Transactional Memory for High-Performance Embedded Systems

Dissertation
zur Erlangung des akademischen Grades eines
Doktors der Ingenieurwissenschaften (Dr.-Ing.)
der Fakultät für Angewandte Informatik
der Universität Augsburg

Universität
Augsburg
University

eingereicht von
Christian Piatka, M.Sc.

Transactional Memory for High-Performance Embedded Systems

Christian Piatka, M.Sc.

Erstgutachter: Prof. Dr. Sebastian Altmeyer
Zweitgutachter: Prof. Dr. Theo Ungerer
Tag der mündlichen Prüfung: 20.03.2023

Kurzfassung

Der immer größer werdende Bedarf an Rechenleistung in eingebetteten Systemen, der für verschiedene Aufgaben wie z. B. dem autonomen Fahren benötigt wird, kann nur durch die effiziente Nutzung der zur Verfügung stehenden Ressourcen erreicht werden. Durch physikalische Grenzen sind Prozessorhersteller dazu übergegangen, Prozessoren mit mehreren Prozessorkernen auszustatten, statt die Taktraten weiter anzuheben. Daher kann die zusätzlich benötigte Rechenleistung aus unserer Sicht nur durch eine Steigerung der Parallelität gelingen.

Hardwaretransaktionsspeicher (HTS) erlauben es ihren Nutzern schnell und einfach parallele Programme zu schreiben. Allerdings wurden HTS nicht speziell für eingebettete Systeme entwickelt und sind daher nur eingeschränkt für diese nutzbar. Durch den Einsatz herkömmlicher HTS steigt die Komplexität und es wird somit schwieriger abzusehen, ob andere wichtige Eigenschaften erreicht werden können.

Um den Einsatz von HTS in eingebettete Systeme besser zu ermöglichen, beschreibt diese Arbeit einen konkreten Ansatz. Der HTS wurde hierzu so entwickelt, dass er eine parallele Ausführung von Programmen ermöglicht und Eigenschaften besitzt, welche für eingebettete Systeme nützlich sind. Dazu gehören unter anderem: Wegfall der typischen Limitierungen herkömmlicher HTS, Einflussnahme auf den Konfliktauflösungsmechanismus, Unterstützung einer abschätzbaren Ausführung und eine Funktion, um Energie einzusparen.

Um die gewünschten Funktionalitäten zu ermöglichen, unterscheidet sich der Aufbau des in dieser Arbeit beschriebenen HTS stark von einem klassischen HTS. Im Vergleich zu dem Referenz HTS, der ebenfalls im Rahmen dieser Arbeit entworfen und implementiert wurde, betrifft die größte Anpassung die Konflikterkennung. Sie wurde derart verändert, dass die Konflikte zentral erkannt und aufgelöst werden können. Hierfür mussten die Cache-Hierarchie und Cache-Kohärenz stark angepasst und teilweise erweitert werden.

Das System wurde in einem taktgenauen Simulator, dem gem5-Simulator, umgesetzt. Zur Evaluation wurden die acht Benchmarks der STAMP-Benchmark-Suite eingesetzt. Die Evaluation der verschiedenen Funktionen zeigt, dass die Mechanismen funktionieren und somit einen Mehrwert für eingebettete Systeme bieten.

Abstract

The increasing demand for computational power in embedded systems, which is required for various tasks, such as autonomous driving, can only be achieved by exploiting the resources offered by modern hardware. Due to physical limitations, hardware manufacturers have moved to increase the number of cores per processor instead of further increasing clock rates. Therefore, in our view, the additionally required computing power can only be achieved by exploiting parallelism. Unfortunately writing parallel code is considered a difficult and complex task.

Hardware Transactional Memories (HTMs) are a suitable tool to write sophisticated parallel software. However, HTMs were not specifically developed for embedded systems and therefore cannot be used without consideration. The use of conventional HTMs increases complexity and makes it more difficult to foresee implications with other important properties of embedded systems.

This thesis therefore describes how an HTM for embedded systems could be implemented. The HTM was designed to allow the parallel execution of software and to offer functionality which is useful for embedded systems. Hereby the focus lay on: elimination of the typical limitations of conventional HTMs, several conflict resolution mechanisms, investigation of real time behavior, and a feature to conserve energy.

To enable the desired functionalities, the structure of the HTM described in this work strongly differs from a conventional HTM. In comparison to the baseline HTM, which was also designed and implemented in this thesis, the biggest adaptation concerns the conflict detection. It was modified so that conflicts can be detected and resolved centrally. For this, the cache hierarchy as well as the cache coherence had to be adapted and partially extended.

The system was implemented in the cycle-accurate gem5 simulator. The eight benchmarks of the STAMP benchmark suite were used for evaluation. The evaluation of the various functionalities shows that the mechanisms work and add value for the operation in embedded systems.

Danksagung

An dieser Stelle möchte ich mich für die hervorragende Betreuung während meiner Promotion bei Herrn Prof. Theo Ungerer und Herrn Prof. Sebastian Altmeyer bedanken. Auch bei Herrn Prof. Jörg Hähner möchte ich mich bedanken, da er als Drittprüfer an meiner mündlichen Prüfung teilgenommen hat.

Ebenfalls möchte ich mich bei meinen Kollegen hier am Lehrstuhl bedanken. Die vielen Gespräche und Diskussionen im Rahmen unserer Mittagsrunde, aber auch zu anderen Gelegenheiten waren immer sehr anregend. Diese werden mir sehr fehlen. Des Weiteren habe ich den immer sehr kollegialen Umgang geschätzt. Besonders bei Rico Amslinger, Florian Haas und Sebastian Weis möchte ich mich an dieser Stelle bedanken, da ich mit ihnen sehr eng zusammengearbeitet habe.

Der größte Dank gilt meiner Familie, insbesondere meiner Frau Helen, unseren drei Kindern sowie meinen Eltern und Geschwistern, da diese Arbeit ohne ihre Unterstützung nicht möglich gewesen wäre.

Christian Piatka
Augsburg im Juli 2023

Acronyms

ACID **A**tomicity, **C**onsistency, **I**solation and **D**urability

ASF Advanced Synchronization Facility

COTS common of the shelf

DMB Data Memory Barrier

FPGA Field Gate Programmable Array

HTM Hardware Transactional Memory

HTS Hardwaretransaktionsspeicher

ISA Instruction Set Architecture

LLC Last Level Cache

LSU Load/Store Unit

OS Operating System

PC program counter

RC Release Consistency

RS read set

SMT Simultaneous Multithreading

STM Software Transactional Memory

TLE Transactional Lock Elision

TM Transactional Memory

TME Transactional Memory Extensions

TMU Transaction Management Unit

TSX Transactional Synchronization Extensions

WCET worst case execution time

WS write set

XI cross interrogates

Contents

1

Introduction

Contents

In 1965 Gordon E. Moore predicted a trend concerning the development of integrated circuits for the computer industry. The prediction is known as Moore's law and implies that the numbers of transistors on a computer-chip can be doubled every 18 to 24 months [52, 31, p. 19].

For nearly 30 years the frequency at which a processor could be run by also doubled whenever the number of transistors doubled. Since the power needed to run a transistor is proportional to its size, it is possible to reduce the power supply needed to run a transistor when it is reduced in size. Another effect of decreasing a transistor's size is that it can be operated at a higher frequency. Since the power consumption of a transistor decreases energy density stays constant, even though more transistors are located within the same space and frequency is higher. This effect is known as Dennard scaling.[17]

In the early two thousands it was no longer possible to further decrease the power supply for transistors. Due to physical limitations a certain voltage is needed to power a transistor even though they continue to decrease in size [21]. Therefore, it was also not possible to further increase frequency, since otherwise the energy density would have increased and problems with heat development would have occurred. Since no further performance gains were possible, but the number of transistors continued to increase, the processor industry shifted from producing single core chips to manufacturing multi-

core chips to enable performance gains though parallel execution [21]. Therefore, nowadays multi-core processors have become ubiquitous.

Supplying adequate software for multi-cores is complex, since writing correct parallel code is a challenging task [29, p. 1]. Transactional Memories (TMs) are synchronization mechanisms which simplify this process. Therefore, this technology has become interesting for research and industry. In the following we will further motivate why we focus on using a Hardware Transactional Memory (HTM) and how we consider using it for embedded systems. Next, we will lay out the objectives and contributions of this work. After we declare where a part of this work is published, we will close this chapter by giving an overview of how this work is structured.

1.1 Motivation

The demand for embedded systems which offer high performance will most likely increase in the near future. Since e.g., cars or other complex embedded systems will have to process a lot of data to provide their functionality, multi-core systems must be exploited to a high degree to provide the necessary performance.

Providing multi-cores with sufficient software is not trivial [29, p. 1]. Writing correct parallel programs which exploit the available computational power is not easy and is therefore considered a difficult task. One of the key problems is to set the right amount of synchronization [39]: Potential parallelism may be lost due to too coarse synchronization settings prohibiting the parallel execution of software. If synchronization is set too fine parallelism may be increased, but errors might occur since this approach is considered very error prone. TMs simplify the process of writing a parallel program and leave it to software or hardware to handle synchronization. In the optimal case a user only must set the beginning and end of a critical section. An instance executing the critical section is called a transaction. TMs use optimistic synchronization which allows multiple concurrent transactions to simultaneous access critical data. In our work we focus on an HTM, which realizes its functionality in hardware. We do not believe that a Software Transactional Memory (STM), which is realized purely in software, allows the adaptations for the functionality we want to offer.

Embedded systems which do not require high performance might not need hardware offering the possibility to execute parallel code, since their workloads can be handled by a single core chip. These systems are not focus of this work, since TMs are not beneficial in such a case. In our work we therefore want to supply an HTM specifically for high-performance embedded systems, since we find that conventional HTMs do not offer the functionality and flexibility needed in modern embedded systems. In the following

we discuss some problems with conventional HTMs and lay out which functionality an HTM for embedded systems should offer in our opinion.

Conventional HTMs often suffer from hardware limitations, meaning a transaction has to be aborted when exceeding the available resources. We believe that such hardware limitations should not be an issue, since otherwise already made computational progress might be discarded. Therefore, we believe that an HTM for embedded systems has to provide a mechanism which allows transactions of arbitrary size. Other transactions should be affected as little as possible if a transaction exceeds available resources. Optimally, they should be able to continue execution until the transaction ends.

A conflict resolution policy determines how the underlying HTM handles a conflict. Most HTMs only offer one conflict resolution policy. Therefore, they are very static meaning that the contention management strategy cannot be adapted even though a different strategy would be more suitable for the execution. If multiple strategies could be offered, more control over the behavior of the execution of software could be provided. Changing the strategy potentially allows further optimizations (performance, power consumption, etc.). Among others, we believe that providing a conflict resolution policy which can handle priorities is important, since embedded systems often require priorities. Therefore, we believe it is important to provide several contention resolution policies to satisfy the needs of an execution performed on an embedded system.

In our opinion writing software for an HTM has to be as simple as possible. Many HTMs require an alternative path of execution if a transaction continuously cannot be executed. The alternative path of execution is often referred to as fallback path. We believe that there should be no necessity of providing fallback paths when writing software for embedded systems. Otherwise, an alternative synchronization mechanism is needed which adds complexity to the development process. The fallback path must also be implemented carefully and raises the question of why the programmer should not use a different synchronization mechanism right away, since it also must be implemented.

Depending on where embedded systems are used, they require additional features. If embedded systems are integrated into high-realiability machines (e.g., planes) software executed on the embedded systems must be analyzable to guarantee e.g., real-time requirements. Also, power reduction mechanisms are important for embedded systems, since they often have power constraints. Conventional HTMs do not consider the possibility to analyze execution and do not provide any power saving mechanisms. Therefore, such features must also be considered when providing an HTM for embedded systems.

As pointed out conventional HTMs are not thought to be used in embedded systems. Therefore, an HTM developed for embedded systems should eliminate the typical limitations of conventional HTMs, offer several conflict resolution mechanisms, allow the investigation of real time behavior, and supply features to conserve energy.

1.2 Objectives and Contributions

With our work we want to offer an HTM which provides functionality useful for modern embedded systems. Our goal is to develop an HTM which is able to execute transactions of arbitrary size, offers multiple conflict resolution strategies, does not require fallback paths, is analyzable and offers the possibility to save power. To provide a holistic analysis of the behavior we targeted to implement our HTM into an cycle-accurate simulator.

The following contributions were made in this doctoral thesis:

- Implementation of an HTM based on MOSI cache coherence
- Implementation of a mechanism to support transactions of arbitrary size
- Provision of multiple hardware contention management strategies
- Investigation of real time behavior
- Special abort-aware execution mode which allows power saving options when executing software
- False conflict detection and reduction
- Holistic evaluation of performance and overheads of the proposed HTM implementation

1.3 Publications

An early stage of the work described in Chapter 5 is published in [44]. In contrast to the work provided by this thesis we there described an extra hardware unit, the Transaction Management Unit (TMU), which provided the functionalities to offer the desired features. As we continued to further develop our system, we eventually integrated the functionalities provided by the TMU to the Last Level Cache (LLC) controller (see Chapter 5 for more details) and therefore got rid of the TMU.

1.4 Overview

The next chapter of this thesis provides the background to better understand how we implemented our work. We here present an overview on TMs as well as cache coherence. In Chapter 3 we present and summarize related work. At the end of this chapter, we provide a table which compares the presented work to our proposal. Chapter 4 describes how we implemented the underlying baseline HTM. Among other things relevant for the Implementation, we here in particular describe the hardware setup, how we adapted the cache coherence protocol and how we adjusted the cache controllers. At the end of this chapter, we evaluate the HTM by executing the benchmarks from the STAMP benchmark suite. In Chapter 5 we explain how we adapted the baseline HTM developed in Chapter 4 to be able to perform conflict detection at the LLC level. In this chapter we also present the details on how we implemented the features the adapted HTM provides. At the end of this chapter, we provide a detailed evaluation. Here we also investigate where potential overheads are generated. We conclude our work by summarizing it in Chapter 6. In this chapter we also give a conclusion and an outlook on future work.

2

Background

Contents

In this chapter we will explain the mechanisms and tools used in this work to allow a better understanding of the following chapters. Additionally, we will provide information on the topics which are discussed later but are not the focus of this work (e.g., STM). In the following we will therefore first give a general overview of TMs. Since we had to make changes to the cache coherence protocol it is important to understand the underlying mechanisms. Therefore, we provide an overview of the techniques needed to keep the caches coherent. To implement our work, we used the gem5 simulator. Therefore, we will next give an overview of the gem5 simulator. Here, we will give a brief overview on how the gem5 simulator is set up and how it can be configured. Furthermore, we will here also lay out the reasons for some design choices we made in Chapter 4. We will close this chapter by giving a brief overview of the STAMP benchmark suite. The suite consists of eight benchmarks which we used for the evaluations in Chapter 4 and 5. Therefore, we present the key characteristic of each of the eight benchmarks.

