increments a shared variable by 1. There are four different functions that are related to the TM system:

- `startTx` marks the beginning of each transaction run.
- `writeTx` writes to a shared memory location transactionally. Here the memory location of the variable `x` is given, meaning that `x` is the shared memory accessed.
- `readTx` reads a shared memory location transactionally.
- `commitTx` commits the transaction if possible, making the changes final in memory. If it is not possible to commit, e.g. because another transaction interfered, then the transaction is rolled back and starts again from the beginning.

Some TM systems also expose an explicit `abortTx` function that allows aborting a transaction programmatically.

The structure listed above is quite low-level for TM systems and thus requires a lot of boilerplate code from the programmer to function. A more high-level approach, used in e.g. [13], is to provide a language extension that delimits transactions by a block such as in the following code listing:

```
atomic {
  x = x + 1;
}
```

This notation is based on [50]. The transaction here is contained in an **`atomic`** block, which allows the TM system to recognise transactions. The TM system will be able to automatically start a transaction whenever this type of block starts. Each variable read and write is instrumented so that they take place transactionally. In this case, the read of `x` on the right-hand side of the assignment is handled as a transactional read, and the assignment to `x` is handled as a transactional write. When the end of the **`atomic`** block is reached, the TM system will attempt to commit the transaction. If the commit fails, the transaction will be retried from the beginning of the block.

The syntax of the **`atomic`** block is thus rewritten by the TM system to match that of the original `startTx`-`commitTx` loop, where each variable access is changed to `readTx` or `writeTx`. In this way, the TM system increases its usability by decreasing the amount of boilerplate code required. However, this approach may lead to an increased overhead of using the TM system, if there are thread-local variables (i.e. variables that are not read or written by any other threads) which get instrumented and thus get handled by the TM system unnecessarily [51].

In Java STMs, a Java method annotation can be used instead of the atomic block [4, 8]:

```
@atomic public void atomicIncrement()
{
  this.x = this.x + 1;
}
```

Using Deuce STM [8], the entire method `atomicIncrement` will become a transaction because it has the `@atomic` annotation. The STM system will rewrite the bytecode of the method to wrap it in a transactional loop and use transactional getters and setters for the referenced fields.

5.6 Conflict Detection

A conflict in a TM system is when concurrent transactions try to access the same shared memory location and at least one of them wants to write to it. In that case, the TM system must detect this conflict to be able to handle it by aborting or blocking one or more of the transactions.

The types of conflicts that transactions may encounter are [10]:

Read-after-write A transaction reads a shared memory value that was written by another concurrent transaction.

Write-after-write A transaction writes to a shared memory location that overwrites a change made by another concurrent transaction.

A TM system can either detect conflicts as they occur, i.e. when a transaction tries to write to a location that another live transaction has read from or written to already, or at a later time. The former is known as *eager conflict detection*, while the latter is known as *lazy conflict detection* when it is done as the transaction attempts to commit. The TM system can also perform conflict detection at some other point in time during the execution of a transaction, where the conflicts are detected during a validation pass that the TM system makes over the live transactions to check for conflicts. This can result in several validations of the same transaction during its execution.

One of the reasons for using eager conflict detection is that it is possible to abort a conflicting transaction early in its execution, thus preventing it from spending further time executing code that is doomed to be rolled back later. However, it requires that whenever a transaction accesses a shared memory location, all other transactions are able to see that this transaction has accessed this location if they subsequently try to access it. We say that the transaction has *acquired* the memory location. This generates additional overhead in the TM system in the case where the transaction does not have to abort.

Lazy conflict detection can be useful to increase optimistic concurrency, i.e. a transaction does not have to acquire shared memory locations until it is ready to commit. By using optimistic concurrency, the TM system can reduce the overhead of successfully committing transactions at the cost of increased wasted execution time on transactions that have to be aborted. This approach complicates the commit phase, as the TM system must then ensure that no transactions can commit changes to the same shared memory location at the same time.

5.6.1 Inevitability

Another interesting concept is *inevitability*, also known as *irrevocability*. [52, 53] Inevitability allows the TM system to guarantee that a transaction will not be forcibly aborted by the TM system. By utilising visible reads and writes for the inevitable transaction, and aborting or blocking all transactions that might conflict, it is possible for the TM system to ensure that the inevitable transaction will not have to be aborted.

Inevitability depends on conflicts being detected early to prevent non-inevitable transactions to commit changes that would cause the inevitable transaction to be aborted. Only one transaction can be inevitable at any given time, but using inevitability it is possible to perform I/O and other irreversible actions in a transaction.

5.7 Direct or Deferred Update

When implementing a TM system its designers can choose between *eager version management* and *lazy version management*. This choice refers to when data is written to the shared memory of the application during a transaction. Eager version management means that the shared memory is accessed directly by each transaction, i.e. direct updates to the memory, while lazy version management means that the TM system stores the written values in a transaction-local location which are then written to the shared memory when the transaction commits, i.e. updates are deferred until later.

The handling of when and how shared memory is updated incurs some overhead, and depending on the contention the choice may have an influence on its performance as described below.

5.7.1 Eager Version Management

When using direct updates, it is necessary to maintain an *undo-log* for each transaction. This log allows the TM system to restore the shared memory to a consistent state should the transaction abort for some reason. The undo-log must contain the original values of each part of the shared memory that was changed by the transaction. These values can then be written back to the shared memory if the transaction is not able to commit, thus undoing the effects of the transaction.

A TM system that uses eager version management will be more efficient if few transactions must abort, since the undo-logs will not have to be consulted very often to undo the effects of a transaction. When a transaction is committed the log of the transaction can just be deleted, but if it is aborted the TM system must read each entry in the log. If the log is implemented as a linked list, the TM system can undo a transaction in linear time by traversing the log backwards and undoing each write done in the transaction.

However, using direct updates may in itself cause more aborts as other transactions may be able to read inconsistent data from transactions that are in progress, where a deferred update scheme might allow reads of shared memory that another transaction has written, but not yet committed to the shared memory, thus the data read will be consistent. [10]

5.7.2 Lazy Version Management

Deferred updates to shared memory necessitates a transaction-local storage of the changes that the transaction would like to make to the shared memory; a redo-log. Whenever a transaction updates some part of the shared memory, the TM system will create a redo-log entry that records the new value of the shared memory instead of letting the transaction alter the shared memory directly.

The effect of having to perform the writes in each commit is of course an increase in the time it takes to commit a transaction; each entry in the redo-log must be inspected and the new values must be written to shared memory. Should the transaction abort, it just needs to discard its redo-log; no changes to the shared memory must be made. So choosing lazy version management improves the time it takes to abort transactions at the cost of increased time to commit them.

Another consideration regarding redo-logs is that it must support quick lookups during the execution of the transaction. Whenever the transaction wants to read a value from shared memory, the redo-log must be consulted to check if the transaction has previously written that part of the shared memory. If it has written a new value, then that value must be returned instead of the actual contents from the shared memory to ensure that the transaction sees its own writes. This makes it necessary to consider the performance of lookups in the redo-log, as a wrong choice could result in very poor performance; e.g. using an unsorted linked list could mean reading every entry in the redo-log for each read performed by the transaction.

5.8 Isolation

Isolation between transactions is required as per our definition of correctness in transactional memory (Section 5.3), but in general purpose programming not every statement is necessarily a transaction. This means that it might be possible for a programmer to access the shared memory without being in a transaction, which could cause problems if inconsistent data is accessed.

An example of where the problem might present itself is shown in Figure 5.2. In this figure, the listing on the left shows a transaction which maintains consistency by ensuring that the wrapping counter `wrapping` is reset to 0 whenever it reaches 10, so assuming no other code writes to the shared resource, the variable should always be less than 10 for any other transaction. However, if the non-transactional code is able to read the value 10 from the shared resource, then we say that the TM system has *weak isolation*, whereas a TM system that guarantees that non-transactional access will not see this inconsistent state is said to have *strong isolation*. In certain literature, this may be called weak or strong *atomicity* instead of isolation, referring to the definition of atomicity in transactions that they should take place entirely or not at all, meaning that intermediate effects of a transaction should not be visible [54].

The TM systems that we have looked at in Chapter 6 do not implement strong isolation. The usual reason for not implementing strong isolation is due to the overhead this guarantee incurs. One example of this overhead is that eager version management cannot be used easily: as discussed in Section 5.7, the shared memory would be changed during transactions making inconsistent data available to non-transactional access. However, if the TM system automatically makes every non-transactional statement in the programming language a transaction, then eager

```
atomic {
  wrapping = wrapping + 1;
  if (wrapping == 10)
  {
    // Inconsistent state where
        wrapping is not less than 10
    wrapping = 0;
  }
}
```

```
current_value = wrapping;
// Do something with current_value
    that assumes 0 <= current_value <
     10
```

Figure 5.2: Example of transactional and non-transactional access to the same shared resource.

version management would be possible again, albeit with the added transactional overhead incurred with running transactions for every statement.

A different approach to achieving strong isolation is used in the STM system Multiverse [55]. Here all data structures used transactionally must be declared for it. The objects are then only available to transactions, and non-transactional use will throw an exception.

Another possibility for transactional and non-transactional access in the same application is when non-transactional access is not used temporally when transactional access is used. By having barriers in every running thread before transactions are used or before non-transactional access is used after transactional access, it becomes possible to use the shared memory non-transactionally. When a barrier is reached, all threads must then only use transactions to access shared memory or not use transactions at all. Of course having these barriers incurs some overhead in that threads must wait for the slowest thread to reach the barrier before being able to run concurrently again. However, this can be seen as isolation by convention and requires the programmer to actively enforce the policy.

5.9 Nested Transactions

As we have already mentioned, one of the biggest selling points of transactional memory is its ability to correctly compose concurrent code. To achieve this, the programmer can call code that contains transactions from inside a transaction to make sure that any intermediate state around inner transactions remains private to the current thread. In the example with the bank account money transferring function in Section 5.1, the intermediate state between the two inner transactions that perform the withdrawal from one account and the deposit to the other must remain private, as at that point the money exists in neither account, which is an invalid state.

A straightforward way to handle nested transactions is to just make them part of the outermost transaction, meaning that when an inner transaction must abort, we simply abort the outermost transaction and later retry that entire outer transaction. This is called *flat nesting*.

Flat nesting can incur significant overhead due to repeated retries of all the code up to the inner transaction, especially if the TM system allows the programmer to explicitly abort the inner transaction while waiting for some condition to hold. An alternative to flat nesting is *closed nesting* which fixes this problem by only aborting the inner-most transaction, which can then be retried without having to redo all the code of the surrounding transaction. However, this model incurs the transactional overhead of having to track the inner transactions separately, which means lower performance compared to flat nesting when aborts rarely occur in the inner transactions.

Another possibility is *parallel nesting* which allows for the possibility of inner transactions to be run concurrently. This approach, however, is out of the scope of this report.

5.10 Hardware or Software

A TM system can, as mentioned, be implemented in software, hardware, or a combination of the two as a hybrid. Each have their own benefits and drawbacks that we will discuss below.

