FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Hardware Transactional Memory for a Real-Time Chip Multiprocessor

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik

eingereicht von

Peter Hilber
Matrikelnummer 0326179

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer: O.Univ.Prof. Dipl.-Ing. Dr.techn. Herbert Grünbacher
Mitwirkung: Assoc. Prof. Dipl.-Ing. Dr.techn. Martin Schöberl

Wien, 15.06.2010

(Unterschrift Verfasser)

(Unterschrift Betreuer)

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.ac.at

Erklärung zur Verfassung der Arbeit

Peter Hilber
Pfeilgasse 3a/374
A-1080 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 15.06.2010

Acknowledgements

I am grateful to my thesis advisors, Prof. Martin Schöberl and Prof. Herbert Grünbacher, for supervising my thesis, for providing advice, and for guiding my research efforts. Earlier work by Martin Schöberl, Florian Brandner and Jan Vitek laid the foundations for my thesis. Martin also provided editorial and technical advice and the used tag memory implementation. I would like to thank Martin for many interesting discussions. I would also like to thank the members of the Real Time Systems Group at the Vienna University of Technology for their constructive feedback. The Department of Computer Engineering at the Vienna University of Technology generously provided the used target hardware. Thanks go to http://www.alteraforum.com user Rysc for sharing a content addressable memory implementation.

I would like to thank my parents, my sisters, my brother and my grandparents for their support during the course of my studies. Special thanks go to Carina for her support while writing this thesis.

Hardware Transactional Memory for a Real-Time Chip Multiprocessor

Abstract

Transactional memory is an alternative to conventional lock-based synchronization. Locks are difficult to use and not composable; transactional memory offers a simple programming model and the high concurrency desired for future multiprocessors. While actually multiple threads concurrently access shared data, the results look as if code sections (transactions) had been executed sequentially. Conflicts among concurrent transactions are automatically resolved.

To our knowledge, there is currently no transactional memory system suitable for hard real-time systems on multiprocessors. *Real-time transactional memory* (RTTM) is a proposal of a time-predictable transactional memory for chip multiprocessors. The main goals of RTTM are a simple programming model and analyzable timing properties. Static analysis detects non-conflicting transactions, which lowers the worst-case execution time bounds. In this master's thesis, RTTM was implemented on an FPGA and the viability of the implementation was evaluated.

For time-predictable execution, RTTM is hardware-based. Each core gets a small, fully associative cache which tracks the memory accesses in a transaction. RTTM was implemented on JOP, a time-predictable chip multiprocessor directly executing Java bytecode. The basic programming interface is the `@atomic` method annotation. Using Java facilitates link-time transformations and the abort of conflicting transactions.

The FPGA-based implementation supports small transactions suitable for synchronization in embedded real-time applications. Up to 12 cores fit on a low-cost Cyclone II FPGA running at 90 MHz with a device utilization of more than 90%. The RTTM hardware is costly due to the fully associative cache, but does not dominate the hardware resource consumption. The close relationship of the processor to the Java Virtual Machine enables some resource-saving optimizations. A part of RTTM was implemented in software in order to make the integration of the CPU nearly transparent and to lower the hardware costs. As a preparation for tool-based worst-case execution time analysis, the execution time of individual RTTM operations was bounded.

Hardware Transactional Memory for a Real-Time Chip Multiprocessor

Kurzfassung

Transactional Memory (Transaktionaler Speicher) ist eine Alternative zur konventionellen Synchronisation mit Locks. Programmierung mit Locks ist aufwändig, fehlerhaft und nicht *composable*; Transactional Memory bietet ein einfaches Programmiermodell und die hohe Nebenläufigkeit, die für künftige Multiprozessoren benötigt wird. Obwohl mehrere Threads gleichzeitig auf gemeinsame Daten zugreifen, entsprechen die Ergebnisse einer sequentiellen Ausführung von Codeabschnitten (Transaktionen). Konflikte von Transaktionen werden transparent aufgelöst.

Nach meinem Wissen gibt es kein für harte Echtzeitsysteme geeignetes Transactional Memory für Multiprozessoren. *Real-Time Transactional Memory* (RTTM) ist der Entwurf eines echtzeitfähigen Transactional Memories für Chip-Multiprozessoren. Die Entwurfsziele von RTTM sind ein einfaches Programmiermodell und analysierbares Zeitverhalten. Statische Analyse der potentiellen Konflikte von Transaktionen ermöglicht eine Verringerung der Worst-Case Execution Time Bounds. In dieser Diplomarbeit wurde RTTM auf einem FPGA implementiert und die Implementierung evaluiert.

Um echtzeitfähig und performant zu sein, ist RTTM hardware-basiert. Ein jedem Prozessor zugeordneter, vollassoziativer Cache verfolgt die Speicherzugriffe in einer Transaktion. Die Implementierungsplattform ist der echtzeitfähige Chip-Multiprozessor JOP, der Java Bytecode direkt ausführt. Die wesentliche Programmierschnittstelle ist die `@atomic` Method Annotation. Die Verwendung von Java vereinfacht Codetransformationen zur Link Time und den Abbruch von in Konflikt stehenden Transaktionen.

Die FPGA-basierte Implementierung ermöglicht Transaktionen zur Synchronisation von eingebetteten Echtzeitanwendungen. Auf einem Cyclone II FPGA können bis zu 12 Prozessorkerne laufen, mit einer Taktfrequenz von 90 MHz und einer Nutzung von über 90% der FPGA-Ressourcen. Die RTTM-Hardware ist wegen des vollassoziativen Caches aufwändig, dominiert aber nicht den Ressourcenverbrauch. Die Verwandtschaft von JOP mit der Java Virtual Machine ermöglicht Ressourcen sparende Optimierungen. Ein Teil von RTTM wurde in Software implementiert, um die Integration der CPU zu vereinfachen und den Ressourcenverbrauch zu vermindern. Als Vorbereitung auf eine toolbasierte Worst-Case-Execution-Time-Analyse wurde die maximale Ausführungszeit der einzelnen RTTM-Operationen analysiert.

Contents

List of Figures

List of Tables

List of Listings

CHAPTER 1

Introduction

Performance of sequential processors has increased exponentially in past decades, but this growth has slowed down in recent years. The gains by heavier use of instruction level parallelism and pipelining are diminishing and complex and fast clocked processors need more power than sustainable [OH05]. Moore's law about the exponential increase of the number of transistors available on an integrated circuit [Moo98] appears to remain valid for the next years, however [LR07].

The past increases in computational power have rendered possible increasingly capable software, and further increases of computational power are highly desirable. *Chip multiprocessing* is seen as a way to continue performance growth by using thread level parallelism. This does however necessitate the parallelization of applications, which is seen as a major problem [OH05]. Parallelization is usually not transparent to the programmer. *Parallel programming* is considered "fundamentally more difficult than sequential programming" [LR07, p. 2]. Concurrency and nondeterminism make it difficult to write parallel programs and reason about them.

In contrast to message passing systems, *shared memory* reduces the need for manual data distribution [KMVR90]. When using task parallelism (as opposed to data parallelism), concurrent threads are then usually coordinated using *lock-based concurrency control* such as semaphores, mutexes and monitors. The employed mechanisms are at a low level of abstraction.

Abstraction[1] and *composability*[2] are very important for handling complex systems [Kop08]. Abstraction and composability are extensively used in software engineering. Traditional, lock-based concurrency control does however not compose [Her06, Ch. 18.1].

[1]"An *abstraction* is a simplified view of an entity, which captures the features that are essential to understand and manipulate it for a particular purpose." [LR07, p. 3]

[2]"*Composition* is the ability to put together two entities to form a larger, more complex entity, which in turn is abstracted into a single, composite entity." [LR07, p. 4]

When manipulating data, it is often necessary to lock multiple exposed objects to guarantee invariants mandated by the application. A classical example is the atomic movement of an item between two data structures. It is necessary to lock both data structures in order to guarantee atomic movement, but this may lead to deadlocks when performing concurrent movements and taking locks in different order.

1.1 Transactional memory

Transactional memory (TM) is an alternative to conventional lock-based synchronization. Transactional memory allows a simple and composable programming model and potentially high concurrency. The name stems from analogies to transactions used in database systems. Database transactions have the ACID properties: (failure) atomicity, consistency, isolation and durability [HR83]. In a transactional memory system, a computation executing in a thread is wrapped in a transaction. Such a transaction has *ACI properties* (durability is not needed for in-memory transactions). *Failure atomicity* means that a transaction either fully completes or does not change anything at all. *Consistency* means that a transaction, when starting from a consistent state, will leave the system in a consistent state. Consistency is application dependent and may be represented as invariants on data structures. *Isolation* ensures that each transaction produces correct results regardless of any concurrently executing transactions [LR07].

The ACI properties make programming parallel systems easier: the results of transactions look as if they had been executed one after the other, while actually they might have been executing concurrently. For example, efficient parallel read and write access to a FIFO queue is difficult with locks because there can be contention when the queue has fewer than two elements [MS96]. When using transactional memory, such a parallel data structure is a non-issue. Ideally, a programmer only needs to mark code blocks which are to be executed atomically. When using other synchronization mechanisms, one has to specify explicitly which shared data is accessed, except when using coarse grain locking such as a single global lock. The semantics of transactional memory is however not fully specified by the ACI properties, since the interaction of transactional and non-transactional code is not defined [LR07]. There is not yet a consensus how the semantics of transactional memory should be described and how strong such semantics should be [MBS+08, Boe09, Luc08, SDMS08]. An often proposed semantics is *single global lock atomicity* [MBS+08], where transactions behave as if each was encompassed by a reentrant single global lock. Transactional memory systems differ in the semantics they provide as well as in their implementation.

In a TM system, a transaction can end with either a *commit* or an *abort*. A commit makes all changes of a transaction visible outside of the transaction, while an aborted transaction has no visible effects. Transactional memory systems are usually a form of optimistic concurrency control and execute multiple transactions in parallel. In case of conflicting accesses to shared resources, some of the involved transactions are aborted and possibly restarted. There are different strategies how to save the *tentative updates* of a transaction and how to *detect conflicts* among transactions. For hardware-based TM implementations (hardware transactional memory, HTM), updates are commonly buffered in a CPU-local write buffer. If a transaction is aborted, the TM system undoes any visible changes. For a HTM, this includes changes to the program counter, the register file etc. HTM implementations usually detect conflicts by recording the *read set* and *write set*, which represent the memory addresses read resp. written during a transaction. Conflict detection often does not preclude some false positives (e.g. if the granularity of conflict detection is a cache line). The time at which conflicts are detected can also vary, since it might not be feasible to detect a conflict immediately after occurrence. This is called *early conflict detection* or *late conflict detection*. A more comprehensive introduction to transactional memory can be found in [LR07].

Transactional memory was first proposed by Herlihy and Moss as a hardware extension to cache coherency protocols [HM93]. There have been many hardware based transactional memory system proposals since then (e.g. [HWC+04, MBM+06, KHR+08, BHHR]). However, transactional memory has not been implemented in available commercial microprocessors yet.[3] As an alternative approach, *software transactional memory* (STM) systems have been proposed, beginning with [ST95]. STM designates transactional memory systems implemented in software. STM systems have the benefit of running on off-the-shelf hardware. STM implementations are also more flexible, which is an advantage given the missing experience with transactional memory [LR07, GZU+09]. Hardware transactional memory (HTM) systems, i.e. systems with only a moderate software component, usually also have some intrinsic limitations due to fixed-size hardware structures and therefore cannot efficiently support transactions with a large read set and write set. Hybrid TM systems combine hardware and software implementation [DFL+06, Lie04].

1.2 Real-time transactional memory (RTTM)

Real-time transactional memory (RTTM) is a proposal of a time-predictable transactional memory by Martin Schoeberl, Florian Brandner and Jan Vitek [SBV10b]. RTTM targets chip multiprocessors with a shared memory. It is to our knowledge the first transactional memory system intended for real-time systems with hard deadlines running on multiprocessors. Other work on transactional memory for real-time systems is discussed in Section 2.2. The main design goals of RTTM are a simple programming model and analyzable timing properties. Design decisions were taken considering their impact on the worst-case execution time (WCET) estimates. High average case throughput is not a goal of RTTM.

[3]Sun's Rock processor has limited support for transactional memory.

RTTM transactions are small, i.e. they only have a small read set and write set. RTTM assumes that there are many pairs of transactions which do not conflict with each other. Static program analysis of conflicts among transactions is then used to provide competitive WCET estimates. A discussion of transactional memory variants related to RTTM is in Chapter 2.

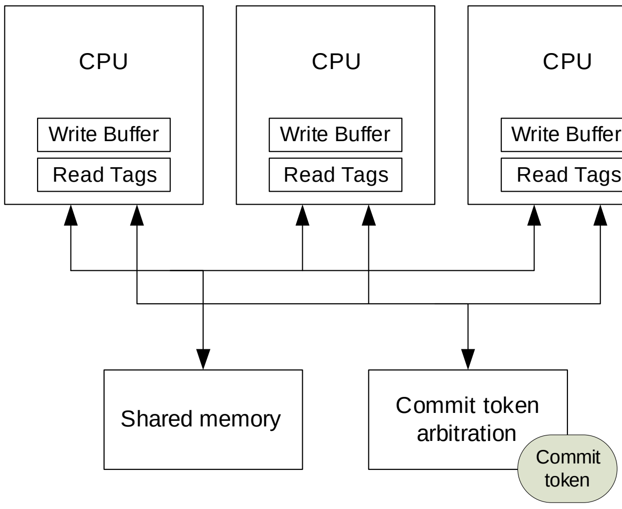

Figure 1.1: RTTM components

RTTM is a hardware-based transactional memory system. It assumes a chip multiprocessor (CMP) with a global shared memory. Each CPU of the CMP is equipped with its own *write buffer* and *read tag memory* (cf. Figure 1.1). The write buffer contains the memory addresses and data written by the corresponding CPU during a transaction (i.e. the write set of a transaction). The read tag memory contains the memory addresses read by the CPU during a transaction (i.e. the read set of a transaction). During a transaction, all writes to the shared memory are buffered in the write buffer.

For time-predictable and efficient memory accesses, the write buffer and read tag memory need to be realized in hardware. To avoid false conflicts, accesses are logged at the data word level. The caches implementing the write buffer and read tag memory are fully associative to avoid unpredictable cache usage. Since fully associative caches are expensive in terms of hardware consumption [Hyd03, Ch. 11.4.2], their size and therefore the read set and write set size are limited. The implications of this restricted size for programmers are discussed in Chapter 5 and [SBV10b].

Commit During a commit, which happens at the end of a transaction, the contents of the write buffer are written to the shared memory. The addresses being written to the shared memory are also broadcasted to all other CPUs (somewhat similar to bus snooping). This is depicted in Figure 1.2 on the next page. Before committing, a CPU $\mathcal{A}$

has to acquire the single *commit token*. During the commit, $\mathcal{A}$ holds the commit token. During $\mathcal{A}$'s commit, all other CPUs which are currently executing a transaction compare the addresses being written and broadcasted by $\mathcal{A}$ to the addresses they have read so far, i.e. their read tag memory. A match on another CPU $\mathcal{B}$ indicates an overlap of $\mathcal{B}$'s read set with the write set of $\mathcal{A}$. Any conflict between transactions will lead to such a match in one of the transactions. On a match, $\mathcal{B}$ will abort its transaction.

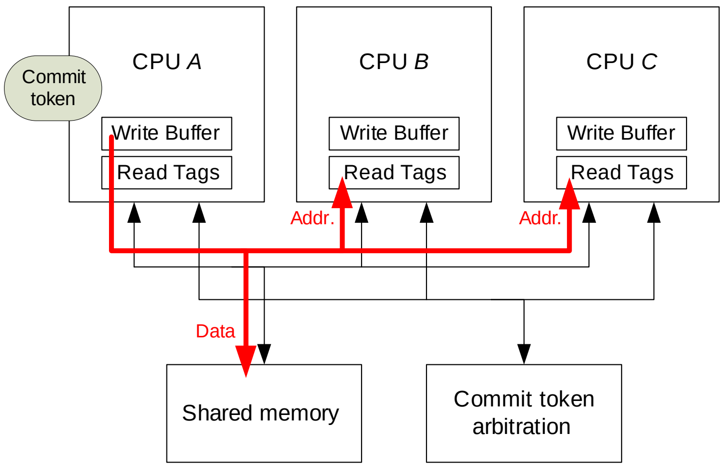

Figure 1.2: RTTM commit

Early commits The situation that the write set resp. read set of a transaction is bigger than the capacity of the write buffer resp. the read tag memory should be avoided by static program analysis. In the exceptional case that such an *overflow* occurs, an *early commit* is done: The CPU will attempt to acquire the commit token. If the CPU succeeds in acquiring the commit token, it will hold the token while continuing the transaction and broadcasting any addresses written during the transaction. The early commit phase stretches until the end of the transaction.

After the commit token has been acquired by a transaction t, no other, possibly non-conflicting, transaction will be able to commit until t has ended. Parallelism is therefore crippled by an early commit, which should however not happen frequently during normal operation. Early commits may also be used to execute I/O operations during a transaction, which usually cannot be rolled back.

From the point of view of other CPUs, there are few differences between a transaction performing a commit and one performing an early commit (disregarding temporal behavior). In both cases the commit token is held and there are writes to the shared memory and corresponding broadcasts.

Conflict detection In RTTM, conflicts are violations of the serialization of memory accesses implied by the commit order. All conflicts among concurrent transactions are detected as overlaps of the write set of some committing transaction t and the read set of another still running transaction. The conflicts are detected during t's commit. Possible overlaps of the read set and write set are depicted in Table 1.1. Two transactions t_1 and t_2, where t_1 commits first, are conflicting iff t_2 read an address from the shared memory before t_1 wrote to it.[4] In all other cases, the serialization is already implied by the commit order.

Committing transaction (t_1)	Running transaction (t_2)	Corresponding data hazard	Conflict resolution
Read	Read	—	No conflict
Write	Write	Write-After-Write	No conflict (order respected)
Write	Read	Read-After-Write	Rollback of running trans.
Read	Write	Write-After-Read	No conflict (order respected)

Table 1.1: Read set/write set overlaps

As a kind of *late conflict detection*, this can be implemented efficiently in hardware, as only a single (committing) CPU broadcasts to the other CPUs at the same time. (By contrast, *early conflict detection*, where writes to the write buffer are immediately broadcasted, requires n CPUs to listen to the $n-1$ other CPUs.) Early and late conflict detection lead to a similar WCET. When a conflict is detected, the committing transaction wins and the other transaction performs a *rollback* and a subsequent *retry* of the transaction.

In RTTM, a (failing) transaction may see inconsistent data if it read memory modified by a conflicting, committing transaction. Similar to [DS07, p. 4], we define a *zombie transaction* as "a transaction that is still running after having read an inconsistent view of global data".[5] In general, zombie transactions may behave incorrectly. Incorrect behavior may include – depending on the implementation characteristics – invalid memory accesses, exceptions, infinite loops or infinite recursions [LR07, Section 2.3.5]. An RTTM implementation must therefore provide *zombie containment* until the conflict was detected and the transaction was aborted. The global state – in particular the shared memory – will not be influenced by a zombie transaction.

WCET analysis The basic programming model of RTTM is the definition of atomic sections. Source code blocks marked with an `@atomic` annotation are executed in a

[4]A transaction performing an early commit may write multiple times to the same address in the shared memory, in which case the time of the last write is relevant.

[5]In the RTTM paper [SBV10b], the term *zombie transaction* has a slightly different meaning. It refers to "transactions that are marked as aborted, but continue to run their transaction".

transaction. Because of the limited size of the read set and write set, (tool-based) static analysis should be employed to assure that no overflows occur. If all transactions are executed as part of a periodic task, WCET analysis is possible [SBV10b]. The number of transaction retries in the worst case depends on the number of conflicting transactions. Under the simplifying assumption of the same WCET for all transactions, the maximum number of retries of a transaction is one less than the number of conflicting transactions. More complex sets of transactions lead to pessimistic but safe bounds. Static analysis should also be used to detect which transactions are not conflicting. In the RTTM paper [SBV10b], static analysis is discussed and considered viable.

1.3 The Java Optimized Processor

The *Java Optimized Processor* (JOP) [Sch05, Sch08] developed by Schoeberl is the prototype implementation platform for RTTM. JOP is a hardware implementation of the *Java Virtual Machine* (JVM) [LY99]. This means it executes Java *bytecodes*[6] directly, rather than interpreting or dynamically translating them. JOP is targeted at real-time systems and aims therefore to provide WCET bounds as low as possible. It is a simple design suitable for embedded systems. The hardware architecture of JOP strives for low resource requirements and low WCET bounds. Common case optimizations such as branch prediction or a conventional data cache have been omitted.

The processor is optimized towards the Java Virtual Machine specification [LY99] and the bytecode instruction set. As the JVM is a CISC (complex instruction set computer), the bytecodes are internally mapped to sequences of simple microcode instructions. Simple bytecodes (e.g. integer addition) are mapped to single microcode instructions. Bytecodes of greater complexity (e.g. field access) are translated to microcode sequences. Very complex bytecodes (e.g. object creation) are implemented in a subset of Java.[7] These mechanisms enable JOP, which is itself a RISC (reduced instruction set computer), to implement a complex instruction set.

As the JVM instruction set is stack-based, an efficient implementation of the stack is important for a hardware implementation. A JOP microcode instruction accesses at most the top two stack elements and a single further stack element. Therefore JOP uses registers for the top two stack elements and an on-chip RAM for deeper positions [Sch09, Section "The Stack Cache"]. The strong guarantees of the JVM also make a special form of an instruction cache feasible: a *method cache* holds entire methods, so that an instruction cache miss may only occur on an invocation or return from a method. This facilitates WCET analysis.