2.1 Transactional Memories

The idea which lead to TMs originates from databases. Here, multiple operations, which perform read or write accesses to the database, are tied together, and treated as a whole. The combined operations are called transactions. Transactions must be executed and treated as one atomic operation. If a conflict occurs with another transaction, the effects of one of the conflicting transactions have to be undone. A conflict occurs if a data dependency is violated (see Section 2.1.3 for more details). To be able to perform a rollback of a transaction, a transaction log has to be maintained. It records the changes which were made to the database. In case of a conflict the log is used to restore the database to a consistent state. Database transactions must fulfill the **A**tomicity, **C**onsistency, **I**solation and **D**urability (ACID) properties (see Section 2.1.1 for more details). The most important characteristic, which can be adopted from database transactions, is the optimistic access to data. Hereby, two transactions can both access the same data if no data dependency is violated. [29, pp. 4-5]

2.1.1 Properties of Transactional Memories

A transaction within TMs is a sequence of operations which performs read and/or write accesses to the memory. Transactions in TMs originate from database transactions but do not entirely fulfill their properties. Therefore, transactions of TMs are referred to as light weighted transactions. In contrast to database transactions, transactions in TMs do not meet the consistency and durability criteria. Since the consistency property refers to data integrity, which depends on how the database tables are set up, it is not a relevant property for transactions in TMs. The durability property is also not relevant for transactions in TMs, since it requires that data changed by a database transaction is permanent and is written to a durable medium such as a disk. Therefore, the criteria is also only relevant for database transactions. If not stated differently, we from now on refer to a transaction in a TM when using the term transaction by itself. In the following we give an overview on the properties which apply to transactions: [30, 29, pp. 5-6, 32, p. 2080, 39]

Atomicity: A transaction can either commit or abort. If a transaction commits, the effects of the transaction become visible to other tasks and transactions. A transaction can only be committed if all its operations are executed without failing. In case an operation could not be executed successfully since a conflict occurred (see 2.1.3 for more details), the transaction has to be aborted and the already performed changes have to be discarded. Furthermore, the TM has to ensure that the effects of the already executed operations of the aborted transaction are undone. This form of atomicity is referred to as failure atomicity. [29, p. 5, 30, 33]

Isolation: The effects of a transaction are only determined by the operations within a transaction and have to be independent of other tasks. Therefore, a transaction always has to produce the same result as if it were executed exclusively. [39]

Due to the properties described above transactions are serializable. Although concurrent transactions as well as concurrent code can be executed interleaved, they always produce a result which could have been produced by an instance of a sequential execution of the transactions and the code. [29, pp. 25-26, 33]

2.1.2 Concurrency Control

Concurrency control defines how data can be accessed by transactions within the transactional execution. Furthermore, it has a major impact on when conflicts are detected and resolved [29, p. 20]. Concurrency control is provided in two different ways:

Pessimistic concurrency control: Access to critical data is only granted to one transaction. Therefore, a potential conflict will not arise, since the access which would cause a conflict is delayed until the data is accessible again. The data can only be accessed by another transaction, when the transaction, which first accessed the data, finishes. [29, p. 20]

Optimistic concurrency control: This form of cocurrency control allows multiple transactions to access the same data. If a conflict arises the TM has to ensure that they are resolved before the transactions are able to commit. This form of concurency control is used for most implementations of TMs. [29, pp. 21-22, 39]

When implementing pessimistic or optimistic concurrency control it is important to note that the mechanisms themselves do not ensure progress. Deadlocks, where two concurrent running transactions wait for the respective other to allow access to the data it currently holds, can occur in faulty implementations of pessimistic concurrency control. Therefore, techniques (e.g., contention management) preventing deadlocks have to be considered during the development process when implementing pessimistic concurrency control. For optimistic concurrency control, livelocks have to be considered. Here two or more transactions may keep aborting each other and no progress is made. This can also be solved by providing contention management. [29, p. 21]

Pessimistic and optimistic concurrency control can also be combined. If a transaction keeps aborting a surpassed a predefined number of aborts, when executed optimistically, it can be executed pessimistic. This is done by protecting the execution of the transaction with a mutex. Now the transaction cannot be aborted. Furthermore, it will cause

the concurrent transaction to abort. This mechanism is used to implement a fallback execution of transactions. [29, p. 21]

2.1.3 Conflict Detection

Defining how the conflict detection has to behave significantly impacts the implementation of a TM. To better understand the terms relevant for conflict detection we provide a definition according to [29, p. 20]:

A conflict **occurs**, if a thread tries to read or write to a memory location a transaction already wrote to. A conflict also occurs when a thread tries to write a memory location which was beforehand read by a transaction. [29, p. 20, 43, 32]

A conflict is **detected** when the underlying TM system identifies a conflict [29, p. 20].

A conflict is **resolved** by the underlying TM system. This can happen in various ways. One possible action the TM-System could take to resolve the conflict, is to abort one of the conflicting transactions. [29, p. 20, 33]

The order of the above-mentioned events is always maintained and can never change. The time when the events happen may vary. [29, p. 20, 33]

Conflict detection can be classified in three categories. The first category concerns the granularity by which the TM system detects conflicts. Conflict detection in HTM usually works on cache or word line granularity. In contrast, STMs detect conflicts at object (depending on how the used language defines objects) level. The granularity has a high impact on the implementation since a fine granularity needs more resources. Furthermore, it affects how many false conflicts are detected. False conflicts are conflicts which are detected even if no data dependencies have been violated. For example, when conflicts are detected on cache line granularity, a conflict is detected if two transactions access it in a conflicting manner. The conflict is detected regardless of whether the transactions access the same words of the cache line. Therefore, a conflict can be detected even though no actual conflict occurred. [29, p. 22, 30]

The second category defines when a conflict is detected. The literature describes two approaches: In **eager conflict detection** conflicts are detected and resolved directly after they occur. In contrast, when **lazy conflict detection** is applied potential conflicts are detected when a transaction tries to commit . Eager conflict detection is a more pessimistic approach. When applied it might abort transactions which would have committed with lazy conflict detection. In lazy conflict detection the transactions run longer since the conflicts are detected later. Therefore, they consume more computational re-

sources than in a TM system with eager conflict detection. [29, pp. 20,22, 32, pp. 2085-2086]

The third category refers to what accesses the underlying TM system detects as a conflict. For systems supporting tentative conflict detection conflicts are detected between actively running transactions. In contrast, committed conflict detection only observes conflicts between actively running transactions and committed transactions. Usually TM systems which detect tentative conflicts are combined with eager conflict detection mechanisms. TMs which feature committed conflict detection are combined with lazy conflict detection mechanisms. [29, pp. 22-23]

To detect conflicts TMs rely on an read set (RS) and write set (WS). RSs and WSs are managed per transaction. The RS contains all read accesses performed by a transaction. The WS contains all write accesses. To identify conflicts the TM system has to detect if an overlap between a transaction WS and the RS or WS of another transaction occurred. [29, p. 151, 33, 43].

2.1.4 Versioning

Another important design choice involves the management of speculative data. The literature provides two general mechanisms which can be applied in TMs:

Eager version management allows writing speculative data in place. Meaning, the memory location is directly updated with the speculative data. This technique requires pessimistic concurrency control, since the transaction needs exclusive access when writing to the concerning memory location. Eager version management therefore requires an undo-log which tracks the old values the transaction updated. The undo-log is required, since it is used to restore the updated values in case the transaction aborts. [29, pp. 21-22, 30, 32, p. 2084]

Lazy or deferred version management requires the TM system to provide a private redo-log. The redo-log can be considered as a private copy of the data the transaction wants to update. It is used to buffer the speculative data. During the time the transaction is running the TM system has to ensure that already written values are read from the redo-log. Otherwise, earlier performed writes are not considered. In case the transaction succeeds and therefore commits the actual memory locations are updated with the values from the redo-log. If the transaction aborts the redo-log is dropped and not further considered. [29, pp. 22-23, 30, 32, p. 2084]

2.1.5 Software Transactional Memories

Although STMs are not the main focus of this work we will provide some basic information to better understand related work. Additionally, we will explain why STMs approaches are widely spread in academia (e.g., [28, 34, 41, 56]) and why it seems attractive to work with an STM instead of an HTM. At the end of this section, we will point out why we decided to continue our work with an HTM.

STMs provide TM functionality in software. Therefore, all functionality, such as ensuring the properties are not hurt, conflict detection and concurrency control are offered by a compiler or library. [10, 32, p. 2083]

Using software for the development of a TM brings some advantages: Compared to the underlying hardware of an HTM the software used to implement an STM can be easily modified. Therefore, adapting or developing an HTM is considered quite challenging compared to adapting or developing an STM. When using software to provide TM functionality another advantage is that a potential developer can also exploit the features the underlying programming language offers. Additionally, STMs are not so much affected by hardware limitations (e.g., cache size). [29, p. 101]

STMs provide two main ways to maintain the logs as well as the metadata. This can either happen object-based or word based. The main difference is that in the object-based approach the metadata is maintained within the object. For the word-based approach the metadata is kept for every word address which is used within a transaction. To detect conflicts STMs provide special functions which identify a read or write access to be transactional. The extended read and write operations cause extra instructions, since the STM has to check if the accessed location was already accessed by another transaction. If the underlying STM detects a conflict it has to react accordingly and initiate measures. [10, 29, pp. 103-104]

Unfortunately, conflicting non-transactional accesses cannot be identified, since they do not use the special augmented read or write instructions. Therefore, STMs typically only support weak atomicity. When weak atomicity is applied only conflicts between transactions are detected. [7, 10, 30, 39]

Even though STMs offer the above-mentioned advantages we decided to proceed our work with an HTM. Due to the overheads STMs generate when transactionally reading or writing data ([29, p. 149]) and the fact that a potential user has to ensure atomicity ([10]), we decided that an STM is not a good fit for our work. Since our goal is to provide an HTM for embedded systems we need more control of the procedures within the transactional system. In our opinion this can only be provided by an HTM.

2.1.6 Hardware Transactional Memory

Since HTMs are the main focus of our work we show how they can be classified and give a brief overview of their main properties. In Chapter 3 we will put the focus on actual implementations and will explain a few implementations in detail.

To detect conflicts most HTM implementations rely on adaptations to the cache coherence protocol Additionally, the Instruction Set Architecture (ISA) is augmented, in one way or the other, to provide the TM functionality. Depending on how the ISA was adapted, HTMs can be classified as follows [29, p. 148]:

Explicitly transactional HTMs have special instructions which are used when writing or reading transactionally. Accesses to memory locations within the transaction not using these special instructions are not considered transactional. Additionally, other instructions for e.g., starting and ending a transaction can be provided (e.g., [14, 33, 38, 59]).

Implicitly transactional HTMs only require the specification of the boundaries of a transaction (i.e., start and end of a transaction). All accesses within the transaction boundaries are considered transactional and therefore added to the RS or WS respectively (e.g., [3, 12, 47, 49]).

As stated earlier (see 2.1.3 for more information) a TM system has to maintain an RS and a WS per transaction to be able to detect conflicts. To provide an RS an HTM has to track the reads performed during a transaction. A popular approach to do this is to modify the data cache, so it is able to mark cache lines, read during a transaction. A WS is maintained by buffering transactional writes. Tracking the WS requires more care, since transactional writes are speculative and therefore cannot be written back to memory. Here HTMs also utilize the cache or some sort of additional buffer [32, p. 2083]. Since caches are involved in maintaining the RS and WS, usually an entire cache line is added to the RS when read or to the WS when written. Read and write operations are recognized by monitoring the accesses to the cache. One important detail to consider is that the RS and WS are usually limited in size. The limitation is related to the associativity of the caches, or the size of the buffers used to maintain the RS and WS. If a cache or buffer is overflowed the affected transaction usually has to abort. [29, pp. 149-150, 32, p. 2083, 43]

To detect conflicts HTMs rely on the cache coherence protocol. The cache coherence protocol is here used to identify read or writes from other cores. A conflict is detected if either a read or write access of another core overlaps with the WS, or a write access of another core overlaps with the RS of the transaction currently executed. [29, p. 152]

If a conflict is detected it sooner or later has to be resolved (see 2.1.3 for more information). Apart from discarding the WS and RS (see 2.1.1 for more information), HTMs also have to consider the register set. The register set has to be set back to the state right before the transaction started. Otherwise, speculative values can still be located in the registers which could lead to a faulty execution. Therefore, HTMs have to provide some sort of mechanism (e.g., software solution or hardware mechanism) to save a copy of the register set when a transaction is started. [29, pp. 152-153]

2.1.7 Challenges and benefits of Transactional Memories

One big challenge within TMs are I/O operations. Since TMs allow optimistic access to data it is not foreseeable if a conflict will occur. If within a transaction a string was printed to a terminal this is problematic, since the transaction may still be aborted. If that is the case a complete rollback of the transaction cannot be performed, since the output operation cannot be reverted. [39, 30]

Another issue TMs face is that the semantics can be different depending on the underlying system. Most STMs support weak consistency and most HTMs support strict consistency. This may cause the same program to produce a different outcome depending on where it was executed. Therefore, it can be challenging to develop software which can be executed on either TM system (STM or HTM). [39]

Until now, it is not easy for programmers to use TM. Especially HTMs are only available in a few processors (see Chapter 3 for more Information). Also, for STMs the hurdles to use it are quite high. Potential programmers might have to completely rewrite a project to be able to integrate transactions. [39]

Especially in state of the art HTMs the limitation of resources can be problematic. Due to limited buffer size HTMs may be forced to abort transactions which exceed the buffer limits. This e.g., can occur if the RSs or WSs become too big. A possible solution for this is presented in Chapter 3 as well as in Chapter 5. [30]

When writing parallel code, where programmers have to ensure mutual exclusion when accessing critical sections, it can be very challenging to set the right amount of synchronization. Providing too much protection by setting the protection to critical sections too wide (also known as coarse-grained locking) limits scalability. Setting the protection of the critical sections very narrow (also known as fine-grained locking) is not easy to implement and might cause implementation faults leading to a faulty execution causing data races or deadlocks. In contrast, TMs provide an abstraction which highly simplifies parallel programming. Here the underlying TM system manages synchronization. Programs using TMs do not suffer from data races or deadlocks. [39]

2.2 Cache Coherence

Cache coherence mechanisms are heavily exploited when implementing TMs. Also, our work, described in Chapter 4 and Chapter 5 heavily relies on the cache coherence protocol. Furthermore, we also added states and messages to the established protocols described in this section (see Chapter 5 for more details). In the following we will first give an overview on cache coherence. Next, we will briefly explain snooping based cache coherence. Since we used directory-based cache coherence for our work, we will provide a detailed explanation of the mechanism. Furthermore, we will describe the states necessary and show in detail how the MOSI protocols work.