STM systems allow a level of flexibility that cannot be provided by hardware implementations. Once hardware has been finalised and put in production, it is not usually possible to alter it. Implementing TM in software can provide this flexibility as changes to the STM library used by an application can just be distributed as an update to the application. A hybrid of software and hardware implementation, HyTM, would allow the software parts of the TM system to be replaced like this.

STM also allows a very high-level access to the data structures of the application, in that it can be written specifically for a specific programming language and thus can interact with the application on a different level. For example, STM systems can track memory at object-granularity as discussed in Section 5.11. HTM is more limited here, as hardware is usually designed to support any number of current and future programming languages [34], which may even support completely different programming language paradigms, and can thus not make as many assumptions about the way memory will be managed by the application. This usually results in a more low-level approach to memory at the hardware address level instead of e.g. objects in a object-oriented programming language.

A drawback to STM is that it must use processing power from the system to work. It will use computation time to manage the shared memory that thus cannot be used to perform the computations of the application. This overhead can be quite significant depending on the implementation of the STM. Every time the programmer uses transactions, the STM library must be invoked to provide the guarantees necessary to correctly handle the shared memory. The system can also provide overhead in other parts of the application if the TM system is to provide guarantees such as strong isolation as discussed in Section 5.8. HTM can work around this by performing the HTM-related work in the same clock-cycles that run the application code [6], however there is still an overhead associated with the way TM functions, in that the transactions may still need to be retried, and thus increase the computational cost of the application.

Most modern general-purpose processors have built-in *caches* that act as a faster alternative than reading and writing directly to the main memory. When memory is read, the processor fetches the part of memory along with surrounding memory into a *cache line* in the cache. This cache line can then be read from and written to many times during the execution of the thread, and then at some point written back to main memory or discarded if there were no changes. In multi-processor systems, the processors must coordinate all their caches to make sure that there are no conflicts. This is done by using a *cache coherence protocol*. [34]

It is clear that the job of a cache coherence protocol is very similar to that of a TM system, so many HTM implementations extend the cache coherence protocols to handle transactions, but this means that the size of the cache becomes a hard limit on the size of transactions that they support. [7] If this is how changes to shared memory are tracked, the HTM will only support a limited amount of reads or writes to shared memory. If the limit is exceeded the HTM is not able to track further changes and will thus not be able to detect conflicts. An effort towards solving this is a hybrid implementation that uses HTM for transactions until they reach the limits of the hardware, after which they are moved to an STM implementation that is slower, but allows for much larger transactions. [12]

5.11 Granularity

The granularity that a TM system uses to track changes decides how precisely a TM system can detect conflicts. TM systems can track changes to the shared memory at several levels depending on the programming language they are used in and how they are tied into the application. The worst level of granularity is using a single lock to protect the entire shared memory. In this way, there can no concurrency between writing transactions, even those that write to entirely distinct parts of the shared memory.

At the other end of the spectrum, there is machine word-based granularity, where each addressable machine word is tracked individually, so conflicts are detected only when transactions actually access exactly the same address of the shared memory. Examples of word-based granularity STMs are [9, 56, 57]. Common for all these STM systems are that they are built for languages with pointers, so they can, as libraries, access the addresses of shared memory locations directly to use as, e.g., keys for metadata arrays. This also allows tracking individual parts of an array so that conflicts do not occur if two transactions access distinct parts of it.

HTMs are usually cache-line based as mentioned in Section 5.10. This level of tracking is closely related to word-based tracking, as cache-lines specifically contain a fixed number of machine words. However, the coarser granularity may cause *false conflicts* which means that the TM system has detected a conflict that is not strictly real. Consider two variables each occupying a machine word in the shared memory. If they are both loaded into the same cache-line, then a change to one of the variables will be seen as a change to the other as well when using cache-line granularity, and if the TM system detects a conflict with another transaction using the other variable, then the TM system has encountered a false conflict. [11]

By storing additional metadata for each cache-line it is possible to provide finer grained access than cache-lines, which can then be consulted in case a cache-line conflict is detected [11]. This form of two-stage granularity allows the performance of the conflict detection to be increased in the case where there are no cache-line conflicts, but it is clear that in the worst-case the performance will be worse than the finer of the two, as it will have to perform both the coarse- and fine-grained conflict detection for each conflict.

A coarser granularity than cache-line granularity is *object* and *field* granularity. This type of granularity is often used in object oriented programming languages, where fields of objects can be the lowest level of shared memory access allowed by the programming language. In a library written in Java, for example, it is not possible to access the memory address of a variable, but it is possible to track individual objects, and thus also their individual fields. Examples of STMs that have object or field granularity are [4, 8, 58, 59].

The best choice for granularity in an STM must thus depend on the specific application. If an application uses very large objects for shared state, where transactions only work on small, often distinct parts of an object, a field-based granularity would be better than object granularity, as this would reduce the number of false conflicts and thereby wasted work due to aborted transactions. However, the cost of finer granularity is increased size of the metadata stored by the STM system.

5.12 Static or Dynamic

Transactions often wrap operations on memory. Which memory will be accessed is either specified statically, that is; it is clear from reading the source code what memory is accessed, or dynamically, which means the memory accessed will be determined during run-time.

The distinction can also be applied to the total number of trasactions and transactional objects in the system. Static in this context means every transaction and transactional object is defined in advance, where dynamic allows for new transactions and transactional objects to be initialised at run-time.

To sum it up, the differences between choosing one scheme over another are [60]:

Static TMs require transactions and transactional objects to be defined statically, as with the memory accessed within transactions.

Dynamic TMs can allocate memory for transactional objects during run-time, transactions can access memory based on live decisions, and also allocate and start new transactions.

```
l = (1, 2, 3, 4)

atomic:
    l.add( wrap(5) )

    for all i in l:
        atomic:
            computation(i)
```

Listing 5.1: Example code showing an atomic code block which is impossible to implement using a static TM, but is indeed possible in a dynamic TM.

Choosing a static TM requires the programmer to be explicit about memory usage. This may seem mundane, but as a consequence new transactional objects cannot be created during run-time in, and neither can actual transactions. Listing 5.1 gives an example of an atomic block which would not be possible to implement using a static TM. It assumes granularity at element-level in the list *l*, and thus every element is wrapped in a transactional object. The mutability of *l* is used to atomically add 5 to the list, and for every element in *l*, it spawns a new (nested) transaction and performs a computation on the element. Although the example code does not serve as practical on its own, it demonstrates a scenario accessing data and manipulating it, and how static TMs hinder one in doing so.

Dynamic TMs allow for code such as this, but static TMs can put a bound on memory usage and will require the programmer to consider which memory is accessed and in which transactions. Such restrictions can be useful in certain application domains where hardware resources are limited, such as embedded systems.

5.13 Blocking or Non-Blocking

STMs are built using the traditional tools available for concurrency, e.g. locks or atomic compare-and-set instructions. Using these low-level tools, an STM system can be designed to provide its services, but the choice of which tools to use allow for certain different types of STMs to be developed, depending on which type of algorithms are used. [8, 4, 57]

5.13.1 Blocking TM

In a *blocking* implementation of a TM system, locks are used to ensure exclusive access to shared memory. This means that each transaction can hold a number of locks and when it is not possible to obtain a lock, the transaction must either wait for the lock to become available or abort itself. Depending on whether the system uses eager or lazy version management, the locks may not be acquired before the transaction tries to commit, at which point it will attempt to acquire all locks to ensure exclusive access to the parts of the shared memory to perform its updates.

Blocking STMs are capable of giving certain guarantees, such as freedom from deadlock and/or livelock. Deadlock-freedom can be guaranteed by ensuring that locks are taken in a specific order, e.g. by using lazy version management to gather all the writes for a transaction in the redo-log, and then taking all the locks on the parts of the shared memory in a system-wide determined order. Another approach is to try to acquire locks eagerly, but whenever a lock cannot be obtained, either wait for a specific amount of time for the lock to become released, or immediately abort the transaction. Livelock-freedom is usually ensured by the contention manager.

One issue with blocking implementations is that all other transactions depending on those locks, are blocked until the thread can release them. If a thread does not release its locks, then it can permanently block all other threads that require that lock from progressing. For example, this could happen in a system with eager version management, where a transaction acquires the lock for a variable and then loops indefinitely. Unless the loop contains a call to the TM system, then the TM system will not be able to abort the looping transaction.

5.13.2 Non-Blocking TM

A solution to this issue is to use a *non-blocking* TM system. Non-blocking TM systems do not use locks to ensure mutual exclusion, and it is thus not possible for a transaction to hold a lock indefinitely. In other words, *"a nonblocking algorithm guarantees that if one thread is pre-empted mid-way through an operation/transaction, then it cannot prevent other threads from being able to make progress"* [11]. A non-blocking TM system can be one of three different types: *wait-free*, *lock-free*, or *obstruction-free*.

Wait-free systems guarantee that all threads will eventually be allowed to make progress, i.e. the system is free of both livelocks and starvation. Lock-freedom is a simpler guarantee that the system as a whole will eventually be allowed to make progress, i.e. the system is free of livelock, but not guaranteed to be starvation-free. Obstruction-free systems give even weaker guarantees than lock-freedom: only if no other threads preempt the transaction, will it be guaranteed progress. This is the most basic guarantee given by any non-blocking algorithm. Because nobody has so far been able to determine a way to design a wait-free or lock-free TM system that performs well enough to be of practical use, the focus of STM research has been in the development of obstruction-free systems [7].

In obstruction-free TM systems, whenever a transaction requires exclusive access to a part of the shared memory, it can try to take ownership of this shared memory. If the shared memory is already owned by another transaction, the TM system will decide which of the transactions will be able to continue. One benefit of the obstruction-free design is that individual threads do not hold locks that they must release before other threads can continue, and thus if a thread crashes unexpectedly a lock-based design could mean that the locks would never be released and other threads would never be able to acquire those locks. An obstruction-free design would allow other threads to revoke ownership of the shared memory held by a crashed thread, thus allowing the application as a whole to progress.

5.14 Contention Management

When conflicts occur in a TM system, the TM system can use a *contention manager* to determine which of the two competing transactions should be aborted or paused and which should be allowed to continue. Whenever a conflict is detected, the contention manager is invoked to resolve the conflict. The contention manager can implement one of a wide variety of strategies to determine which transaction should be allowed to continue. Without contention management transactions would be able to continuously abort each other, which in extreme cases can lead to a livelock.

A contention manager can prevent livelocks by ensuring that at least one of the running transactions will eventually be allowed to commit. It can also provide certain fairness properties to prevent starvation, for example, by letting transactions that have been aborted many times commit, thus preventing scenarios such as where short-running transactions will continuously be able to abort a long-running transaction.

5.14.1 Strategies

To decide how a conflict should be resolved, i.e. which transaction to abort when a conflict between two transactions is detected, the contention manager utilises a contention management strategy. There are several different strategies in existence, which have different characteristics:

Passive A transaction which detects it conflicts with another will abort *itself*.

Aggressive A transaction which detects it conflicts with another will abort the *other*.

Polite If a transaction detects a conflict, it will wait for an increasing amount of time before trying again, specifically, by using bounded exponential back-off. When the bound is reached, the until-now-waiting transaction will abort its competitor.