JOP provides a base to implement the *Connected Limited Device Configuration* (CLDC) [Sun03] subset of Java. The JOP runtime system provides a real-time profile

[6]The instruction set of the Java Virtual Machine and other virtual machines is denoted as *bytecode*.

[7]This subset of Java gets translated into other bytecodes.

similar to *Safety Critical Java* [HHL+09].[8]

Pitter has developed a chip multiprocessor version of JOP providing a shared memory [Pit09]. A priority based preemptive scheduler is provided, with a timer interrupt used for scheduling. A *memory arbiter* provides the JOP cores with access to the shared main memory. Several arbitration strategies have been analyzed. When using a time division multiple access (TDMA) based memory arbitration strategy, WCET analysis is possible.

FPGA-based systems are considered a "viable platform for CMP research" [WCN+07]. JOP runs on several low-cost FPGAs and is easy to adapt. This and several other features – related to the implementation and discussed in Section 6.3 – make JOP an appropriate platform for an implementation of RTTM.

1.4 The SimpCon SoC interconnect

The *SimpCon* interconnection standard [Sch07] is used in JOP to connect modules (e.g. coprocessors) and peripherals (e.g. a memory controller). SimpCon is designed for on-chip interconnections of system-on-chip (SoC) components and is fully synchronous.

SimpCon specifies a point-to-point master-slave connection. The slave informs the master using a *ready counter* signal (`rdy_cnt`) how many cycles are at most left to finish a read or write access (possibly an unbounded number). Through this, SimpCon supports pipelining of read and write operations. The ready counter is used for an early restart of the pipeline in the master. Different levels of pipelining are possible.

In the uniprocessor JOP, the CPU and memory controller communicate using the SimpCon standard (with the CPU acting as master). In the CMP version, each CPU is individually connected to the memory arbiter. The arbiter is in turn connected as SimpCon master to the memory controller. This is depicted in Figure 1.3 on the next page. The arbiter forwards reads and writes from the CPUs to the memory controller according to the arbitration strategy. In the RTTM implementation, the RTTM specific hardware is inserted by redirecting the SimpCon interconnects between each CPU and the memory arbiter.

1.5 Problem statement

To our knowledge, no transactional memory system for multiprocessors has been considered for use in hard real-time systems. Transactional memories for real-time systems will be discussed in Section 2.2. RTTM is a proposal of a time-predictable hardware TM for chip multiprocessors. The basic RTTM functionality has been simulated in a behavioral

[8]Safety Critical Java is JSR (Java Specification Request) 302, `http://jcp.org/en/jsr/detail?id=302`.

[9]Figure adapted from [Pit09].

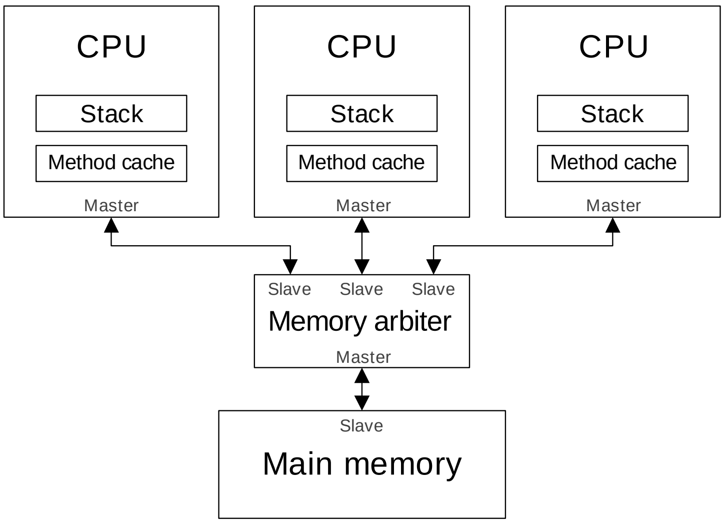

Figure 1.3: JOP chip multiprocessor. The connections use the SimpCon interconnect standard. The stack and the method cache are CPU-internal.[9]

level simulation of the JOP CMP [SBV10b, Muc09]. These simulations have indicated that the read set and write set size is small when using appropriate programming styles.

The objective of this master's thesis is the first implementation of RTTM and the evaluation of its viability. Given the high costs for a fully associative tag memory (and the limited capacity of the target, a low-cost FPGA), the resource consumption is of critical importance. Since a main goal of RTTM is a simpler programming model for parallel systems, the semantics and the limitations of transactional code are also of interest. As the RTTM proposal targets hard real-time systems, all operations in the implementation should have analyzable execution time bounds. A full, tool-based WCET analysis is considered future work.

1.6 Overview

The remainder of this thesis is organized as follows: Chapter 2 discusses related transactional memory systems and compares them to RTTM. Chapter 3 describes the semantics, the programming interface and other characteristics of the implementation. Chapter 4 describes the implementation and justifies implementation decisions. Chapter 5 gives some advice on programming using RTTM. Chapter 6 evaluates the resource consumption and performance of the implementation and RTTM in general, where possible. Chapter 7 concludes and gives an overview of the directions of future work.

CHAPTER 2

Related Work

2.1 Hardware transactional memory

First, we describe some hardware transactional memory systems which are related to RTTM. None have been considered for use in real-time systems.

An architecture for mostly functional languages

Knight [Kni86] describes a hardware system which speculatively parallelizes *sequential* code written in a functional language on a multiprocessor. It is credited with being the first paper "to use caches and cache coherence to maintain ordering among speculatively parallelized regions of a sequential code in the presence of unknown memory dependences" [LR07, Ch. 4.3.2].

The hardware organization bears similarities to many later HTM systems, including RTTM. The functionality of the *depends cache* and the *confirm cache* roughly corresponds to that of the *read tag memory* and *write buffer*, respectively. Cache coherency is maintained by the depends cache observing values written by other processors and dropping incorrect speculation. An important difference to later transactional memory proposals is that transactions are always committed in the order of the sequential code regions.

Transactional memory

Transactional memory: Architectural support for lock-free data structures [HM93] coined the term *transactional memory*. The paper proposed an extension of a MESI-style[1] cache coherence protocol ([Goo83]) to detect conflicts among transactions. New

[1]MESI stands for the possible cache line states: modified, exclusive, shared, invalid.

instructions for transactional loads and stores are introduced. A small *transactional cache* tracks memory accesses and buffers stores. The transactional cache has, in addition to a MESI state, a special cache line state. Both the original and the tentative value are stored in the cache and are dropped on a commit or abort, respectively. Conflicts are detected on their occurrence. Commit and abort are CPU-local operations, i.e. they produce no cache coherency traffic or memory traffic. The proposal requires explicit validation to detect conflicts and allows inconsistent reads. In our understanding (and as suggested in [LR07, Ch. 4.3.5]), an asynchronous abort on a detected conflict would also have been possible. In the described implementation, a transaction is aborted when it performs a conflicting write (when it tries to revoke access from another transaction). The implementation does not guarantee forward progress, but relies on software-level adaptive backoff instead. Due to cache coherency mechanisms, non-transactional memory accesses could also be executed as if each was a transaction.

Similar to RTTM, the transactional cache is fully associative. On an overflow, a transaction is aborted (unless a larger transactional cache is emulated by software). The size of transactions is therefore limited.

Transactional Memory Coherence and Consistency

Transactional Memory Coherence and Consistency (TCC) is a hardware transactional memory system [HWC+04]. All code is executed in atomic transactions. TCC supports both optimistic synchronization of parallel programs and speculative parallelization of sequential programs. While MESI-style cache coherency protocols need to support updates of small cache lines with low latency, TCC replaces this hardware by a high bandwidth broadcast bus used to commit transactions by reporting the transactions' write set. The aim of TCC is to combine the simpler hardware used for and the implicit synchronization provided by message passing systems and the easier programming model offered by shared memory systems without relaxed consistency.

TCC and RTTM have a similar mechanism to guarantee atomicity and isolation of transactions. Writes are buffered locally and the read set is tracked through the cache. To commit, a processor tries to obtain the global commit token and, if successful, broadcasts the write set of the transaction to all other processors over a high bandwidth bus. These bandwidth requirements limit the scaling of TCC. The broadcast may also include the modified data, so that caches can be updated directly. However, this requires a higher broadcast bandwidth [HWC+04]. Conflict detection is similar to RTTM: If another processor detects an overlap of the broadcasted write set with its read set, it will perform a rollback.

When using TCC, the programmer inserts *transaction boundaries* into the code. The only requirement for error-free execution in the value domain is that "transaction breaks should never be inserted during the code between a load and any subsequent store of a shared value (i.e. during a conventional lock's critical region)" [HWC+04]. Other inappropriate choices of transaction boundaries may however lead to frequent rollbacks and degraded performance. In general, transactions should be large, as long as conflicts are not frequent and the read set and write set do not become too big. When choosing transaction boundaries, it is possible to trade off development effort and performance. Transaction boundaries may also be inserted automatically. To preserve partial atomicity in this case, the programmer needs to mark sections where no boundary may be inserted.

Optionally, transactions may be ordered by assigning *phase numbers* to them. A transaction may only commit once all transactions with older phase numbers have been committed. If used, phase numbers are included in the broadcasts. Phase numbers may be used to avoid starvation of long transactions.

While it bears many similarities to RTTM, TCC is not geared towards time predictability. While RTTM expects the programmer to use small atomic sections for synchronization only, TCC executes all code in transactions. TCC hereby also avoids issues in the interaction of transactional and non-transactional code.

In simulations of a range of server applications adapted for TCC, the read set size and write set size was determined to be in the order of 6-12 KiB resp. 4-8 KiB for almost all transactions *with 64 Byte cache lines*.[2] There were a number of transactions and benchmarks with higher demands, however. Due to the need for fully associative tags, RTTM only supports smaller read sets and write sets. When using RTTM, the programmer only marks small code sections as transactional. In the real-time domain, the read set and write set size is also expected to be smaller [SBV10b]. While TCC's conflict detection has cache line granularity, RTTM has data word granularity to avoid false positives. In a TCC implementation, caches are likely not fully associative, but a victim buffer is used instead [MCC+05]. Similar to RTTM, an overflow will trigger an early commit, which limits parallelism. In RTTM, an early commit should be excluded by performing static analysis of the read set size and write set size.

Interestingly, there is also an FPGA-based prototype implementation of TCC (using a high-performance multi-FPGA board) [WCN+07]. This was the "first FPGA-based framework for research on CMPs with hardware support for transactional memory" [WCN+07]. The prototype is intended as a replacement of slow simulation; some FPGA-specific implementation issues are reported. We will briefly discuss similarities and differences of the implementations in Section 4.2.3 and Section 4.2.4.

2.2 Transactional memory for real-time systems

To our knowledge, transactional memory implementations which are considered suitable for real-time systems have so far been limited to software transactional memory. These transactional memory implementations are restricted to single core processors, with the exception of [SQV09], which targets soft real-time systems. According to

[2]We did not find data on how many words were actually changed.

[SQV09], “most of [the] existing solutions for real-time scheduling consider either tasks in multiprocessor systems or transactions in database systems, but not both together”.

Preemptible Atomic Regions for Real-time Java

Preemptible Atomic Regions (PARs) [MBC+05] is a concurrency control abstraction for real-time systems. In order to minimize blocking time, PARs can be preempted by higher priority tasks. On preemption, the effects of a PAR are undone. PARs are a restricted form of software transactional memory. The original PAR design is for uniprocessors only. PARs were motivated by the possibility of interference between the non-real-time and real-time code in an RTSJ (*Real-time Specification for Java* [GB00]) environment. The authors report that, depending on semantics, programs can run faster and experience less jitter when using PARs instead of locks.

Since PAR is for uniprocessors, code within a PAR can update memory in place. On each write, address and original contents of the written location are also recorded in an *undo buffer* (undo log). A commit only resets a pointer to the undo buffer. When a PAR is aborted due to preemption, the original contents recorded in the undo buffer are restored in reverse order. Similar to RTTM, it is assumed that there are a limited number of writes in a PAR. While the undo buffer is maintained in ordinary memory, it must be small enough for a quick rollback on preemption. Since a PAR is aborted on every context switch, at most one PAR is aborted on a context switch. The worst-case blocking time is determined by the maximum number of writes performed in a PAR.

PARs avoid several issues of lock-based mutual exclusion: deadlocks, violation of isolation due to programmer errors and lock acquisition overhead. PARs also compose easier than locks and might require fewer context switches.

Similar to the RTTM implementation, the basic programming interface is a method annotation. PARs are an STM variant for uniprocessors, while RTTM is an HTM system for CMPs. While PARs were devised to minimize blocking time for higher priority tasks, there is no notion of priority in RTTM.

Inside a PAR, duplicate methods which maintain the undo buffer through additional instructions are invoked instead of the original methods. The PAR implementation does also target a real-time Java Virtual Machine. Similar to the RTTM implementation, the PAR implementation uses the exception mechanism of Java to abort a PAR and does not track writes to local variables. Some issues were encountered when integrating the PAR semantics into the (virtual) machine. Some modifications of the virtual machine kernel state should not be undone when aborting a PAR. Virtual machine kernel code is therefore compiled without logging. The exception handling mechanism was modified to specially treat exceptions aborting a transaction. Other integration issues are also described in the paper. Similar to RTTM, there are restrictions to the code executed in PARs. On an undo buffer overflow, interrupts are disabled, which is similar to an early commit.

Other related real-time capable work

Real-Time Support for Software Transactional Memory (RT-STM) [SQV09] is, to our knowledge, the only transactional memory system besides RTTM where concurrent real-time transactions run on a multicore. The paper focuses on the scheduling of these transactions. While RTTM strives to guarantee deadline compliance, RT-STM reduces the number of deadline violations of soft real-time transactions. The paper formalizes real-time transactions and introduces a deadline-based real-time scheduler. Existing STMs are adapted and the performance with different real-time scheduling policies is compared.

Supporting lock-free synchronization in Pfair-scheduled real-time systems [HA06] bounds the number of retries when using lock-free synchronization on multiprocessor systems using *proportionate fairness* real-time scheduling. Such lock-free operations are however not intended as a replacement for locking synchronization. Instead they should allow more efficient implementation of simple data structures, such as queues. Assumptions regarding the behavior of the lock-free algorithms are also made for the analysis: when a lock-free operation is retried and how long the operation takes at most.

Response time analysis of software transactional memory-based distributed real-time systems [FRJ09] considers distributed systems where each node executes separated STMs. An algorithm to bound the response time is presented. There is also work on real-time database transactions (e.g. [ARMJ97]) and lock-free data structures for real-time systems (e.g. [HA06], [ARJ97]).

CHAPTER 3

Characterization of implementation

Transactional memory systems differ greatly both in their semantics and in their implementation [LR07, Ch. 2]. In the following, we will describe the semantics, the programming interface and the implementation characteristics of the RTTM prototype implementation developed as part of this thesis. The paper proposing RTTM [SBV10b], as presented in Section 1.2, does not specify details of the behavior and leaves some decisions to the implementation.

Table 3.1 on the following page explains some terms used in the following which were not introduced yet. With respect to other terminology, the thesis follows [LR07].

3.1 Semantics

RTTM transactions satisfy the ACI semantics – failure atomicity, consistency and isolation – that is the key property of transactional memory systems [LR07, Ch. 1]. ACI semantics have been introduced in Section 1.1. In Section 1.2, esp. page 5, I have argued that RTTM satisfies these properties. Atomicity is fulfilled because each transaction performs its changes only in a non-interruptible step (during the commit/early commit), if at all. Isolation is fulfilled because there is only a single commit token passed in the serialization order of transactions. In the current implementation, transactions have *exactly-once semantics* (and not at-most-once semantics). In the temporal domain, the number of retries of a transaction is bounded by the number of conflicting transactions (see page 6).

If a program is *segregated*, i.e. “all mutable shared memory locations are accessed either exclusively inside or exclusively outside a transaction” [MBS$^+$08], there are no problems with the interaction of transactional and non-transactional code [MBS$^+$08]. If

Term	Definition
Commit/Committing transaction	From the point of view of other CPUs, there are few differences between a committing and an early committing transaction (disregarding temporal behavior). In both cases the commit token is held and there are writes to the shared memory and corresponding broadcasts. Therefore the term commit resp. committing transaction also refers to an early commit resp. early committing transactions, unless otherwise noted.
Preceding transaction	All successful transactions are totally ordered by the order in which they acquire the single commit token. A transaction t_2 is *preceded* by another transaction t_1 iff t_2 has not yet acquired the commit token when t_1 does.
Succeeding transaction	A transaction t_2 succeeds another transaction t_1 iff t_2 acquires the commit token after t_1.
Nested transaction	Any transaction which is executed as part of another transaction.
Doomed transaction	A transaction t_d which is conflicting with a *preceding transaction* t_1 (i.e. t_d read from an address which will be broadcasted by t_1). t_d will fail (at the latest) during t_1's commit.
Current write set/ *Current* read set	The shared memory addresses written/read by a transaction until a certain point in time.

Table 3.1: Glossary of used terms

a program is not segregated, the semantics are more involved. In the following, we will discuss the semantics w.r.t. the interaction of transactional and non-transactional code.

RTTM implements the $write \overset{hb}{\rightarrow} read$ model presented in [GMP06, Section 3.3]. In this model, the *happens-before* (hb) relationship between transactions only contains happens-before edges from transactions writing a certain memory location to transactions reading a certain memory location. In RTTM, all transactional reads and writes satisfy the happens-before relation, as illustrated in Table 1.1 on page 6. The $write \overset{hb}{\rightarrow} read$ model supports the *data handoff* idiom outlined in Table 3.2 on the facing page. The RTTM model does also support a number of interactions between transactional and non-transactional code which are not supported by the $write \overset{hb}{\rightarrow} read$ model [GMP06, Section 3.4].

Transactional memory systems are said to provide *strong isolation* if each access during non-transactional execution does behave as if the access was executed in an individual transaction [LR07]. Many software TM systems do not provide strong isolation, since it apparently has high overheads [SMDS07]. RTTM also does not provide

Initially, ready = **false**

Thread 1	Thread 2
`data = 42;`	
`atomic {`	
`  ready = true;`	`atomic {`
`}`	`  tmp = ready;`
	`}`
	`if (tmp) {`
	`  r1 = data;`
	`}`

Table 3.2: Example of *data-handoff* from [GMP06]. `tmp == true` $\Rightarrow$ `r1 == 42`

strong isolation. *Weak isolation* only isolates transactions from each other. In the case of weak isolation, one could demand that transactional and non-transactional accesses to a memory location must not overlap in time [SMDS07]. This requirement implies a partition into *shared* objects and objects *private* to some thread at any point in time [SMDS07]. A transaction may conduct *privatization* of an object O by modifying the shared data such that O will not be accessed by any subsequent transactions (e.g. remove an item from a linked list). This may however lead to *privatization problems*, which occur in some software TM systems [SMDS07]. In [MBS$^+$08, Section 3], *privatization safety* is defined as the requirement "that an STM must respect a happens-before ordering relation from a transactional access S1 to a conflicting non-transactional access S2".[1] In the case of RTTM, the happens-before ordering relation is established by a transaction in the same task as S2 (preceding S2). This is depicted in Table 3.3 on the next page. RTTM provides privatization safety.[2]

Publication safety is defined as the requirement "that an STM must respect a happens-before ordering relation from a non-transactional access S1 to a conflicting transactional access S2" [MBS$^+$08]. In RTTM, the happens-before ordering relation is established by a transaction in the same task as S1 (following S1). This is depicted in Table 3.4 on the following page. Publication safety is not supported by RTTM, since data races can occur (see the example in Table 3.5 on the next page).

We think that RTTM also implements *encounter-time lock atomicity* [MBS$^+$08, Section 7]. In a "semantically equivalent lock-based [execution] where each transaction is protected by some minimal set of locks such that two transactions share a common lock if and only if they conflict", these locks are acquired "at any point before the

[1]In this context, *conflicting* means that the transactional and non-transactional access go to the same memory location.

[2]RTTM is not affected by privatization problems, since it uses deferred updates and a single commit token. The use of a single commit token implies *commit linearization*, which provides privatization safety [MBS$^+$08, Section 4.1.2].

Thread 1	Thread 2
`atomic {`	
`  S1;`	
`}`	
	`atomic {`
	`  // privatization`
	`}`
	`S2;`

Table 3.3: Privatization safety (from [MBS+08]). `S1` and `S2` are conflicting.

Thread 1	Thread 2
`S1;`	
`atomic {`	
`  // publication`	
`}`	
	`atomic {`
	`  S2;`
	`}`

Table 3.4: Publication safety (from [MBS+08]). `S1` and `S2` are conflicting.

Initially data = 42, ready = **false**, val = 0

Thread 1	Thread 2
	`atomic {`
	`  tmp = data;`
`data = 1;`	
`atomic {`	
`  ready = true;`	
`}`	
	`  if (ready)`
	`    val = tmp;`
	`}`

Table 3.5: Data race during publication (from [MBS+08]). `val == 42` is allowed by the RTTM semantics.

corresponding data is accessed” [MBS+08]. Encounter-time lock atomicity does support *publication in a conditional* (see Table 3.6).[3] [MBS+08] lists some idioms stronger than encounter-time lock acquisition which are not supported by the RTTM implementation.