2.2.1 Overview

Cache coherence always refers to a certain memory location and defines the behavior of reads and writes to it. Therefore, it is an important tool to enforce memory consistency. Memory consistency defines the order of accesses to memory locations, considering all memory accesses. The applied memory consistency model can be observed by a potential programmer. Its enforcement is therefore essential to write correct parallel programs. Memory consistency and cache coherence are not the same, since cache coherence alone cannot determine the behavior of shared memory architectures. Due to the fact that the processor pipeline can also reorder memory accesses, use buffers, etc. it also has to be considered when enforcing consistency in shared memory architectures. If not thought of it is possible that in such architectures, memory consistency might be hurt even though cache coherence works correctly. [31, p. 379, 57, pp. 15; 20-21; 91]

To enforce cache coherence, cache coherence protocols are applied. As Figure 2.1 indicates every cache coherence protocol offers some sort of interface for the cores to interact with it. The way the core interacts with the cache hierarchy and how the cache coherence protocols work may vary. There are two ways to enforce cache coherence. In the following we describe the two approaches and point out the main difference:

Consistency-directed coherence: Here a write operation of a core is acknowledged before all locations of that address were invalidated or updated. Therefore, out-of-date values might be visible for other cores. To enforce coherence consistency-directed approaches rely on the cache coherence protocol as well as on the cores. [57, p. 11]

It is possible, but challenging, to apply this form of coherence to different forms of consistency. Especially in systems, enforcing consistency requiring the data to be consistent only at synchronization boundaries (e.g. Release Consistency (RC)), consistency-directed coherence can be suitable. [18, 19, 57, p. 12]

Consistency-directed coherence is also referred to as lazy coherence, since write operations are lazily updated [19].

Consistency-agnostic coherence: In this form of cache coherence an illusion is created in which the core interacting with the cache hierarchy is made belief it is interacting with one atomic memory. The main identification criteria for this form of cache coherence is that the effects of a write operation is made visible to all other cores before it is acknowledged. Since, this is the form of cache coherence we used in our implementation (see Chapter 4 for more information) we provide Figure 2.1 for better understanding. Furthermore, we refer to a system applying consistency-agnostic coherence when mentioning cache coherence. [57, p. 11]

Consistency-agnostic coherence is also referred to as eager coherence, since write operations eagerly invalidate shared locations [19].

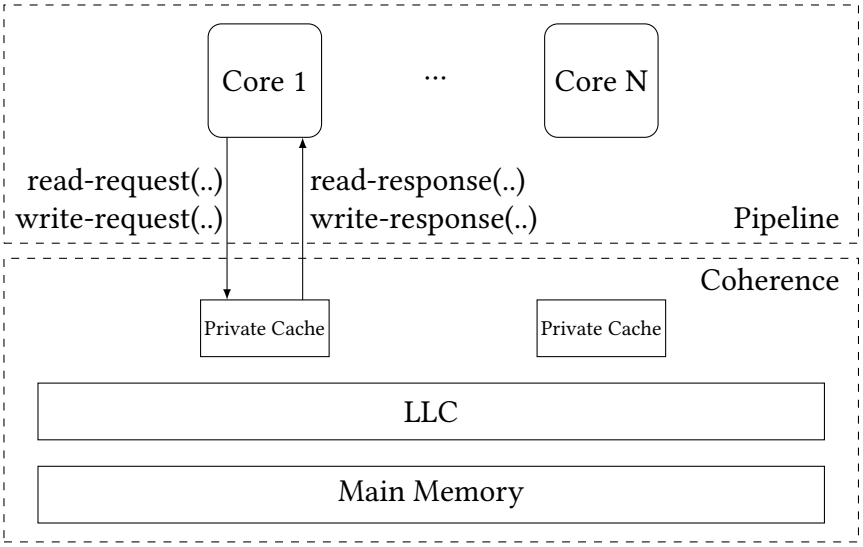

Figure 2.1: Coherence involves all caches (private caches and the last level cache (LLC)) as well as the main memory. For cores to interact with the coherence protocol it offers an interface. The coherence protocol provides two functions. The read-request returns the value for a given memory location. The write-request expects a memory location and a value. If the write operation was successful, it returns an acknowledgment. The cache coherence protocols let the actual memory hierarchy look way simpler, since they completely abstract away the caches. Therefore, the cores access the memory as if only one atomic main memory existed. [57, p. 11] The Graphic is inspired by [57, p. 12].

2.2.2 Invariants

It is important to determine when a memory hierarchy can be called coherent. Literature therefore provides multiple definitions (e.g., [57, p. 15, 45]). In the following we present the definition presented in [31, p. 378], since we consider it as the most holistic and comprehensible. According to [31, p. 378] coherence is achieved if the following three criteria are met:

1. If a core A writes to a memory location and afterwards reads from the same location, A should always receive the value it has written beforehand. This rule applies as long as no other core performed a write operation between the consecutive write and read operation of core A. [31, p. 378]

2. A value written to a memory location by core A should be available to core B when it is reading that exact same memory location. Here also time has to be considered, since the time between the write and read operation has to be sufficient. Additionally, the value written by core A should only be received by core B if no other core performed a write operation to that memory location between the write operation of core A and the read operation of core B. [31, p. 378]

3. Write operations to the same memory location are serialized. Therefore, all cores always observe the same order of writes to a memory location. It can never happen that two or more cores observe a different order of write accesses to a certain memory location. [31, p. 378]

To maintain the above-mentioned invariants cache coherence protocols were developed. Many of these protocols contain an invalidation mechanism. Therefore, they are often referred to as "invalidation protocols". The underlying idea of these protocols is that a core can only write to a location if its cache contains the only valid copy of the concerned memory location. Therefore, the core has to send an invalidation message to all sharers of that particular location. Reads to a memory location can be performed by multiple cores simultaneously. To track the shares the core has to perform a read request. [57, p. 14]

In theory, cache coherence is not bound to a particular granularity since it could vary from 1 to 64 bytes. In reality this would entail high hardware costs. Therefore, coherence is usually enforced at cache line level which is dependent on the memory architecture. In our work we also implemented cache coherence at cache line level (see Chapter 4 for more information). [57, p. 14]

2.2.3 States

Cache coherence protocols rely on state machines to maintain cache coherence. For this purpose, usually the cache hierarchy is augmented to support states for every line contained. Since, we, as well as many others, use a subset of the MOESI protocol, originally proposed by [60], for our work (see Chapter 4 for more information), we provide a brief overview of the stable states of the protocol in the following. [57, pp. 97-98 100]

Modified: A cache line marked as modified is valid but dirty since it most likely was modified through a write access. The cache line is held exclusively by the cache which contains it. The copy of the cache line in the LLC or memory is most likely outdated. Therefore, requests to the cache line have to be answered by the cache containing the modified cache line. [57, p. 98, 60]

Owned: A cache marks a cache line with the owned state if it responded to a request concerning a cache line it held in the modify state. Since the cache line was modified beforehand the cache contains the most recent version and has to respond to requests concerning that cache line. Without another state change the cache cannot perform any writes to the cache line, since it is no longer held exclusively by the cache. [57, p. 98, 60]

Exlusive: If marked as exclusive the copy of the cache line is exclusively held by the cache containing it. This can be beneficial since no further communication is necessary when updating the state of the cache line. Note that the state can therefore be instantly transformed from exclusive to modified. [57, p. 98, 60]

Shared: A cache line in shared mode was read by several cores. In every local cache the cache line is therefore marked as shared which indicates that the cache line is not held exclusively. A cache line in shared mode may only be read. [57, p. 98, 60]

Invalid: A cache line may be invalid for two reasons. The first reason is that the cache line is currently not present in the cache. The second reason is that the cache line was updated by another core and therefore all locations of the cache line but the one in its local cache had to be invalidated. [57, p. 98]

In general, a cache line may be valid or invalid. Further distinctions are made for valid cache lines. In Figure 2.2 we provide a diagram which shows the characteristics of the states in detail.

Some state transitions of stable states (e.g., S $\longrightarrow$ M) cannot be performed atomically, since the state change requires some communication. Therefore, the state of the concerning cache line may has to be set to an intermediate state. Depending on the protocol the number of intermediate states varies. We are going to present intermediate states in more

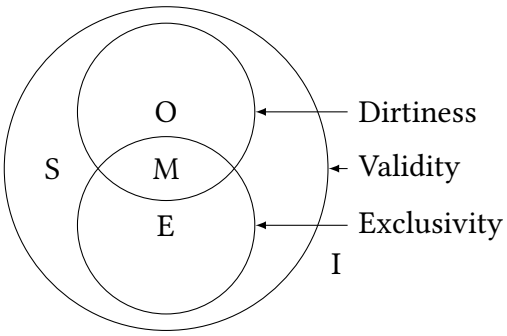

Figure 2.2: If a cache line is not valid it is marked as invalid (represented by the invalid state (I)). For invalid cache lines no further distinctions are made. The states shared (S), modified (M), exclusive (E) and owned (O) represent the states for which a cache line is considered valid. Validity can further be divided into exclusivity and dirtiness. A cache line which is not dirty, but exclusive is marked with the exclusive state (E). A dirty and exclusive cache line is marked with the state modified (M). If the modified cache line was shared it is not exclusive anymore but remains dirty. The state is then changed to the owned state (O). [57, p. 99] This graphic was originally provided by [60] and the labeling was adopted from [57, p. 99].

detail in Section 2.2.6, where we explain how the MOSI protocol can be implemented. [57, p. 99]

2.2.4 Snooping Based Cache Coherence

The main idea with snooping-based cache coherence protocols is that the states of the cache lines are solely managed by the cache controllers, meaning there is no central point where the cache coherence is managed. Therefore, all cache controllers are interconnected via a shared medium like a bus. To maintain cache coherence the participants, snoop the shared medium to keep the states of their cache lines up to date. Whenever a cache controller wants to change the state of a cache line it broadcasts a request. The other cache controllers notice the state change by snooping the bus. If a request concerns a cache line located in the cache they perform the necessary actions and collaboratively ensure cache coherence. [57, p. 107, 31, p. 380]

For snooping to work the order of the requests to the cache lines is very important. To enforce the ordering snooping-based approaches rely on ordered broadcast networks. By broadcasting every coherence request to all participants, it is ensured that all requests can be observed by all cache controllers. [57, p. 107]

Snooping based protocols do not scale well, which is why directory-based consistency is more often used in multi-core architectures. [57, p. 151]

2.2.5 Directory Based Cache Coherence

In directory-based cache coherence the states of the cache lines are managed by a central directory. The directory saves the current state and the exact location of a certain cache line. Furthermore, it either saves the sharers, the exact location, or both for a particular cache line. Requests to cache lines are first directed to the directory. From there they are either directly handled or re-directed to the current location of the cache line. In contrast to snooping based approaches coherence requests are not broadcasted to all participants. Since the directory contains information to where caches lines are located coherence messages can be sent more precisely. [57, p. 151]

For directory-based approaches ordering is also important. In contrast to snooping based approaches, where an ordered broadcast network ensures ordering, the coherence requests are ordered when reaching the directory. Here they are serialized and put in order. [57, p. 152]

Directory based approaches scale better but might inquire more steps for certain state changes which can cause higher latencies. The reason for the additional steps is that in contrast to the snooping-based approach a potential state change possibly has to be acknowledged by several shares, since otherwise ordering cannot be enforced. [57, pp. 152-153]

For our work we implemented directory-based cache coherence (see Chapter 4 for more information).

2.2.6 MOSI Cache Coherence Protocol

Due to some benefits, we expected when implementing the MOSI cache coherence protocol (see Chapter 5 for more information), we chose to use it for our work. Implementing cache coherence protocols can be very complex and error prone. Therefore, we oriented our implementation at the one provided by [57, pp. 151-190]. To give an accurate insight on how the protocol is implemented we here provide two tables also provided by [57, pp. 169-170]. Table 2.1 gives an overview on how the cache controller behaves within the MOSI cache coherence protocol. Table 2.3 gives an overview of how the directory controller has to react depending on which message arrives and what state the cache line is in. Throughout our work we slightly adapted Table 2.1 since we believe that oth-

erwise we would have not been able to operate the protocol (see Table 2.1 for more information).

	Load	Store	Replacement	Fwd-GetS	Fwd-GetM	Inv	Put-Ack	Data from Dir (ack=0)	Data from Dir (ack>0)	Data from Owner	Ack Count from Dir	Inv-Ack	Last-Inv-Ack
I	Send GetS to Dir / IS^D	Send GetM to Dir / IM^{AD}											
IS^D	Stall	Stall	Stall			Stall		-/S		-/S			
IM^{AD}	Stall	Stall	Stall	Stall	Stall			-/M	-/IM^A	-/M		ack–	
IM^A	Stall	Stall	Stall	Stall	Stall							ack–	-/M
S	Hit	Send GetM to Dir / SM^{AD}	send PutS to Dir / SI^A			send Inv-Ack to Req / I							
SM^{AD}	Hit	Stall	Stall	Stall	Stall	send Inv-Ack to Req / IM^{AD}		-/M	-/SM^A	if(ack>0) -/SM^A else -/M*		ack–	
SM^A	Hit	Stall	Stall	Stall	Stall							ack–	-/M
M	Hit	Hit	send PutM to Dir / MI^A	Send data to Req / O	Send data to Req / I								
MI^A	Stall	Stall	Stall	Send data to Req /OI^A	Send data to Req /II^A		-/I						
O	Hit	Send GetM + data to Dir / I	send PutM to Dir / OI^A	Send data to Req / O	Send data to Req / IM^{AD}								
OM^{AC}	Hit	Send GetM to Dir / IM^{AD}	Stall	Stall	Send data to Req / IM^{AD}						if(ack>0) -/OM^A else -/M*	ack–	-/M*
OM^A	Hit	Stall	Stall	Stall	Stall						ack––	-/M	
OI^A	Stall	Stall	Stall		send Inv-Ack to Req /II^A		-/I						
SI^A	Stall	Stall	Stall			send Inv-Ack to Req /II^A	-/I						
II^A	Stall	Stall	Stall				-/I						

Table 2.1: This table provides an overview of the MOSI cache coherence protocol from the perspective of a cache controller. It shows all stable and transient states used in the protocol (first column). The first row shows actions relevant for coherence which can reach a cache controller. Depending on the current state and the desired action the table specifies the state transitions. The table further provides which steps (e.g., if messages have to be triggered) have to be performed to ensure a correct state change. The table was copied from [57, p. 169]. We performed a few changes marked by a star. We believe that without these adaptations the protocol cannot work correctly. Due to the space limitation, we provide more details in the following paragraphs.