Karma Whenever a transaction accesses a transactional object, it accumulates *karma*, or priority. The intuition is the more objects a transaction has accessed, the more work it has performed and thus its karma increases. If a conflict is detected, the transaction with the least amount of karma is aborted. To prevent starvation of aborted transactions, karma is not reset when aborted, so it will continue to increase until the transaction is allowed to commit.

Eruption Like Karma, transactions accumulate priority by the number of objects accessed. Upon detecting a conflict, the priorities of the two competing transactions are compared: if the conflict-detecting transaction has the higher priority, the competitor is aborted; if the competing transaction has the higher priority, it will have its priority increased by that of the conflict-detecting transaction, and then put the conflict-detecting transaction to sleep. Ideally, the competing transaction is then less likely to be aborted in case of a subsequent conflict with another transaction, thus increasing the chance the now-sleeping transaction is allowed to commit during the next retry.

Priority Transactions are assigned priorities based on their starting time—the older the transaction, the higher priority. If a conflict is detected, the transaction with the highest priority (oldest) is allowed to commit and the other one is aborted.

A Comprehensive Strategy In the article A Comprehensive Strategy for Contention Management in Software Transactional Memory [61], a strategy is developed that the authors claim works well both under high contention and low contention. It is developed specifically for RSTM [9], but it is unclear how their benchmarks were performed, and thus the validity of their claim is also unknown.

It is worth noting that while contention managers *can* prevent livelocks from occurring, certain strategies will not provably guarantee this. Of the mentioned strategies, Karma, Priority, and Polite do not fall into the category of strategies which provably prevent livelocks. They are, however, regarded as livelock-free in practice. [61]

Each strategy involves different amounts of bookkeeping, and each contributes overhead in doing so. In applications where transactions suffer little or no contention, using a contention manager is not desirable since nothing is gained and overhead is increased.

Chapter 6

Transactional Memory Systems

In this chapter, we look at two existing transactional memory systems available for general purpose computing. As the focus for this project is Java, we only look at STM systems for Java. We also present our own implementations of STM for Java. The purpose of this chapter is to show that we can use the design and implementation choices described in Chapter 5 to analyse TM systems and describe the characteristics of systems that we have studied and of systems we have built ourselves.

We have also looked at additional TM and TM-like systems in relation to real-time systems, but these will be discussed in Section 7.1.

The systems we will look at here are DSTM2 in Section 6.1, Deuce STM in Section 6.2, and our own implementations of STM algorithms in Section 6.3.

6.1 DSTM2

DSTM2 [4] is the successor to the first dynamic STM system by Herlihy *et al.*: DSTM [60]. It is built as a research platform that allows STM developers to design and test in practice different approaches to a TM system, e.g. contention management strategies. It is designed with the idea that *"no single run-time mechanism is ideal for all kinds of applications"* [4]. DSTM2 improves on the usability of the system over DSTM by using bytecode rewriting of methods annotated with `@atomic` as described in Section 5.5.

Because DSTM2 is implemented as a library and uses bytecode rewriting, it can be used with a standard Java compiler and run on a standard JVM.

6.1.1 Factories

A factory in DSTM2 is used to create the objects that will be used in the transactions of the application. The factory is initialised with an interface that the objects that will be used transactionally. The interface can contain method signatures of the following types:

```
    T getField();
    void setField(T value);
```

Where `Field` is the name of the field chosen by the programmer using the TM system, and `T` is the type of the field. Defining this kind of interface allows the factory to create standard getter and setter methods with these names at runtime that allow using the object. In a simple linked list of integers, for example, the following interface might be used to implement the nodes of the list [4]:

```
@atomic public interface INode {
  int getValue();
  void setValue(int value);
  INode getNext();
  void setNext(INode value);
}
```

The interface is declared atomic using the `@atomic` annotation, which is required of all objects that might be shared between concurrent transactions.

When the factories are initialised with the interfaces, they can then be used repeatedly to create new objects in transactions for use in the application.

Bytecode rewriting is performed when instrumenting the critical sections in a Java application. When the bytecode is rewritten, the different methods of the factory and the DSTM2 `Thread` package that contains the factory used to create transactions are called at various points to allow the STM to handle the transactions, e.g. record the access to and current value of a field in a transaction.

Many details of how the STM operates are defined by which factory is used by the application, and it is thus difficult to classify DSTM2 with regards to the design choices we listed in Section 5.4.

Operational Structure DSTM2 uses the `@atomic` annotation together with factory objects to discern transactions and transactional objects.

Conflict Detection The conflict detection strategy used depends on the factory chosen. DSTM2 comes with factories that have early conflict detection, i.e. eagerly acquiring objects both when reading and writing, and a more optimistic concurrency model based on invisible reads, where the read-set is validated at commit. It is also possible to implement a factory that performs lazy acquiring of objects for writing as well.

Direct or Deferred Updates Whether the system performs direct or deferred updates to memory depends on the factory.

Isolation Isolation guarantees depend on the specialised getters and setters provided by the factory.

Nested Transactions Nested transactions are not supported [4].

Granularity Object-level granularity is supported to detect conflicts.

Static or Dynamic DSTM2 is a dynamic STM.

Blocking or Non-Blocking One factory provided in DSTM2 uses the same obstruction-free algorithm that was introduced with the original DSTM. Another factory called `shadow` uses locks to protect critical sections of the TM system, and avoids the use of indirection used in the obstruction-free factory. It is thus possible to use DSTM2 as both a blocking and a non-blocking TM system, and the supplied factories can determine the specific algorithm used.

Contention Management Contention managers are pluggable entities that can be chosen at runtime.

6.2 Deuce STM

Deuce [8] is a Java STM framework that, like DSTM2, is intended as a research platform for designing TM algorithms. It also uses bytecode rewriting to achieve the same operational structure as DSTM2, i.e. the `@atomic` annotation. The main purpose for Deuce STM is to provide a noninvasive STM for Java that allows more practical use in that preexisting Java libraries can be used from transactions without having to modify them. It also provides a better performing base for Java dynamic STM research than DSTM2 [8].

The notion that Deuce is noninvasive compared to DSTM2 means that it does not have the same factory system that requires DSTM2 users to build their runtime objects using factories to be able to use them transactionally. Deuce automatically instruments all objects that can be accessed from a transaction with special getters and setters in a synthesised parallel method that is called instead of the original method when called from a transaction [8]:

```
...
public void setForward(int level, Node next) {
  forward[level] = next;
}

public void setForward(int level, Node next, Context c) {
  Node[] f = forward__Getter$(c) {
  forward__Setter$(f, level, c)
  }
...
```

The top method in this code example is the original method as it was written in the program, while the bottom method has been overloaded to take a transactional `Context` object that stores the context of the current transaction. It uses the special getters and setters to access the `forward` array. The overloaded method is created during bytecode rewriting and is only used by rewritten bytecode.

Deuce can run as a Java *agent* allowing it to hook into the class initialisation routines of a running JVM. This allows Deuce to instrument the classes once as they are first initialised by the JVM, and objects can then be created from this class. It allows plugging in different algorithms to handle the various aspects of the STM. It is therefore, like DSTM2 (Section 6.1), difficult to categorise:

Operational Structure The `@atomic` annotation is used to delimit transactions. Shared memory accesses are instrumented automatically.

Conflict Detection Depending on the algorithm chosen conflict detection can be done both eagerly and lazily. Deuce comes with an implementation of the TL2 algorithm [57] and the LSA algorithm [62]. The former uses lazy validation by acquiring locks at commit time and aborting if all locks cannot be acquired, and the latter uses a two-phase locking system where locks are acquired eagerly.

Direct or Deferred Updates The two approaches to conflict detection also use both strategies for updating shared memory. The TL2 algorithm uses deferred updates and writes only after it has acquired all the locks to the shared memory locations it wants to modify, while the LSA algorithm writes data to shared memory as soon as it has acquired the lock eagerly during the transaction and then rolling back changes if it must abort.

Isolation Deuce STM has weak isolation because the overhead of ensuring strong isolation was deemed too high [8].

Nested Transactions Transaction nesting is supported, but it is not clear from the article [8] how it is implemented. We suspect flat nesting is used based on the source code.

Granularity Field-level granularity was chosen to reduce the number of false conflicts and increase concurrency.

Static or Dynamic Deuce STM is a dynamic STM.

Blocking or Non-Blocking Both the TL2 and LSA algorithms described in the paper use locks to protect shared memory locations, so they imply a resulting TM system that is blocking. However, as far as we can tell from the source code, it should be possible to design a non-blocking implementation as well.

Contention Management The contention management introduced in the paper is based on a simple exponential back-off strategy that should break practically any livelock, however, the system allows plugging in more advanced contention management strategies as required.

6.3 Own Experiments

To better understand the basics of STM design, we also created a few simple STM libraries for Java, which are based on algorithms found in [10]. We implemented both a lock-based blocking implementation, a compare-and-set non-blocking implementation, and a reference single global lock implementation that just serialises all transactions.

The lock-based implementation uses a two-phase locking approach, where locks are acquired eagerly whenever a variable is written. The system uses invisible reads with validation of the read-set whenever a new variable is read and during commit. It is a more or less direct implementation of the algorithm found in [10] adapted slightly for Java. However, since the obstruction-free version is the one we spent the most time on, it will be the focus of this section.

All versions must be used directly with calls to the library methods. This means that using the STM requires explicit writing of the boilerplate code of the transactional loop and wrapping calls to shared memory in read and write methods. An example of the use of the obstruction-free implementation is shown here:

```java
do
{
  try
  {
    c.txNew();

    // This is the body of the transaction
    c.txWrite(counter, (c.txRead(counter) + 1));

    c.txCommit();
  }
  catch (AbortException e)
  {
    if (c.txRethrowAbort())
    {
      throw e;
    }
  }
} while (!c.txIsCommitted());
```

This code shows how we have accommodated for several different features of the TM system. The code uses a context variable `c` that is the `Thread` object of the running thread extended with methods for transactions.

The `txNew` method is called to mark the beginning of a transaction run. This method initialises the transaction, so it must be called before any of the other transactional calls. After this initial call, the transactional body begins. In this example, we have an integer counter that is incremented atomically by reading its current value, adding 1, and writing the result back to the counter.

With the `txCommit` call, the transaction will attempt to commit its changes. If any of the `txRead`, `txWrite`, or `txCommit` calls fail because the transaction has been aborted, they will throw an `AbortException`.

In the `AbortException` handler, the method `txRethrowAbort` is called to determine whether the exception should be rethrown. This is part of our support of flat nesting: when `txNew` is called, the thread will check if it is already running a transaction. If it is, a nesting-depth counter is incremented instead of allocating a new transaction object. Whenever an inner transaction attempts to commit, the counter is decremented until it reaches zero, at which point the actual commit functionality of the transactional object is used to attempt to commit the changes. The `AbortException` handler also decrements the counter until it reaches zero and rethrows the exception to the next handler to allow only the outermost while loop to actually perform a retry.

The while loop allows program execution to continue from the beginning of the transaction with a new transactional object whenever the transaction fails.