Initially data = 42, ready = **false**, val = 0

Thread 1	Thread 2
`data = 1;`	**`atomic`** `{`
`atomic` `{`	`  `**`if`**` (ready)`
`  ready = `**`true`**`;`	`    val = data;`
`}`	`}`

Table 3.6: Publication in a conditional (example from [MBS+08]) is supported by RTTM. `val != 42`

Since the RTTM implementation tracks memory accesses at the word level, there are no problems with the granularity of transactional/non-transactional memory accesses, i.e. *granular safety* [MBS+08] is provided. *Observable consistency* [MBS+08] and *speculation safety* [MBS+08] is supported.[4] For an in-depth discussion of the implications of the various safety properties supported or not supported by RTTM, see [MBS+08].

Stronger semantics? More publication patterns could be supported if *all* writes to the shared memory would be broadcasted (which would be feasible in the prototype implementation). I assume that the semantics would then correspond to *asymmetric lock atomicity* [MBS+08]. The possible abort of transactions by non-transactional code would need to be accounted for in the WCET analysis, however. A notable property of RTTM is that conflicts are asymmetric, i.e. if transaction t_1 possibly aborts t_2, this does not imply that t_2 possibly aborts t_1. A transaction aborts automatically only if its read set overlaps with the write set of a preceding transaction. This prevents a behavior more similar to *single global lock atomicity*, where the program executes as if each transaction was protected by a single global lock.

An RTTM implementation could support *strong isolation* (for data types residing in a single memory word),[5] if, in addition, commits happened atomically. Else, one read could already see the modifications of a transaction and a successive read could see a value not yet updated by the transaction. As an alternative, static analysis could be considered to ensure segregation into transactional and non-transactional memory.

[3]Speculative code motion is then disallowed. The RTTM implementation does not use compiler or hardware reordering.

[4]Observable consistency and speculation safety are supported because the update of the shared memory is deferred.

[5]Only `long` and `double` types do not reside in a single memory word.

Transaction nesting The implementation allows transactions to be nested. Nested transactions are *flattened*, i.e. aborting a nested transaction causes an outer transaction to abort, and committing a nested transaction has no effect. To support not flattened transactions, a more advanced write buffer and read tag memory would be necessary.

Exceptions Exceptions always abort, rather than commit, the current transaction. Terminating exceptions, which attempt to commit the transaction, are not supported. Handling of exceptions inside a transaction (`catch` or `finally` blocks) is also currently not supported, since it interferes with the transaction abort mechanism.

3.2 Implementation characteristics

In this section, we characterize the implementation techniques used for the RTTM prototype. This is mostly done with regard to the taxonomy presented in [LR07, Ch. 2.3]. Most characteristics follow from the RTTM proposal.

Granularity of conflict detection RTTM conflict detection has *word granularity*, detecting conflicting accesses to a memory word. In the implementation platform JOP, all Java data items (such as elements of a `boolean[]`) and all implementation specific data structures modified in a transaction occupy one or more dedicated memory words. This excludes false positives. The conflict detection is currently largely language agnostic.

Direct or Deferred Update RTTM uses *deferred update* (the write buffer) during normal operation, as most HTM systems do. Memory words are *updated in place*. The single commit token makes updating the shared memory easy. On a buffer overflow/early commit, direct update is used.

Concurrency control RTTM uses *optimistic concurrency control*. W.r.t. *progress guarantees*, RTTM provides *wait freedom* if there is a single thread per CPU and as long as certain schedulability conditions are met [SBV10b] (and as long as no thread stalls while performing an early commit).

Conflict Detection RTTM detects conflicts among transactions *late* (i.e. not when they appear), during *validation* of a transaction. For each transaction, a read set and a write set is maintained, which remains private to the transaction. RTTM uses *lazy invalidation*, as defined in [Sco06]. According to [Sco06], this is the weakest consistency-ensuring conflict definition.

```
public class RingBuffer<T> {
  // ...

  @atomic public T read() {
    if (rdPtr == wrPtr) {
      return null;
    }
    T val = data[rdPtr++];
    if (rdPtr == data.length) {
      rdPtr = 0;
    }
    return val;
  }
}
```

Listing 3.1: Example of atomic method

Contention management The *contention resolution policy* of RTTM is to always abort the conflicting transaction(s) which is (are) not committing. As argued in [SBV10b], the execution remains time-predictable if certain schedulability conditions are met.

3.3 Programming interface

Transactions are created by the programmer using an `@atomic` *method annotation*, as depicted in Listing 3.1. The `@atomic` annotation is the sole interface needed to use the functionality of RTTM, providing a simple interface for programming parallel systems.[6] The annotation provides the code executed in and invoked by the method with semantics as discussed in Section 3.1. Most Java language features are supported inside a transaction, as will be discussed in Section 5.2. The restriction to entire methods instead of code blocks is a pragmatic decision.[7] Annotated methods are modified at link time. A drawback of the restriction to *atomic methods* is that it may lead to fragmentation of code.

As an alternative, transactions may also be implemented using a special low-level method directly accessing the memory-mapped RTTM hardware interface, which will be described at page 31. An example is in the appendix (Listing B.2 on page 76). I believe that the use of a language feature is substantially less laborious and less error-prone.

[6]Since Java method annotations do not alter the method signature, they can be transparently added without compromising an interface.

[7]Code transformations are simpler and most local variables do not need to be restored on a transaction retry, as detailed in Section 4.3.1.

3.3.1 Software commands

RTTM *software commands* provide additional functionality to the programmer. Software commands are invoked as Java methods. Their use can complicate the programming model of RTTM.

`retry()` The `retry()` statement, as introduced in [HMPJH05], lets the programmer roll back and restart the current transaction, as is also done transparently by RTTM if a conflict is detected. `retry()` can serve as a mechanism to coordinate transactions. [HMPJH05] proposes to delay the transaction restart until the data accessed has been changed. In an RTTM transaction, such a behavior can be attained by looping until an anticipated condition becomes true: `while (!condition);` . If a shared memory value evaluated in the condition is updated, the transaction will automatically retry.[8]

If an *early commit* was initiated earlier because of a buffer overflow or by the application, any writes before the `retry()` software command were already written to the shared memory and cannot be undone, violating the atomicity of transactions.[9]

`abort()` The `abort()` statement aborts the current transaction. It performs a rollback of all changes in the transaction and then exits the atomic method throwing a Java exception.[10] It has not been investigated which criteria should be used to abort a transaction. As in the case of the `retry()` statement, if an early commit was initiated earlier, any writes before the `abort()` software command will violate the atomicity of transactions.[11]

`earlyCommit()` An *early commit* is tried *a*) if the read tag memory or write buffer overflows or *b*) upon executing I/O operations (which usually cannot be rolled back). The implementation does not automatically try to commit before an I/O operation. Instead, the programmer needs to manually invoke the `earlyCommit()` software command before an I/O operation. Listing 3.2 on the next page shows an example use of the `earlyCommit()` software command to output an atomic snapshot. Because it blocks other transactions while generating output, this style of use is at most suited for diagnostic purposes.

If the `earlyCommit()` software command succeeded,[12] a subsequent abort of the transaction might violate the atomicity of the transaction, since all writes go directly to the shared memory during an early commit. An abort might happen *a*) if an exception terminating the current transaction is thrown due to a programming error or *b*) might be caused by the application (using the `abort()` or `retry()` software command).

[8]This programming style will not work if an early commit has been performed in the transaction.

[9]`retry()` could be ignored during an early commit.

[10]As the `AbortException` is an unchecked exception derived from `RuntimeException`, an atomic method is not required to declare it in the `throws` clause.

[11]`abort()` could also be ignored during an early commit.

[12]Impossible for a transaction failing due to a conflict, since it will never acquire the commit token.

```java
@atomic public void snapshot() {
  // grab commit token
  rttm.Commands.earlyCommit();
  // can do I/O now

  System.out.println("Linked list contents:");
  // iterate through linked list
  for (LinkedObject o = head; o != null; o = o.getNext()) {
    System.out.println(o.getData());
  }
}
```

Listing 3.2: Example use of `earlyCommit()` software command

3.3.2 Diagnostics

For diagnostic purposes some basic per-CPU statistics, including the transaction commit and retry count and the maximum read set and write set size, may be measured without a probe effect.[13]

3.4 Scheduling

Context switches during a transaction are currently not possible. While executing a transaction, all interrupts are disabled. It would be straightforward to support context switches/interrupts during a transaction. Such a context switch/interrupt would simply abort the current transaction.[14] Transactions violating a deadline could then be aborted.

[13]See class `rttm.Diagnostics`.

[14]In JOP, interrupts are issued in a manner similar to exceptions (see page 53). In the RTTM implementation, Java exceptions are used to abort a transaction (see Section 4.3.2). When switching back to the thread running the interrupted transaction, an exception could be thrown to retry the transaction.

CHAPTER 4

Implementation

The RTTM prototype is implemented on the *Java Optimized Processor* (JOP), a *Java Virtual Machine* in hardware (see Section 1.3). RTTM is a hardware transactional memory system and language independent, as most HW TM systems [LR07, Ch. 4.1]. The prototype implementation does however rely on some of the strong guarantees and features of the Java programming language and the Java Virtual Machine (JVM), as will be discussed in the following.

Significant parts of the functionality are implemented in software, which is also eased by the properties of the implementation platform. Figure 4.1 on the following page shows the layers in the RTTM implementation. The interfaces between them and the layers' implementation will be discussed in this chapter.

4.1 HW/SW-Partitioning

Next, we justify the partitioning in hardware and software and give an overview of the RTTM implementation. Memory accesses and conflict detection need to be implemented in hardware to be time-predictable and efficient. If the flush of the write buffer during the commit was to be performed by the CPU, additional HW-SW interfaces would be needed. On the other hand, the abort of transactions is currently implemented using the exception handling mechanism of Java – similar to the approach of the *Preemptible Atomic Regions* software TM [MBC$^+$05]. The implementation does not introduce new CPU instructions.

The hardware part of RTTM has a simplified view of transactions. The hardware ignores the control flow associated with a transaction abort and has no concept of the retry of a transaction or of transaction nesting. The decision to keep the hardware simple has the benefit of keeping the interface between the CPU and the RTTM-specific hardware

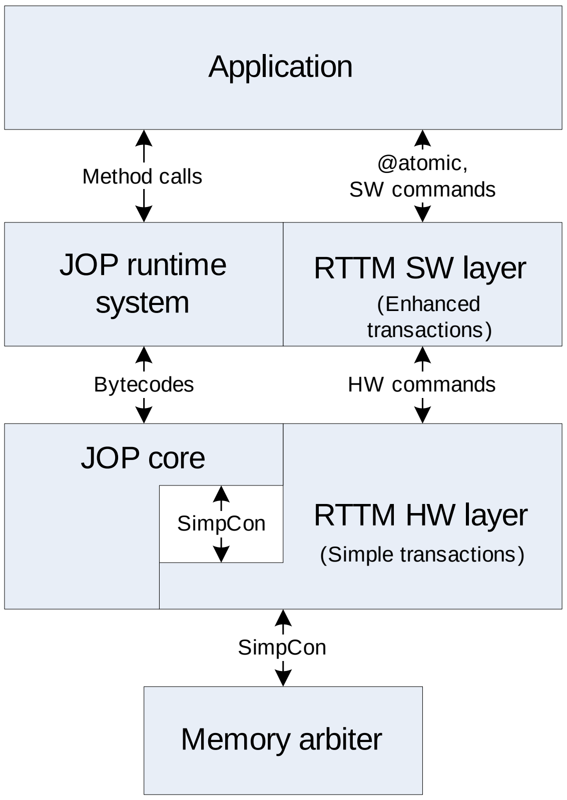

Figure 4.1: RTTM implementation layers

simple and of requiring no changes to the CPU. It also avoids race conditions between the CPU and RTTM which would make maintaining the nesting count consistent difficult.

The software part of RTTM is implemented in a *transaction wrapper* which is transparently added to *atomic methods* at link time. The transaction wrapper implements *a*) the abort and retry of transactions and *b*) the nesting of transactions and *c*) saves and restores relevant CPU state. The use of software makes the implementation more flexible, e.g. w.r.t. the retry/abort strategy and the handling of Java exceptions.

4.2 Hardware layer

RTTM has been implemented by extending the CMP version of JOP (see page 8 and [Pit09]). In the conventional JOP CMP, each CPU is individually connected to the memory arbiter, as shown in Figure 4.2 on the next page. I/O is individually connected to each CPU.

[1]Figure adapted from [Pit09].

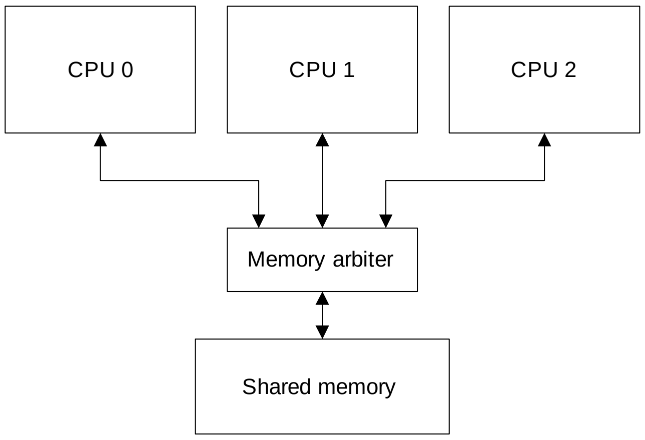

Figure 4.2: Conventional JOP CMP components[1]

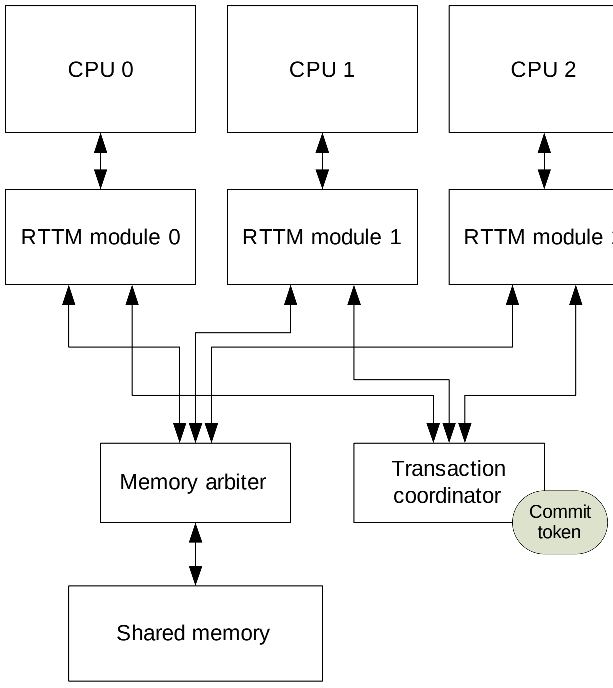

Figure 4.3: Hardware components related to RTTM implementation

In the RTTM implementation, depicted in Figure 4.3 on the preceding page, an *RTTM module* has been inserted between each CPU and the arbiter. The RTTM module comprises most of the RTTM functionality implemented in hardware. It contains the *write buffer* and *read tag memory* introduced in Section 1.2. The *transaction coordinator* grants the single *commit token* to an RTTM module requesting it.[2] An RTTM module which has been granted the token holds the token until it has completed the transaction.

The RTTM implementation does not introduce new (JVM bytecode or microcode) instructions. Communication of the RTTM SW layer with the RTTM module is memory-mapped.[3]

4.2.1 The RTTM module

The interfaces of an RTTM module are depicted in Figure 4.4 and will be described next.

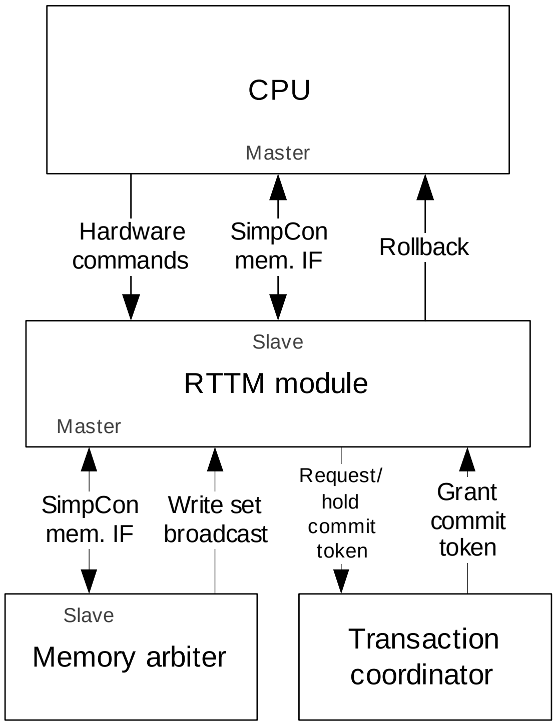

Figure 4.4: Interfaces of an RTTM module

[2]In case of conflicting requests, the commit token is currently granted to the RTTM module with the lowest CPU ID.

[3]As the Java language is memory safe, JOP-specific special bytecodes are used to access memory addresses mapped to the RTTM module.

HW Command	Description
`start_transaction`	Start transaction
`end_transaction`	Attempt to obtain commit token and commit
`early_commit`	Programmer tries to obtain commit token for the remainder of the transaction using the `earlyCommit()` software command
`aborted`	Ends a hardware-side transaction. Either the RTTM SW layer confirms that it has interrupted the (doomed) transaction or the programmer performed an `abort()` or `retry()` software command.

Table 4.1: Hardware commands

Hardware commands The hardware commands *start*, *commit* or *abort* a transaction in the RTTM module. Each atomic method body is transparently supplemented by a *transaction wrapper* at link time. The transaction wrapper, executing on the CPU, transparently issues the memory-mapped hardware commands.[4] Some hardware commands may also be issued by the application (indirectly, using the *software commands* presented in Section 3.3.1), but this is not needed in a basic transaction.[5] The hardware commands are listed in Table 4.1 and will be described in the context of their use in the RTTM module state machine.

SimpCon memory interface Memory accesses are handled by the SimpCon SoC interconnect introduced in Section 1.4. Table 4.2 on the following page lists the SimpCon signals used to connect *a*) the CPU (master) and RTTM module (slave) and *b*) the RTTM module (master) and memory arbiter (slave). The following signals are RTTM-specific:

- The `tm_cache` flag is only used on the CPU to RTTM module interconnect. If cleared, it indicates that a memory access should bypass the read tag memory and write buffer. This flag can be cleared for several types of memory accesses (see Section 4.2.10) and is important because of the restricted size of fully associative caches.

- The `tm_broadcast` flag is only used on the RTTM module to arbiter interconnect. It indicates that an address being written to the shared memory should be

[4]Hardware commands do not actually use dedicated signals, but are mapped into an unused part of the memory address space which is routed through the RTTM module.

[5]As a not recommended alternative, the hardware commands may also be issued directly (see Listing B.2 on page 76 in Appendix B).

Signal	Width	Set by	Description
`address`	23	Master	Memory address[6]
`wr_data`	32	Master	Data written
`rd_data`	32	Slave	Data read
`rd`	1	Master	Start read access
`wr`	1	Master	Start write access
`rdy_cnt`	2	Slave	Cycles until memory access finishes (3 ≡ unbounded; used for pipelining)
`tm_cache`	1	Master (CPU)	Indicates if memory access is tracked (only on CPU to RTTM module interconnect)
`tm_broadcast`	1	Master (RTTM module)	Indicates memory address broadcast (only on RTTM module to arbiter interconnect)

Table 4.2: SimpCon signals in RTTM implementation

broadcasted to all other RTTM modules, as described on page 4.

Write set broadcast If an RTTM module issues a write to the memory arbiter with the `tm_broadcast` flag set, the arbiter will broadcast the write address to all RTTM modules. These will check for overlaps with their *current read set* (if executing a transaction). Broadcasts and shared memory reads are scheduled such that any conflicts are detected and no false conflicts are detected.

Rollback If the RTTM module detects a conflict to a committing transaction, it sets the `rollback` signal to initiate a rollback in the software layer.

Request/hold commit token & **Grant commit token** The signal to *request/hold* the commit token is set by the RTTM module when (i) the CPU issues an `end_transaction` or `early_commit` HW command or (ii) if the read tag memory or write buffer overflows (see page 5). The request/hold signal remains set until *a*) the transaction is aborted (due to a conflict or upon an explicit request by the application) or *b*) until the transaction has been committed. In the case *b*), the RTTM module has been *granted* the commit token by the transaction coordinator before committing. When the request/hold signal is cleared, the commit token returns to the transaction coordinator.

[6] 23 bits is the typical JOP address space width.

4.2.2 The memory arbiter

The RTTM implementation adapts the memory arbiter of the JOP CMP. For the JOP, multiple memory arbiter versions implementing different arbitration policies are available [Pit09]. A time division multiple access (TDMA) policy was found to be best for a time-predictable processor in [Pit09]. This policy was adapted as follows for RTTM: When a transaction performs a commit, all writes to the shared memory are broadcasted to the other RTTM modules. During the commit, all TDMA slots of processors executing transactions are temporarily reallocated to the committing processor. This reallocation of slots limits the inconsistent state observed by conflicting transactions, as will be discussed in Section 4.3.3.

Memory arbiter behavior requirements The RTTM implementation makes some assumptions about the order in which memory accesses are performed by the memory arbiter and the shared memory to guarantee detection of conflicts between transactions. If an RTTM module $\mathcal{M}$ has not found a *broadcasted* address $\mathcal{A}$ in its *current read set*, as tracked through the read tag memory – i.e. no conflict has been detected –, a read of $\mathcal{A}$ by $\mathcal{M}$ must return the new value written to $\mathcal{A}$ (or an even newer value).