For better understanding of Table 2.1 and 2.3 we provide the Tables 2.2 and 2.4. Table 2.2 provides an overview on the coherence messages used. Table 2.4 gives an overview on the abbreviations used in Table 2.1 and 2.3.

The first column of Table 2.1 shows the stable states M, O, S and I as well as all possible transient states e.g., IS^D. For better understanding we want to look at the transient state IS^D in more detail. Since the basic scheme of the transient states follows the same pattern, the description is sufficient for all transient states.

IS^D represents the state of a cache line which is set when it is transitioning from invalid (I) to shared (S). Since the data has to be requested and the controller has to be registered as a sharer of the cache line it cannot be instantly set to shared. The controller first has to send a GetS request (second column, second row). Then it is set to the transient state IS^D. ^{D} hereby indicates that data is expected to be received. Once the data arrives the controller is certain it was registered as sharer and may now change the state of the cache line to shared (S) (ninth column, third row). If a transient state is marked with an ^{A} e.g., SM^A the controller expects one or more acknowledgment message(s). The controller knows how many acknowledgment messages it has to receive, since the directory transmits the number of sharers to the cache controller after receiving a GetM request. ^{C} indicates the transient state is expecting the shared counter. The only case where this is necessary is when a state is transitioned from owned (O) to modify (M), since the controller needs to know for how many acknowledgments it has to wait, before the state can be switched.

Message	Explanation:
GetS	Message sent by cache controller to indicate it wants to read a cache line
GetM	Message sent by cache controller to indicate it wants to write a cache line
Fwd-Get(S/M)	Get(S/M) request is forwarded to owner of cache line
Put(S/M/O)	Cache controller sends this message if it wants to invalidate a cache line
Put-Ack	Coherence message sent by the directory to confirm prior sent Put message
Inv	Coherence message sent by directory if cache line has to be invalidated
Inv-Ack	Coherence message sent by the cache controllers if a cache line was invalidated
send data	The cache controllers as well as the directory controller can send data, the directory always adds the number of sharers of the concerned cache line to the message

Table 2.2: Coherence messages used for MOSI cache coherence protocol

	GetS	GetM from from Owner	GetM from Non-Owner	PutS NotLast	PutS-Last	PutM+data from Owner	PutM+data from Non-Owner	PutO+data from Owner	PutO+data from Non-Owner (ack=0)
I	Send data to Req, add Req to Sharer /S		Send data to Req, set Owner to Req /M	Send Put-Ack to Req	Send Put-Ack to Req		Send Put-Ack to Req	Send Put-Ack to Req	Send Put-Ack to Req
S	Send data to Req, add Req to Sharers		Send data to Req, send Inv to Sharers, set Owner to Req, clear Sharers /M	Remove Req from Sharers, send Put-Ack to Req	Remove Req from Sharers, send Put-Ack to Req /I		Remove Req from Sharers, send Put-Ack to Req		Remove Req from Sharers, send Put-Ack to Req
O	Forward GetS to Owner, add Req to Sharers	send Ack-Count to Req, send INv to Sharers, clear Sharers /M	Forward GetM to Owner, send Inv to Sharers, set Owner to Req, clear Sharers, send Ack-Count to Req /M	Remove Req from Sharers, send Put-Ack to Req	Remove Req from Sharers, send Put-Ack to Req	Remove Req from Sharers, copy data to mem, send Put-Ack to Req, clear Owner /S	Remove Req from Sharers, send Put-Ack to Req	Copy data to mem, send Put-Ack to Req, clear Owner /S	Remove Req from Sharers, send Put-Ack to Req
M	Forward GetS to Owner, add Req to Sharers /O		Forward GetM to Owner, set Owner to Req	Send Put-Ack to Req	Send Put-Ack to Req	Copy data to mem, send Put-Ack to Req, clear Owner /I	Send Put-Ack to Req		Send Put-Ack to Req

Table 2.3: This table shows an overview of the MOSI cache coherence protocol from the perspective of the directory controller. The first column shows the stable states M, O, S and I. The first row represents the actions relevant for coherence. Depending on the state and the requested action the table provides information on how the state has to change and which actions have to be triggered. Note that no transient states at directory level are necessary. The table was taken from [57, p. 170].

Abbreviation	Meaning and explanation:
Req	stands for requestor, usually refers to cache controller from which message originated
Dir	stands for directory
ack	stands for acknowledgment, in the cache controller it may also stand for acknowledgment counter

Table 2.4: List of abbreviations used in Tables 2.1 and 2.3

According to the description above we implemented the MOSI cache coherence protocol into the gem5 simulator. Since cache coherence is highly important for HTM, the implementation provides the basis for our work further described in Chapter 4 and 5.

2.3 The Gem5 Simulator

The gem5 simulator is the result of the union of the m5 and GEMS simulator. Both simulators originally were independently developed but openly available. The m5 simulator was developed at the University of Michigan and used to model networked systems. The GEMS simulator was created to evaluate the performance of multiprocessors and developed at the University of Wisconsin. Combining both simulators made the gem5 simulator a full-system simulation tool. After the merge of the two projects the gem5 simulator continued to be openly available, and the development process became community oriented. Therefore, researchers and specialists were able to continuously contribute to the project. Over time also the development process was professionalized by using version management, continues integration and other tools. This made it easier and more standardized to contribute to as well as use the gem5 simulator. [5, 6, 40, 42]

Five years ago, in 2017 when this work started the gem5 simulator already supported several ISAs such as ARM, ALPHA, MIPS, Power, SPARC, and x86 (later versions 20.0+ also support RISC-V and GPU-ISAs [40]) [6]. For our work we focused on the ARM ISA (see Chapter 4 for more information).

Apart from offering different ISAs the gem5 simulator also offers great flexibility in the CPU model. Among other simpler models, the gem5 offers accurate pipelined in order as well as out of order CPU models. For our work we used the O3 model, which is the pipelined out of order CPU model. In contrast to the simpler models, the pipelined models are more accurate and therefore take longer to evaluate. [6]

The gem5 offers two system modes to run applications. The System-call Emulation allows a bare metal execution of programs by emulating system calls. In Full-System mode a complete system consisting of an operating system and devices is modeled. For our work we used the System-call Emulation mode since the Full-System mode complicated the evaluation process, due to the very long evaluation times. [6]

One of the main reasons we chose the gem5 simulator for our work is that it can be freely augmented and adjusted. Since most components are already available (CPU, memory, buses, etc) we only had to focus on adding the functionality relevant for our work. Apart from that advantage the gem5 simulator is extremely popular and well established in the academic world. [6, 40]

Due to merge conflicts when we fetched the latest version of the gem5, caused by our adjustments (bug fixes, implementation), we were not able to continuously fetch the most recent versions of the gem5. Therefore, our version of the gem5 more or less matches (we performed the last merge on 01.08.2018, where we had quite a few merge conflicts) the version of the gem5 simulator between 2017 and 2018. Unfortunately, it became too difficult to solve the merge conflicts, which is why we stuck to our version, since a re-

implementation of our work to a newer version of gem5 was out of scope of the time constraints of our project.

2.4 STAMP Benchmark Suite

The authors of [13] developed the STAMP benchmark suite, which contains realistic benchmarks to evaluate TM-Systems. In their work, the authors describe eight benchmarks which provide a wide range of challenges for TMs. This includes frequent or rare use of transactions, short and long transactions, and high or low contention. The benchmark suite consists of eight benchmarks and is highly portable since it is widely used to evaluate transactional memories (e.g., [61, 22, 14]). In the following we will give a brief description of the benchmarks which we derived from [13]:

bayes: This benchmark learns the structure of a Bayesian network. It uses a hill-climbing strategy which combines global and local search. The application can be assigned to the machine learning domain. The transactions executed within the application are long. They generate big read and write sets. The contention between the transactions is high. Additionally, the time spent in transactions is also high.

genome: Is a program which matches gene sequences. The application can be divided into two phases. Within the first phase duplicates of DNA segments are removed. In the second phase the application tries to match unmatched segments. Transactions are used in both phases. The transaction size as well as the size of the RS and WS are medium. Overall, the time spent in transactions is high and contention is low.

intruder: This application emulates a design of an network intrusion detection system. Here network packages have to run through three phases. This is done in parallel. Two of the three phases utilize transactions. The size of the RS and WS is medium. The transaction length is short. The contention between the transactions is considered high by the authors.

kmeans: Is a clustering algorithm which groups objects in a N-dimensional space. Hereby the objects are grouped in K clusters. The contention in the transactional execution heavily depends on the value K, since the transactions protect the update of the clusters centers which are updated iteratively by every thread while processing a partition of the objects. If K is big the probability of two transactions working on the same cluster center is low. These properties are beneficial for the execution with transactions since the optimistic synchronization can be exploited. According to the authors, contention can therefore be considered low. Addition-

ally, the transactions' length is short, and the time spent in transactions is low. The RS and WS are small.

labyrinth: Within this benchmark the threads pick a start and an end point within a maze represented by a three-dimensional uniform grid. The threads now have to add paths from the starting to the end points in parallel. This is protected by transactions. If two paths overlap a conflict is detected. Calculating a path and adding it to the global maze is protected by a single transaction. Therefore, the RS and WS can be considered as large. Also, the transaction length is long. The time spent in transactions as well as contention is high.

ssca2: The Scalable Synthetic Compact Applications 2 (SSCA2) usually consists of four kernels which operate on large, directed, weighted multi-graphs. For the STAMP benchmarks the authors focused on one kernel. Here, transactions are used to protect operations to the arrays which are used to save the graphs. Since the graphs can become quite big contention is low. Graph operations are not very long and therefore the size of the RS and WS is small, the transactions themselves are short, and the time spent in transactions is short.

vacation: This application represents a travel booking system. All necessary data is saved in trees. Coarse-grain transactions are used to protect writes to the database. Overall, the application spends a lot of time in transactions. According to the authors, contention can be considered as medium or even low. The size of the RS and WS and the size of the transactions is medium.

yada: The benchmark creates a Delaunay mesh using Ruppert's algorithm [50]. The basic data structures here used are graphs. They store the mesh triangles, a set which holds the mesh boundaries, and a work queue which holds the skinny triangles which are the triangles which have to be refined. Accesses to the work queue as well as the refinement of the skinny transactions are protected with transactions. Therefore, transactions are long, and the time spent in transactions is high. Since some triangles are visited and later changed during triangulation, the RSs and WSs are large. The authors claim that overall contention can be considered as medium.

Table 2.5 summarizes the information given above focusing on the most important details.

Benchmark	Tx length	R/W Set	Tx Time	Contention	Domain
bayes	long	large	high	high	machine learning
genome	medium	medium	high	low	bioinformatics
intruder	short	medium	medium	high	security
kmeans	short	small	low	low	data mining
labyrinth	long	large	high	high	engineering
ssca2	short	small	low	low	scientific
vacation	medium	medium	high	low/medium	online tx processing
yada	long	large	high	medium	scientific

Table 2.5: This table summarizes the properties of the eight stamp benchmarks. Here, "Tx length" refers to the length of the transactions executed within the benchmarks. The Column "R/W Set" indicates how big the transactions RSs and WSs get. "Tx Time" refers to the time the benchmark spend executing transactions. The higher the contention the more conflicts occur during execution. The column "Domain" indicates to which domain the benchmark belongs. The information used for this table is given in [13].

2.5 Summary

This chapter describes the background for the work presented in the Chapters 3, 4 and 5. The first part of the chapter concentrates on TMs. Here we provide an overview on the properties of transactions, which options for concurrency control exist, how conflict detection works and how several versions of data can be handled. In the remainder of the section, we also describe the difference between STMs and HTMs. The section concludes by outlaying the challenges and benefits of transactional memories.

Next, we put the focus on cache coherence. Especially the cache coherence protocol has a significant relevance for our work. We start this section by providing an overview on the topic after giving a general introduction to cache coherence. Then we present the invariants cache coherence protocols have to fulfill. After describing widespread states used by many cache coherence protocols, we continue the section by describing snooping- and directory-based cache coherence. The section ends by giving a detailed description on how the MOSI cache coherence protocol can be implemented.

We realized our work by using the gem5 simulator. Therefore, we provide an overview on the gem5 simulator and give some insights on our design choices. Also, we describe the benefits of the gem5 simulator and also explain why we use an older version of the gem5.

The last section of this chapter focuses on the STAMP benchmark suite. We used the benchmark suite to evaluate our work (see Chapters 4 and 5 for more information). Therefore, we gave a brief overview of all the benchmarks included and roughly explained what they do. The section finishes by providing a table with the properties of each benchmark.

3

Related Work

Contents

Since the original proposal of TMs by [33], TMs were utilized for many approaches. Therefore, a wide variety of TMs exist. Since they were partly redesigned and augmented in different ways, they may strongly vary from the original proposal. Often the focus of the initially intended use, which is benefiting a parallel execution through optimistic synchronization, shifted to other functionalities. Further developed TMs were e.g., used to enable fault tolerance (e.g., [26], [2]). Here the conflict detection and the rollback mechanism are heavily exploited to detect and recover from faults. TMs can be implemented in software and hardware. Also, hybrid approaches exist which are partly implemented in software and hardware. The concepts of TMs were also picked up by chip manufacturers and are therefore available in some common of the shelf (COTS) hardware.

In the following we present work which describes different mechanisms for TMs as well as actual TM implementations. We therefore first present some HTM approaches developed by the industry. Next, we will provide an overview of the work done on contention management strategies. Thirdly, we will look at approaches which support unbounded transactions. Since we focus on embedded systems, we will then look at an approach

which also focuses on embedded systems. Before summarizing this chapter, we will discuss real time transactional memory approaches.

3.1 Hardware Transactional Memory Approaches Devolved by Industry

TMs offer features which are interesting for a variety of chip manufacturers. Therefore, several manufacturers proposed their own version of an HTM. A few of these approaches were available through COTS processors (e.g., Intel TSX). In the following we provide a brief overview.

3.1.1 Sun Microsystems

The authors of [12] present Rock: a high performance sparc chip-multithreaded processor. It supports speculative execution and enables an execute-ahead-mode, simultaneous speculative threading and transactional memory. The authors claim that Rock is the first commercial processor which offers support for HTM. Although the processor reached a high degree of development it was never commercially available [46].

In their work the authors describe an architecture which enables an execute ahead mode. Here the system can speculatively execute instructions. To support the execute ahead mode the architecture was designed to allow checkpoints. Therefore, the system can perform a rollback in case of a problem.