In our code, we use `AtomicReference` objects to hold a references to our STM metadata which also contains the actual shared memory data. `AtomicReference` is in the `java.util.concurrent.atomic` package, which can be used to create non-blocking data structures[1] Objects of `AtomicReference` contain a method that allows to use hardware compare-and-set instructions on supported platforms by the `compareAndSet` method. The following code shows the usage of our metadata in the code for writing to shared memory:

```
public <T> void write(AtomicReference<TValue<T>> value, T newValue) throws
    AbortException
{
  TValue<T> v = value.get();
  TValue<T> result = new TValue<T>(this, v.oldValue, newValue);

  if (v.owner != this && v.owner != null)
  {
    v.owner.state.compareAndSet(State.LIVE, State.ABORTED);
    if (v.owner.state.get() == State.COMMITTED)
    {
      result.oldValue = v.newValue;
    }
  }

  if (state.get() != State.LIVE || !value.compareAndSet(v, result))
  {
    // Transaction has been aborted, or someone else acquired the variable
    abort();
  }
}
```

The metadata is stored in a `TValue` object, which has fields for: the transaction that has currently acquired the object, the old value of the shared data, and the new value of the shared

[1]The implementation of `java.util.concurrent.atomic` may use locks internally if the platform does not support the necessary instructions for an efficient non-blocking implementation. [63]

data. When a transaction wants to write to shared data it will first create a new `TValue` object with itself as owner. The old value is copied from the old `TValue` object, and the new value is set to the value that the transaction would like to write.

Having the new `TValue` object, the transaction will attempt to acquire the memory location. If the current owner of the `TValue` object is this transaction, or there is no owner, then this is not necessary, but in the other case the transaction that is currently listed as the owner will have to be either aborted, if it is currently live, or the `newValue` from the old `TValue` object will have to be copied over if it was committed. In this way, the current transaction makes sure that the metadata is consistent with the timeline of the previous transactions.

In the last part, the transaction checks that it has not been aborted by some other transaction, and then tries to atomically swap in its `TValue` object making its claim to the shared memory with its tentative write along with the old value of the shared memory.

Reads in the system are handled similarly to writes. The metadata does not distinguish whether the owner has acquired the resource for reading or writing, so two readers will conflict just as two writers do. This allows us simpler metadata and simpler code for conflict detection, but increases the contention of the system significantly. We did not do any significant benchmarking of the implementation, because of its obvious inefficiency. Due to this we also did not experiment with any contention management strategies.

To conclude, we will now summarise the main points of our obstruction-free STM:

Operational Structure Calls must be made directly to the TM library to create transactions and access memory transactionally.

Conflict Detection The system uses eager conflict detection with visible reads and writes, and readers can produce false conflicts because they can conflict with each other.

Direct or Deferred Updates Because of the datastructure used to hold the shared data, one could consider the updates both deferred and direct: reading from `oldValue` will always provide consistent results that were committed at some point prior to the read, while the updated values of aborted and live transactions are available in `newValue`, which means that it is possible to read inconsistent values there.

Isolation The system provides no guarantees towards strong isolation.

Nested Transactions Nested transactions are provided through flat nesting.

Granularity Granularity is object-based in that the entire objects are acquired when reading or writing them.

Dynamic or Static The system is dynamic.

Blocking or Non-Blocking Depending on platform support, the system is obstruction-free.

Contention Management Contention is handled aggressively by immediately aborting the other transaction when detecting a conflict. No contention manager is implemented.

Chapter 7

Real-Time Transactional Memory

As we have seen in Chapter 5, the benefits of TM are many when programming concurrent systems. In addition to the general purpose programming benefits, TM can also improve areas specifically interesting to embedded and/or real-time systems: TM can alleviate priority inversion [64], reduce blocking time and thus increase responsiveness of high priority tasks [14], and prevent crashed threads from blocking the rest of the system (Section 5.13).

These reasons make TM an appealing alternative to locks in RTS. However, as we have seen in Chapter 5, TM systems do not generally provide any guarantees as to when a transaction will be able to successfully commit. This is unacceptable in RTS where worst-case performance is critical to the correctness of the system. In this chapter, we discuss existing work and our own ideas on how to bound the response times of tasks containing transactions.

We begin in Section 7.1 by looking at the work that has already been done in the field to get an overview of what has been accomplished and what still needs accomplishing before a TM system can be considered real-time compatible. Then we give our own thoughts on how to use the existing work to improve on the state-of-the-art in Section 7.2. Finally, in Section 7.3, we analyse our idea to see that if our claims hold, then our approach can be used as a foundation for a real-time STM.

7.1 Existing Work

In this section we look at the existing work that has been done to be able to use transactional memory in the development of real-time systems. We have already discussed how TMs use a loop to retry transactions until they successfully commit. The existing work in the field has focused on bounding this loop through various means.

We start by looking at an article that we have concluded to show the closest practical example of STM for RTS, before looking at another article that presents a real-time HTM. We also look briefly at a third article that presents an STM for soft real-time applications, as it holds a couple of ideas that we have found usable in our own approach which we present in Section 7.2. In the remainder of the section, we give a general overview of other related work that we use for our approach.

7.1.1 Preemptible Atomic Regions

The article on Preemptible Atomic Regions (PARs) [14] describes an atomic construct for programming concurrent applications for real-time systems. The idea presented is very close to an STM, although concurrency is sacrificed in order to provide the guarantees required for hard real-time systems. The article describes an implementation for Java that offers the same easy approach to critical sections as the previously mentioned STMs (Chapter 6), i.e. the methods that must be executed atomically are annotated by the programmer and then the system takes care of handling concurrency correctly (Section 5.3).

Implementation

The system is implemented using some of the same ideas found in Deuce STM discussed in Section 6.2: by using bytecode rewriting, the authors of the article are able to, as a library, instrument the `@PAR` annotated methods to use specialised getters and setters on objects and encapsulate them in a loop to allow retries.

The system uses direct updates on changes to the shared memory, and stores original values in an undo-log that is used if an atomic region is preempted by another thread. This is where PARs differ from STMs: there is no conflict detection, conflicts are assumed as soon as a higher priority thread is released. This also means that PARs provide strong isolation in that any writes are undone before another thread can read them. Nesting is handled flatly.

PARs are tied into the scheduler of the machine to allow it to abort any atomic region running in the thread being preempted. Because only one PAR may be running at a time, multi-processors are not supported. Only one PAR may be aborted per preemption, which is how the number of retries is bounded.

Analysis

The authors of the article build on the original response time formula to get the following equation [14]:

$$R_i = C_i + \max_{j \in lp(i)} U_j + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{p_j} \right\rceil (C_j + U_i + W_i) \tag{7.1}$$

Where the first term is the computation time C (WCET) of task i, the second term is the blocking time from lower priority tasks, and the last term is the interference from higher priority tasks.

Since there are no locks to release, the maximum time a task can be blocked is the time it takes to undo the changes of the currently running lower priority task. In the worst case, the task with the highest abort time is preempted, which is why the second term uses the max function to find the highest undo time U of the set of tasks that have a lower priority than i.

The interference that can be experienced is the sum of the cost having to run every higher priority task (C_j), of having to undo the current transaction (U_i) each time another task gets to run, and then having to redo the work in the current transaction after the higher priority task is done (W_i). This is further increased in case the higher priority task may be released several times during the execution of the current task, thus the factor of $\lceil R_i/p_j \rceil$ is introduced, where p_j is the period of task j.

The formula can be calculated as described in Section 3.3.

	High	Medium	Low
C	2300	2350	2450
P	13000	14000	15000
$R(PAR)$	2316	7023	12048
$R(PIP)$	4750	7100	7100
$R(PCE)$	4750	7100	7100

Table 7.1: Response time analysis (in μs) of a microbenchmark with three tasks using PAR, PIP and PCE. The maximium measured abort time, U, is $16\mu s$. P refers to the period of the three tasks. [14]

Performance

In the article, Equation 7.1 is used to compare the response time of a system using PAR with the same system using locks and a priority inheritance (PIP) or ceiling (PCE) protocol. The result is shown in Table 7.1

From these results, the authors of the article note that the response time of higher priority tasks is much lower at the expense of increased response time of the low priority task. This conclusion is again drawn from other experiments that are conducted in the article using larger applications, including PRISMj, the control software for the Boeing ScanEagle UAV, that was rewritten to use PAR.

The conclusion of the article is that PARs allow stronger correctness guarantees than lock-based synchronisation, while being practical for actual real-time applications using Java. The results show that in cases where there are few writes in critical sections, which they believe is the common case for real-time systems, PARs perform similarly or better than a lock-based implementation.

7.1.2 RTTM

The RTTM: Real-Time Transactional Memory paper [15] presents an HTM implemented as a modified JOP that can be used in hard real-time applications.

Implementation

RTTM allows bounding on the number of retries by having a contention manager that makes sure to abort all but one of the running transactions, so that the number of active transactions in a critical instant where all transactions conflict and run concurrently will be decreased for every round of retries. Figure 7.1 shows this scenario for a set of six threads τ_1 to τ_6 concurrently running six transactions.

The system uses deferred updates with a global commit lock to ensure that when a transaction acquires the lock to commit, it will always succeed unless another transaction has already committed that round. Whenever a transaction commits all other transactions will be blocked from committing and then when the commit lock is released they have either been invalidated, and must abort when they detect the conflict, or remain unaffected. This commit lock can also be acquired before committing to ensure inevitability should the transaction have to do I/O, or if the TM metadata of the transaction is full so that it must perform direct updates to continue.

The operational structure of the system is very simple: at the beginning of the transaction, a call to `RTTM.start()` is made, and at the end, a call to `RTTM.end()` is made. The hardware implementation will then ensure that transactions can be retried without having an explicitly

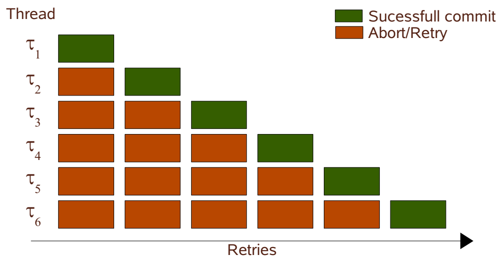

Figure 7.1: One transaction is committed each round, reducing the number of active transactions. [15]

defined loop in the source code. Any memory accesses are automatically considered to be transactional, and memory changes are tracked at the word-level to ensure no false conflicts can occur.

Conflicts are detected by evaluating the read and write sets after the commit lock has been acquired. If another transaction has already committed that round and invalidated the redo-log, it will discard the log and release the commit lock. This means that it uses lazy conflict detection.

Due to the global commit lock, the system is a blocking TM.

When transactions conflict, the contention manager is queried to decide which transactions are aborted and which one is allowed to commit. The policy applied is straight forward: the first transaction to acquire the commit lock is allowed to commit, and subsequently abort and restart all live conflicting transactions. This gives the behavior described earlier and illustrated in Figure 7.1.

Analysis

The analysis of the system is the main contribution of the article. The analysis determines the upper bound on the number of retries a transaction can experience in the worst case. Because at least one transaction will be allowed to commit of a set of concurrently running transactions, the number of concurrent transactions will decrease for each retry as long as they all commit before the next release of one of the tasks.

The maximum number of retries r is thus:

$$r = n - 1 \tag{7.2}$$

Where n is the number of threads each with one transaction.