The requirement is satisfied if the following conditions are met: *a*) All (read or write) memory accesses are serialized at some point in the memory arbiter, which precedes the point where writes are broadcasted, if applicable. *b*) Consider a serialization where the read access is scheduled by the memory arbiter before the write access/broadcast. This is the only possible conflict due to *a*), i.e. the only case where the old value of $\mathcal{A}$ is returned.[7] The read tag memory must now already have been updated w.r.t. the access to $\mathcal{A}$ at the time when the broadcast is processed. If so, the conflict will be detected.

A written address must also be broadcasted by the memory arbiter at the latest in the cycle when it sets `rdy_cnt` below 3 (to guarantee conflict detection in case the commit token is released immediately afterwards and granted to a conflicting transaction, which is waiting for the commit token).

These conditions are met by the *transaction cache* implementing the read tag memory and write buffer and by the used TDMA arbiter and their interconnect.

4.2.3 Influence of target technology

The RTTM prototype implementation targets *field programmable gate arrays* (FPGAs). FPGAs have a predefined structure and therefore logic feasible on *application-specific integrated circuits* (ASICs) might not synthesize efficiently on FPGAs [WCN⁺07]. RTTM requires fully associative tags to track the read set and write set. In the case of the read set, the tag memory is necessary to detect conflicts to the committing transaction. In

[7] See also the data hazards in Table 1.1 on page 6.

the case of the write set, the tag memory is necessary when values written during a transaction are read again. Full associativity is required to avoid tag memory overflows. Fully associative tags are expensive to implement in hardware [Hyd03, Ch. 11.4.2], since each tag needs to be compared to (a part of) the accessed address.

The main target for the RTTM prototype implementation is the Altera Cyclone II EP2C70 FPGA. This medium-size low-cost FPGA provides ~68000 *logic cells* (LCs) and in addition *memory blocks* with a total capacity of ∼1 Mibit. Each logic cell supports combinatorial logic through a 4 bit lookup table and can save the result in a single flip-flop. Since each logic cell can hold at most 1 bit of memory, large memories cannot be efficiently implemented using logic cells. The memory blocks can implement memories of different word size and capacity. Each of the 250 memory blocks on the EP2C70 has a capacity of 4 Kibit. One or more memory blocks can serve as the main component of various types of memories such as multi-port RAMs and FIFOs. These memories are either instantiated using intellectual property blocks provided by the manufacturer or during hardware synthesis (if appropriate coding styles are used).

In order to hold the addresses and the data in the read tag memory and the write buffer, an RTTM implementation needs a few Kibit of memory per processor core. In the typical case that shared memory addresses are 19 bit wide and data words are 32 bit wide, a write buffer with 64 entries needs $64 \times 19 = 1216$ bits of memory for the tag memory and $64 \times 32 = 2048$ bits of memory for the buffered data. The RTTM implementation uses FPGA memory blocks whenever feasible. All the memories, except the tag memory, are implemented using memory blocks.

4.2.4 Tag memory implementation

The fully associative tag memory needed for both the read tag memory and the write buffer is the most resource consuming part of the RTTM implementation (cf. measurements in Table 6.2 on page 62). Each tag needs to be compared for hit detection. The functionality of fully associative tags corresponds to those of a simple *content addressable memory* (CAM), where the memory deduces storage addresses from data words. Current FPGA technology seems not to be well suited for CAM/fully associative tags implementation [WCN+07, GLD00]. One of the challenges encountered in the FPGA-based implementation of the TCC prototype [WCN+07, Section 5] was the implementation of memory structures with high associativity.

The fully associative tag memory cannot directly be implemented using memory blocks, since each tag needs to be compared for hit detection. When using only logic cells, in the typical case mentioned in the last section, the tag memory occupies at least 1216 logic cells, since each only contains a single flip-flop [GLD00, Section 3]. The actual implementation[8] needs 1630 LCs in this case, and hit detection for at least 64

[8]The implementation of the tag memory was adapted from a prototype created by Martin Schoeberl.

```
  l <= (others => '0');
  for i in 0 to lines-1 loop
    if h(i)='1' then
      l <= to_unsigned(i, way_bits);
      exit;
    end if;
  end loop;
```

Listing 4.1: Line encoder with priority. h, an array with the tag comparison results, is encoded to l.

```
  l <= (others => '0');
  for i in 0 to way_bits-1 loop
    for j in 0 to lines-1 loop
      n := to_unsigned(j, way_bits);
      if n(i)='1' and h(j)='1' then
        l(i) <= '1';
      end if;
    end loop;
  end loop;
```

Listing 4.2: Line encoder without priority. h is encoded to l.

entries is feasible in a single cycle (see Section 6.1).[9] Since the write buffer and read tag memory are invalidated at the end of each hardware transaction or on a buffer overflow, a simple "replacement" strategy using a strictly increasing counter is used.

A subtle issue is the encoding of the lines once the tags have been compared. Since at most one tag matched and was valid, there is no need for a priority encoder (cf. Listing 4.1 and Listing 4.2).

Comparison to implementation using memory blocks A fully associative tag memory of considerable address width and depth can be implemented in FPGA memory blocks using a special technique [auR08, BG02]. The saved addresses are encoded in the memory blocks in a sparse manner.

In the simple variant, the width of the memory word read from a memory block corresponds to the number of cache entries. The saved address is split up in parts, each of which has at most the width required to index the words in a memory block. To represent a saved address, a bit $\mathcal{B}$ is set in each memory block. This bit $\mathcal{B}$ is set in the memory word indexed by the corresponding part of the split up saved address. In the memory

[9]RTTM needs 4 cycles for a memory operation with a tag memory match. The TCC prototype needs 13 cycles on a TCC cache hit, because a hardcore processor integrated into the FPGA was used.

word, the index of $\mathcal{B}$ corresponds to the cache line for the saved address. A lookup is then performed by bitwise ANDing the words read from each memory block (with the address split up as described). A set bit in the AND result indicates that the memory location is cached at this line. In effect, logic cells used to save cached addresses and to implement comparators are traded against a significant number of memory bits, where the number increases exponentially with the width of the split up address part. The number of cache entries can be increased with an approximately linear increase of resource consumption by replicating the described structure. A generic implementation for Altera FPGAs is in Listing B.3 on page 77. Using the memory blocks implementation technique, a tag memory with 32 entries would require 3 memory blocks on the Cyclone II: each memory block (with a size of 4 KiB) can implement a memory with 32 bit (read) word size and 128 words (i.e. 7 bits of the split up address). The shared memory addresses with a width of 19 bits must be split up on 3 memory blocks.

Memory blocks in several FPGA series, including the Cyclone II, lack a "gang reset" capability. This caused difficulties for the TCC implementation [WCN+07], where each of a large number of state bits had to be reset in a dedicated cycle. The requirement for a fast reset before a new hardware-side transaction would also complicate a tag memory implementation using memory blocks:[10] the bits set for each cache line need to be reset before the line is reused.

An in-depth discussion of the memory block implementation technique is contained in [BG02].[11] FPGA manufacturer Xilinx provides such a content addressable memory implementation [BG02]. I did not find other fully associative tag memory implementation techniques offering fast update and hit detection for current FPGAs [Bre99, Xil08, GLD00].

Table 4.3 on the facing page shows that the fraction of the available memory blocks consumed by the memory block variant is comparable to the fraction of available logic cells consumed by the logic cell variant. Note that the memory block variant also requires logic cells for the encoding of the lines and the reset of the tag memory. Since the JOP CPU uses a greater fraction of the memory blocks,[12] the logic cell variant is implemented.

4.2.5 Interfaces of the read tag memory and write buffer

The interface of the *read tag memory* is depicted in Figure 4.5 on the facing page. Since the read tag memory only needs to detect a conflict, it is not necessary to return which tag matched the provided address. The *write buffer*, depicted in Figure 4.6 on the next page, buffers writes during a transaction. The write buffer therefore needs to identify which tag matched a provided address when data is written or re-read during a transaction.

[10] But the reset would not necessarily cause a delay when using `valid` bits synthesized as logic cells.

[11] [BG02] refers to a Xilinx FPGA series, but I think most considerations are also applicable to other current FPGAs.

[12] Cf. total resource consumption in Table 6.3 on page 63 (where the logic cell variant is used).

Depth	Logic cells variant	Memory block variant
Words	% LCs of FPGA	% mem. blocks of FPGA
16	0.6%	–
32	1.2%	1.2%
64	2.4%	2.4%
96	–	3.6%
128	4.7%	4.8%
256	9.3%	9.6%

Table 4.3: Tag memory variants: comparison of (per-CPU) resource requirements

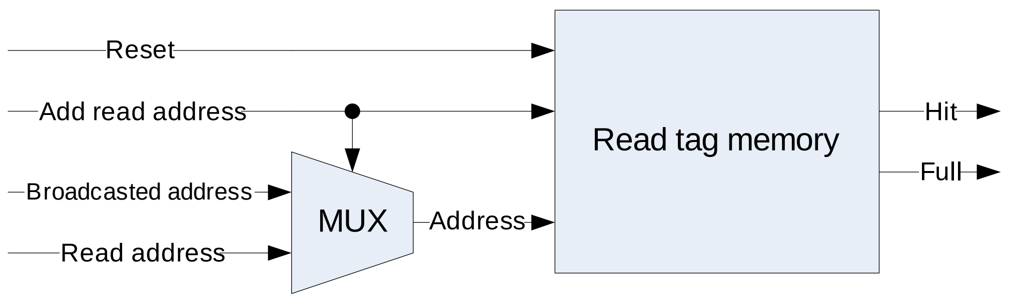

Figure 4.5: Basic read tag memory interface

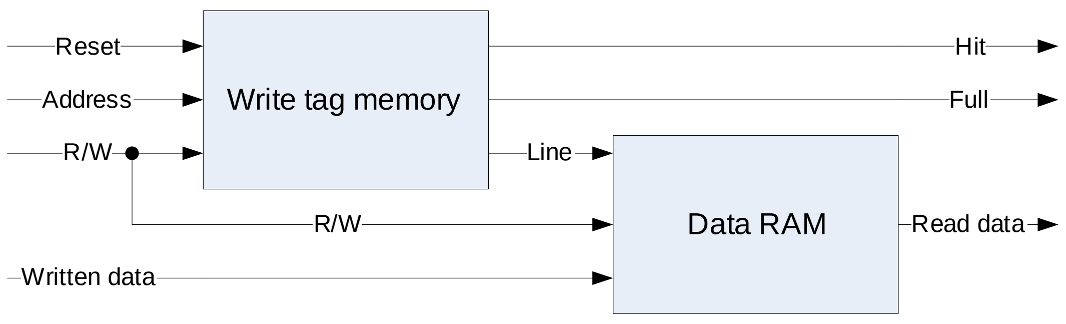

Figure 4.6: Basic write buffer interface (without commit logic)

4.2.6 The transaction cache

Microbenchmarks run on a behavioral simulation of an RTTM implementation on the JOP CMP [SBV10b, Muc09] showed that the read set is typically bigger than the write set. The microbenchmarks also indicated that there is often a large overlap between the read set and the write set of a transaction. As proposed in [SBV10b], a common tag memory which tracks both the read set and write set was implemented. Additional `read` and `dirty` bits are used to distinguish the read set and write set. The hardware structure combining the functionality of the write buffer and read tag memory is designated as *transaction cache*. The transaction cache is the central data structure of the RTTM module. For each memory location tracked, the transaction cache contains the fields listed in Table 4.4.

Item	Width	Description	Implementation
`valid`	1	Validity of entry	Logic cells
`address`	Varies[13]	Address of data word	Logic cells
`read`	1	Indicates address was read in transaction	Mem. blocks
`dirty`	1	Indicates address was written in transaction	Mem. blocks
`data`	32	Data word	Mem. blocks

Table 4.4: Transaction cache fields

Since implementing the write buffer means buffering the tentative values of a transaction and since the tag memory of the write buffer and the read tag memory is combined in the transaction cache, it is a cheap enhancement to also cache the data *read* during a transaction. This ensures there is at most one miss for each element of the read set during a transaction.

The interfaces and the basic components of the transaction cache are depicted on the next page. The tag memory saves the addresses of both the read set and the write set. It also detects conflicts with addresses written/broadcasted by a committing transaction.[14] The data RAM caches values written or read during the transaction. The write addresses FIFO stores the addresses of the write set of the transaction. An address is added to the FIFO when it is first written during a transaction. On a commit, the corresponding data is written to the shared memory (logic not depicted). The data cache organization of the TCC prototype is similar [WCN+07, Fig. 1].

[13]The width (typically 19 bits) corresponds to the number of data words in the shared memory. The processor actually has a larger address space. Memory not read/written during a transaction (flash) and I/O is mapped into other parts of the address space.

[14]These unsynchronized activities are performed interleaved without an impact on the temporal behavior. This will be discussed on page 46.

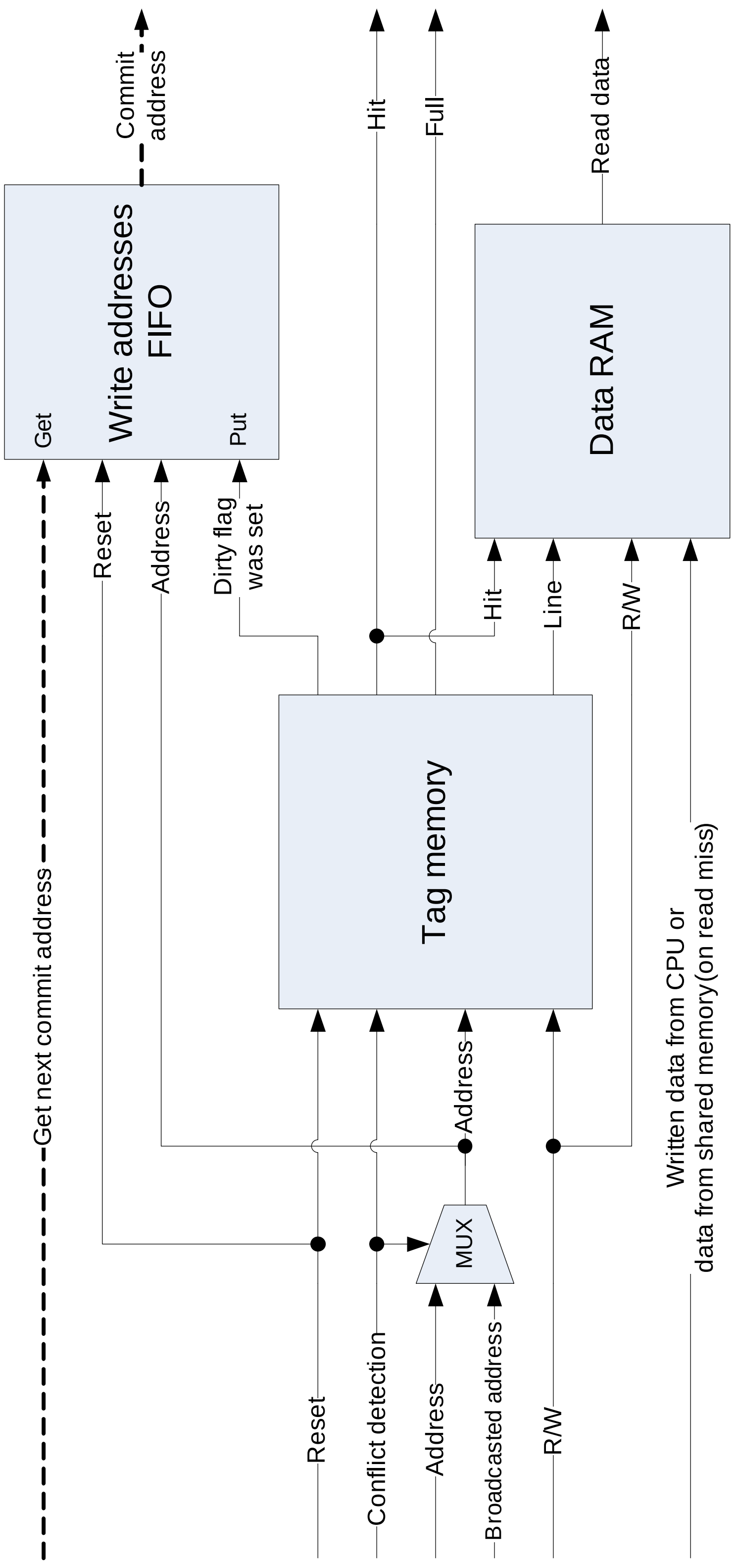

Figure 4.7: Transaction cache.
Dashed lines are only used to flush the write set during the commit phase.

On a transaction start, the tag memory is invalidated and the write addresses FIFO is emptied. On each transactional memory access which is a cache miss, the next cache line is used. In the exceptional case that the tag memory is full, the RTTM logic stalls the processor and tries to perform an early commit.

The addresses of the write set are also stored in the tag memory. But the tag memory is implemented using logic cells. In order to not only perform hit detection, but also read out the addresses in the tag memory, additional logic would be required. The addresses could be read out using random access or by shifting the addresses through the tag words. In both cases, this additional logic needs an equivalent of about 40% of the tag memory LCs.[15] A FIFO is implemented very efficiently using the FPGA's memory blocks, so the addition of the write addresses FIFO saves hardware resources.[16] When using the FIFO, the commit time does not depend on the read set size.

4.2.7 The state machine

The RTTM module behavior is governed by the state machine on the facing page. The transaction cache is accessed or bypassed according to the state machine. The state machine resembles the simplified view of transactions by the hardware layer.

Outside of a transaction the RTTM module is in state `BYPASS`. A normal *successful* transaction traverses the states in this order: `BYPASS`, `TRANSACTION`, `WAIT_TOKEN`, `COMMIT`, `BYPASS`. Figure 4.9 on page 42 is a sequence diagram of a successful transaction. A failed (and usually retried) transaction normally traverses the states `BYPASS`, `TRANSACTION`, `ABORT`, `BYPASS` or the states `BYPASS`, `TRANSACTION`, `WAIT_TOKEN`, `ABORT`, `BYPASS`.

4.2.8 Transaction states

`BYPASS` This state is active outside of transactions. It is also active in some parts of a software layer transaction.[18] In this state, memory accesses bypass the transaction cache and any broadcasted addresses are ignored. When an atomic method is invoked outside of a transaction, the transaction wrapper issues a `start_transaction` hardware

[15]With 64 words and 19 bits address width, the tag memory needs 1630 LCs without support for reading addresses, 2260 LCs with the shifting variant and 2300 LCs with the random access variant.

[16]In Section 6.1, we will observe that the memory blocks are depleted earlier than the LCs on the used FPGA. But reading the memory addresses from the tag memory would shift the balance and lead to an even earlier depletion of LCs. (Consider that we save an equivalent of 40% of the tag memory LCs with a single memory block.)

[17]The CPU is stalled by delaying the current memory access. There is always such a memory access when the CPU needs to be stalled.

[18]Software layer transactions will be discussed in Section 4.3.1.

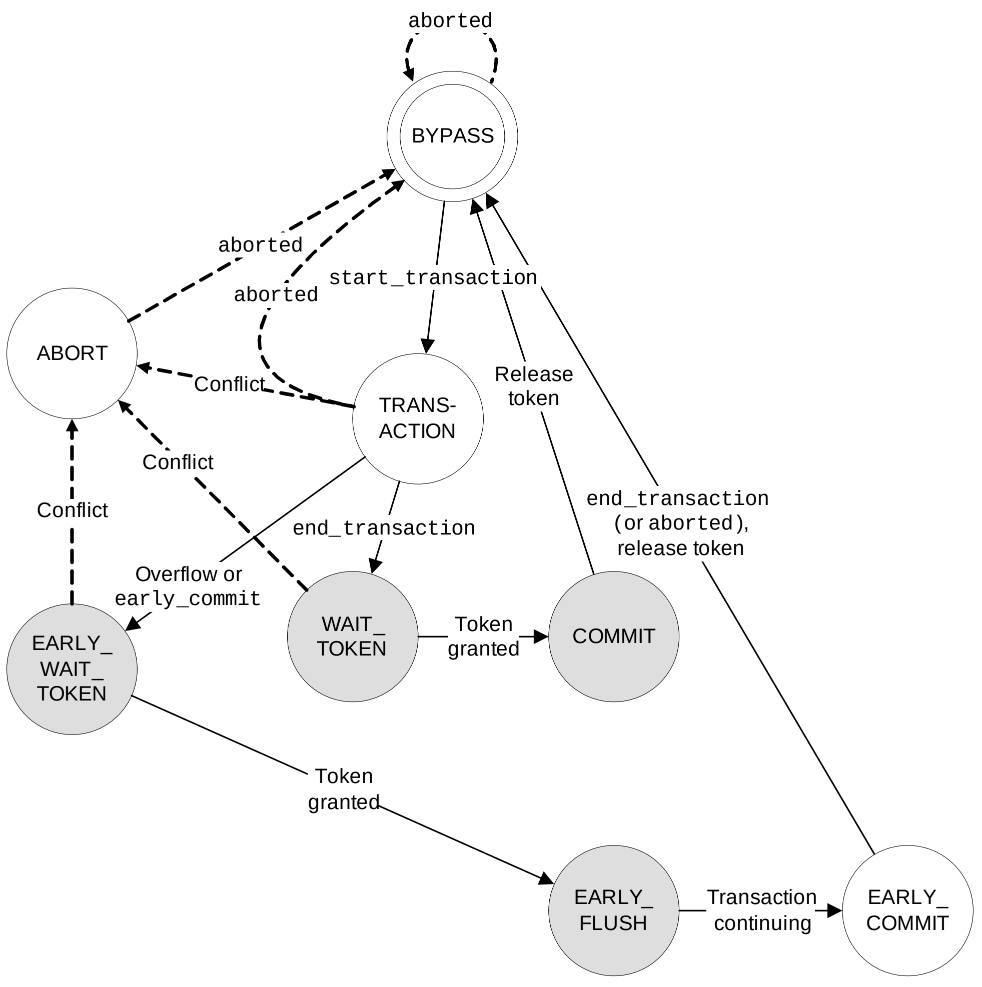

Figure 4.8: RTTM module state machine. States during which the CPU is stalled[17] have a solid background. Hardware commands issued by the RTTM software layer are `monospaced`. Dashed lines indicate aborting transactions.