A checkpoint is automatically taken if an instruction cannot be executed immediately due to e.g., a cache miss. After a checkpoint is taken all subsequent instructions are executed in speculative mode. Therefore, the values are written to shadow copies of the actual used registers. The instruction which initially could not be executed is moved to a deferred instruction queue. Additionally, the destination register of the instruction which could not be executed is marked as not available (NA). Therefore, all subsequent instructions which consume the register are also moved to the deferred instruction queue. As soon as the instruction can be executed, the instruction and its dependent successors are fetched from the deferred instruction queue and executed. The authors call this the replay process.

If the replay execution works with no disruptions the system detects that the speculative execution was successful and marks the speculative shadow copies of the registers as the architectural state. The authors call this the join operation.

To be able to detect potential conflicts the cache lines are marked with an additional s-bit when read in speculative mode. If a cache line with an s-bit is invalidated or evicted speculation fails, and the execution resumes from the checkpoint. The speculation has to fail, since otherwise the total store order cannot be guaranteed.

The speculation ahead mode can be augmented to exploit simultaneous multithreading. If activated the join operation can be performed by a second thread. This can be beneficial for execution and is called simultaneous speculative threading by the authors.

The ability to create checkpoints as well as the ability to perform a rollback, in case speculation does not succeed, is exploited to provide an HTM. In addition to the already existing features, the L2-cache is augmented. Here all addresses which a transaction wrote are sent to the L2 cache. The L2 cache is then able to detect conflicts. If the L2 cache detects a conflict it informs the concerning core which then initiates an abort. Unfortunately, it is not clear which conflict resolution policy is applied. When a transaction tries to commit, the L2 cache locks the cache lines the transaction wrote and prevents them from being read or written by other threads during the commit. The list of addresses sent during the execution of the transaction is used to do this. Then the core committing its transaction, starts to write back the content of its write queue, since this is where it stores its speculative values. The last write to a locked line releases the lock of that line and enables it to be read or written by other threads again.

To start and stop transactions, the system was augmented to support two new instructions. The checkpoint instruction indicates the start of a transaction. Here the system expects an address to the code which should be executed if the transaction fails. The commit instruction indicates the end of the speculative part and therefore signals the transactions end. A transaction is aborted if a conflict is detected, or the hardware resources are not sufficient to execute the transaction.

3.1.2 Proposal of Hardware Transactional Memory by AMD

In [15] AMD proposed the Advanced Synchronization Facility (ASF) which is an extension for lock-free data structures and transactional memory. The proposal is not available in real hardware but was implemented in a simulator. The ASF offers commands to start, abort and commit a transaction. Additionally, the ASF offers the possibility to reduce the transactional footprint. Here transactional and therefore speculative loads and stores can be explicitly marked. Other instructions are not considered as part of the transaction.

The RS and WS are managed in the L1 cache by providing extra bits per line. Conflict detection is performed eager. The authors refer to the contention management strategy as "attacker wins". This is a straightforward technique, where the core which detects the conflict aborts its transaction. After a random back-off time the transaction is

re-executed. A potential user has to offer a fallback strategy, since the system cannot overcome capacity conflicts.

The proposal also suggests the additional use of the load store queue to manage the transactional data. This feature allows bigger transactions and can be used in combination with the functionality of reducing a transaction's footprint. If the feature is activated the speculative data is kept in the L1 caches as well as in the load and store queue. This implies some complexity but also enlarges the minimum transaction length, which is determined by the set associativity of the L1 cache.

3.1.3 Hardware Transactional Memory approaches by IBM

The authors of [62] describe the hardware support for TM provided by the Blue Gene/Q machine. Every core supports Simultaneous Multithreading (SMT) and can therefore run multiple transactions at the same time. To save the speculative footprint of the transactional execution the L2 cache supports multiple versions of each cache line. Conflicts are detected eager and performed at L2 cache level. For this purpose, the system supports speculative IDs. 128 IDs are provided. Since the IDs are a scarce resource, the IDs have to be frequently recovered. With a process called ID scrubbing the IDs are checked every 132 cycles and made available again if possible. Conflict resolution can take the transaction start time into account and favors the transactions which started earliest. The speculative state can get as big as 20 MB, since 10 ways of the 16 way set associative 32 MB L2 cache can be used for speculation without an eviction. The system supports two execution modes: A short running mode and long-running mode.

The short running mode evicts a speculatively written cache line from the L1 cache. All following accesses to that cache line have to be serviced by the L2 cache. A request is then answered with the thread specific data. Additionally, the answer signals the core to directly save the data to a register. This mode is good for small transactions which sporadically access memory locations.

The long-running mode supports up to five versions of a cache line in the local cache. Here the authors describe how they use the Translation Lookaside Buffer to do this, since the L1 caches themselves are not modified. In this scheme the memory requests are not forwarded and therefore the L1 cache is flushed on transaction entry. This is important so the L2 cache is able to notice which cache lines are consumed by the transaction. This scheme can be used for longer transactions which access many memory locations, since it better exploits temporal and spatial locality within the transaction.

The system also supports an irrevocable mode. Transactions which repeatedly failed to commit or cannot be executed speculatively are executed in irrevocable mode. After the

thread acquired an irrevocable token, the transaction is executed in serialized mode and cannot be aborted.

Apart from the HTM IBM developed for the Blue Gene/Q machine the company also developed an HTM for the system z [35]. In contrast to the prior presented HTM, here mainly the Load/Store Unit (LSU) to which the L1 cache and the store queue are integrated, was augmented to provide an HTM.

The system has a 96 KB 4-way associative L1 cache which is coupled to a private 1 MB 8-way associative L2 cache. The L1 as well as the L2 cache are store through which means that their cache lines always match with the ones stored in the L3 cache. The L3 cache is shared by a central processing chip which consist of 6 cores. It has a capacity of 48 MB. The system also provides an off chip L4 cache with a capacity of 384 MB to which six central processing chips are connected. Cache coherence is provided with a variant of the MESI protocol not further described. The authors call coherency requests generated by the protocol cross interrogates (XI).

To mark a cache line as part of a transaction the L1 cache provides extra bits to mark a cache line as read or written within a transaction. Since the L1 and L2 cache are store through the authors proposed a gathering cache. The purpose of the gathering cache is that the transactional writes are not written to the L2 and L3 cache. The authors claim that if the speculative cache lines were located anywhere else than the L1 cache the effort to perform a cleanup in case of a transaction abort would be unacceptable for performance. The gathering cache is managed as subset of the L2 cache. The maximum size of a transactions' footprint is limited by the gathering cache and is bound to its size (64 x 128 bytes) and the associativity of the L2 cache, since the gathering cache is managed as a subset of the L2 cache.

Conflict detection is performed eager. If an XI indicates a conflict to an LSU by changing the coherence state of a cache line it aborts the transaction currently running on the connected core. The LSU does not have to directly abort a transaction, it can also reject the XI. This mechanism is called stiff-arming and the idea is that a transaction might finish before the request is reissued. To guarantee progress an XI-counter is provided which aborts the transaction if it is about to exceed a certain threshold.

IBM also developed architectural support for HTM for the power architecture [8, 1]. Besides the basic functionality of an HTM the authors implemented support for suspending and resuming transactions. Once a transaction is suspended, it enters the suspended transaction mode. While executing in this mode all accesses are performed non-transactionally. During this time the system prevents the start of a new transaction. If a conflict occurs, it is detected but handling it is deferred to when the transaction resumes. The mechanisms allow transactions to survive interrupts. Additionally, the authors propose rollback-only transactions. This technique is used for speculation. The basic idea of this technique is to only provide the possibility of being able to perform a rollback. In

contrast to conventional transactions, rollback-only transactions are not used to perform conflict detection. Access to shared data has to be prohibited in software. Unfortunately, the authors do not discuss or explain the hardware setup. Therefore, apart from the fact that the transactional footprint is saved in the L2 cache [1], no details on how the HTM is implemented are known.

3.1.4 Intel TSX and ARM TME

Intel as well as ARM both offer HTM implementations [16, 4, 58]. Unfortunately, hardly any implementation details were released by the hardware manufacturers. This mainly concerns the internal handling of transactional execution, since usage and abstract functionality is well documented. Therefore, no statements upon how the manufacturers manage transactional execution can be made.

In general Intel TSX as well as ARM TME provide two software interfaces (normal hardware transactions and Transactional Lock Elision (TLE)) with which transactional regions can be defined [16, 58]. The support for Intel TSX started with the fourth-generation Intel core processor [27]. ARM provides an HTM implementation for the Arm A-Profile Architecture and announced such support in 2019 [58]. Unlike Intel, to our knowledge currently no COTS ARM processor exists which features the TME.

Since we had access to machines which provide Intel TSX we were able to perform some experiments in which we tried to examine some implementation details (e.g., contention management strategy, maximum size of transactions, etc.). Unfortunately, we were not able to generate reliable results which would have given us some insights into implementation details. In general, the transactional execution seemed unstable, and the received error codes sometimes were misleading and not necessarily beneficial.

3.2 Contention Management Strategies

Past work suggests quite a few contention management strategies which will be presented in detail in the following. The mentioned proposals developed contention management approaches mainly for STMs. Although we focus on HTMs, we provide an overview of the theory and mechanisms, since the findings are relevant for our work.

[25, 53, 54] developed several contention management strategies for STMs. In the following we list and explain some of the proposed strategies. We based our selection on the selection made by [29, S. 51-52] and [53], as we consider the choice to be sensible, since the authors focused on the more promising strategies:

passive: The transaction which detects the conflict aborts itself and re-executes.

***polite*:** Within the polite manager a transaction aborts and tries to resolve the conflict by using an exponential backoff. After a certain number of failed attempts the transaction aborts the concurrent transactions and itself finishes transactional execution.

***karma*:** Whenever a transaction accesses a new object the priority of that transaction increases. Therefore, the idea behind the manager is to try to take the already achieved work into account. When the transaction commits the priority is set back to 0. If it aborts the value remains which can be considered as karma, since it increases the possibility of committing when re-executed. An aborted transaction will try to access the object which caused the conflict until the conflicting transaction commits or accessing the object lead to a higher priority than the conflicting transaction. The aborted transaction is backed off for a fixed time until it is re-executed.

***eruption*:** Similar to *karma* a transaction gains priority by accessing objects. In contrast to *karma*, a transaction additionally gains priority if it blocks another transaction. The gain in priority is dependent on the blocked transaction, since it transfers its priority. The priority is then added to the priority of the blocking transaction. The idea behind this manager is that a transaction which blocks many other transactions will gain a high priority and therefore will commit fast.

***kindergarten*:** The idea behind this manager is that transactions take turns when accessing an object. This is done with a hitlist. Initially the hitlist is empty. When a transaction detects a conflict, and the conflicting transaction is contained in the hitlist it the manager aborts the conflicting transaction. If the conflicting transaction is not part of the hitlist the current transaction is aborted and the conflicting transaction is added to the hitlist. After a backoff time the transaction is re-executed.

***timestamp*:** When a transaction is started the time of the transaction start is saved. If two transactions conflict the transaction which started earlier is able to continue.

***publishedTimestamp*:** Works like *timestamp* with the difference that it is able to abort long-running transactions which appear to be inactive.

polka*:** Is a combination of the managers ***pol*ite* and ***ka****rma*. Here the transactions are able to build up a priority. Additionally, an exponential backoff is used, in case a transaction aborts, to determine when the transaction is re-executed.

***greedy*:** The Authors who presented Greedy were able to show that for n transactions sharing s resources the runtime is a factor $s(s+1)/2$ of the optimal runtime achieved

by an offline scheduler [25]. All transactions started are set up with a timestamp. In contrast to the contention management strategy *timestamp* a transaction keeps the assigned timestamp, even if it was aborted. Additionally, every transaction tracks if it is currently waiting for another transaction. A transaction is marked as waiting if a conflicting transaction has a higher priority and is currently not waiting for another transaction. Therefore, transaction A aborts transaction B if it was started earlier or if transaction B is waiting for another transaction.

3.3 Unbounded Conflict Detection

In conventional HTMs the size of transactions is limited by the underlying hardware. To solve this problem different approaches exist. The approaches vary in complexity. In the following we give an overview of the most relevant proposals for our work.

3.3.1 Unbounded Transactional Memories

The authors of [3] first introduced the idea of unbounded transactions. To enable unbounded transactions, they provide two proposals. The first implementation they describe is UTM. UTM is an in hardware implemented mechanism which supports transactions of arbitrary size which can be nearly as big as the virtual memory. Additionally, the authors state that transactions are able to survive time slice interrupts as well as process migration from one core to another. The mechanism is quite complex and requires significant changes to the cores and the memory subsystem.

To save the register-set the authors avoid saving a snapshot of all registers. Instead, they propose a mechanism in which they save a snapshot of the rename table of the registers. With this technique the authors are able to detect all physical registers which are in use. These registers are blocked from further use until the transactions commits or aborts. This is done by moving the specially marked registers to a Register Reserved List instead of the normal Register Free List. If the transaction commits the list is flushed and registers a moved to the Register Free List. When the transaction aborts, the rename-table is set to the state when the transaction started.

The memory state is managed by a data structure named xstate. The xstate data structure saves transactions logs for every transaction. The transaction logs consist of a commit record and a vector of log entries. The commit record reveals the status of a transaction (pending, committed or aborted). Log entries contain a block pointer which is a pointer to a memory block. If the value of a memory block was changed the old value is saved by the corresponding log entry. Additionally, a reader list is maintained in the log entries. This is

important for conflict detection. The authors claim that conflicts are resolved by using a timestamp (also saved by the transaction log) which allows favoring the older transaction in case of a conflict. To find the corresponding xstate entry the system provides a log pointer and an RW bit for every block in memory.

To implement and maintain the xstate data structure a lot of changes to the processor and memory would have to be made. Due to the complexity of these changes UTM was not implemented. Instead, the authors offer a second proposal named LTM which implements the fundamental ideas of UTM but is simpler. Here overflowing cache lines are written to a hash table which is located in uncached main memory. All speculative data is therefore kept in the local caches and the hash tables. In contrast to UTM the running transaction is aborted if the corresponding hardware detects a conflict. If the uncached hash table located in main memory is too small, the operating system is informed, and the transaction is restarted. Additionally, the size of the hash table will be increased by the operating system. LTM was implemented into a cycle-level simulation using UVSIM.

The authors evaluated LTM. Here only one benchmark, which focuses on small transactions, was evaluated in parallel. The findings of the authors concerning this experiment is that the proposed TM scales better than an approach using locks. Furthermore, the authors also evaluated the SPECjvm98 benchmarks by running them in their simulation. This was performed on only one core and also shows that the proposed HTM performs better than a lock-based approach. A third experiment focuses on a trace driven study which shows that the chosen benchmarks use transactions but only a few transactions can be considered to overflow. In the authors opinion the findings of the evaluation suggest that the underlying assumptions behind UTM and LTM are correct. Since the evaluation focused on single cores no sophisticated multi-core evaluation was presented.