To further tighten this bound, the authors of RTTM describe a method to statically analyse the program to determine which transactions can actually conflict. This allows the number of retries for a transaction to be lowered to the number of threads each with one transaction that may possibly conflict with that transaction (excluding itself).

7.1.3 Deadline-Aware Scheduling

The article Deadline-Aware Scheduling for Software Transactional Memory [65] presents an STM that uses deadline annotations on individual transactions to support soft real-time applications.

Implementation

To accomplish their goal of creating an STM that can be used in soft real-time applications, the authors of the article have modified the STM system TinySTM [56] and the scheduler of the Linux operating system.

TinySTM was modified to include an *adaptive mode switching* module which allows the STM to dynamically change the level of optimism used in transactions. The highest level of optimism in the system is when invisible reads are used to increase concurrency. However, if this level of optimism makes it hard for a transaction to commit because of contention from other transactions, the system can switch to use visible reads for that transaction to increase its chance of success as the contention manager will be able to abort conflicting transactions before they invalidate its read-set. However, it is still possible for transactions to conflict if one manages to commit a write to a variable before it is added to the read- or write-set of the other transaction but after the other transaction has started. When a transaction still fails to commit and its deadline is nearing, the final and most pessimistic level of optimism can be used: inevitability, which further decreases the concurrency of the system, but guarantees that the transaction will be allowed to commit. Only one transaction can be inevitable at a time, so transactions may have to wait in line to become inevitable.

To determine when the STM must switch modes, a *transaction length measurement* module is used to determine the total running-time of transactions. The measurements provided by this module allows the system to switch to less optimistic modes when the deadline is nearing by letting the system see that the transaction cannot be retried many times without missing the deadline. Transaction times are measured at runtime and a pool of individual transaction measurements are used to predict the execution times of future transaction runs.

The changes to the operating system scheduler allows the system to provide better estimates of transaction running times, as the scheduler will attempt to avoid preempting running transactions. Because the system supports multi-processors, conflicts may still occur even if a transaction is not preempted in the case that another transaction runs in parallel on a different processor core. Only estimates of successfully committed transaction runs that were not preempted are used to predict future transaction execution times.

Another benefit of preventing transactions from being preempted is that transactions cannot be allowed to commit writes if an inevitable transaction is running. Preempting an inevitable transaction would mean that no writing transactions would be able to make progress for an extended period of time while waiting for the inevitable transaction to finish. By keeping the scheduler from preempting inevitable transactions, they will commit as soon as possible and allow other concurrent writing transactions to commit again.

Analysis

Because the system is not able to provide any guarantees about the accuracy of the execution time predictions of transactions and how long a transaction must wait to attain inevitability, it is not possible to give any sound analysis of the schedulability of an application using the STM. However, as the STM was designed for soft real-time applications, this is acceptable.

In a hard real-time context, this is not enough, but some results are useful in isolation for hard real-time applications:

- If a transaction is inevitable, it will be able to commit in the span of its execution time provided it is not preempted.
- Involving the scheduler in the TM system allows preventing preemption of transactions.

The system presented allows most transactions to run with high concurrency, i.e. optimistically, which ensures high throughput, and then handles starvation by letting the programmer specify deadlines for individual transactions that the system will try to meet. This differs from the approach we have discussed in Section 5.14, where the contention manager is responsible for preventing starvation.

7.1.4 Non-Blocking vs Blocking

Blocking TMs can have properties such as wait-free operations and strong progressiveness. However, as [10] notes, blocking TMs do not handle crashes very well. The wait-free property only covers individual operations and not the entire system. Given two tasks t_1 and t_2, t_1 then successfully acquires lock on a resource A. Then t_1 is suspended, and t_2 attempts to acquire A. The blocking property results in a potentially unwanted situation where t_2 is blocked until t_1 releases its lock. This phenomenon becomes critical in a real-time setting, if t_2 has higher priority than t_1—a lower-priority task blocking a higher-priority task (priority inversion) is not desired behavior of an RTS. This discussion is not specific to non-blocking or blocking TMs only, but it is nonetheless important to consider.

Choosing a non-blocking approach will alleviate these issues. Either the shared data structures can be non-blocking or the TM itself can. Effectively, it means that tasks or threads concurrently accessing the shared data structure will not block each other. That is, if t_1 crashes given the same context as above, t_2 is not blocked.

The idea to use non-blocking data structures in real-time systems was first noted by P. Sorensen and V. Hemacher in [66, 67]. Their work revolved around using wait-free read/write buffers for real-time communication, but the implementation relied on blocking functionality within the operating system, and thus was not a complete solution to the issues initially described. To the best of our knowledge, the first attempt to use non-blocking data structures in hard real-time systems was done by Anderson, J. H. *et al.* [68], and they found that the general case was that lock-free data structures imposed less overhead than lock-based synchronization. However, lock-free data structures still suffer from the unpredictable aspect, since they do not provide bounded guarantees of progression.

Early work on moving non-blocking characteristics from data structures to transactions in real-time systems was initially thought of by Anderson, J. H. *et al.* (1997) [69] which requires hardware support for transactions which makes it a hybrid TM.

7.1.5 Worst-case Execution Time (WCET)

Determining the WCET for tasks in a real-time TM environment is non-trivial given the nature of transactions: they can run concurrently and retry a potentially unbounded amount of times, thus increasing the execution time for each retry. We say that TM is not *time-predictable*.

Manson *et al.* [14] took the consequence and coined Preemptible Atomic Regions for Java, described in Section 7.1.1. Calculating the WCET for tasks using PARs is not significantly different from that of locks, the only difference being determining the upper bound of writes in each PAR. This is necessary in order to reason about the execution time of an undo operation if the PAR is preempted.

While PARs can be seen as transactions, determining the WCET for general transactions has also been of interest [15, 70]. We conclude that this is due to the fact that PARs are only relevant for uni-processor setups, while general transactions can run concurrently. RTTM [15] defines hardware-level support for transactions to provide higher predictability and lower WCETs, namely by bounding the number of aborts a transaction can suffer. Calculation techniques for RTTM are proposed in [71].

7.1.6 Uni- vs Multi-Processor

As of this writing, embedded real-time systems use multi-processor hardware in increasing numbers. Applications can be computationally complex and, as such, require more resources to run in a timely fashion, but increasing the number of cores instead of the clock frequency also has power-related advantages. As with most transitions from uni-processor to multi-processor hardware, new techniques must be employed to reason about correctness. In the context of real-time systems, correctness refers to functional and temporal correctness. [22] This will be used in our discussion in Section 7.2 when considering our approach in a multi-processor setting.

In this section, we describe how *proportionate fair scheduling* (pfair) has become attractive as a global scheduling scheme, and summarise the work done in this field.

Reasoning about real-time systems on multi-processor hardware is only feasible when considering the simple task model. We presented the general task model in Section 3.1, which applies to uni-processor environments. This model is an improvement over the simpler one which is more restrictive. Their differences are described in Section 3.1, but as an example, it assumes the following:

- Only periodic tasks.
- Complete inter-task independency.
- Tasks complete before their next release ($D = T$).
- Tasks do not exhibit blocking behavior.

While this model applies to many applications, it does impose restrictions. Sporadic tasks are often used in error handling, for which it does not necessarily apply that they complete before their next release. Failure to meet these assumptions can lead to unsound proofs about schedulability, as stated in [22]. However, sporadic tasks have a fixed minimum inter-arrival time, meaning there is a delay between how often they can be released. Knowing this, sporadic tasks can be modelled as periodic tasks with a period equal to the minimum inter-arrival time [22], and thus removing the need for explicity allowing sporadic tasks.

Traditionally, a rate-monotonic or EDF approach has been used in both uni- and multi-processor environments. While they are optimal on uni-processor hardware, they both result in low processor utilization on multiple processors [2]. Pfair was the first optimal scheduling scheme for periodic tasks [72, 73, 74, 75, 76]. In [72], it is shown that pfair is capable of scheduling any set of periodic tasks with utilization up to the available number of identical processors. However, recent research has shown that pfair schedules induce a high amout of preemption between the tasks. Inspired by EDF, an algorithm called U-EDF was developed for scheduling periodic tasks across multiple identical processors [77]. Accordingly, the algorithm generates schedules with a maximum of one preemption per task in their simulations. They prove the correctness of the algorithm, but not with respect to optimality. Their algorithm for allocating tasks onto processors does not consider shared resources either.

Utilization-based schedulability tests are, however, not exact and do not apply to the general task model. They do not consider blocking or interference, which arguably become much more relevant in multi-processor environments. Assuming any two tasks can execute in parallel is only possible if both tasks do not require exclusive access to a resource. That is why multi-processor scheduling only applies to the simple task model where tasks are independent of each other.

7.1.7 Increasing Predictability

To bound the otherwise unpredictable nature of transaction interleavings and execution time, different techniques have been developed. Inevitability (Section 5.6.1) is one approach that has been leveraged in [65] where the optimism in the concurrency control is degraded when a transaction is closing in on its deadline. At the last stage, the transaction in question is made inevitable.

Providing bounds on maximum number of aborts, and thus also subsequent retries, is also a common technique [71, 15].

Considering the scheduler to assist the contention manager is a third technique. Providing the tasks with temporal properties, as described in Section 3.1, the contention manager can use these in its decision making [78, 79]. The former describes contention management algorithms for FPS and EDF scheduling schemes, and introduces the term *slack*. Slack is the remaining time a transaction has to retry in before its deadline is missed. Using this information the contention manager can decide whether or not a given transaction can meet its deadline if it is considered as an abort candidate. It does not consider scheduling in a multi-processor setting, though. According to [80], notable work on using non-blocking synchronization in CMP real-time systems is [81, 82]. It is argued that a blocking TM implementation provides less overhead, but the possibility of priority inversion does not seem to be considered. Decisions are made based on a similar set of design criteria as we give in Section 5.4.

7.2 Our Approach

Based on findings throughout this project, we have established assumptions regarding how software transactions can exhibit time-predictable behavior, provide reliable and correct concurrency, and even be analyzable with respect to response time. The assumptions revolve around STM design choices.

First, the central issues in introducing STM in real-time systems are reiterated, and lastly, the general idea of our intended approach to make real-time enabled STM is presented.

7.2.1 Issues

Priorities in real-time systems dictate some form of *importance* of tasks relative to each other. As we covered in Chapter 3, tasks with higher priorities should not be blocked or preempted by tasks with lower priorities. TM as-is does not provide any notion of relative importance, and can thus result in a very different interleaving pattern than that of the tasks executing them. A situation where tasks are scheduled correctly according to their temporal properties, but the transactions are running in an undesirable manner where they commit little or no work, can easily occur. Even when the contention manager can provide increased fairness in how transactions are being aborted, this is not sufficient for hard real-time systems as we have seen in Section 7.1.

Bounding the number of times a transaction can be aborted provides a higher degree of predictability. It is inspired by the work on bounding loops in real-time systems [83], and has been adopted in TM applications as well as described in the related work in Section 7.1.7—in both cases to make the WCET analysis realistic.