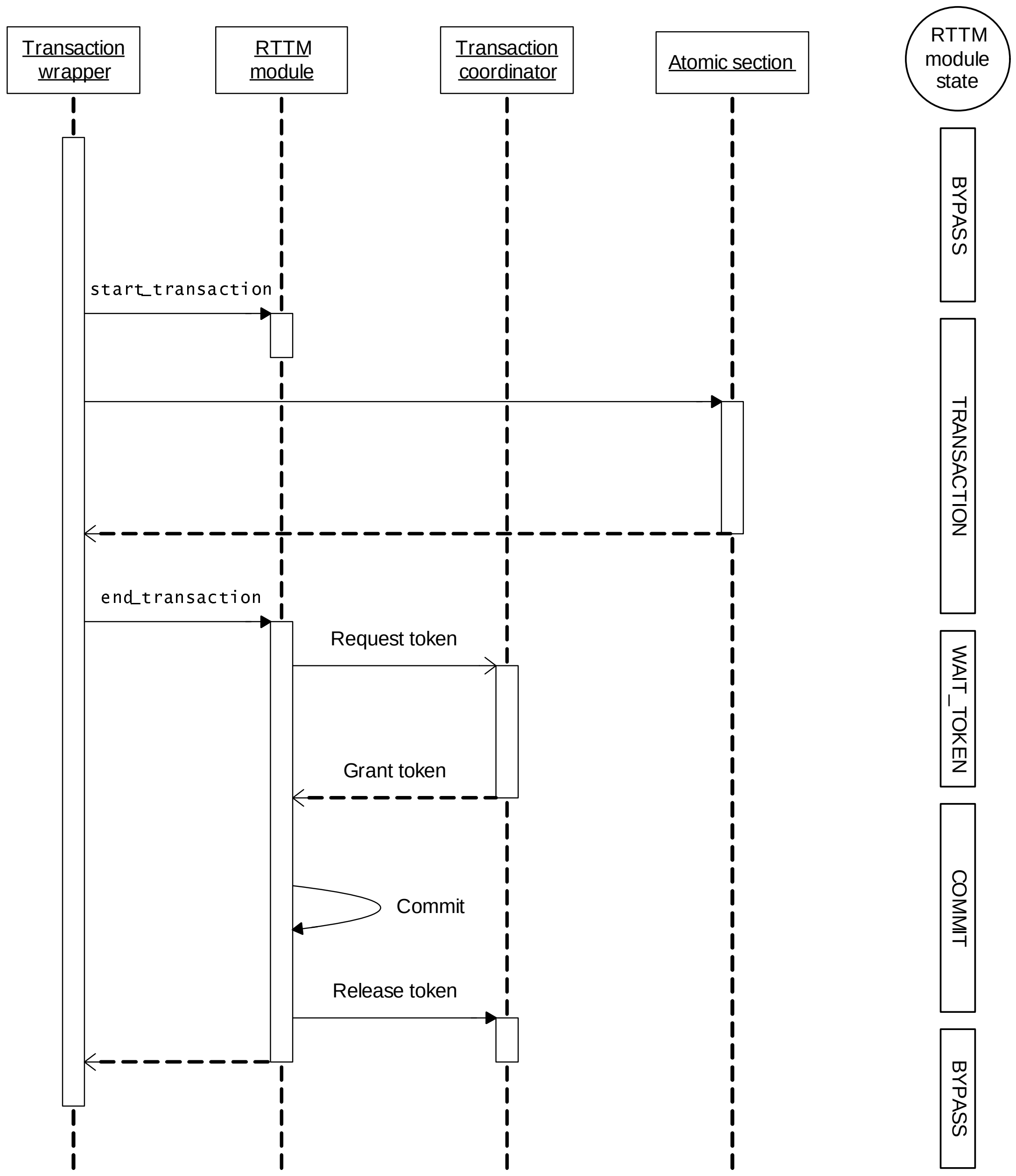

Figure 4.9: Software and hardware components interacting in a normal successful transaction.

command. Then, $\mathcal{M}$ transitions to state `TRANSACTION`.[19]

`TRANSACTION` In this state, all memory accesses for which the CPU sets the `tm_cache` flag are buffered in the *transaction cache*, which comprises the role of both the write buffer and the read tag memory. Other memory accesses bypass the transaction cache. For the rest of this paragraph, assume that the `tm_cache` flag is set. A write is cached in the transaction cache. If a memory read generates a cache miss, the data is read from the shared memory and also stored in the transaction cache. `read` and `dirty` bits associated with the cache entries track the *current read set* and *current write set*.

If a broadcasted address is in the current read set of an RTTM module $\mathcal{M}$, $\mathcal{M}$'s transaction is conflicting with the committing transaction and $\mathcal{M}$ will therefore enter state `ABORT`. The CPU can also initiate a transaction abort by issuing an `aborted` HW command, triggering a direct transition to `BYPASS`. Possible causes for a direct abort are an explicit transaction abort by the application or a thrown Java exception.[20] After both state transitions, the transaction cache is invalidated and the rollback is performed by the transaction wrapper executing on the CPU.

Upon returning normally from a not nested atomic section, the software transaction wrapper issues an `end_transaction` HW command, triggering a transition to `WAIT_-TOKEN`. In the exceptional case that *a*) the transaction cache overflows or *b*) the application performs an early commit (see Section 3.3.1), $\mathcal{M}$ goes into state `EARLY_WAIT_TOKEN`.

A conflict may be detected while any of the other state transition conditions is enabled. The conflict will get priority over all other transitions except the transition with condition `aborted`. No two other state transitions can be enabled concurrently.

`WAIT_TOKEN` The CPU is stalled in this state. The RTTM module $\mathcal{M}$ tries to acquire the single commit token from the *transaction coordinator*. If the commit token is eventually granted to $\mathcal{M}$, $\mathcal{M}$ goes into state `COMMIT`. If a broadcasted address is in the read set of $\mathcal{M}$ – tracked through the transaction cache – $\mathcal{M}$'s transaction is conflicting with the committing transaction and $\mathcal{M}$ will therefore enter state `ABORT`. Any conflict will be detected while the commit token is still held by another RTTM module committing a preceding transaction.[21]

`COMMIT` If an RTTM module $\mathcal{M}$ has reached state `COMMIT`, its current transaction will succeed. $\mathcal{M}$ holds the single commit token while in state `COMMIT`. $\mathcal{M}$ writes the write set of the transaction, as tracked through the transaction cache, to the shared memory.

[19]The HW command `aborted` is ignored when a transaction has already been aborted earlier and $\mathcal{M}$ is back in state `BYPASS`. See page 53, esp. footnote 40, for an explanation why `aborted` may be issued twice.

[20]See Section 4.3.2.

[21]Discussed on page 5.

The write set addresses are also simultaneously broadcasted to all other RTTM modules, causing any conflicting transactions to abort. After the write set has been written to the shared memory, the transaction changes are visible to all other transactions and to non-transactional code. $\mathcal{M}$ then releases the commit token and goes back to state `BYPASS`. The CPU remains stalled while in state `COMMIT` and resumes execution when $\mathcal{M}$ goes back to state `BYPASS`.

`ABORT` An RTTM module $\mathcal{M}$ enters state `ABORT` if a conflict with the committing transaction is detected (i.e. a broadcasted address is in the current read set of $\mathcal{M}$). $\mathcal{M}$ remains in this state until it is assured that the CPU has observed the conflict and aborted the execution of the transaction. Upon entering `ABORT`, $\mathcal{M}$ sets the `rollback` signal to initiate a rollback on the CPU side, which will be discussed in Section 4.3.2.

While in state `ABORT`, the behavior of the transaction must be contained, since the transaction is not always aborted immediately. Memory writes are discarded, since a (write) buffer overflow might have occurred in an earlier state or might occur in this state and the transaction must not become visible. Memory is always read from the shared memory for pragmatic reasons.

An exception thrown due to the `rollback` signal, or possibly an exception thrown earlier during the transaction, eventually causes the CPU[22] to issue an `aborted` HW command. This signals to the RTTM module that the execution of the atomic section has been aborted. $\mathcal{M}$ will reset to state `BYPASS`.

`EARLY_WAIT_TOKEN` This state is entered in the exceptional case that *a*) the transaction cache overflows or *b*) the application performs an early commit. The CPU is stalled in this state.[23] The RTTM module $\mathcal{M}$ tries to acquire the single commit token from the transaction coordinator. If the commit token is eventually granted to $\mathcal{M}$, $\mathcal{M}$ goes into state `EARLY_FLUSH`.

If a broadcasted address is in the current read set of $\mathcal{M}$, $\mathcal{M}$'s transaction is conflicting with the committing transaction and $\mathcal{M}$ will therefore enter state `ABORT`. Any conflict will be detected while the commit token is still held by another RTTM module committing a preceding transaction.

`EARLY_FLUSH` In `EARLY_FLUSH`, operations similar to `COMMIT` are performed. The CPU remains stalled in state `EARLY_FLUSH`. The main difference is that the transaction continues execution upon exiting this state. If an RTTM module $\mathcal{M}$ has reached state

[22]The `aborted` HW command is not issued by the transaction wrapper, but earlier by the code implementing the exception throwing in order to avoid asynchronous exceptions during exception handling. Exception handling includes lock unlocking and stack manipulation.

[23]The CPU waits *a*) for the current memory access to complete or *b*) for the `early_commit` HW command to finish.

`EARLY_FLUSH`, its current transaction will succeed.[24] $\mathcal{M}$ holds the single commit token while in state `EARLY_FLUSH` and the subsequent state `EARLY_COMMIT`. $\mathcal{M}$ writes the *current* write set of the transaction, as tracked through the transaction cache, to the shared memory. The current write set addresses are also simultaneously broadcasted to all other RTTM modules, causing any *currently* conflicting transactions to abort.

After the current write set has been written to the shared memory, a part of the transaction changes has become visible. The transaction is then continued in state `EARLY_COMMIT`.

`EARLY_COMMIT` An RTTM module continues to hold the commit token while in state `EARLY_COMMIT`. The transaction cache is not used any more in this state, as a buffer overflow might have happened. Instead, all writes are immediately propagated to the shared memory. The addresses written are also simultaneously broadcasted to all other RTTM modules and cause any conflicting transactions to abort.

Upon returning normally from a not nested atomic section, the transaction wrapper issues an `end_transaction` HW command, triggering a transition to `BYPASS` and the release of the commit token. All transaction changes are now visible to all other transactions and to non-transactional code.

A transaction in state `EARLY_COMMIT` might still be aborted in an exceptional situation, either *a*) if an exception terminating the current transaction is thrown due to a programming error or *b*) by the application (using the `abort()` or `retry()` software command). In this cases, the transaction's changes to the shared memory will not be rolled back and violate the atomicity of the transaction.

4.2.9 Pipelining

The JOP CPU uses the *SimpCon* SoC interconnect for memory access. SimpCon is a fully synchronous point-to-point master-slave connection (see Section 1.4 and page 31). SimpCon supports pipelined operations [Sch07]. Early restart of the master is supported by using a `rdy_cnt` signal driven by the slave. This signal indicates how many cycles are at most left to finish a read or write access (possibly an unbounded number).

The RTTM module, acting as a SimpCon slave for the CPU (see Figure 4.4 on page 30), supports moderately pipelined memory accesses. All memory accesses bypassing or accessing the *transaction cache* span multiple clock cycles in order to support a high clock frequency. At most two memory accesses and two addresses broadcasted by a committing transaction are processed concurrently.

Since the interconnect delay between the CPU, the RTTM module and the memory arbiter is an issue, even memory accesses bypassing the transaction cache are delayed.

[24]Unless the transaction is manually aborted by the application or by an exception thrown due to a programming error.

Reads therefore need two or three cycles more and writes one cycle more than in the conventional JOP CMP.[25] This is no disadvantage since a higher clock frequency is possible and since worst-case shared memory access times are high with many cores [Pit09].

In the transaction cache, there is a potential structural hazard when the tags need to be compared for both *a*) a memory access and *b*) to match an asynchronously *broadcasted* address against the current read set (indicated by the multiplexer in Figure 4.7 on page 39). To prevent the CPU from initiating too many memory accesses without having to set `rdy_cnt` too high, only SimpCon pipeline level 2 for read and write memory accesses is provided by the RTTM module. Pipeline level 2 means that the CPU is allowed to initiate a new memory access only when the `rdy_cnt` signal is at most 1, i.e. when the old memory access will be completed in the next cycle. Since the RTTM module sets the `rdy_cnt` signal to 2 or higher in the cycle after a memory access start, the CPU is allowed to issue memory requests at most every 2 cycles. The memory arbiter is also allowed to broadcast addresses at most every 2 cycles.[26]

The hazard can then be resolved without affecting the timing of memory accesses by delaying the matching of broadcasted addresses by 1 cycle in case of a collision. The restriction of the rate of memory accesses also has the benefit of reducing potential data hazards, since all memory accesses are separated by more than one cycle.

4.2.10 Memory access classification

An efficient and predictable implementation of the write buffer and read tag memory requires a fully associative cache. Fully associative caches are expensive and therefore the read set and write set of a transaction must be small. Memory areas where no conflict can arise do not need to be included in the read set and write set. If, in addition, changes to a memory area during a failed transaction can be ignored, there is no need to buffer modified data.

The *Java Virtual Machine* (JVM) specification defines *runtime data areas* [LY99, Ch. 3.5], some of which can be excluded from the read set and from write buffering. The JOP core reproduces most of the memory structure of the JVM in a direct way (using hardware and microcode), which makes the exclusion of memory areas easy. Some memory areas where no conflicts can happen are listed in Table 4.5 on the facing page, together with the corresponding JVM runtime data areas. The *Method Area* and the per-class/per-interface *Runtime Constant Pool* are constant in the JOP implementation.

[25] For a more detailed discussion of the memory access timing differences, see the analysis of worst-case temporal behavior in Section 6.2.

[26] The JOP CPU and the used TDMA memory arbiter satisfy these requirements. There is no impact on performance.

[27] Arguments to an atomic method invocation executing a not nested transaction must be restored on a retry if they might have been modified.

Memory area	JVM runtime data area	No conflicts	Ignore writes	Implemented
Instructions	Method Area	✓	✓	✓
Stack	JVM stack	✓	✓[27]	✓
Constants	Runtime Constant Pool	✓	✓	✗
Object representation	(Implementation specific)	✓	✓	✗

Table 4.5: Non-conflicting memory areas

The implementation specific representation of objects, where e.g. the size of an array or the pointer to the method vector base is saved, will also not change during a transaction (see [Sch09, Ch. Runtime Data Structures]).[28] The runtime constant pool and the object representation are not yet excluded from the read set. Since these memory areas are accessed by dedicated microcode blocks, a filtering mechanism to exclude these areas from the read set could be incorporated into the microcode and for some bytecodes in the memory interface.[29]

The JVM specification defines the *Java virtual machine stack* [LY99, Ch. 3.6] as thread-private. Each method can only access its own *stack frame*. JOP holds the stack in a dedicated CPU-internal memory. Writes to the stack are therefore not buffered during a transaction. The fact that only atomic methods – as opposed to code blocks – are transparently supported and the nature of the stack mean that most changes to the stack during a failed transaction can be ignored and no elements of the stack need to be restored when rolling back a transaction.[30] JOP uses a special form of an instruction cache, a *method cache* (see Section 1.3). The method cache is loaded by dedicated hardware, which clears the `tm_cache` SimpCon flag.

Since the size of the read set of a transaction should be bounded by static analysis, the memory addresses which are actually excluded from the read set should correspond to the addresses ignored by the static analysis.

An alternative would be conflict detection at the object level instead of the memory word level by tracking accesses to object handles (or caching of objects similar to [KHR+08]). This could support a bigger read set.

[28]At least as long as garbage collection during transactions is not supported.

[29]In the current implementation – which does not support garbage collection – only part of the memory accesses of the following JVM bytecodes needs to be tracked: `getfield`, `getstatic`, `putfield`, `putstatic`; `aaload`, `baload`, `caload`, `daload`, `faload`, `iaload`, `laload`, `saload`; `aastore`, `bastore`, `castore`, `dastore`, `fastore`, `iastore`, `lastore`, `sastore`.

[30]Method arguments are an exception, as they might be modified. Method arguments are – if needed – saved and restored by the software transaction wrapper (see Section 4.3.1).

4.2.11 Summary of hardware integration

The changes to the JOP CMP hardware consist mainly in the addition of the RTTM module and the transaction coordinator and changes to the memory arbiter. The interface of the RTTM hardware to the conventional part of the CMP consists of those interfaces of the RTTM hardware which interact directly with a non-RTTM component (see Figure 4.10). It consists of the SimpCon interconnects to (i) CPU and (ii) shared memory for memory accesses, (iii) the (memory-mapped) hardware commands and (iv) the `rollback` signal which signals a detected conflict with a committing transaction to the CPU.

On the SimpCon interconnect (i), the CPU sets the `tm_cache` flag as described in the previous section. The semantics of the other lines of (i) as well as the semantics of SimpCon interconnect (ii) remains unchanged.

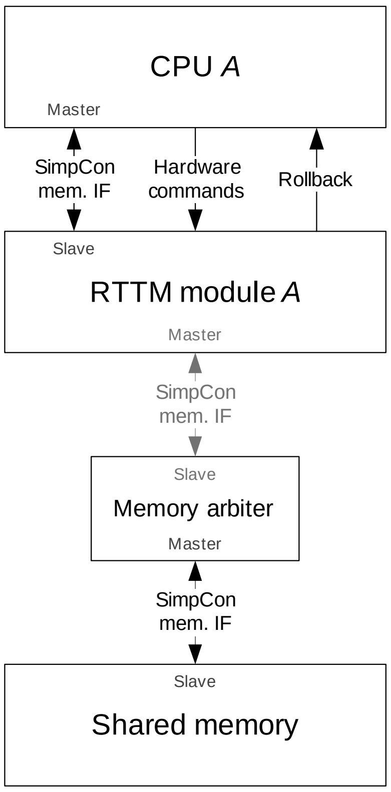

Figure 4.10: Interfaces of an RTTM module and the memory arbiter to the conventional part of the CMP

The abort of transactions in response to the `rollback` signal is implemented using the Java exception handling mechanism which executes on the CPU. The throwing of some conventional Java exceptions, such as the `StackOverflowError` and the

`NullPointerException`, is also triggered by hardware mechanisms. A mechanism to raise exceptions by hardware is therefore available [Sch08, p. 271]. Due to the asynchrony of conflict detection, an exception may be thrown concurrently or nearly concurrently by the RTTM module and by CPU hardware mechanisms. The hardware exception generation mechanism is adapted to always give priority to the exception generated by the RTTM module.

In addition, signals for inter-CPU synchronization are registered to enhance performance. There are no further modifications to the conventional CMP. Changes of the temporal behavior will be discussed in Section 6.2.

Restrictions on processor behavior The current RTTM implementation poses some restrictions on the behavior of the processor:

- To avoid structural hazards: the CPU memory SimpCon interconnect has at most pipeline level 2 (see Section 4.2.9) and the memory arbiter broadcasts addresses at most every 2 cycles.
- To guarantee conflict detection: the memory arbiter and the shared memory serialize memory accesses and the update of the read tag memory and broadcasts are fast enough for conflict detection (see page 33).

4.3 Software layer

The hardware layer of the RTTM implementation implements a simplified model of transactions corresponding to the state machine on page 41. The hardware does not handle the control flow associated with transaction abort, the retry of a transaction or transaction nesting. The role of the software layer is mainly to transparently start, abort and retry transactions, to initiate commits and to support (flattened) nested transactions.[31]

The behavior of Java programs [GJSB05] is thoroughly specified by the Java Virtual Machine (JVM) [LY99]. The JVM may also be implemented in hardware (e.g. [MO98, CW05]). Such hardware directly executes the JVM instruction set, called *bytecode*. JOP, the prototype implementation platform for RTTM, is a time-predictable JVM in hardware.

The RTTM implementation relies on Java language features (exception handling) and some of the strong guarantees and restrictions of Java and the JVM (known and limited instruction set, e.g. only method-local jumps). This is uncommon for a hardware TM[32] [LR07, Ch. 4.1] but reduces hardware resource consumption and simplifies the implementation.

[31]Some small changes to the runtime system are also necessary, as will be discussed in Section 4.3.4.
[32]There are multiple STMs which use language features, e.g. [HMPJH05].

4.3.1 The transaction wrapper

The *transaction wrapper* implements most of the software functionality of RTTM. The transaction wrapper encloses the original method body of *atomic methods*, but retains the original method signature. It starts, aborts and retries transactions, initiates commits and supports (flattened) nested transactions, i.e. is mostly glue code. The transaction wrapper consists of JVM bytecode. It is added at link time.