3.3.2 Virtualizing Transactional Memory

To avoid or reduce non-transactional conflicts that occurred due to the underlying hardware (e.g., capacity conflicts) great knowledge of the hardware specifications is necessary. This is not satisfactory for the authors of [48], since they consider this as major hurdle to use transactional memories. Therefore, they propose a hardware transactional memory approach which abstracts the functionality of the transactional memory from the underlying hardware. To achieve this the authors, utilize the mechanisms used for virtualizing memory. This leverages a potential user to use the transactional memory without needing specialized knowledge of the hardware. The complexity of handling everything correctly is concealed by virtualization mechanisms.

With their work the authors want to provide a hardware transactional memory which allows a transaction to survive context switches, page faults and exceeding hardware re-

strictions. To achieve these goals the authors, introduce Virtual Transactional Memory (VTM). Here the Authors do not focus on the HTM, since this can vary. More important is how the approach handles scenarios causing the regular HTMs to abort their transactions. The underlying HTM therefore only plays a minor role.

The VTM consists of a Transaction Status Word (XSW) which holds the current execution status of a transaction. It is part of a thread's state and is located in virtual memory. The Address Data Table (XADT) manages overflowed parts of the transactional execution. To quickly determine if a cache request causes a miss, the authors introduce a filter (XF) for the XADT. The XF is realized as a bloom filter and can reliably detect if an entry is not contained in the XADT. If the XF detects that an entry is contained in the XADT, the VTM walks the XADT to detect if a conflict really occurred. Both structures are implemented in software and are contained in the application's virtual address space. Figure 3.1 shows how the authors assume the VTM system. The authors assume that the VTM system, implemented in hardware or microcode, manages the above-mentioned structures. This e.g., includes resizing the data structures if they are too small to fit the overflowed data. Unfortunately, the approach was not implemented and therefore not evaluated.

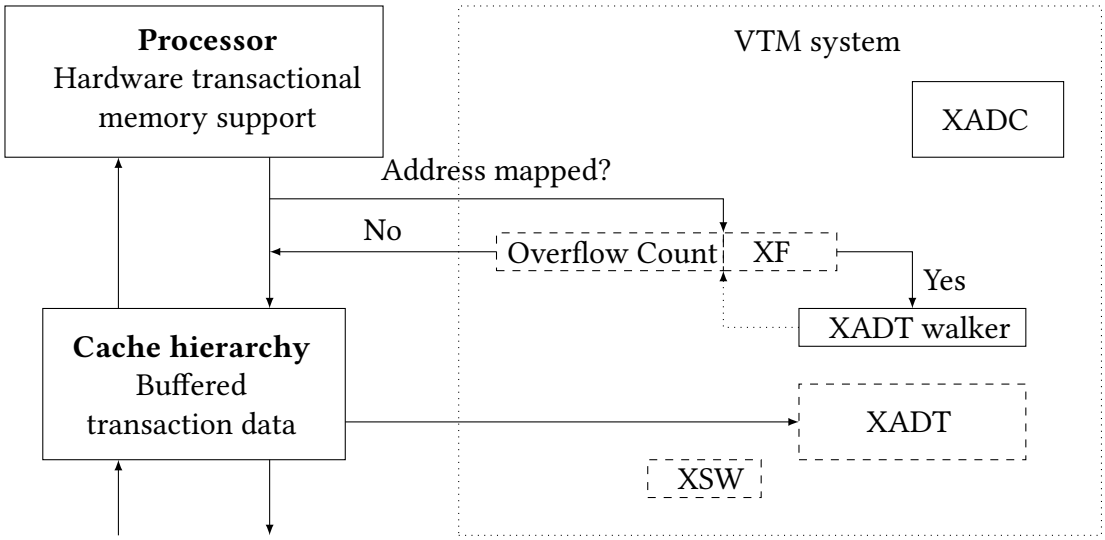

Figure 3.1: The dashed boxes contain structures the authors assume to be located in virtual memory. The hardware structures are represented by solid boxes. According to the Authors the XADC unit caches remapping information for faster accesses of subsequent accesses to overflowed addresses. The overflow count is used to quickly determine if an address overflowed. This is important for conflict detection, since VTM has to determine if a conflict with an overflowed address exists. The graphic was originally provided by [48] and was here reused for explanation.

3.3.3 LogTM-SE: Decoupling Hardware Transactional Memory from Caches

The approach presented by the authors of [63] is based on LogTM [43]. LogTM is a Log based HTM approach, which updates values in place. In case of a transactional conflict a valid system state can be restored by setting back memory following the logged changes. Among other improvements the authors extended LogTM by implementing a signature mechanism inspired by [11].

LogTM-SE relies on a MESI directory protocol for cache coherence and exploits it to detect conflicts. To ensure the correct behavior the authors had to adapt the MESI coherence protocol. This concerns mechanisms to not update the directory entry in case a cache line is evicted from the L1 cache. Since signatures are managed at L1 cache level this measure ensures that conflicts can still be detected. Additionally, the authors developed a mechanism to ensure correctness if a cache line is evicted from the L2 cache. Here, subsequent coherence requests have to be treated specially, since information of cache line distribution is lost. Therefore, the directory state is rebuilt on subsequent requests concerning that cache line. To do so the subsequent coherence request is broadcasted to the L1 caches. According to the answers the directory state can be restored. If a conflict is detected the L2 directory puts the cache block into a special state, which requires signature checks for all subsequent requests. As soon as the conflict is resolved the directory resets the state for the cache block.

Conflicts are currently resolved by using a straightforward resolution policy. A requesting transaction has to abort in case the core the request was forwarded to, detects a conflict.

To track the RSs and WSs the approach uses signatures. Here a part of the address is used to set a bit in the signature. Since a system cannot provide the resources to mark a bit for every possible address the authors presented different approaches. Although all approaches can cause false conflicts, correctness is assured, since a real conflict is never missed.

The system is optimized for commits, since they can be handled fast. In case of an abort the memory operations have to be undone with support of the log. LogTM-SE is able to handle transactions of arbitrary size, thread context switches and migrations as well as paging.

3.3.4 Directory-Based Conflict Detection in Hardware Transactional Memory

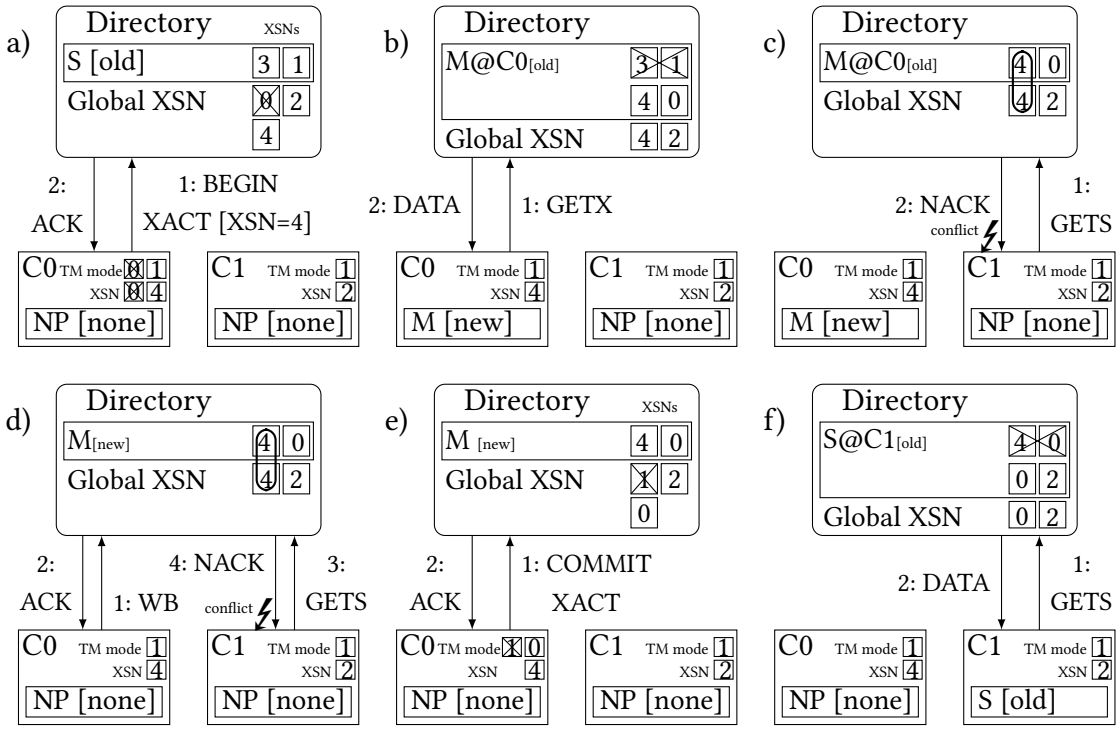

Figure 3.2: In subfigure a) Core 0 (C0) starts a transaction. Therefore, the Global XSNs, managed by the directory, are updated as soon as the begin request hits the directory. Core 0 also marks that it is running a transaction and generates the XSN for this transaction. If core 0 wants to write a cache line it sends the corresponding request (GETX) to the directory. The directory checks if the request can be serviced and updates the XSNs to match execution (see subfigure b)). This is important for conflict detection depicted by subfigure c). Here core 1 (C1) tries to read the cache line core 0 has modified. The directory detects the conflicting access by matching the XSNs stored for the cache line and the global XSNs. Consequently, it informs core 1 about the conflict. Only after Core 0 wrote back the cache line and committed its transaction (subfigure d) and e)) core 1 is able to read the cache line. When core 0 commits the transaction (subfigure e)) also the global XSNs are updated. Therefore, no conflict is detected when core 1 reads the cache line in subfigure f). The graphic was originally provided by [61].

The Authors of [61] developed an approach for an HTM where the conflict detection is shifted to the directory. Instead of managing an explicit RS and WS they used a Transaction Serial Number (XSN) and the state of the cache line to identify conflicts.

The authors describe two schemes, a basic and a more advanced one. The base detection scheme allows the directory to keep track of where a transaction is currently running. Therefore, the directory is explicitly notified whenever a transaction starts or commits. The Authors claim that this information is sufficient to start conflict detection. Since this simple setup would cause a lot of false conflicts the caches are flush cleared at transaction begin. Now the read and writes detected by the directory can be considered as the RS and WS. With their proposal the authors want to support transactions of arbitrary size. Therefore, the cache lines do not necessarily remain local. The directory then cannot tell where the speculative data comes from or if it was read by a transaction. To solve this issue the authors, introduce XSNs.

XSNs are small, reusable per-core identifiers. Every core has an XSN register. The XSN is incremented after every transaction start. The directory saves a global list of all currently active transactions by saving their XSN. Additionally, the directory is also able to save a vector of XSNs for every entry. Old entries are removed lazily. Figure 3.2 shows in detail how this mechanism works.

The authors augmented their basic scheme to avoid flushing the local caches. Here every local cache line can be marked as part of a transactional execution. If a cache line was added to the RS or WS the directory is informed and a bit for the cache line is set locally. Unfortunately, the authors do not explain how a modified cache line which was evicted from a local cache is restored if the transaction, in which it was changed, aborts.

The authors evaluated their approaches. For this they implemented their work into a simulation environment and compared it to LogTM-SE ([63]). With the evaluation the authors were able to show that they were able to reduce execution time between 25 and 55%. On average the proposed system achieved a 15% improvement.

3.3.5 Making the Fast Case Common and the Uncommon Case Simple in Unbounded Transactional Memory

The authors of [7] try to avoid the necessity of having to offer a complex mechanism for unbounded transactions. To reduce the potential number of unbounded transactions they therefore first present a permissions-only cache. The permissions-only cache is an extra cache which augments the regular cache infrastructure. The idea behind the permissions-only cache is that it only saves which cache line belongs to the RS or WS of a transaction. It does not store the actual data. The actual data is saved in the L1 or L2 cache. This technique enables the cores to run much larger transactions, since the RS and WS can grow much larger as the baseline system would allow it. Only if the permissions-only cache overflows an unbounded transaction has to be handled. The authors claim

that a 4 KB permissions-only cache allows a transaction to consume up to 1 MB of data without overflowing.

To handle unbounded transactions the authors, offer two mechanisms. The first mechanism is called OneTM-Serialized. It is the simpler of the two approaches, since it stalls all cores but the one who is executing an unbounded transaction. This way the core running the unbounded transaction is able to use the entire memory hierarchy by itself. For this purpose, the authors propose a shared as well as a private status word. The shared status word can be read by all threads and indicates if currently an overflowed transaction is executed. If a thread detects that currently overflowed transaction is executed, it stalls. Once the overflowed transactions ended the other threads continue execution. According to the authors conflicts are detected during the execution of an overflowed transaction and resolved in favor of the overflowed transaction.

The second, more complex mechanism, is called OneTM-Concurrent. In contrast to OneTM-Serialized it allows other cores to continue their execution when a transaction overflowed. During the execution only one overflowed transaction is allowed. So, if a second transaction overflows it is stalled until the overflowed transaction finished execution. To manage the correct execution the authors, provide per cache line metadata. The metadata consists of 2 bytes. It contains the information if a cache line was read or written (each one bit) by an overflowed transaction as well as a 14-bit identifier. The metadata travels with the data meaning it is additionally provided for every cache line. This increases the data payload. Since only one overflowed transaction is allowed, a thread wanting to execute, and overflowed transaction always first has to check if currently an overflowed transaction is executed. If a transaction switches to overflowed mode all cache lines read or written have to be marked as part of an overflowed transaction. The easiest way to do this is to restart the transaction in overflowed mode. An alternative proposed by the authors is to walk the cache and the permissions only cache to set the metadata accordingly. The authors also propose an approach where the data is adapted gradually. Here only the metadata of overflowed transactions is updated, and non-overflowed cache lines are marked before a context switch occurs. The last two proposals were not evaluated.

The authors evaluated their approaches and were able to show that their implementation had the targeted results. When additionally, the permissions-only cache was activated nearly all overhead due to unbounded transactions, could be eliminated.

3.4 Embedded-TM

The work described in [22] focuses on developing an HTM for embedded systems. The authors focus on energy consumption and complexity of the architecture. Therefore, the work relies on a simple setup of the underlying architecture. The actual implementation

of the baseline transactional memory is not described in detail but can be considered as the ones described in [33] and [23]. The baseline architecture consists of a core, a small scratch pad memory, a snooping device, a bus access, a L1 cache and a transaction cache (TC) or victim cache (VC).