7.2.2 General Idea

Our idea for a solution to these issues involves letting non-blocking transactions inherit the priorities of tasks. To that end, we can let the transaction, say *B*, with the highest priority of the currently running transactions commit, and force other conflicting transactions to wait. Should an even higher priority transaction, say *A*, start before *B* has committed, *A* will immediately preempt *B*. After *A* has committed, *B* will resume. One of two things will happen during the execution of *A*: (a) *B* modified a shared object, and actions towards ensuring *A* reads consistent data must be taken, or (b) *B* either read a shared object also referenced in *A*, or modified resources which *A* does not reference, and thus no action is required to be performed at this point. This effectively eliminates priority inversion and the blocking time experienced following from counter-measures such as original and immediate ceiling priority protocol. Based on the knowledge gained throughout this project, we establish a set of claims regarding how TM can be applied in hard real-time systems. Even though we have not implemented a TM for hard real-time systems at this point, we provide claims which capture the uncertainties we have regarding design choices. The uncertainties have been addressed separately (Section 7.1), but the combination of them have not.

We state the following claim, which applies to uni-processor platforms:

Claim 1:
By assigning non-blocking transactions priorities inherited from the tasks starting them, and preventing other than the highest-priority transaction from committing, we can allow higher-priority tasks to start executing their critical sections faster than with lock-based synchronization.

However, to ensure that there are no same-priority transactions that can livelock each other we present the following claim:

Claim 2:
By using distinct priorities for threads, only one of the running transactions at any point in time will have the highest priority and will always be allowed to commit successfully.

The basis of Claim 1 is found in Preemtible Atomic Regions (Section 7.1.1), where critical regions can begin executing as soon as the effects of the preempted critical region has been undone. Although in the case where not all critical sections conflict, our approach can show a higher level of concurrency and less wasted CPU time. In PAR, preempted regions consistently undo their changes immediately, regardless of whether or not the involved regions access the same shared objects. In our approach, a transaction need only be aborted if a conflict is detected, i.e. when the transactions access one or more of the same shared objects. Preempted transactions which do not conflict are still preempted, but not aborted and will continue at the preemption point later on. Of course, the worst-case situation will still result in the same behavior as PARs, where the preempted transactions must be aborted because there is a conflict. That is, when each transaction accesses one or more of the same shared objects.

Using ideas from RTTM (Section 7.1.2), however, we can limit this worst-case situation during the response-time analysis:

Claim 3:
Using static analysis, we can determine sets of transactions that can run concurrently without conflicts. If the schedulability analysis takes this into consideration it will lead to a tighter bound on the worst-case response time, as there might be fewer retries to consider.

In RTTM, a static analysis is performed to determine the possibilities of conflicts. Using the same approach for an STM, it would be possible to lower the worst-case scenario from that of

PARs, in that the fact that transactions that access disjoint memory locations will never conflict results in them not being able to abort each other. However, RTTM is an HTM, so it requires hardware support from the platform. Using a software solution will allow the use of existing hardware without alterations.

In addition to the unpredictability of STM, another problematic issue is I/O. Since I/O performed is usually irreversible, it is not possible to perform these calls in transactions that might need to be retried. To accommodate the use of I/O in critical sections, we propose the following:

Claim 4:
Since the transactions of the running task with the highest priority in our system are effectively inevitable, I/O can be performed here. If applications are designed to use only the highest priority thread to perform I/O that must be in critical sections, then our idea becomes viable for a wider range of real-time software.

In RTTM, this is handled by acquiring the commit lock early, which makes the transaction inevitable as soon as I/O calls are encountered. However, this approach reduces concurrency of the system, as other, even non-conflicting, transactions are blocked from committing until the inevitable transaction runs to completion and releases the lock.

Multi-Processing

Extending these thoughts to multi-processor systems requires considering the issues of scheduling tasks onto multiple processors/cores. As is argued in Section 7.1.6, non-blocking concurrency becomes extra attractive in this setting, but the scheduling much more difficult. It is also noted that useful results have only been produced for the simpler task model in multi-processor environments, whereas uni-processor approaches allow for a much more sophisticated task model. The differences between the two task models are described in Section 3.1.

Given a task set which is schedulable either by global or partitioned allocation [22], we open the discussion here whether or not our approach will apply in a multi-processor environment. Assuming static analysis reveals which transactions can run concurrently without conflicts (Claim 3), this information can be used to denote which transactions can run in parallel, i.e. on separate cores or CPUs at the same time.

Since shared resources are not considered according to the simple task model, neither is the fact that tasks can be blocked because a requested resource is acquired by another task. If non-blocking data structures were used as shared entities between tasks, then the need for (blocking) critical sections would be alleviated. This would in turn leave each operation on the non-blocking data structure with an increased overhead in terms of higher memory usage and potential retries in case of a conflict.

As we covered in Section 7.1.6, the Pfair scheduling algorithm is able to schedule periodic tasks on multiple identical processors using the notion of fairness. It is also argued how Pfair induces an unreasonable amount of task preemptions, and is the motivation behind U-EDF, which decreases the amount of preemptions.

It is difficult to say whether we can take steps towards applying our approach in multi-processor environment. Every optimal multi-processor scheduling algorithm we have investigated does not allow for dependencies between tasks. Using non-blocking synchronization, and considering Claim 1–4, our approach could be applicable to multi-processor architectures. However, the scope of this project is to replace lock-based synchronization techniques with STM to achieve less error-prone applications and, in some cases, a higher level of concurrency on uni-processor hardware. Deeper research on existing work within scheduling of real-time systems on multi-processor hardware would have to be performed.

7.3 Response Time Analysis

To analyse the response time of our approach (Section 7.2), we will use the sources on which our claims are based. However, we also need to look closer at the design choices (Section 5.4) we will use to design an STM based on our idea:

Operational Structure This choice has no impact on our analysis. Any operational structure will be rewritten to the same basic boilerplate code that will be used as a basis for the analysis.

Conflict Detection We have already shown how the highest-priority transaction will be inevitable. Chosing more optimistic conflict detection strategies for the other running transactions has no impact on the worst-case scenario, as they are not guaranteed to commit before becoming inevitable themselves, so any performance gain here is merely an average-case optimisation.

Direct or Deferred Updates This design choice is less important to our analysis: if direct updates are used, whenever a transaction is aborted its work must be undone, however, if deferred updates are used, whenever a transaction commits its redo-log must be used to make the changes visible. In both cases, the maximum size of the metadata is constant, so the time required in both cases is constant. However, the performance of the redo-log must be considered to determine the worst-case time of a transaction, as it must be consulted for each shared memory read.

Isolation We will not look into strong isolation, as the overhead associated with ensuring strong isolation gives added complexity to the overall system making it more difficult to analyse.

Nested Transactions Composability is an important part of the motivation behind using STM in an application, so we will strive to support this. Flat nesting allows the easiest analysis as inner transactions can never retry without retrying the outermost transaction and thus do not contribute to the unpredictability of the STM.

Hardware or Software TM The purpose of this project is to provide a real-time enabled implementation of an STM, so this choice is already given.

Static or Dynamic To determine the worst-case execution time of a task, it must have a bound on the number of transactions that can be run per release of the task. It is also necessary to determine the shared memory locations that can be accessed to determine which transactions can conflict (Claim 3). This information is also required to determine the maximum size of the STM metadata, which allows bounds on the undo- or redo-logs to make them predictable. The conclusion is that our STM will be static, although static analysis tools will be able to determine the information needed to make it appear dynamic to the programmers using the STM in their applications.

Granularity The static analysis to find possibly conflicting transactions will require that the granularity provides no false conflicts, or a predictable set of false conflicts. Using either object or field granularity, which we have seen is common in Java TMs (Chapter 6), will allow predicting conflicts reliably, as the code determines when there is a conflict.

Blocking or Non-Blocking Because our claims rely on a non-blocking implementation to hold, we will use an obstruction-free design.

Contention Management Our approach relies on the highest-priority thread being allowed to commit inevitably, so it will use an aggressive contention management strategy aborting any conflicting transactions as it runs. Since the STM already guarantees that all transactions will be allowed to commit, it is not necessary to use a contention manager to provide livelock and starvation avoidance.

Given these choices we will now look at how we expect to compute the response time of an application:

Retries As the number of retries is bounded in the same fashion as in RTTM (Section 7.1), we conclude that this allows a sound response time analysis. If only one transaction is allowed per task, and they all have equal length and can conflict, the critical instant becomes as shown in Figure 7.1, where the highest-priority transaction is guaranteed to commit in its first attempt, and the lowest-priority transaction will run at most $n - 1$ times, where n is the number of tasks in the system.

Worst-Case Execution Time The amount of time taken to execute a transaction depends on boilerplate code for: (a) starting a transaction, (b) reading from a shared variable, (c) writing to a shared variable, (d) committing a transaction, and (e) the remaining body of the transaction. Abort costs are considered in the overall response time when other transactions can interfere with the execution. Because of the choices we have made and our experience with STMs, we conclude that we will be able to determine upper bounds for each of these boilerplate code parts. Choosing either the HVM (Section 4.3) or JOP (Section 4.1) will give us a way of determining the exact cost of executing each instruction related to transaction management. Including these bookkeeping costs into the WCET analysis of our STM enabled applications, we can still express the total response time correctly. For example, given that the lowest-priority transaction will run at most $n - 1$ times, where n is the total number of tasks, the WCET here equals $C_T(n-1)$, where C_T is the transaction related cost.

Chapter 8

Evaluation

In this chapter, we describe our own experience with researching the field of transactional memory and related work, the difficulties we encountered, and how our work lead us to our own approach described in Section 7.2.

8.1 Transactional Memory

Our bibliography shows that there is a tremendous amount of research in TM. Much of our work has been to sort through this research to find the commonalities of individual TM systems and how they set themselves apart. The results of this work is the list of design and implementation choices that we have compiled in Chapter 5.

A large obstacle we had to overcome to get the necessary overview of the field and compile this list was finding the right entry point to the research in the field. While many articles start by explaining generally what TM is and why it is beneficial, they quickly delve deeply into specific issues making the article hard to follow for someone who has not yet grasped the concepts of TM. It was not until we found the books [10, 11] that we were able to fully understand the foundations of TM and the significance of individual papers.

We have not encountered detailed concurrency theory throughout our time at the university until this point. Especially correctness was a topic we came upon several times when researching TM in Section 5.3, which was crucial to understand in order to grasp the properties of transactional memory. Identifying the hierarchy of non-blocking progress guarantees was also a challenge, as described in Section 5.13, but provided us with the necessary understanding of the characteristics of non-blocking and blocking concurrency.

Our experience was further expanded by implementing concrete algorithms found in [10]. The algorithms are described as being too simple and inefficient for practical use, but building a TM from scratch allowed us to see exactly how TM functions and analysing the implementations allowed us to see where and why they are inefficient and what other implementations of TM do to be efficient in practice. Our first attempt at understanding TM in practice involved looking at open source implementations like DSTM2 and Deuce STM, however, the size of the code base of these systems is significant and difficult to understand due to complexities like bytecode rewriting. Working with a simple implementation allowed us to play with individual options and see how individual properties are ensured, e.g. obstruction-freedom.

Another obstacle we encountered in our mapping of the field is that different terms are sometimes used to describe the same principle. For example, the terms strong isolation and strong atomicity are both used in the field, so when attempting to determine if a TM system supports this feature, it was necessary to keep an eye out for either of these terms. In [11], both terms were used to introduce the concept to avoid confusion, which is also how we chose to handle the issue in this report.