The transaction wrapper catches any exceptions thrown in the original method body and processes them appropriately. In a not nested transaction, this try-catch block is executed in a loop until the transaction has committed or has been aborted. The behavior of the transaction wrapper is different if the transaction is nested. Upon method invocation, the transaction wrapper determines whether it is executing a nested transaction by reading a CPU-local boolean `inTransaction`.

Not nested transactions In the method invocation corresponding to a not nested transaction, the transaction wrapper issues *hardware commands* to the *RTTM module*. The RTTM module stalls the CPU while executing some hardware commands and during memory accesses which overflow the transaction cache (see state machine on page 41). The RTTM module signals conflicts to the transaction wrapper by throwing a Java `RetryException`.

A conceptual model of the transaction wrapper, which assumes that the transaction is not nested, is on the facing page. First, the transaction wrapper performs some initializations. The method arguments are copied to dedicated local variables if they might be modified inside the atomic method.[33] Interrupts are disabled. The transaction wrapper then executes the original method body in a loop until the transaction *a*) commits or *b*) is explicitly aborted by the user or *c*) a conventional exception is thrown. In the loop body, the transaction wrapper first signals the RTTM module to start a transaction. Then the original method body is executed. If no exception was thrown in the original method body, the transaction wrapper signals the RTTM module to try to commit. The CPU will then block until *a*) the commit has succeeded or *b*) a conflict with a preceding transaction has been detected. In case *b*), the RTTM module did never acquire the commit token. If the commit succeeded, the transaction wrapper re-enables the interrupts. Then the method returns normally.

The transaction fails if an exception was raised while executing *a*) the original method body, *b*) any methods invoked by it or *c*) the `END_TRANSACTION` hardware command. The transaction is then *rolled back*. Depending on the thrown exception, the transaction is retried or the exception is propagated to the invoker of the transaction. If a conflict was detected or the programmer invoked the `retry()` software command, the method arguments are restored, if they might have been modified, and the loop body is executed

[33]It is determined at link time if any method arguments might be modified in the original method body.

```
ResultType notNestedTransactionWrapper(int arg1, Object arg2) {
  int arg1Copy = arg1;
  Object arg2Copy = arg2;

  Native.wrMem(0, IO_INTERRUPT_ENABLE);
  ResultType result;

  while (true) {
    Native.wrMem(START_TRANSACTION, IO_HW_COMMAND);

    try {
      // original atomic method body here,
      // with returns redirected to next statement

      Native.wrMem(END_TRANSACTION, IO_HW_COMMAND); // commit
    } catch (Exception e) {
      // exception handling issues ABORTED HW command
      if (e instanceof RetryException) {
        arg1 = arg1Copy;
        arg2 = arg2Copy;
        continue;
      } else {
        Native.wrMem(1, IO_INTERRUPT_ENABLE);
        throw e;
      }
    }
    break;
  }

  Native.wrMem(1, IO_INTERRUPT_ENABLE);
  return result;
}
```

Listing 4.3: Conceptual transaction wrapper for not nested transaction

again. If a conventional exception was raised due to a programming error or if the programmer invoked `abort()`, the exception is propagated to the invoking method.

Nested transactions are transparent to the hardware layer. The implementation only supports flattened nested transactions: in a nested transaction, the transaction wrapper does not perform a commit and only rethrows exceptions. The functionality of the *nesting count* of transactions is replaced by a CPU-local and a method-local variable.[34] The transaction wrappers maintain a CPU-local boolean `inTransaction` indicating whether the current thread is executing a transaction. Each invocation of an atomic method has a method-local boolean `isNotNestedTransaction` (added by the transaction wrapper) indicating whether the invocation corresponds to a not nested transaction. `isNotNestedTransaction` is deduced from `inTransaction` on method invocation. Java code corresponding to the entire generated transaction wrapper is shown in Listing B.1 on page 75 in the appendix.

4.3.2 Transaction rollback

Transactions are rolled back by a Java exception. An exception may be thrown due to the following causes:

- the RTTM module detected a conflict to a committing transaction or
- a programming error or
- an explicit transaction retry/abort by the application.[35]

If a transaction is rolled back, all its changes that violate failure atomicity must be undone. A failed transaction might have affected (i) CPU internal state, (ii) the RTTM module and (iii) the lock implementation (for a `synchronized` statement or method). The shared memory is not modified by a failed transaction.[36]

The changes are undone as follows: The *transaction wrapper* resets the RTTM module state (ii) by issuing an `aborted` hardware command. The RTTM module invalidates the transaction cache and goes to the initial `BYPASS` state. Locks acquired during a transaction (iii) are unlocked during the rollback of the transaction. Since the rollback uses the Java exception handling mechanism and the exception handling mechanism

[34]Maintaining the nesting count is made difficult by race conditions between CPU and RTTM module: the RTTM module could initiate a rollback of the transaction immediately before or after the nesting count is modified.

[35]*Zombie transactions* seeing an inconsistent global state cannot cause an exception due to e.g. a division by zero or a null pointer dereference, since they are aborted early enough (after one executed bytecode).

[36]Unless an early commit was performed.

already performs lock unlocking [LY99, Ch. 3.11.11], no changes are required for lock support. The CPU internal state (i) comprises *a*) the stack and *b*) special purpose registers (program counter, several pointers to the stack and the current method and class).[37] Since the rollback uses the Java exception handling mechanism, no changes to restore the special purpose registers are necessary.

Stack rollback The JVM specification defines the *JVM stack* [LY99, Ch. 3.6] as thread-private. The (JVM) stack holds *frames* created on a method invocation and destroyed on method invocation completion. A frame holds local variables (including method arguments), the *operand stack*[38] and the saved context of the invoking method.[39]

JOP holds the stack in a special purpose memory. The nature of the stack means that most changes to the stack during a failed transaction can be ignored. Once a transaction is being rolled back, frames created during a transaction only need to support abrupt method completion through an exception. As only atomic methods, instead of code blocks, are transparently supported by the implementation, changes to most local variables can be ignored, as a retry of the transaction will re-initialize these local variables. Method arguments are an exception, since they are part of the local variables and therefore mutable in the JVM. Method arguments are – if needed – saved and restored by the transaction wrapper.

Asynchronous exception handling When the RTTM module detects a conflict, it initiates a transaction retry by setting the `rollback` signal. In response, the CPU generates a Java `RetryException`. The conflict detection is asynchronous to the execution of instructions on the CPU. In JOP, hardware generated exceptions replace the JVM *bytecode* executed next. An asynchronous exception thrown due to a detected conflict is therefore still *precise*. As for any other exception, the exception handling mechanism will issue the hardware command `aborted` to quit the hardware transaction.[40]

4.3.3 Zombie bytecodes

During a commit, an RTTM transaction $\mathcal{S}$ updates the shared memory *in place*. This can cause a conflicting transaction $\mathcal{Z}$ to observe inconsistent data, i.e. the *isolation* property (see page 2) is violated for $\mathcal{Z}$. The occurrence of inconsistencies is illustrated in

[37]Since the JVM is stack-based, there are relatively few registers whose state has to be kept between executed bytecodes.

[38]The JVM is a stack-oriented machine.

[39]See [Sch09, Section "Runtime Data Structures"] for details about the stack implementation in JOP.

[40]Any code following the hardware command `aborted` will therefore not get interrupted. This exception handling code includes lock unlocking and stack manipulation. `aborted` itself may be issued twice if another exception is already being thrown when a second exception is thrown due to a detected conflict.

Figure 4.11. This inconsistencies may cause incorrect behavior of the *zombie transaction* $\mathcal{Z}$. The RTTM module executing $\mathcal{Z}$ will notice the conflict during the commit of $\mathcal{S}$ (see page 5) and retry the transaction.

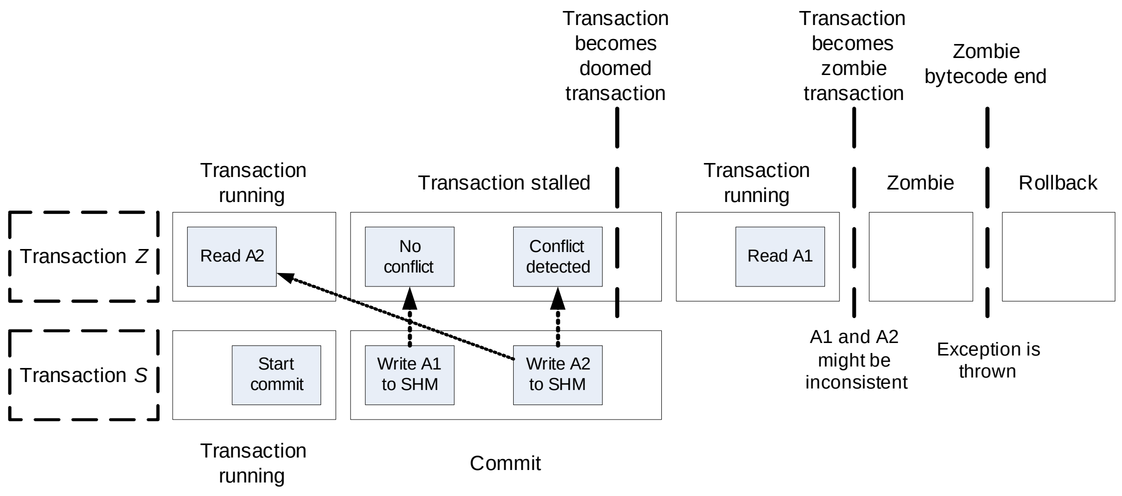

Figure 4.11: Transaction $\mathcal{Z}$ reads mutually inconsistent values of A_1 and A_2 and therefore becomes a zombie transaction.

In some hardware TM systems and in some software TM systems which are targeting safe languages such as Java, zombie transactions may also occur [LR07]. Incorrect behavior such as invalid memory accesses, exceptions, infinite loops or infinite recursions can be contained in such systems [LR07]. STMs can rely on closed memory systems and managed runtime environments [DSS06] and HTMs use computer architectural means.

The RTTM *implementation*, however, needs to restrict the behavior of a zombie transaction. Some functionality, such as garbage collection and I/O, is not transparently supported inside a transaction. The control flow must remain consistent, since the Java exception handling mechanism is used to roll back a transaction. The following data areas must not appear inconsistent during and after a zombie transaction:

- Method invocation contexts (JOP-specific, see [Sch09, Section “Runtime Data Structures”])
- Other JVM internal data structures (object structure, array structure, class structure [Sch09, Section “Runtime Data Structures”]) to avoid corrupting processor state or corrupting memory
- Local variables used by the *transaction wrapper*

No garbage collection and object allocation is allowed while a (zombie) transaction is running. This assures that object references dereferenced by a (zombie) transaction will point to a consistent structure. A committing transaction therefore updates only instance/class fields and the components of arrays in the shared memory.

Zombie behavior The *memory arbiter* providing access to the shared memory normally follows a TDMA scheduling strategy to make memory accesses time-predictable. But when a processor C has the commit token, all slots of other processors which are executing a transaction are reallocated to C. Other processors executing a transaction are then stalled on a shared memory access. This implicates that no other processor can observe inconsistent states while C has the commit token. An RTTM module $\mathcal{D}_m$ will also detect a conflict of its transaction to the committing transaction while C has the commit token. $\mathcal{D}_m$ will then initiate the throwing of a `RetryException` on its processor $\mathcal{D}_p$.

This exception is thrown similar to conventional hardware-generated exceptions by replacing the JVM *bytecode* executed next. Since the exception throwing is initiated during C's commit, at most one bytecode will observe inconsistent state, reducing the zombie phase of a transaction to a *zombie bytecode*. The control flow will therefore not be modified by a zombie transaction. This implicates that code not supported in a transaction, such as I/O, will not be executed *due to* a zombie transaction. Since bytecodes accessing Java objects (i.e. mutable shared memory) take their arguments from the operand stack and the operations performed by a single bytecode[41] are limited, a zombie bytecode will not even throw an exception *due to* the zombie inconsistencies. The `RetryException` will supersede any exception which is about to be thrown.

Java bytecodes are implemented as microcode sequences. The microcode control flow is not corrupted due to a zombie transaction.[42] Only the `double` and `long` data types occupy multiple data words in the RTTM implementation and the values are not used in the bytecodes in which they are loaded onto the operand stack. References to local variables are hardcoded in the bytecodes. The zombie containment of shared memory accesses is provided by the RTTM hardware. During state `TRANSACTION`, the transaction cache buffers any writes to the shared memory. During state `ABORT`, all writes to the shared memory are discarded. These are the only possible states during a zombie transaction.

We think that the restriction of the zombie transaction to a single instruction would also enable the use of an unsafe language for an RTTM implementation. With the reallocating arbitration strategy, an early commit stalls not conflicting transactions which are not yet waiting for the commit token. If the arbiter did not reallocate slots of other transactions during a commit, the zombie phase of transactions could span multiple

[41] Some complex bytecodes are implemented using a subset of simpler bytecodes. We only need to consider the simpler bytecodes then.

[42] This follows immediately from the observations about microcode branches in [Sch09, Section "Microcode Path Analysis"], but more generally also because only instance/class fields and components of arrays are updated.

Modification	Explanation
When throwing an exception, issue hardware command `aborted`	Acknowledge transaction abort due to detected conflict to RTTM module; abort transaction on conventional exception. Prevent RTTM module from throwing exception while exception handling mechanism updates JVM internals non-atomically.
Bytecodes `putfield`, `putstatic`, `aastore`: Remove garbage collection support	Garbage collection support might lead to deadlocks when interfering with early commits

Table 4.6: Modifications to JOP runtime system

bytecodes. This might lead to corrupted control flow and wrong method invocations, which would be a problem since not all functionality is supported inside a transaction. It is possible to tolerate zombie transactions executing more than a single bytecode while seeing inconsistent state, but this requires the code reachable with corrupted control flow from a transaction to be appropriately limited (no I/O, no object allocations...).

4.3.4 Summary of runtime system integration

The (small) modifications to the JOP runtime system are summarized in Table 4.6. In addition, some RTTM-specific classes, exceptions and constants were added.[43]

4.4 Link time transformations

The original bodies of *atomic methods* are enclosed by a *transaction wrapper* at link time. The *Byte Code Engineering Library* (BCEL)[44] is used to add a header and footer similar to Listing B.1 on page 75 to the original method body and redirect `return` statements in the original method body. The method signature is retained. Non-atomic methods invoked by an atomic method do not need to be modified.

[43]See file `readme-rttm.txt` in the source code.
[44]See `http://jakarta.apache.org/bcel/`.

CHAPTER 5

Programming for RTTM

Following [LR07, p. 15], transactional memory is, although often presented in the form of a software library or processor instructions, fundamentally a programming abstraction and should serve its users, the programmers. Boehm [Boe09] argues that the programmer should see the synchronization operations just as locks and that there should be no other constructs exposing properties of the implementation. In RTTM, the basic functionality is provided by the `@atomic` method annotation alone. Although the RTTM semantics does not closely correspond to that of a single global lock (see Section 3.1), I believe it is sufficiently intuitive in small transactions used for synchronization. But the programming style must also consider the limitation of the *read set* size and *write set* size of transactions. Transgressions of these maxima as well as I/O in a transaction will degrade performance. In the current implementation, there are also limitations to code executing in a transaction, which will be discussed in this chapter.

Existing lock-based programs cannot in general be directly translated to transaction-based RTTM programs. This is because of the differing semantics and because there are programs where different threads might need to reside concurrently in critical sections in order to make progress, while transactions are serializable [BLM05]. But it is questionable if a code section which requires such concurrency would be converted into an atomic section, which should have a bound on its WCET. RTTM can also be viewed as an extension of the *compare-and-swap* instruction "to simplify the implementation of non-blocking communication algorithms" [SBV10b].

As a hardware transactional memory, RTTM requires a processor with added RTTM hardware, which is not a big issue as long as reprogrammable FPGAs are used for a research prototype. The changes to the conventional JOP design flow are small.[1]

[1] See file `readme-rttm.txt` in the source code.

5.1 Recommended programming style

The limited size of the read set and write set of a transaction must be considered by the programmer, since an overflow of the transaction cache causes an early commit. Simple accesses to data structures such as bounded queues, linked lists and balanced trees have a bounded read set and write set. A practical way to reduce the read set of a transaction is to pass data as arguments to an atomic method, since the stack is excluded from the read set.[2]

The read set and write set size as well as potential conflicts between transactions should be determined using static analysis. With data structures that might be reallocated, the read set and write set size cannot be bounded [SBV10b, Section 5.2]. If big runtime libraries are used or programs are big, the analysis of potential conflicts might become computationally infeasible [SBV10b, Section 5.2].[3]

Interaction of transactional and non-transactional code If transactions are only used to exchange data over the shared memory and access to each memory location is either entirely transactional or entirely non-transactional, RTTM behavior is intuitive.[4] When using bounded queues, all access to the data in the queues should be performed in a transaction. When using linked lists, the *data handoff* idiom [GMP06] (see Table 3.2 on page 19) can be used to access data only outside of transactions.

When a memory address is accessed by both transactional and non-transactional code, nonintuitive behavior is possible (see semantics in Section 3.1). Two basic patterns are *privatization* and *publication*, which convert "objects" between private and shared state. RTTM provides *privatization safety*. This means that changes by a transaction will be seen by all non-transactional code following a succeeding transaction.

Publication safety is not supported. This means that if a memory location was modified in a task before a transaction t_1, a succeeding transaction t_2 might not see the update of the memory location (see Table 3.5 on page 20). Publication works if the access to the publicized data is preceded by a *condition variable* set in t_1 (see Table 3.6 on page 21).

If transactions need to be executed in a specific order, a transaction t_b can wait for another transaction t_a by reading a condition variable (to be set by t_a) in a loop. t_b will then automatically retry once t_a commits. Another possibility is to use the `retry()` statement (see page 24).

[2]Modifications to data which is only read before the transaction will not be detected during the transaction.

[3]A more detailed discussion of static analysis is found in [SBV10b].

[4]In this case, the program is segregated, and *weak isolation* corresponds to *strong isolation* [MBS$^+$08].

5.2 Limitations

The RTTM implementation imposes limits on code running inside and outside of transactions. *Garbage collection* is not supported while any transaction is running. Any functionality potentially involving garbage collection, i.e. direct or indirect use of the `new` keyword, is therefore unavailable while any transaction is running. This is because a garbage collector might reallocate an object seen by a transaction and corrupt the internal representation of the object (e.g. the size of an array, potentially leading to a corrupted state).

In the current implementation, support for *exceptions* inside a transaction is very limited. Exceptions abort rather than commit a transaction.[5] `try-catch-finally` statements are not yet supported inside a transaction, since this is complicated by the (ab)use of the exception handling mechanism for transaction rollback.

`synchronized` statements or methods are allowed inside a transaction, as long as the transaction does not perform an early commit. Failing transactions will also acquire locks and therefore impair performance. If an early commit is performed before the `synchronized` statement, the transaction may deadlock with another transaction inside a `synchronized` statement. Due to the speculative execution of transactions, `wait()` and `notify()` are not allowed in a transaction.

Once a transaction performing an early commit has obtained the commit token,[6] its changes will be written to the shared memory and cannot be undone on a rollback. Such a rollback might only happen in exceptional situations (see page 45). The RTTM implementation does not automatically try to do an early commit before an I/O operation. Instead, the early commit must be performed by the program. Access to special memory areas (i.e. flash) is not supported inside transactions.

Zombie transactions are very restricted in their behavior (as discussed in Section 4.3.3) and the RTTM programmer does not need to consider their occurrence.

5.3 Testing and debugging

Testing and debugging of RTTM programs poses some theoretical and practical difficulties, as does the testing and debugging of concurrent systems in general [HcTlH95]. It might be difficult to reproduce RTTM application executions, since the temporal ordering of transactions may vary if there are any variations in the temporal behavior of the application.

A transaction might fail due to a bug in two ways: *a*) unintended transactional shared memory accesses can lead to additional detected conflicts and retries (but they should be

[5]Thrown exceptions should not contain any data, since the data might have been invalidated by the transaction rollback.

[6]Such a transaction is therefore not a doomed transaction.

detected by static analysis) or *b*) an exception thrown during the transaction is propagated outside of the transaction. Due to a program bug, a transaction might also overflow the read tag memory/write buffer and acquire the commit token early (this should again be detected by static analysis). If the transaction fails during an early commit, all its writes up to now have already become visible in the shared memory. Unintended invocations of `retry()`, `abort()` and `earlyCommit()` might also cause unwanted behavior.

The RTTM hardware can optionally gather basic per-CPU statistics, such as transaction commit and retry count and the maximum read set and write set size.[7] As a hardware TM system, an RTTM implementation would become more complex if it provided TM-specific debugging functionality [LM06].

The current version of JOP does not support a debugger. Common debugger functionality such as breakpoints is problematic inside transactions, since a transaction may abort [LM06] or see inconsistent states. As proposed in [LM06], a breakpoint should probably be delayed until the transaction is guaranteed to observe consistent data by obtaining the commit token. Similar to other TM systems [LM06], RTTM can be used to get *atomic snapshots* of multiple variables while the system is running by reading them inside a single transaction (and can also set multiple variables in an atomic operation).

[7] See class `rttm.Diagnostics`.

CHAPTER 6

Evaluation

This chapter contains a first evaluation of the RTTM implementation. The costs and the performance of the RTTM hardware are analyzed. Differences of the worst-case behavior w.r.t. the conventional JOP CMP are discussed. Last, the suitability of Java/JOP for a resource-efficient implementation of RTTM is discussed.