It is important to know if the system supports a TC or a VC, since the functionality of the approach is determined by this. Two functionalities are proposed and evaluated by the authors. If the architecture contains a TC for every core, all transactional data is stored here. The TCs are fully associative. Therefore, the TCs are very small (512B) and the event that it is overflowed by transactional data is likely. To still be able to execute large transactions the authors additionally offer a serial mode which shuts off all the other cores and allows a specific core to exclusively execute its transaction. The serial mode is not only used if a transaction is too big, but also when a transaction exceeds a predefined number of tolerated failed attempts. The authors refer to this approach as TM-vanilla scheme. Another scheme called TM-shutdown-L1WB is the same as TM-vanilla but the TC is shut down after every transaction commit. Since the authors optimize energy consumption, this makes sense.

If the hardware configuration contains a victim cache, the transactional data is stored in the L1 cache. This could be beneficial, since it is much bigger. The limiting factor for the transaction size here is the cache associativity. Since increasing the associativity of the L1 cache causes a higher consumption of energy the authors considered a 4-way set associative cache. If a transaction overflows, the victim cache, which is also (like the TC) fully associative, is utilized to store the overflowed data. The authors chose this strategy to better overcome size limitations. When it is not used it is powered down. If the victim cache overflows or the transaction exceeds a predefined number of tolerated failed attempts the system also switches to serial mode. The authors refer to this approach as TM-victim scheme.

To reduce abort rates the authors proposed to implement a lazy over an eager conflict resolution scheme. Here the conflicts are detected eagerly but are resolved lazily. This means that transactions are only aborted if another transaction commits. Therefore, the committing transaction always wins.

The work was evaluated by implementing it to the MPARM simulation framework which provides a cycle-accurate, multi-processor simulator. The framework also offers power models to estimate power consumption. The findings of the authors were that they achieved the best results for power and complexity when using the TM-victim scheme. The authors recommend that the conflict resolution scheme (eager or lazy) should be configured depending on the workload, since the lazy scheme adds complexity but improves performance for high conflict workloads.

3.5 Real-Time Transactional Memories

The work of [55] presents a timing-predictable HTM. To store the WS of transactions the authors use a transaction buffer. To keep track of RS the read addresses are marked. The size of the transaction buffer is limited since it is fully associative and therefore costly. The authors assume that the programmers are aware of this and will choose to use small transactions. In case the buffer overflows the transactions are serialized. Conflict detection can be performed eagerly or lazily. Since it is not relevant for estimating the WCET and because the authors consider eager conflict detection as expensive, they propose to use lazy conflict detection.

One major aspect of determining a actual WCET concerns program analysis. To detect potential conflicts the authors analyze a program by using a points-to analysis. To determine the size of the RS and WS the authors suggest a symbolic analysis which models all possible program states.

In the evaluation the authors, are able to show that their program analysis is able to detect conflicts. This is an important finding since the tightness of a potential WCET is heavily dependent on the analysis. The authors provide mathematical proof that the WCET for a task t_{wcet} can be calculated with the following formula:

$$t_{wcet} = t_c + rt_a$$

t_c refers to the total cost when execution the task which includes the execution time of the atomic section t_a. r refers to the number of times an atomic section has to be executed before it is committed.

The authors of [51] present real-time support for a STM (RT-STM). The authors did not aim at creating a timing analyzable STM. Instead, they focused on maximizing the number of transactions meeting their deadlines. To achieve this goal the authors enhanced Fraser's STM [24] in the way that it can handle parameters when starting transactions. Additionally, the authors adapted the conflict resolution in a way that contention is resolved in favor of the transaction which is closer to its deadline.

To transfer information linked to a transactions execution, RT-STM was integrated LITMUS^{RT} [9] library. LITMUS^{RT} is a real time operating system. With the underlying Operating System (OS) the authors are able to initialize transactions by transferring the start time and information concerning the deadline of the transaction. In an evaluation the authors were able to show that RT-STM can better guarantee that transactions meet their deadline. The authors compared their approach to other STMs (EnnalsSTM [20], DSTM [34]) and were able to apply different scheduling policies (G-EDF, P-EDF and PD^2).

3.6 Summary

Even though it seems that manufacturers lose interest in TMs, since e.g., Intel drives back efforts to advertise their HTM implementation, the topic still seems relevant. ARM announced the availability of transactional memory extensions in 2019 [58], which shows that the topic is still relevant and research trying to further improve the technology is worthwhile.

Approach	Implementation	Support Unbounded transactions	Contention Management Strategies	Priorities for Transactions	Real-time Features	False Conflict Detection
Sun (Rock) [12]	real hardware	✗	✗	✗	✗	✗
AMD [15]	simulator	✗	✗	✗	✗	✗
IBM Blue Gene/Q [62]	real hardware	✗	✗	✗	✗	✗
IBM system z [35]	real hardware	✗	✗	✗	✗	✗
Intel TSX [16]	real hardware	✗	✗	✗	✗	✗
UTM [3]	simulator	✓	✗	✗	✗	✗
VTM [48]	no implementation	✓	✗	✗	✗	✗
LogTM-SE [63]	GEMS + Virtutech Simics	✓	✗	✗	✗	✗
DirCD [61]	GEMS + Virtutech Simics	✓	✗	✗	✗	✗
OneTM-Serialized [7]	GEMS + Virtutech Simics	✓	✗	✗	✗	✗
OneTM-Concurrent [7]	GEMS + Virtutech Simics	✓	✗	✗	✗	✗
Embedded-TM [22]	MPARM simulation framework	✗	✗	✗	✗	✗
RTTM [55]	FPGA	✗	✗	✗	✓	✗
Our Approach	gem5	✓	✓	✓	✓	✓

Table 3.1: This table provides a comparison of a selection of the approaches discussed in this chapter. The last line refers to the approach we developed in Chapter 5. The first column in the table refers to the approach we refer to. The next column specifies how the approach was implemented (simulator, real hardware or FPGA). The remaining columns refer to the features we are interested in. An ✗ means that the feature is not available. A ✓ means that the approach supports the feature.

To provide an overview of proposals provided by the industry, we present several HTM approaches developed by actual chip manufacturers, at the beginning of this chapter. Some companies (e.g., Sun, AMD, IBM) presented their work in very detail. A few of the presented HTMs were made available through COTS processors (e.g., Intel Tsx).

Next, we present work which deals with contention management strategies. Here we present a selection of policies and explain these briefly. Following the presentation of the contention management strategies, we discuss several approaches which provide unbounded transactions and an HTM proposal which was developed for embedded systems. In the last section of this chapter, we look at two real-time approaches which use TMs.

We conclude this chapter by presenting Table 3.1 which gives an overview of the features which we consider useful for our work. To allow a comparison of different approaches every line represents an approach of a selection of the systems discussed in this chapter. Additionally, we give an outlook on which features the HTM we describe in Chapter 5 provides. Note that we left out the HTM implementation of IBM offering support for the power architecture as well as the ARM TME implementation, since we were not able to gain enough information about these systems. Furthermore, we also left out all STM approaches, since a comparison would not be useful since our focus lies on HTMs.

4

Implementing a Hardware Transactional Memory exploiting MOSI Cache Coherence

Contents

In this chapter we designed and implemented a complete HTM. In contrast to Chapter 5 the focus does not necessarily lie on embedded systems even though we kept them in mind. Instead, we here focus on an efficient and robust HTM implementation. The HTM implemented exploits MOSI cache coherence, since we consider it beneficial when designing the HTM specifically considered for embedded systems (see Chapter 5 for more information). Furthermore, we need the implementation provided in this chapter as baseline to evaluate our findings in Chapter 5.

For our implementation we chose to use the gem5 simulator, since it is a holistic system which offers many options and is highly configurable (see 2.3 for more information). Nevertheless, we had to reimplement cache coherence to be able to implement our version of an HTM.

The HTM was realized without having to perform any changes to the cores. An interface to use the HTM is offered in software. Most functionality of the HTM is provided by the cache controllers. In this chapter we first give a detailed description of the underlying system. Then we explain how we implemented cache coherence. After laying out some design choices and describing the integration of the HTM, we explain the interface. We close the chapter by providing a short summary.

4.1 Basic System

This section describes the underlying system providing the basic framework for the HTM integration (see Section 4.4 for more details). Further it also describes what adjustments had to be made to the original gem5 simulator (see Section 2.3 for more details) to best prepare the implementation of the HTM. The setup of the baseline system was chosen to resemble a high-performance embedded system. Therefore, the baseline system is a multi-core system with private L1 caches and a shared L2 cache (LLC). The L2 cache is non-inclusive non-exclusive. Therefore, transaction aborts due to evicts caused by the L2 cache are avoided. We further assume that the directory is able to contain entries for every cache line contained in one of the L1 caches. The number of cores can vary and the system is also attached to main memory. Figure 4.1 depicts and describes the baseline system in more detail.

In 2017, when this work started, the gem5 simulator (see Section 2.3 for more details) did not offer an HTM. To implement an HTM we developed an approach, for which mainly the memory hierarchy of a system has to be adjusted. Most parts of the memory hierarchy of the gem5 simulator is provided by the Ruby Memory System Simulator [36]. The ruby memory system simulator consists of three parts:

1. Interconnection network, simulating the communication within the simulated multi-core processor.

2. Structures for simulating caches and memory.

3. Coherence controllers, ensuring cache coherence.

To simulate the memory hierarchy and facilitate implementation of the HTM, the goal was to reuse as much infrastructure as possible of the provided memory system simulator.

With minor adjustments the basic system uses the provided interconnection network as well as the provided cache and memory structures. The provided coherence controllers were replaced for the following three reasons:

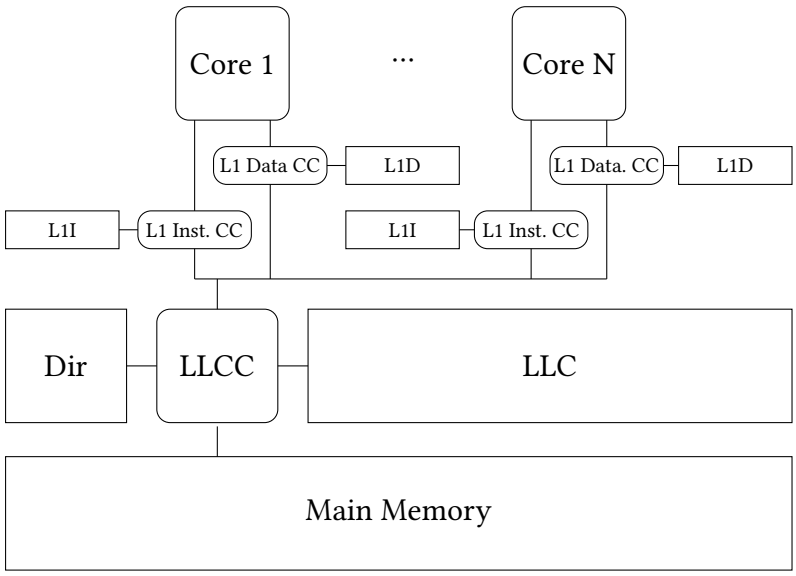

Figure 4.1: The baseline system consists of up to N cores. Every core has a private L1 data (L1D) as well as an instruction (L1I) cache. Each cache is managed by its own cache controller (L1 Data CC, L1 Inst CC or LLCC). The cores access their caches through the L1 cache controllers. If they do not contain the requested data, the request is forwarded to the LLC. The LLC is shared and is accessed through the LLC controller (LLCC). If the LLC cannot service, the request it is redirected to main memory. Communication ways of the system are indicated by the lines of connection but is described in more detail later in this section. Cache coherence is directory based, indicated by the availability of the directory (Dir). The directory is also accessed via the LLC controller (LLCC). The graphic is inspired by [57, S. 154]

1. The cache coherence in the gem5 simulator is provided by a domain specific language. It was not foreseeable if it could be used and augmented to implement an HTM.

2. The implementation of an HTM strongly relies on the full control and detailed technical knowledge of the cache controllers, which is ensured by our own development.

3. We assumed that the use of a more common programming language would increase comprehensibility of our work.

Due to the above-mentioned reasons the cache controllers as well as cache coherence were completely reimplemented (see Section 4.2 for more details).

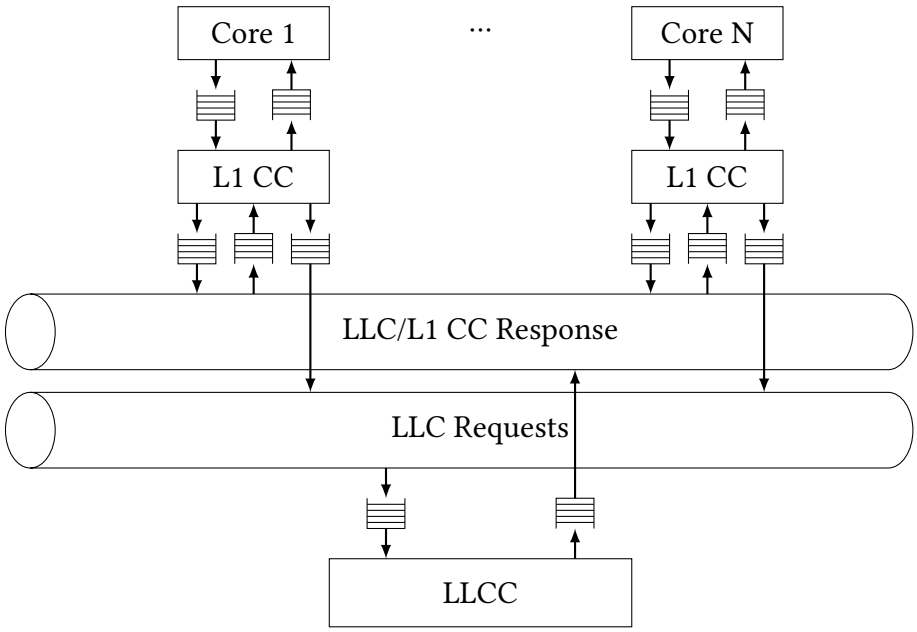

Figure 4.2: To exchange messages and data, the basic system relies on two interconnection networks and queues. Each core is connected to a queue allowing it to send requests to the L1 cache controller (L1 CC). To answer the requests the L1 cache controller also uses a queue. The L1 cache controllers are connected to two interconnection networks. One network (LLC Requests) handles the communication of the L1 cache controllers to the LLC controller. This concerns requests which could not be serviced by the L1 cache (e.g., a read request, to a cache line not contained by the L1 cache). The second network (LLC/L1 CC Response), handles communication the other way around, meaning from the LLC controller to the L1 cache controller. It also allows the L1 cache controllers to send messages to the other L1 cache controllers. This concerns e.g., forward requests of owned cache lines (see Section 2.2 for more details). The messages in the interconnection network and queues are strictly ordered. As indicated, requests are buffered before they are routed to their destination, where they are buffered again until serviced by the hardware unit. Graphic inspired by [36].