The result of our mapping of the field proved very useful, however, as we were able to use our understanding of TM concepts to successfully analyse TM systems. Using this understanding of TM we were also able to recognise how existing work in bringing TM to RTS works and where we will be able to contribute.

A final observation we would like to make on the field of TM is that development is currently taking place mostly in academia. Although several companies like Intel, Microsoft and Oracle are actively working on TM [84, 85, 4], we have, as of yet, found very little evidence of commercial use of TM.

8.2 Real-Time Software

In order to understand the difficulty in bringing TM to RTS, we also looked at RTS generally. As we already had previous experience with RTS, it was easier to describe this field, however the challenge here was to find the parts that would be relevant when analysing real-time TMs.

We decided to look at two analysis tools for real-time Java in Chapter 4 that we have found should be suitable for the work that lies ahead in implementing a real-time STM. However, as we did not have any previous experience with these tools, we had to acquire it during our research.

It is noted when describing the HVM in Section 4.3 that the documentation for it is under development. The website[1] further states that a *HelloWorld* example exists in the SDK jar linked on the website, but it only contains the SDK and a compiled version of the example. Kasper Søe Luckow[2] is a co-author on TetaJ [41], and he has extensive experience with the HVM. He was able to show us how to get started with the HVM in practice. TetaJ was straightforward in getting to work for the supplied example, and Kasper was also kind enough to enlighten us on the component architecture thereof.

The JOP was developed as a part of Schoeberl's Ph.D. thesis [35] which has extensive details about analysis, design, and implementation of the JOP. As such, we did not need further instructions on how to develop for that platform.

Even though we did not experiment in practice with the JOP and HVM, and the analysis tools, we still became aware of their existence and established a rough baseline for using them.

8.3 Real-Time Transactional Memory

In our own approach that we have presented in this report, we build upon the existing work that has been done in making TM schedulable. The existing work in the RTTM article has already shown that it is possible to use TM in RTS.

Issues often arise when developing software based on new ideas. As far as our research has shown, there are no implementations of our idea or similar notions in existence today, so obstacles during development are to be expected. However, due to the foundations of our claims in Section 7.2 having working implementations, we conclude that it will be possible for us to combine these claims to create a novel implementation of a real-time STM.

[1]http://icelab.dk/
[2]luckow@cs.aau.dk

The greatest challenge we foresee for the development of our real-time STM is to make it scalable on multi-processors. We have seen in Section 7.1 that multi-processor scheduling is difficult, so this is an obstacle that we must overcome if we are to live up to all the promises of STM. However, even without scalability, using STM in RTS still brings immediate benefits like composability and avoiding the use of manually implemented locking protocols. The work on scalability would then be considered future work after the initial implementation of our real-time STM that is immediately ahead of us.

Chapter 9

Conclusion

In this report, we looked at the possibility of bringing transactional memory to real-time software. To approach the subject, we first refreshed our knowledge on real-time software, and then investigated the concepts of transactional memory to learn why it is interesting and how it works. The final part of the report was then used to discuss the existing work in creating real-time enabled transactional memory, and give our thoughts on how the existing work can be used to create a software transactional memory for use in real-time software.

During our work on refreshing our knowledge of real-time software, we looked at what characterises this class of software and what is required of the tools used during development of real-time software. The conclusion here is that the main focus of hard real-time software is whether it is schedulable, and how to show that a given system is schedulable. For this schedulability analysis, we looked at a couple of tools that can analyse applications written in Java, the programming language we selected as the focus for the project. Java was chosen due to its widespread use and because it has support for real-time software. The tools for real-time Java applications we analysed are SARTS and TetaJ. Both allow analysing the schedulability of an application. SARTS is made for the specialised Java processor: the JOP, whereas TetaJ is made for the HVM, a virtual machine designed to run on general-purpose Atmel processors. The conclusion here was that both tools can be used to analyse real-time transactional memory implementations used in real-time software.

The next focus was transactional memory. We determined that transactional memory is interesting because concurrency is becoming increasingly important due to the rise in multi-core processors being used. Transactional memory was shown to be easier to use than traditional lock-based concurrency control, however, it has not yet reached widespread use and much research is still taking place to develop the concept. After having looked at some general foundations for transactional memory: defining transactions and what it means for their execution to be considered correct, we compiled a list of design and implementation choices that we found to be common for transactional memory systems. During this process we chose to have software transactional memory as a focus for our efforts, as this allows widespread use on existing hardware and has fewer limitations in terms of hard limits on transaction sizes. We then used our list of design and implementation choices to analyse a few implementations of Java software transactional memory systems to show how this list was useful and how existing implementations were designed. In addition, we also implemented a few of our own software transactional memory systems to further understand the concepts.

Using the knowledge about real-time systems and transactional memory, we then looked at how these concepts could be combined. To this end, we started by analysing existing work to see what had already been achieved. We described three different systems: Preemptible Atomic Regions for Real-Time Java [14], RTTM: Real-Time Transactional Memory [15], and Deadline-Aware Scheduling for Software Transactional Memory [65]. The first we found to use concepts found in transactional memory, but the optimistic concurrency that allows transactions to run concurrently had been removed for schedulability. The second is a real-time hardware transactional memory, however our interest lies in achieving a pure STM where RTTM relies on hardware support. The last is a software transactional memory for soft real-time applications, which means it is not directly usable for hard real-time applications, however, we found the concept of inevitability used in this system interesting, as it allows guaranteeing that a transaction will be able to commit.

While Preemptible Atomic Regions (PARs) consistently undo their changes when preempted, regardless of whether the involved regions actually conflict, our approach only results in undoing changes should two transactions conflict. This results in less wasted CPU time and a higher level of concurrency in ideal cases. Using transactions to wrap I/O operations which are irreversible is not appropriate, and as such we propose to have all I/O operations running in a highest-priority task, since the transactions within this task will effectively be inevitable.

Using these systems and other related work in real-time scheduling and concurrency concepts, we outlined our idea for a design of an obstruction-free software transactional memory for hard real-time systems. Our ideas initially apply to uni-processor platforms only, but we discuss measures towards applying our ideas on multi-processor hardware as well.

The next step from here is to use the knowledge we have gathered in this report to implement our idea. Certain claims were made in the formulation of our idea that we will be able verify during the implementation. During our analysis of our idea, we concluded that because the foundations of the idea have been shown to work in practice, the implementation should be workable to create software transactional memory for hard real-time systems.

Bibliography

[1] Kim G. Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer (STTT)*, 1:134–152, 1997. 10.1007/s100090050010.

[2] John Carpenter, Shelby Funk, Philip Holman, Anand Srinivasan, James Anderson, and Sanjoy Baruah. A categorization of real-time multiprocessor scheduling problems and algorithms. In *Handbook on Scheduling Algorithms, Methods, and Models*. Chapman Hall/CRC, Boca, 2004.

[3] Bryan Cantrill and Jeff Bonwick. Real-world concurrency. *Queue*, 6:16–25, September 2008.

[4] Maurice Herlihy, Victor Luchangco, and Mark Moir. A flexible framework for implementing software transactional memory. *SIGPLAN Not.*, 41:253–262, October 2006.

[5] Simon P. Jones. *Beautiful Concurrency*. O'Reilly Media, Inc., 2007.

[6] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21:289–300, May 1993.

[7] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers, 2008.

[8] G. Korland, N. Shavit, and P. Felber. Noninvasive concurrency with java stm. In *Third Workshop on Programmability Issues for Multi-Core Computers (MULTIPROG-3)*, 2010.

[9] Luke Dalessandro, Virendra J. Marathe, Michael F. Spear, and Michael L. Scott. Capabilities and limitations of library-based software transactional memory in c++. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Transactional Computing*. Portland, OR, Aug 2007.

[10] Rachid Guerraoui and Michał Kapałka. *Principles of Transactional Memory*. Morgan & Claypool, 2010.

[11] Tim Harris, James R. Larus, and Ravi Rajwar. *Transactional Memory, 2nd edition*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2010.

[12] Torvald Riegel, Patrick Marlier, Martin Nowack, Pascal Felber, and Christof Fetzer. Optimizing hybrid transactional memory: The importance of nonspeculative operations. In *SPAA'11*, pages 53–64, 2011.

[13] Derin Harmanci. Tmjava. `http://www.tmware.org/tmjava`. [Online; accessed Dec 8 2011].

[14] Jeremy Manson, Jason Baker, Antonio Cunei, Suresh Jagannathan, Marek Prochazka, Bin Xin, and Jan Vitek. Preemptible atomic regions for real-time java. In *In 26th IEEE Real-Time Systems Symposium*, 2005.

[15] Martin Schoeberl, Florian Brandner, and Jan Vitek. Rttm: real-time transactional memory. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 326–333, New York, NY, USA, 2010. ACM.

[16] Oracle Java Community Process. Real-time specification for java (rtsj). `http://www.rtsj.org/specjavadoc/book_index.html`. [Online; accessed Oct 9 2011].

[17] X/Open Base Working Group. Real-time extension in unix. `http://www.unix.org/version2/whatsnew/realtime.html`. [Online; accessed Dec 6 2011].

[18] H. Kopetz, J. C. Laprie, Brian Randell, and B. Littlewood, editors. *Predictably Dependable Computing Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1995.

[19] Continental Automotive GmbH. Osek specification 2.2.3. `http://portal.osek-vdx.org/files/pdf/specs/os223.pdf`. [Online; accessed Nov 21 2011].

[20] Inc. Intermetrics. Ada 95 specification. `http://www.adaic.org/resources/add_content/standards/95lrm/RM.pdf`. [Online; accessed 26 Nov 2011].

[21] IBM. Posix threads explained. `http://www.ibm.com/developerworks/linux/library/l-posix1/index.html`. [Online; accessed 11 Nov 2011].

[22] Alan Burns and Andy Wellings. *Real-Time Systems and Programming Languages*. Addison Wesley, fourth edition, 2009.

[23] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2007.

[24] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20:46–61, January 1973.

[25] K. Albers and F. Slomka. An event stream driven approximation for the analysis of real-time systems. In *Real-Time Systems, 2004. ECRTS 2004. Proceedings. 16th Euromicro Conference on*, pages 187 – 195, june-2 july 2004.

[26] S. Chakraborty, S. Kunzli, and L. Thiele. Approximate schedulability analysis. In *Real-Time Systems Symposium, 2002. RTSS 2002. 23rd IEEE*, pages 159 – 168, 2002.

[27] U.C. Devi. An improved schedulability test for uniprocessor periodic task systems. In *Real-Time Systems, 2003. Proceedings. 15th Euromicro Conference on*, pages 23 – 30, july 2003.

[28] Fengxiang Zhang and A. Burns. Schedulability analysis for real-time systems with edf scheduling. *Computers, IEEE Transactions on*, 58(9):1250 –1258, sept. 2009.

[29] E. Bini, G.C. Buttazzo, and G.M. Buttazzo. Rate monotonic analysis: the hyperbolic bound. *Computers, IEEE Transactions on*, 52(7):933 – 942, jul 2003.