6.1 Hardware resource consumption and performance

The resource consumption of the RTTM prototype was determined by compiling for an Altera Cyclone II EP2C70F896C6 FPGA using Altera Quartus II (see Appendix D). The most relevant characteristics of this FPGA were introduced in Section 4.2.3. For this target, a single core minimal version of JOP has a maximum clock frequency (F_{max}) of 107 MHz.[1] This maximum clock frequency is a guiding value which the RTTM implementation should try to achieve. The minimal JOP core requires ~3000 logic cells (LCs). Table 6.1 on the following page lists more reference values of the implementation platform.

We measured the resource consumption (used logic cells and memory blocks) of selected components and the maximum clock frequency.[2] The target F_{max} was 100 MHz. The size of the transaction cache tag memory (corresponding to the size of the union of the read set and write set) and the number of cores was varied. Components of interest are the CPU-local RTTM modules and the transaction cache tag memory contained in the RTTM modules. Other RTTM-specific hardware does not consume significant resources or limit the maximum frequency.

[1]In the source code accessible as described in Appendix C, these optimizations are found in revision 7cf39dd75a29dd0d370737dec53cdd9d72f67f7b.

[2]The maximum clock frequency was estimated using the *Classic Timing Analysis* of Quartus II.

FPGA capacity			Shared mem.	Minimal JOP core		
LCs	Mem. blocks	Mem. bits	Addr. width	LCs	Mem. bits	F_{max}
68416	250	1024000	19 bits	~3000	~65000	107 MHz

Table 6.1: Implementation platform reference values

Tag memory			RTTM other	RTTM memory		Ratio RTTM/Total		F_{max}
Words	LCs	LCs ratio	LCs	KiB	Mem. blocks	LCs	Mem. blocks	MHz
16	440	1	540	0.1	4	23%	19%	100
32	830	1.9	560	0.21	4	30%	19%	102
64	1630	3.7	560	0.41	4	41%	19%	95
128	3210	7.3	580	0.83	4	54%	19%	88
256	6390	14.6	580	1.66	6	69%	26%	81
512	12680	29	580	3.31	9	81%	35%	73

Table 6.2: Scaling with tag memory size on a quad-core. All absolute resource numbers are per-CPU.

Tag memory sizes up to 64 words do not limit the maximum clock frequency much (see Table 6.2). Higher clock frequencies for big tag memories could be supported by expanding the hit detection to more than one cycle. The tag memory does not need many logic cells more than the minimum implied by the memory requirements (e.g. 1630 logic cells instead of 1216 logic cells; cf. column 2 of Table 6.2 and page 34). The ratio of the resources consumed by the RTTM hardware to the total consumed resources is depicted in columns 7 and 8. For tag memory sizes up to 64 words, the RTTM hardware does not dominate the hardware resource consumption. 32 words suffice to atomically try to move an item from a linked list to another using a simple object-oriented implementation.[3] The resources used for the tag memory grow linearly with the number of words, as can be seen from the relative size of the tag memory in column 3. The number of logic cells used for other RTTM hardware is nearly constant (column 4). While an RTTM implementation needs few memory bits to cache the data and the write set (column 5) – usually a single memory block for each one suffices – it also uses dedicated memory blocks to independently set the `read` and `dirty` bits (column 6). The RTTM hardware still consumes a bigger fraction of the logic cells available in the FPGA than of the memory blocks available (not depicted).

The horizontal scalability of the design is also satisfying, as the maximum clock

[3]See `rttm.tests.LinkedListContention.snatch()` and `rttm.common.LinkedList` in the source code. Further examples of read set and write set sizes are found in [SBV10b, Muc09].

Cores	F_{max} (MHz)	LCs used	Mem. blocks used
2	102	16%	17%
4	95	31%	34%
6	97	47%	50%
8	96	63%	67%
10	96	78%	84%
12	91	93%	100%

Table 6.3: Scaling with # of cores (with a tag memory size of 64 words)

frequency does not drop much even with a high number of cores and a corresponding high resource utilization (see Table 6.3). As the expensive part of the RTTM hardware is CPU-local, the RTTM/total resource consumption ratio does not shift significantly with the number of cores.

JOP CMP performance enhancements In the JOP CMP supporting RTTM, an RTTM module is inserted into each point-to-point SimpCon connection from a CPU to the memory arbiter. By registering all SimpCon signals in the RTTM module, a significant speedup is achieved compared to the non-RTTM JOP CMP using the TDMA arbiter, especially if a high number of cores is used. I think that the non-RTTM JOP CMP could also benefit from an additional stage in the TDMA memory arbiter, since the higher operating frequency should easily compensate for 1 or 2 additional cycles per uncached memory access.[4]

6.2 Worst-case temporal behavior

JOP is designed to make *worst-case execution time* (WCET) analysis easy. The WCET analysis of JOP is described in [SPPH10]. It consists of *microcode WCET analysis* and *Java application WCET analysis*. The microcode WCET analysis results in WCET bounds for individual bytecodes. It needs to be performed only once for a particular processor (hardware/microcode) version. Remarkably, there are no timing dependencies between bytecodes. Therefore it is easy to calculate the WCET of basic blocks.[5] Java application WCET analysis is performed at the bytecode level for application code. It is tool-based and requires some annotations at the source code level to bound complex loops.

[4]The signals for inter-CPU synchronization were also registered in the RTTM implementation. I think that this leads to a looser coupling of the CPUs with respect to their placement on the FPGA.

[5]In [SPPH10], basic blocks are sequences of instructions without any jumps or jump targets.

In the CMP version of JOP [Pit09], the main memory is accessed through a memory arbiter; WCET bounds for bytecodes accessing the main memory differ from the single core version. WCET analysis is extended to the JOP CMP using a time division multiple access (TDMA) memory arbitration policy in [Pit09, Ch. 5], where the WCET of bytecodes which access the main memory is bounded.

For this preliminary discussion, we assume that only the basic `@atomic` programming interface is used. Transactions are aborted if they conflict with a committing transaction and are (re-)executed until they commit. We also assume that no early commits occur – which should be assured through static analysis – and that each memory access performed by the arbiter takes at least 4 cycles.

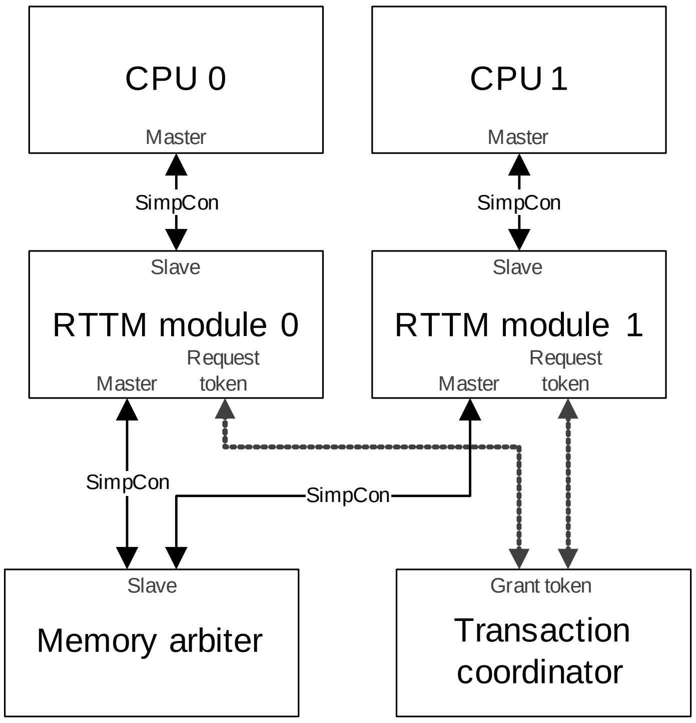

Figure 6.1: Components and interfaces which affect temporal behavior

Memory accesses The insertion of the RTTM module between CPU and memory arbiter (see Figure 6.1) and the addition of the transaction cache change the timing of memory accesses. The memory accesses use the synchronous SimpCon interconnect (see Section 1.4). Of particular interest for the worst-case behavior is the `rdy_cnt` SimpCon signal. It indicates the number of cycles at most left until the memory access is finished (3 ≡ unbounded) and determines when the CPU restarts execution as a memory access is finishing.

Compared to [Pit09, Section 5.4.4], the analysis of non-transactional memory accesses[6] needs to be modified as follows: Insert one additional cycle before each memory access is scheduled on the arbiter. In the case of a read access, insert another additional cycle after each memory access finishes (i.e. `rdy_cnt` = 0) on the arbiter.[7] In the case of a write access, the `rdy_cnt` signal generated by the RTTM module for the CPU is derived from the memory arbiter to RTTM module signal as depicted in Table 6.4.

Memory arbiter to RTTM module	3	2	1	0
RTTM module to CPU	3	3	0	0

Table 6.4: During a non-transactional write access, `rdy_cnt` values to the CPU are derived from `rdy_cnt` values to the RTTM module.[8]

For transactional memory accesses:[9] if a read access is a transaction cache miss, insert two additional cycles before each memory access is scheduled on the arbiter and insert another additional cycle after each memory access finishes. In this case, the RTTM module delays the `rdy_cnt` signal set by the memory arbiter for one cycle. For each memory location in the read set, there is at most one miss during a hardware transaction. If a read access is a transaction cache hit, 4 cycles are required to complete the access and the sequence of `rdy_cnt` values supplied by the RTTM module to the CPU is X,[10] 3, 1, 0. In the case of a transactional write access, 4 cycles are required to complete the access and the `rdy_cnt` values are also X, 3, 1, 0.

To calculate the WCET of bytecodes, this information should be integrated into the algorithm outlined in Listing 5.1 in [Pit09]. If the memory controller of the CPU would normally restart during `rdy_cnt` = 2, the possible absence of this value needs to be accounted for in the WCET analysis. There is also an assumption regarding the SimpCon interconnect between RTTM module and memory arbiter: the memory arbiter generates a `rdy_cnt` sequence passing through all `rdy_cnt` values smaller than 3, i.e. X, (3, . . . , 3,) 2, 1, 0 (so that the RTTM module can react to a finishing memory access without delay).[11]

The RTTM *hardware commands* (see page 31) are translated to the implementation-specific bytecode `jopsys_wrmem` at link time. This bytecode directly writes to the

[6]Memory accesses which are executed in a state other than `TRANSACTION` or with the `tm_cache` flag cleared.

[7]The RTTM module generates the following sequence of `rdy_cnt` values: X (not yet valid), 3, . . . , 3, 2, 1, 0.

[8]The `rdy_cnt` signal generated by the RTTM module for the CPU is allowed to run ahead of the `rdy_cnt` signal generated by the memory arbiter for the RTTM module because the next memory access will be delayed anyway.

[9]Memory accesses which are executed in state `TRANSACTION` with the `tm_cache` flag set.

[10]Since SimpCon is a synchronous interconnect, `rdy_cnt` is not yet valid (X) during the cycle in which the memory access is started.

[11]This assumption is satisfied in the current implementation.

specified memory address.[12] The RTTM module intercepts these memory accesses. The execution time of `jopsys_wrmem` depends on the hardware command:

- `start_transaction` and `aborted` take 7 cycles.
- The duration of `end_transaction` and `early_commit` depends on how long it takes to acquire the commit token or detect a conflict.

During a commit, each element of the write set is written to the shared memory once. The RTTM module issues each write one cycle after the previous write finishes (with `rdy_cnt` = 0). The first write is issued 4 cycles after the RTTM module has received the commit token. When the memory arbiter finishes the last write, the RTTM module will return `rdy_cnt` 0 in the following cycle.

Interference of commits There is also a timing difference in the behavior of the modified TDMA memory arbiter (see Section 4.2.2) w.r.t. the conventional TDMA memory arbiter analyzed in [Pit09]: while the transaction coordinator has granted the commit token to an RTTM module, time slots of CPUs executing a transaction might be reallocated to the committing CPU, i.e. all read memory accesses of CPUs executing a transaction might be stalled by the memory arbiter.

The registering of signals for inter-CPU synchronization causes lock allocation and a CMP synchronization signal to be delayed by one cycle compared to the hitherto implementation.[13]

6.3 Discussion

Next, we discuss how features of Java and JOP enable a simple and more resource-efficient implementation of RTTM.

Benefits of using Java Java and the JVM have some characteristics which facilitate the implementation of RTTM. The Java exception handling mechanism can be used to roll back transactions. The memory model makes it possible to exclude often-used memory locations such as the stack from the read set and write set. Because the instruction set of the JVM does not contain difficult to analyze instructions that are used in unsafe environments (such as non-local jumps), transformations can be performed at link time (and a standard Java compiler can be used). Link-time modification of transactions is easy

[12]The memory address and value are taken from the operand stack.

[13]The described signals use the port `sync_out_array` in `cmpsync.vhd` in the source code.

for the bytecode instruction set.[14] It would be straightforward to generate a customized transaction wrapper for each transaction which leaves out unnecessary operations, such as the handling of nested transactions. The implementation does not rely on the memory safety of Java, since the behavior of zombie transactions is restricted by other means.

Using Java also eases the tool-based static analysis of the read set and write set size and of conflicts among transactions [SBV10b].

Suitability of JOP CMP RTTM is nearly transparently integrated into the JOP CMP by inserting the RTTM module in the individual SimpCon interconnects of CPUs and memory arbiter. In the core, no structural changes were required.[15] The implementation of a part of RTTM in software also supports this transparency. I think it is also beneficial to use a hardware implementation of the JVM, which directly executes Java bytecode, since it is relatively cheap to distinguish different types of memory accesses in hardware, some of which can be excluded from the read set. The JOP architecture also contains dedicated memories for the stack and the method cache, both of which can be excluded from the read set and from write buffering.

Implementation limitations The RTTM implementation is a research prototype. I think that the restriction of (transparent) atomic sections to entire methods – instead of code blocks – might lead to fragmentation of code (or to transactions which have a big read set/write set). The restriction to atomic methods also leads to unnecessary method invocations. Method invocations are also associated with a transaction retry.[16]

[14]There are multiple frameworks for analyzing and modifying Java class files, which contain the bytecode instructions. The JOP build toolchain uses the *Byte Code Engineering Library* (BCEL, see `http://jakarta.apache.org/bcel/`).

[15]See Section 4.2.11.

[16]The method invocations are `com.jopdesign.sys.JVMHelp.except()` and `com.jopdesign.-sys.JVMHelp.handleException()` and `com.jopdesign.sys.JVM.f_athrow()`.

CHAPTER 7

Conclusion

Real-time transactional memory (RTTM) is the first transactional memory system intended for hard real-time systems running on chip multiprocessors (CMPs). In this thesis, RTTM was implemented on the JOP CMP, which is a Java Virtual Machine (JVM) implemented in hardware. A medium-size, low-cost FPGA such as the Altera Cyclone II EP2C70 can implement a 12-core version of JOP with support for RTTM. This configuration consumes almost all available resources while still running at 85% of the single core speed. A fully associative tag memory is necessary for predictably tracking addresses read and written in a transaction. This tag memory can apparently not be implemented efficiently in an FPGA and consumes most of the RTTM-specific hardware. The operating frequency is not significantly limited by a big tag memory. Using a tag memory tracking 64 words, transactions of moderate complexity can be implemented. Thus we conclude that the implementation of RTTM on FPGAs is feasible [SH10]. The relatively high hardware requirements for the fully associative tag memory are however a drawback of RTTM.

A FIFO replacement strategy can be used for the tag memory. The performance of the tag memory implemented using logic cells is good (single cycle hit detection). The logic cell variant was preferred to a variant using FPGA memory blocks, since *in the case of the JOP CMP* the inefficient use of additional memory blocks would deplete the available memory blocks when using many cores.

Part of RTTM was implemented in software. The motivation for this is to ease the integration of RTTM into the CPU and to lower the hardware resource requirements. In a pure hardware implementation, additional hardware would be required to save and restore CPU state. Instead, the Java exception mechanism is used to restore the CPU state. RTTM can then be integrated into the JOP CMP in a nonintrusive, nearly transparent way. JOP implements much of the JVM architecture directly in hardware and microcode. This makes it possible to use the strong guarantees of the JVM (e.g. stack is thread-local, memory accesses can be classified according to the bytecode causing them) to obtain an implementation consuming less hardware.

In RTTM, zombie transactions may occur if a transaction is continued after a shared memory location has been modified by another transaction. A modified memory arbitration strategy limits the zombie part of a transaction to a single bytecode without requiring changes to the core. This simplifies the reasoning about zombie transactions.

The main design goals of RTTM are *a*) a simple programming model and *b*) analyzable timing properties [SBV10b]. The basic programming model of the implementation is the `@atomic` method annotation. Assuming tool-based analysis of the read set and write set size and a *segregated* program, i.e. there are either only transactional or non-transactional accesses to a memory location, the programmer does not need to consider many limitations when using transactions for synchronization only. When accessing a memory location both inside and outside of a transaction, access to a *publicized* memory location must be guarded with a conditional due to the restricted semantics. I think that the restriction of the `@atomic` annotation to entire methods can lead to fragmentation of code; as an alternative, annotations of code blocks could be supported.

The differences in the temporal behavior of the conventional JOP CMP and the RTTM implementation have been analyzed. While all operations have a bounded execution time, there are significant differences to the WCET analysis of the conventional JOP CMP. A full WCET analysis of the RTTM implementation is considered future work.

Summary of contributions

The contributions of this thesis are the following:

- The first implementation of a time-predictable hardware transactional memory for chip multiprocessors is described and evaluated.
- Hardware savings in the TM implementation which are permitted by a safe language (Java) and a virtual machine (Java Virtual Machine) are discussed.
- The semantics of RTTM is analyzed.
- Restrictions of transactional code in the implementation are discussed.
- Different FPGA implementation techniques for a fully associative tag memory are compared in the context of the JOP CMP.

Future Work

The next step in the development of RTTM should be the adaptation of multi-threaded real-word workloads with real-time constraints for the prototype implementation and their evaluation. Both applications with lock-based synchronization and wait-free algorithms could be adapted to use transactions. Implementation of real-word workloads should also enable better assessment of the suitability of the RTTM programming model. Currently, the implementation restricts (transparently created) atomic sections to entire methods, since Java does not allow annotations at the code block level. A workaround would be to modify the Java compiler [Caz10]. Instead, we propose to (ab)use the `synchronized` code block together with a special purpose class to indicate atomic sections.[1]

Use of a single global lock is a simple alternative to obtain semantics comparable to RTTM. An important research question is under which circumstances the RTTM implementation shows better performance than such coarse-grained locking. If there are only "few" cores and there is "low" contention for the single lock, the coarse-grained locking is preferable, since it avoids the need for expensive special hardware. This research question does not only cover measured execution times, but also WCET estimates. To calculate WCET bounds, the bytecode WCET estimation from [Pit09] should be adapted to RTTM. Transaction commits, retries and the wait time for the commit token need to be integrated in the WCET analysis. The static analysis of the size of the read set and write set and the conflicts among transactions should be tool-based.[2]

In CMP versions of JOP with many cores, the bandwidth of the main memory is the bottleneck due to missing data caches [SPPH10, Section 8.6]. Possibilities to implement such a data cache with analyzable benefits even in the worst case are being investigated [SPPH10]. Because of the high costs of the fully associative tag memory, it should be reused as part of such a data cache – possibly the *object cache* evaluated in [SBV10a] – during non-transactional operation.

Other points of interest include the possibility of the integration of the RTTM implementation with a future real-time garbage collector for CMPs and the implementation cost on an ASIC instead of an FPGA (esp. w.r.t. the fully associative tag memory).

[1]The appropriate restrictions of the JVM bytecode and the control flow inside a `synchronized` block should make it possible to transform such code in a transaction safely and without great effort during link time.

[2]RTTM can be used even without a static analysis of *a*) conflicts among transactions and *b*) the read set and write set size. But due to the indeterministic nature of multithreaded programs, *a*) an unexpected number of retries or *b*) early commits performed due to read set/write set overflows may lead to deadline violations not observed during testing.

APPENDIX A

Acronyms

ASIC	Application-Specific Integrated Circuit
CAM	Content Addressable Memory
CISC	Complex Instruction Set Computer
CLDC	Connected Limited Device Configuration
CMP	Chip Multiprocessor
FIFO	First In, First Out
FPGA	Field-Programmable Gate Array
hb	happens-before
HTM	Hardware Transactional Memory
HW	Hardware
IF	Interface
JOP	Java Optimized Processor
JVM	Java Virtual Machine
LC	Logic Cell
MUX	Multiplexer
PARs	Preemptible Atomic Regions

RISC Reduced Instruction Set Computer

RTTM Real-Time Transactional Memory

SoC System-on-Chip

STM Software Transactional Memory

SW Software

TCC Transactional Memory Coherence and Consistency

TDMA Time Division Multiple Access

TM Transactional Memory

VHDL Very High Speed Integrated Circuit Hardware Description Language

WCET Worst-Case Execution Time

APPENDIX B

Code Listings

Listing B.1: Java code similar to generated *transaction wrapper* (see Section 4.3.1)

```java
public static int atomicMethod(int arg0) throws RetryException,
  AbortException, Throwable {
  int arg0Copy = 0xdeadbeef; // make compiler happy
  boolean isNotNestedTransaction =
    !Utils.inTransaction[Native.rdMem(IO_CPU_ID)];

  if (isNotNestedTransaction) {
    arg0Copy = arg0; // save method arguments
    Native.wrMem(0, IO_INT_ENA); // disable interrupts
    Utils.inTransaction[Native.rdMem(IO_CPU_ID)] = true;
  }

  while (true) {
    if (isNotNestedTransaction) {
      Native.wrMem(TM_START_TRANSACTION, MEM_TM_MAGIC);
    }

    try {
      // Not really a method invocation
      // The original method body is inserted here, return
      // statements in it are redirected to the next statement
      int result = originalMethodBody(arg0);

      if (isNotNestedTransaction) {

        // try commit
```

```
          Native.wrMem(TM_END_TRANSACTION, MEM_TM_MAGIC);
          // no exceptions happen after here

          Utils.inTransaction[Native.rdMem(IO_CPU_ID)] = false;
          Native.wrMem(1, IO_INT_ENA); // re-enable interrupts
        }
        return result;
      } catch (Throwable e) {
        // exception handling issues ABORTED HW command
        // e is RetryException, AbortException or other exception

        if (isNotNestedTransaction) {
          // reference comparison is enough for singleton
          if (e == RetryException.instance) {
            // restore method arguments
            arg0 = arg0Copy;
          } else {
            // transaction was manually aborted or a bug triggered
            Utils.inTransaction[Native.rdMem(IO_CPU_ID)] = false;
            Native.wrMem(1, IO_INT_ENA); // re-enable interrupts
            throw e;
          }
        } else { // nested transaction: propagate exception
          throw e;
        }
      }
    }
  }
}
```

Listing B.2 shows how a basic transaction consisting of a code block can be implemented non-transparently using *hardware commands* (see page 31). HW commands are invoked using the JOP-specific special method `Native.wrMem()`. During linking, calls to this method are replaced by a JOP-specific special bytecode. The nesting and the explicit abort of transactions is however not supported by the listed code. It is also assumed that no interrupts will occur.