[30] M Joseph and P Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(5):390–395, 1986.

[31] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A.J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284 –292, sep 1993.

[32] CISS. Indlejrede systemer skal tala java. `http://ciss.dk/dk/projekter/afsluttede_projekter_-_case_stories/indlejrede_systemer_skal_tale_java.htm`. [Online; accessed Dec 28 2011].

[33] Martin Schoeberl. JOP Handbook. `http://jopdesign.com/doc/handbook.pdf`. [Online; accessed Dec 6 2011].

[34] Andrew S. Tanenbaum. *Structured Computer Organization*. Prentice Hall, fifth edition, 2006.

[35] Martin Schoeberl. *JOP: A Java Optimized Processor for Embedded Real-Time Systems*. PhD thesis, Vienna University of Technology, 2005.

[36] P. Puschner and Ch. Koza. Calculating the maximum, execution time of real-time programs. *Real-Time Syst.*, 1:159–176, September 1989.

[37] Thomas Bøgholm, Henrik Kragh-Hansen, Petur Olsen, Bent Thomsen, and Kim G. Larsen. Model-based schedulability analysis of safety critical hard real-time java programs. In *JTRES '08: Proceedings of the 6th international workshop on Java technologies for real-time and embedded systems*, pages 106–114, New York, NY, USA, 2008. ACM.

[38] Thomas Henties, James J. Hunt, Doug Locke, Kelvin Nilsen, Martin Schoeberl, and Jan Vitek. Java for safety-critical applications. In *2nd International Workshop on the Certification of Safety-Critical Software Controlled Systems (SafeCert 2009)*, 2009.

[39] Martin Schoeberl and Wolfgang Puffitsch. Nonblocking real-time garbage collection. *ACM Trans. Embed. Comput. Syst.*, 10:6:1–6:28, August 2010.

[40] Christian Frost, Casper Svenning Jensen, Kasper Søe Luckow, and Bent Thomsen. Wcet analysis of java bytecode featuring common execution environments. In *Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '11, pages 30–39, New York, NY, USA, 2011. ACM.

[41] Christian Frost, Casper Svenning Jensen, and Kasper Søe Luckow. Wcet analysis of java bytecode featuring common execution environments. Master's thesis, Aalborg University, 2011.

[42] Herb Sutter. The free lunch is over. *Dr. Dobb's Journal*, 30(3), 2005.

[43] Edward A. Lee. The problem with threads. Technical Report UCB/EECS-2006-1, EECS Department, University of California, Berkeley, Jan 2006. The published version of this paper is in IEEE Computer 39(5):33-42, May 2006.

[44] Christos H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26:631–653, October 1979.

[45] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12:463–492, July 1990.

[46] Christos H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26:631–653, October 1979.

[47] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28:690–691, September 1979.

[48] Rachid Guerraoui and Michał Kapałka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 175–184, New York, NY, USA, 2008. ACM.

[49] W. E. Weihl. Local atomicity properties: modular concurrency control for abstract data types. *ACM Trans. Program. Lang. Syst.*, 11:249–282, April 1989.

[50] Tim Harris and Keir Fraser. Language support for lightweight transactions. *SIGPLAN Not.*, 38:388–402, October 2003.

[51] Aleksandar Dragojevic, Pascal Felber, Vincent Gramoli, and Rachid Guerraoui. Why STM can be more than a Research Toy. *Communications of the ACM*, 54:70–77, 2011.

[52] Adam Welc, Bratin Saha, and Ali-Reza Adl-Tabatabai. Irrevocable transactions and their applications. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, SPAA '08, pages 285–296, New York, NY, USA, 2008. ACM.

[53] Michael F. Spear, Michael Silverman, Luke Dalessandro, Maged M. Michael, and Michael L. Scott. Implementing and exploiting inevitability in software transactional memory. In *Proceedings of the 2008 37th International Conference on Parallel Processing*, ICPP '08, pages 59–66, Washington, DC, USA, 2008. IEEE Computer Society.

[54] Michael F. Spear, Virendra J. Marathe, Luke Dalessandro, and Michael L. Scott. Privatization techniques for software transactional memory. In *Proceedings of the 26th PODC ACM Symposium on Principles of Distributed Computing*. Aug 2007.

[55] Peter Veentjer. Multiverse. `http://multiverse.codehaus.org/`. [Online; accessed Dec 28 2011].

[56] Pascal Felber, Christof Fetzer, Patrick Marlier, and Torvald Riegel. Time-based software transactional memory, 2010.

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

[58] João Cachopo and António Rito-Silva. Versioned boxes as the basis for memory transactions. *Sci. Comput. Program.*, 63:172–185, December 2006.

[59] Typesafe inc. Software transactional memory (scala). `http://akka.io/docs/akka/1.2/scala/stm.html`. [Online; accessed Dec 28 2011].

[60] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, PODC '03, pages 92–101, New York, NY, USA, 2003. ACM.

[61] Michael F. Spear, Luke Dalessandro, Virendra J. Marathe, and Michael L. Scott. A comprehensive strategy for contention management in software transactional memory. *SIGPLAN Not.*, 44:141–150, February 2009.

[62] Torvald Riegel, Pascal Felber, and Christof Fetzer. A lazy snapshot algorithm with eager validation. In *In Proceedings of the 20th International Symposium on Distributed Computing (DISC'06)*, pages 284–298, 2006.

[63] Oracle. Package java.util.concurrent.atomic. `http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/atomic/package-summary.html`. [Online; accessed Dec 20 2011].

[64] Damian Dechev and Bjarne Stroustrup. Reliable and efficient concurrent synchronization for embedded real-time software. In *Proceedings of the Third IEEE International Conference on Space Mission Challenges for Information Technology*, pages 323–330, Washington, DC, USA, 2009. IEEE Computer Society.

[65] W. Maldonado, P. Marlier, P. Felber, J. Lawall, G. Muller, and E. Riviere. Deadline-aware scheduling for software transactional memory. In *Dependable Systems Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*, pages 257 –268, june 2011.

[66] P. Sorensen. *A Methodology for Real-Time System Development*. PhD thesis, University of Toronto, 1974.

[67] P. Sorensen and V. Hemacher. A real-time system design methodology. *INFOR*, 13(1):1–18, February 1975.

[68] J.H. Anderson, S. Ramamurthy, and K. Jeffay. Real-time computing with lock-free shared objects. In *Real-Time Systems Symposium, 1995. Proceedings., 16th IEEE*, pages 28 –37, dec 1995.

[69] James H. Anderson, Srikanth Ramamurthy, Mark Moir, and Kevin Jeffay. Lock-free transactions for real-time systems, 1997.

[70] Sherif F. Fahmy, Binoy Ravindran, and E. D. Jensen. Response time analysis of software transactional memory-based distributed real-time systems. In *Proceedings of the 2009 ACM symposium on Applied Computing*, SAC '09, pages 334–338, New York, NY, USA, 2009. ACM.

[71] Martin Schoeberl, Bent Thomsen, and Lone Leth Thomsen. Towards transactional memory for real-time systems. Technical report, 2009.

[72] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15:600–625, 1996. 10.1007/BF01940883.

[73] J.H. Anderson and A. Srinivasan. Early-release fair scheduling. In *Real-Time Systems, 2000. Euromicro RTS 2000. 12th Euromicro Conference on*, pages 35 –43, 2000.

[74] J.H. Anderson and A. Srinivasan. Mixed pfair/erfair scheduling of asynchronous periodic tasks. In *Real-Time Systems, 13th Euromicro Conference on, 2001.*, pages 76 –85, 2001.

[75] M. Moir and S. Ramamurthy. Pfair scheduling of fixed and migrating periodic tasks on multiple resources. In *Real-Time Systems Symposium, 1999. Proceedings. The 20th IEEE*, pages 294 –303, 1999.

[76] Anand Srinivasan and James H. Anderson. Optimal rate-based scheduling on multiprocessors. *J. Comput. Syst. Sci.*, 72:1094–1117, September 2006.

[77] G. Nelissen, V. Berten, J. Goossens, and D. Milojevic. Reducing preemptions and migrations in real-time multiprocessor scheduling algorithms by releasing the fairness. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2011 IEEE 17th International Conference on*, volume 1, pages 15 –24, aug. 2011.

[78] António Barros and Luís Miguel Pinho. Managing contention of software transactional memory in real-time systems. Technical report, CISTER Research Center, Polytechnic Institute of Porto, Portugal, 2010.

[79] Walther Maldonado, Patrick Marlier, Pascal Felber, Adi Suissa, Danny Hendler, Alexandra Fedorova, Julia L. Lawall, and Gilles Muller. Scheduling support for transactional memory contention management. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 79–90, New York, NY, USA, 2010. ACM.

[80] S.F. Fahmy, B. Ravindran, and E.D. Jensen. On bounding response times under software transactional memory in distributed multiprocessor real-time systems. In *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, pages 688 –693, april 2009.

[81] Philip Holman and James H. Anderson. Supporting lock-free synchronization in pfair-scheduled real-time systems. *J. Parallel Distrib. Comput.*, 66:47–67, January 2006.

[82] Jennifer Mankin, David Kaeli, and John Ardini. Software transactional memory for multicore embedded systems. In *Proceedings of the 2009 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, LCTES '09, pages 90–98, New York, NY, USA, 2009. ACM.

[83] P. Puschner and Ch. Koza. Calculating the maximum execution time of real-time programs. *Real-Time Systems*, 1:159–176, 1989. 10.1007/BF00571421.

[84] Robert Ennals. Cache sensitive software transactional memory. Technical report.

[85] Microsoft Corporation. Tim harris. `http://research.microsoft.com/en-us/um/people/tharris/`. [Online; accessed Dec 6 2011].

Appendix A

Example: Calculating Response Time

Here we give a short example of how to calculate the response time for a task using the information found in Section 3.3.

Task	Period (T)	Computation time (C)	Priority (P)
a	4	2	2
b	10	3	2
c	15	4	1

Table A.1: Example task set.

Given the task set in Table A.1, we can establish the worst-case response time for the highest-priority task *a* equals its computation time, that is $R_a = 2$. The response times of the remaining tasks are calculated using Equation 3.2. The iterations needed for calculating the response time for task *b* is defined here:

$$\begin{aligned}
w_b^0 &= 3 \\
\\
w_b^1 &= 3 + \left\lceil \frac{2}{4} \right\rceil 3 \\
w_b^1 &= 6 \\
\\
w_b^2 &= 3 + \left\lceil \frac{6}{4} \right\rceil 3 \\
w_b^2 &= 9 \\
\\
w_b^3 &= 3 + \left\lceil \frac{9}{4} \right\rceil 3 \\
w_b^3 &= 12
\end{aligned}$$

$$
\begin{aligned}
w_b^4 &= 3 + \left\lceil \frac{12}{4} \right\rceil 3 \\
w_b^4 &= 12
\end{aligned}
$$

The fourth iteration w_b^4 yields the same result as the previous iteration w_b^3, and thus the response time for task b in the example task set equals 12. Finally, asserting whether or not the task is schedulable depends on whether $R \leq T$ for the given task, and since $12 \leq 12$ task b is deemed schedulable with respect to both its own computation time and the worst-case interference from task a.