Listing B.2: Basic transaction directly using the memory-mapped RTTM hardware interface

```
import com.jopdesign.sys.Native;
import static com.jopdesign.sys.Const.*;
```

```
public class SimpleTransactionWithDirectHWAccess {

  public void nonTransactionalMethod() {
    // non-transactional code...

    {
      // save any locals written in transaction

      while (true) {
        Native.wrMem(TM_START_TRANSACTION, MEM_TM_MAGIC);

        try {
          // transactional code block

          Native.wrMem(TM_END_TRANSACTION, MEM_TM_MAGIC);
          break;
        } catch (Throwable e) {
          // restore any locals written in transaction
        }
      }
    }

    // non-transactional code...
  }

}
```

Listing B.3: Generic content addressable memory (corresponding to fully associative tags) using memory blocks from [auR08]. The M4K component is the memory block. The address conversion is performed in line 3 and 4 of the `PORT MAP` of `m4k_inst`.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;
USE IEEE.STD_LOGIC_MISC.ALL;

-- A single M4K can make a 32 word, 7-bit wide CAM. This script can be parameterized to
    chain them together into wider/deeper CAMs.
-- Please keep the depth a multiple of 32, and the width a multiple of 7. If you don't
    need a width that is a multiple of 7, just tie off the extra
-- bits so they always match(both when writing and matching)

ENTITY M4K_CAM IS
GENERIC (
  NUM_WORDS : INTEGER := 64; -- Must be multiple of 32
  WRADDR_WIDTH : INTEGER := 6; -- Enter log2(NUM_WORDS).
  WIDTH : INTEGER := 21 -- Must be multiple of 7
);
PORT
  (
  -- Write Ports
  wrdata     : IN std_logic_vector(WIDTH-1 DOWNTO 0);
  wraddress  : IN std_logic_vector(WRADDR_WIDTH-1 DOWNTO 0);
  wrdelete_n  : IN std_logic;
  wren     : IN std_logic;
  wrclock    : IN std_logic;

  -- Match Ports
  matchdata  : IN std_logic_vector(WIDTH-1 DOWNTO 0);
  matchen    : IN std_logic;
  matchclock  : IN std_logic;
  matchaddr  : OUT  std_logic_vector(WRADDR_WIDTH-1 DOWNTO 0); -- encoded output. Remove
       this if unnecessary, as lot of logic needed to encode.
  match_onehot : BUFFER std_logic_vector(NUM_WORDS-1 DOWNTO 0); -- one-hot output
  match    : OUT std_logic -- Determines if there is a match
  );
end M4K_CAM ;

ARCHITECTURE arch OF M4K_CAM IS

COMPONENT M4K
  PORT
  (
    data   : IN STD_LOGIC_VECTOR (0 DOWNTO 0);
    wren   : IN STD_LOGIC := '1';
    wraddress  : IN STD_LOGIC_VECTOR (11 DOWNTO 0);
    rdaddress  : IN STD_LOGIC_VECTOR (6 DOWNTO 0);
    rden   : IN STD_LOGIC := '1';
    wrclock  : IN STD_LOGIC ;
    rdclock  : IN STD_LOGIC ;
    q   : OUT STD_LOGIC_VECTOR (31 DOWNTO 0)
  );
END COMPONENT;


CONSTANT NUM_M4KS_DEPTH : INTEGER := NUM_WORDS/32;
CONSTANT NUM_M4KS_WIDTH : INTEGER := WIDTH/7;

SIGNAL data: std_logic_vector(0 DOWNTO 0);
SIGNAL address: std_logic_vector(WIDTH+4 DOWNTO 0);
SIGNAL m4k_wren : STD_LOGIC_VECTOR(NUM_M4KS_DEPTH-1 DOWNTO 0);
TYPE single_match_array IS ARRAY (NUM_M4KS_WIDTH-1 DOWNTO 0) OF STD_LOGIC_VECTOR(
    NUM_WORDS-1 DOWNTO 0);
SIGNAL single_match_onehot : single_match_array;

begin

  data(0) <= wrdelete_n;

instantiate_M4Ks_depth : FOR i IN 0 TO NUM_M4KS_DEPTH-1 GENERATE

  m4k_wren(i) <= wren WHEN (to_integer(unsigned(wraddress(WRADDR_WIDTH-1 DOWNTO 5))) = i)
       ELSE '0';
  instantiate_M4Ks_width : FOR j IN 0 TO NUM_M4KS_WIDTH-1 GENERATE

    m4k_inst : M4K
    PORT MAP (
        data(0) => data(0),
        wren  => m4k_wren(i),
        wraddress => wrdata(7*j+6 DOWNTO 7*j) & wraddress(4 DOWNTO 0) ,
        rdaddress => matchdata(7*j+6 DOWNTO 7*j),
        rden => matchen,
        wrclock => wrclock,
        rdclock => matchclock,
        q => single_match_onehot(j)(i*32+31 DOWNTO i*32)
        );

  END GENERATE instantiate_M4Ks_width;
END GENERATE instantiate_M4Ks_depth;

-- For a match, all the M4Ks must output a 1 on the same bit. This ANDs them together to
     find the correct output. For example, if the word is 14 bits long,
-- and we do a match on "0000001_0000011" then the first 7 bits might match many
    locations and the second 7 bits might match many locations. We only want
-- the locations that match both bits(i.e. AND them together)
PROCESS(single_match_onehot)
  variable var_match_onehot : std_logic_vector(NUM_WORDS-1 DOWNTO 0);
  BEGIN
  var_match_onehot := single_match_onehot(0)(NUM_WORDS-1 DOWNTO 0); -- need to
      initialize with a 1 the ones that could possibly match up
  FOR k IN 0 TO NUM_M4KS_WIDTH-1 LOOP
    FOR m IN 0 TO NUM_WORDS-1 LOOP
      var_match_onehot(m) := var_match_onehot(m) AND single_match_onehot(k)(m);
    END LOOP;
  END LOOP;
  match_onehot <= var_match_onehot;
END PROCESS;

-- This process encodes the match output. So 000...0001000 becomes 11. (Bit 3 is on, so
    output a binary 3)
-- Note that this does not do priority encoding. So if there are multiple matches, this
    will give erroneous data.
PROCESS(match_onehot)
  variable code : STD_LOGIC_VECTOR(matchaddr'RANGE);
  BEGIN
    code := (others => '0');
    for N in match_onehot'RANGE loop
      if match_onehot(N) = '1' then
        code := code OR std_logic_vector(to_unsigned(N, code'LENGTH));
      end if;
    end loop;
  matchaddr <= code;
END PROCESS;

match <= OR_REDUCE(match_onehot);

end;
```

APPENDIX C

Source code availability

The source code for the implementation and evaluation of RTTM is licensed under the GNU General Public License, Version 3. The source code is part of the Java Optimized Processor JOP and may be obtained from `http://www.jopdesign.com/`. The implementation uses the VHDL and Java programming languages. The file `vhdl/rttm/readme-rttm.txt` in the source code gives an introduction.

APPENDIX D

Measurements

The resource consumption was determined by compiling for the EP2C70F896C6 FPGA using Altera Quartus II 9.0sp2 Web Edition, which is freely available at `http://www.altera.com/`. The maximum clock frequency was estimated using the *Classic Timing Analysis* of Quartus II.

The source code of the RTTM implementation is in a Git repository accessible as described in Appendix C. The revision used for the measurements can be obtained by running the commands in Listing D.1; the current revision, as of May 2010, should perform slightly better w.r.t. resource usage and maximum clock frequency. The changes in the file `vhdl/rttm/performance-measurements.patch` disable unused hardware and disable RTTM instrumentation, which is not implemented efficiently yet.

```
#!/bin/bash
git checkout a5501f726629e5c2d3ad9bfc1545f62ea4711fe5
git show a8e3223174c9acb53d4dcc3cae14e97ce80b8df9:vhdl/rttm/
   performance-measurements.patch|git apply -
```

Listing D.1: Script to obtain source code used for measurements

Index

Bibliography

[ARJ97] James H. Anderson, Srikanth Ramamurthy, and Kevin Jeffay. Real-time computing with lock-free shared objects. *ACM Trans. Comput. Syst.*, 15(2):134–165, 1997.

[ARMJ97] James H. Anderson, Srikanth Ramamurthy, Mark Moir, and Kevin Jeffay. Lock-free transactions for real-time systems. In *In Real-Time Databases: Issues and Applications*, pages 107–114. Kluwer Academic Publishers, 1997.

[auR08] alteraforum.com user Rysc. Parameterized CAMs. Posted at `http://www.alteraforum.com/forum/showpost.php?p=7735&postcount=6`, 2008.

[BG02] Jean-Louis Brelet and Lakshmi Gopalakrishnan. Using Virtex-II Block RAM for High Performance Read/Write CAMs. Xilinx Application Note 260, `http://www.xilinx.com/support/documentation/application_notes/xapp260.pdf`, 2002.

[BHHR] Jayaram Bobba, Mark Hill, Tim Harris, and Ravi Rajwar. Transactional memory bibliography. `http://www.cs.wisc.edu/trans-memory/biblio/`.

[BLM05] Colin Blundell, E Christopher Lewis, and Milo M. K. Martin. Deconstructing transactions: The subtleties of atomicity. In *Fourth Annual Workshop on Duplicating, Deconstructing, and Debunking*. 2005.

[Boe09] Hans-J. Boehm. Transactional memory should be an implementation technique, not a programming interface. In *HotPar '09: Proc. 1st Workshop on Hot Topics in Parallelism*, 2009.

[Bre99] Jean-Louis Brelet. An Overview of Multiple CAM Designs in Virtex Family Devices. Xilinx Application Note 201, `http://www.xilinx.com/support/documentation/application_notes/xapp201.pdf`, 1999.

[Caz10] Walter Cazzola. @Java. A Java Annotation Extension. `http://homes.dico.unimi.it/~cazzola/atjava.html`, 2010.

[CW05] P. Capewell and I. Watson. A RISC hardware platform for low power Java. In *VLSI Design. 18th International Conference on*, pages 138–143, 2005.

[DFL$^+$06] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid transactional memory. *SIGPLAN Not.*, 41(11):336–346, 2006.

[DS07] Dave Dice and Nir Shavit. Understanding tradeoffs in software transactional memory. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, pages 21–33, Washington, DC, USA, 2007. IEEE Computer Society.

[DSS06] Dave Dice, Ori Shalev, and Nir Shavit. Transactional Locking II. In *In Proc. of the 20th Intl. Symp. on Distributed Computing*, 2006.

[FRJ09] Sherif F. Fahmy, Binoy Ravindran, and E. D. Jensen. Response time analysis of software transactional memory-based distributed real-time systems. In *SAC '09: Proceedings of the 2009 ACM symposium on Applied Computing*, pages 334–338, New York, NY, USA, 2009. ACM.

[GB00] James Gosling and Greg Bollella. *The Real-Time Specification for Java*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition)*. Addison-Wesley Professional, 2005.

[GLD00] Steve Guccione, Delon Levi, and Daniel Downs. A reconfigurable content addressable memory. In *IPDPS '00: Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing*, pages 882–889, London, UK, 2000. Springer-Verlag.

[GMP06] Dan Grossman, Jeremy Manson, and William Pugh. What do high-level memory models mean for transactions? In *MSPC '06: Proceedings of the 2006 workshop on Memory system performance and correctness*, pages 62–69, New York, NY, USA, 2006. ACM.

[Goo83] James R. Goodman. Using cache memory to reduce processor-memory traffic. In *ISCA '83: Proceedings of the 10th annual international symposium on Computer architecture*, pages 124–131, New York, NY, USA, 1983. ACM.

[GZU+09] Vladimir Gajinov, Ferad Zyulkyarov, Osman S. Unsal, Adrian Cristal, Eduard Ayguade, Tim Harris, and Mateo Valero. QuakeTM: parallelizing a complex sequential application using transactional memory. In *ICS '09: Proceedings of the 23rd international conference on Supercomputing*, pages 126–135, New York, NY, USA, 2009. ACM.

[HA06] Philip Holman and James H. Anderson. Supporting lock-free synchronization in Pfair-scheduled real-time systems. *J. Parallel Distrib. Comput.*, 66(1):47–67, 2006.

[HcTlH95] Gwan-Hwan Hwang, Kuo chung Tai, and Ting lu Huang. Reachability testing: An approach to testing concurrent software. *International Journal of Software Engineering and Knowledge Engineering*, 5:493–510, 1995.

[Her06] Maurice Herlihy. *The art of multiprocessor programming*. ACM, New York, NY, USA, 2006.

[HHL+09] 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.

[HM93] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300. 1993.

[HMPJH05] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, New York, NY, USA, 2005. ACM.

[HR83] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, 1983.

[HWC+04] L. Hammond, V. Wong, M. Chen, B.D. Carlstrom, J.D. Davis, B. Hertzberg, M.K. Prabhu, Honggo Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on*, pages 102–113, 2004.

[Hyd03] Randall Hyde. *Write Great Code: Understanding the Machine*. No Starch Press, San Francisco, CA, USA, 2003.

[KHR+08] Behram Khan, Matthew Horsnell, Ian Rogers, Mikel Luján, Andrew Dinn, and Ian Watson. An object-aware hardware transactional memory system. In *HPCC '08: Proceedings of the 2008 10th IEEE International Conference on High Performance Computing and Communications*, pages 93–102, Washington, DC, USA, 2008. IEEE Computer Society.

[KMVR90] C. Koelbel, P. Mehrotra, and J. Van Rosendale. Supporting shared data structures on distributed memory architectures. In *PPOPP '90: Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming*, pages 177–186, New York, NY, USA, 1990. ACM.

[Kni86] Tom Knight. An architecture for mostly functional languages. In *LFP '86: Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 105–112, New York, NY, USA, 1986. ACM.

[Kop08] Hermann Kopetz. The complexity challenge in embedded system design. In *ISORC '08: Proceedings of the 2008 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing*, pages 3–12, Washington, DC, USA, 2008. IEEE Computer Society.

[Lie04] Sean Lie. Hardware support for unbounded transactional memory. Master's thesis, 2004. Massachusetts Institute of Technology.

[LM06] Yossi Lev and Mark Moir. Debugging with transactional memory. In *Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*. 2006.

[LR07] Jim Larus and Ravi Rajwar. *Transactional Memory (Synthesis Lectures on Computer Architecture)*. Morgan & Claypool Publishers, 2007.

[Luc08] Victor Luchangco. Against lock-based semantics for transactional memory. In *SPAA '08: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 98–100, New York, NY, USA, 2008. ACM.

[LY99] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[MBC+05] Jeremy Manson, Jason Baker, Antonio Cunei, Suresh Jagannathan, Marek Prochazka, Bin Xin, and Jan Vitek. Preemptible Atomic Regions for Real-Time Java. In *RTSS '05: Proceedings of the 26th IEEE International Real-Time Systems Symposium*, pages 62–71, Washington, DC, USA, 2005. IEEE Computer Society. Extended version available at `http://www.cs.purdue.edu/homes/jv/pubs/rtss05.pdf`.

[MBM+06] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. LogTM: Log-based Transactional Memory. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, pages 254–265. 2006.

[MBS+08] Vijay Menon, Steven Balensiefer, Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Bratin Saha, and Adam Welc. Single global lock semantics in a weakly atomic STM. *SIGPLAN Not.*, 43(5):15–26, 2008.

[MCC+05] Austen Mcdonald, Jaewoong Chung, Hassan Chafi, Chi Cao Minh, Brian D. Carlstrom, Lance Hammond, Christos Kozyrakis, and Kunle Olukotun. Characterization of TCC on chip-multiprocessors. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 63–74. IEEE Computer Society, 2005.

[MO98] H. McGhan and M. O'Connor. PicoJava: a direct execution engine for Java bytecode. *Computer*, 31(10):22–30, 1998.

[Moo98] G. E. Moore. Cramming More Components Onto Integrated Circuits. *Proceedings of the IEEE*, 86(1):82–85, 1998.

[MS96] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proc. 15th ACM Symp. on Principles of Distributed Computing*, pages 267–275, 1996.

[Muc09] Michael Muck. Transactional Memory Beispiele für den Jop Multiprozessor Simulator. Term paper, 2009.

[OH05] Kunle Olukotun and Lance Hammond. The future of microprocessors. *Queue*, 3(7):26–29, 2005.

[Pit09] Christof Pitter. *Time-Predictable Java Chip-Multiprocessor*. PhD thesis, Vienna University of Technology, 2009.

[SBV10a] Martin Schoeberl, Walter Binder, and Alex Villazon. Object cache evaluation. Technical report, 2010. Available at `http://www.jopdesign.com/doc/troceval.pdf`.

[SBV10b] Martin Schoeberl, Florian Brandner, and Jan Vitek. RTTM: Real-time transactional memory. In *Proceedings of the 25th ACM Symposium on Applied Computing*, Sierre, Switzerland, 2010. ACM Press. Available at `http://www.jopdesign.com/doc/rttm.pdf`.

[Sch05] Martin Schoeberl. *JOP: A Java Optimized Processor for Embedded Real-Time Systems*. PhD thesis, Vienna University of Technology, 2005.

[Sch07] Martin Schoeberl. SimpCon - a simple and efficient SoC interconnect. In *Proceedings of the 15th Austrian Workshop on Microelectronics, Austrochip 2007*, Graz, Austria, 2007.

[Sch08] Martin Schoeberl. A Java processor architecture for embedded real-time systems. *J. Syst. Archit.*, 54(1-2):265–286, 2008.

[Sch09] Martin Schoeberl. *JOP Reference Handbook: Building Embedded Systems with a Java Processor*. CreateSpace, 2009. ISBN 978-1438239699. Available at `http://www.jopdesign.com/doc/handbook.pdf`.

[Sco06] Michael L. Scott. Sequential specification of transactional memory semantics. In *ACM SIGPLAN Workshop on Transactional Computing*. 2006. Held in conjunction with PLDI 2006.

[SDMS08] Michael F. Spear, Luke Dalessandro, Virendra J. Marathe, and Michael L. Scott. Ordering-based semantics for software transactional memory. In *OPODIS '08: Proceedings of the 12th International Conference on Principles of Distributed Systems*, pages 275–294, Berlin, Heidelberg, 2008. Springer-Verlag.

[SH10] Martin Schoeberl and Peter Hilber. Design and Implementation of Real-Time Transactional Memory. In *FPL '10: Proceedings of the 20th International Conference on Field Programmable Logic and Applications*, Milano, Italy, 2010.

[SMDS07] Michael F. Spear, Virendra J. Marathe, Luke Dalessandro, and Michael L. Scott. Privatization techniques for software transactional memory. In *PODC '07: Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 338–339, New York, NY, USA, 2007. ACM.

[SPPH10] Martin Schoeberl, Wolfgang Puffitsch, Rasmus Ulslev Pedersen, and Benedikt Huber. Worst-case execution time analysis for a Java processor. *Software: Practice and Experience*, accepted for publication, 2010.

[SQV09] Toufik Sarni, Audrey Queudet, and Patrick Valduriez. Real-Time Support for Software Transactional Memory. *Real-Time Computing Systems and Applications, International Workshop on*, 0:477–485, 2009.

[ST95] Nir Shavit and Dan Touitou. Software transactional memory. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213, New York, NY, USA, 1995. ACM.

[Sun03] Sun Microsystems, Inc. *Connected Limited Device Configuration Specification, Version 1.1*, 2003. `http://jcp.org/aboutJava/communityprocess/final/jsr139/`.

[WCN+07] Sewook Wee, Jared Casper, Njuguna Njoroge, Yuriy Tesylar, Daxia Ge, Christos Kozyrakis, and Kunle Olukotun. A practical FPGA-based framework for novel CMP research. In *FPGA '07: Proceedings of the 2007 ACM/SIGDA 15th international symposium on Field programmable gate arrays*, pages 116–125, New York, NY, USA, 2007. ACM.

[Xil08] Xilinx. Content-Addressable Memory v6.1. Xilinx product specification, `http://www.xilinx.com/support/documentation/ip_documentation/cam_ds253.pdf`, 2008.