Although the basic functionality of the interconnection network was not adapted, some changes were made. The changes relate to adding new message types and the topology of the communication network. To enable cache coherence basic message types had to be added (see Section 4.2 for more details). Later extra message types were added for the HTM implementation (see Section 4.4.2 for more details). To allow the necessary communication, the topology of the communication network was customized. The following communication ways were established by setting up the network accordingly:

- Core to L1 cache controller and vice versa

- L1 cache controller to L1 cache controller
- L1 cache controller to LLC controller and vice versa

The network was set up by setting a configuration file within the gem5 simulator. A detailed description of the communication and an overview of the topology is given by Figure 4.2.

The structures for the caches and the memory, provided by the original gem5 simulator, have been adapted and used. The baseline functionality of these structures was not adapted. Features such as the replacement strategies or the management of the associativity remained unchanged. Changes concerning the structures resembling e.g., the cache lines were augmented, which is described later (see Section 4.4 for more details).

As stated earlier, the cache coherence was replaced by our implementation. For this the cache controllers were completely redesigned and reimplemented. The cache coherence is implemented to the L1 cache controllers and the LLC controller. The baseline system no longer requires the use of the domain specific language for cache coherence. Section 4.2 gives a detailed description of the implementation.

4.2 Implementing Cache Coherence

This section provides an overview of how the memory hierarchy was adapted to enable cache coherence. The cache coherence protocol itself is only handled briefly. For a more detailed description of how cache coherency works, and which states and messages are needed see [57, S. 151-190] or Section 2.2.

The implementation of an HTM strongly relies on cache coherence (e.g., for conflict detection). To ensure cache coherence the MOSI cache coherence protocol (see Section 2.2 for more details) was implemented for the baseline system. We chose a directory-based approach since it provides better scalability for multi-cores with a high core count compared to a snooping-based approach [57, S. 151]. The MOSI cache coherence protocol was applied since it brings some advantages when later adapting the baseline HTM. This concerns silent state changes in the L1 cache controllers (see Section 5.2.3 for more details). The implementation of the cache coherence protocol is based on the detailed description provided by [57, S. 151-190]. To enable cache coherence, the cache controllers and the directory were adjusted and augmented. Additional message types were added to the network, to be able to map the communication correctly.

Concerning cache coherence, the controllers have to be able to complete four main tasks:

1. Changing the state of a cache line.

2. Triggering a new request (e.g., a read request for a cache line which is currently invalid).

3. Delaying a request until the state of a cache line indicates that request can be handled.

4. Forwarding of data or other information (e.g., an invalidation acknowledgment).

All cache controllers were adapted to accomplish these tasks.

The L1 cache controllers were augmented to support new message types. This means that the L1 cache controllers are able to handle incoming messages and know how to react when receiving a message with such a type. Incoming messages to the L1 cache controller can either originate from the corresponding core or the LLC controller and relate to a specific cache line. To make cache coherence work every cache line has to have a state. Therefore, the structure representing a cache line was extended to support states. The L1 cache controller can set a state for every cache line held by the L1 cache. To save the information representing the state, four extra bits are needed. This is because the cache lines in the L1 cache can either be in one of four stable states (modified, owned, shared or invalid) or in one of 11 intermediate states (see Section 2.2 for more details). The state of a cache line and the type of the incoming message ultimately defines how the L1 cache controller behaves.

The LLC controller has to be handled differently, since the directory is integrated at this level of the memory hierarchy (see Figure 4.1). Additionally, incoming messages originate from the L1 cache controllers or the main memory, which means that other message types have to be used. Just as the L1 cache controller the LLC controller now supports all messages necessary for the functionality of cache coherence. The LLC controller was also extended so it can manage the coherency states correctly. In contrast to the L1 cache controller only four stable cache states (modified, owned, share, invalid) are necessary. Because the MOSI protocol demands the LLC controller to know where certain cache lines are located, more information for every cache line has to be saved. Besides the state also an owner as well as a list of sharers has to be stored. This is necessary to inform potential sharers about a state change or to redirect a request to the owner of a cache line. To save this information the directory was utilized. Every entry allows to save the necessary information for a cache line.

For cache coherence in sum 13 message types and 15 states are supported by the L1 cache controllers. Since some combinations, existing of incoming message and state of the cache line are not possible, the L1 cache controllers have to provide the correct behavior for 159 cases. The LLC controller only supports four states and 9 incoming mes-

sages. Since here also a few combinations are impossible the LLC controller provides a predefined behavior for 27 cases (see Section 2.2.6 for more information).

4.3 Design Choices for Hardware Transactional Memories

When designing an HTM there are multiple options to set up conflict detection, conflict resolution and version management. The proposed baseline system was designed to match commercial HTMs (e.g., Intel TSX [16]). This section describes how the key functionalities are set up and how they work.

4.3.1 Conflict Detection

The conflict detection is performed eagerly (see Section 2.1 for more details). Conflicts are detected on cache line granularity. In the baseline HTM a conflict can only be detected by an L1 cache controller. Whenever an L1 cache controller detects a violation of a real data dependency a conflict is detected. Conflict detection is performed by the means of the MOSI cache coherence protocol and the RS and WS. The coherence protocol ensures that all problematic read or writes are delegated appropriately. This means that it is ensured by the protocol that every conflicting access can be noticed by the L1 cache controllers. If the cache controller receives a write request to a cache line contained in the RS or WS of the transaction running on the corresponding core, a conflict is detected. The same applies if the L1 cache controller receives a read request to a cache line contained in the WS. A read request to a cache line in the RS is not problematic and can be serviced, since no real data dependency is violated.

Another source of conflicts relates to the capacity of the L1 cache. If a cache line, which is part of the RS or WS, is evicted from the L1 cache, because the associativity or size of the L1 cache is exceeded, a conflict occurs. The reason for this is that the data is speculative and therefore cannot be written back. This is because of the isolation criteria. If the data were written back, it would get violated, because a conflicting request cannot be detected by the LLC controller and would therefore be serviced. Allowing other cores to process speculative data is bad practice, since it can lead to erroneous behavior. Continuing the transaction without writing back the speculative data is also no option. If the transaction would reaccess the cache line, the data would be reloaded from the LLC ignoring the already performed updates. This could cause unpredictable behavior. Therefore, it is unavoidable to abort the transaction.

4.3.2 Conflict Resolution

In the baseline HTM conflicts are resolved following a simple but widely spread strategy. The L1 cache controller, which detects the conflicts, aborts the transaction running on its corresponding core. This approach is simple and can be easily applied. More complex conflict resolution policies imply more communication making execution inefficient. Therefore, they are not considered to be provided by the baseline HTM. Chapter 5 looks at the difficulties and describes how more complex conflict resolution strategies can be applied for HTMs. The implemented resolution policy cannot guarantee any progress, which is why the software executing transactions has to provide an alternative path of execution for their critical sections. The alternative path of execution can be executed, when the system detects that the transactions will not commit. An abort counter can define after how many failed attempts the transaction is considered to not commit. When the number of aborts exceeds the counter the alternative path of execution is chosen instead of rerunning the transaction. Since deciding if the alternative execution path should be taken is determined in software, no hardware extensions are needed to offer this functionality.

4.3.3 Versioning

The ability to allow at least two versions of a cache line is a key condition to implement an HTM as described. To perform a rollback in case of an abort, the data has to be restored to the values before the transaction has started. The system must be able to allow two versions of the same data. One is speculative and used within the transaction. The other version contains the data before the transactional execution. The speculative version of the cache line is usually held by the L1 cache, since this is where the cache line is used. The backup of the cache line is held by the LLC controller. In case of an abort, the L1 cache controllers invalidate all cache lines contained in the WS. This causes the L1 cache controller to reload the cache line if it is needed again. The speculative data will be discarded.

4.4 Integrating the Hardware Transactional Memory

In this section we describe the integration of the baseline HTM. The proposed approach only requires changes to the memory hierarchy. No changes to the cores are required. We first describe how the RS and WS is managed. Then the necessity of additional coherence messages and their implementation is explained. The last part of this section describes how the cache controllers were modified to be able to manage transactional executions.

4.4.1 Managing the Read and Write-Set

Managing the RS and WS is an essential functionality for an HTM. Conflict detection only works when combining the cache coherence protocol with the RS and the WS. The RS contains all cache lines read by a transaction and the WS contains all cache lines written by a transaction.

To maintain this information every cache line was extended to hold two extra bits of information. The bits are set if a cache line is added to the RS or the WS. The bit for the RS is set by the L1 cache controller, if it detects that a transaction is running, and a read request is serviced successfully by the L1 cache. A request can be handled when the L1 cache contains the corresponding line, and the state indicates that it can be read. Same applies for the bit set when adding the cache line to the WS. It is also only set if a transaction is running on the corresponding core and a write request was successfully serviced.

In the baseline HTM the RS and WS is managed locally by the L1 cache controllers. The size of the RS and WS is bounded by the size of the L1 cache. The lines added to the RS and the WS can increase as long as no cache line contained in the RS or WS is evicted from the L1 cache. Therefore, the maximum size of the RS and WS is equivalent to the number of cache lines the L1 cache can fit. The number of cache lines contained in the RS and WS is limited by the associativity of the L1 caches.

4.4.2 Additional Coherence Messages

To operate the HTM the system needs to support additional messages compared to the basic system. This is because the HTM needs some additional functionality, which cannot be achieved with the existent message types. To obtain concurrency with an HTM the characteristics of optimistic synchronization have to be considered, which is why four new message types were added. Table 4.1 lists all of them and offers a brief description of their functionality. A more detailed explanation is given in the following paragraphs.

Message type:	**Brief description:**
putMnoData	Allows to update the state of a cache line at LLC level without changing its data.
updateMO	Enables the L1 cache controllers to update a cache line at LLC level without changing its state.
redirectRW	Redirects a read or write request which failed due to a conflict to the LLC controller.
sendData	Acts as a response to *redirectRW* message. It enables the LLC controller to send the data without any other information.

Table 4.1: New message types for the baseline HTM

If a cache line is written inside a transaction, the state of the cache line will be set to the modified state by the L1 cache controller. Consequently, that L1 cache controller will be set as the owner of the cache line in the corresponding directory entry. It is important to notice that the changed data of a cache line, which is contained in the WS, is not valid until the transaction commits. This can be problematic if the transaction aborts. The reason for this is that the cache lines contained in the WS have to be invalidated in the L1 cache. Additionally, the ownership of the cache line then has to be updated in the directory. Usually this is done by utilizing a *putM*-message. The problem with the *putM*-message type is that it initiates a write back of the data of the corresponding cache line. In case of a transaction abort, the cache lines contained in the WS cannot be written back, since the data is corrupted, and the atomicity criteria would be hurt. Therefore, the new message type *putMnoData* was implemented, which only straightens out the ownership of the cache line in the directory. This is important because redirected requests concerning the cache lines in the WS can no longer be serviced by the L1 cache controller. With the new message type, the data of the cache line remains untouched. Since the ownership was updated, requests to the corresponding cache line will no longer be forwarded to that L1 cache controller. Instead, future requests will be served by the LLC until the ownership of the cache line changes.

A modified or owned cache line contained by an L1 cache holds the most recent data. The version contained in the LLC is deprecated and remains outdated until updated. Therefore, the version in the L1 cache cannot be lost. Unfortunately, exactly this can happen in a transactional execution if no precautions are taken. The problem occurs if the modified or owned cache line is consumed by a transaction, which later aborts. Because the transaction invalidates its WS, the data will be lost, and future read or write requests are directed to the outdated version in the LLC. Therefore, the cache line has to be updated in the LLC before it is consumed by a transaction. If the cache line is updated before it is consumed by a transaction no data loss will occur if it aborts, since the version in the LLC now contains the most recent data. To update data a message type was implemented, which only updates the data of the corresponding cache line in

the LLC and leaves the state and ownership of the cache line untouched. The name of the new message type is *updateMO*.

The coherence protocol always expects an answer for a request. In an HTM environment this needs extra attention. The following example shows why this aspect cannot be left unhandled and why an extra coherence message is necessary: If a transaction wants to read a cache line, which is not yet included in its corresponding L1 cache, the request is forwarded. For this example, it is assumed that the cache line is marked as modified in the directory. Therefore, the LLC controller redirects the read request to the L1 cache controller which is set as owner for the cache line in the directory. If the cache controller detects that its corresponding core is also running a transaction and the cache line is part of the WS a conflict is detected (see Section 4.3.1 for more details). In this situation the request has to be redirected again, since the cache line cannot be forwarded, because it was modified, and the transaction has not yet been completed. The regular protocol does not provide such functionality, which is why the new message type *redirectRWreq* was added to the system allowing such a redirect. When the redirected message reaches the LLC controller it is serviced with the data contained in the LLC, which is the most recent valid version of that data.

The last message type added to the system is sent from the LLC controller to the L1 cache controller and is called *sendData*. It is used to answer the redirected request discussed in the previous paragraph. The message is necessary, since redirected requests are not handled by the underlying cache coherence protocol. The only information the new message type contains is the data as well as the reference to which cache line it belongs. Other information concerning coherence (e.g., acknowledgment count) have already been sent to the L1 cache controller, when the request was originally stated. If they were sent again, the local information received after the original request would be overwritten and unpredictable behavior could occur (e.g., deadlock, because the L1 cache controller waits for data it already received).

4.4.3 Modifying the Cache Controllers

The transactional execution requires the cache controllers to offer some additional functionalities. To feature them the cache controllers had to be adapted. This mostly concerns the L1 cache controllers, since most parts of the transactional execution is managed here.

The L1 cache controllers have been adapted, so that they offer a small data section. The data section is used to save and manage transaction related data. Values saved in the transaction configuration block are:

1. Transaction depth (max. one, meaning no nested transaction can be executed)

2. Address of the abort handler
3. Abort reason
4. Transaction control value

Currently, it is also used to back up the registers when a transaction is started. The general-purpose register r0-r16 as well as the floating-point registers s0-s31 are saved. The transaction configuration block is managed in the L1 cache controller. Note that saving the registers can be easily shifted to the stack. Then, only the location of the first register saved would have to be stored in the transaction configuration block. To save all information (incl. using registers) 208 bytes are needed.

To guarantee correct transactional execution, the L1 cache controllers need to know when their corresponding core is running a transaction. Therefore, the system supports a memory mapped interface to start and commit a transaction (see Section 4.5 for more details). Depending on the address of a specific memory operation, the L1 cache controller can identify which action was performed and updates the transaction configuration block accordingly. When a transaction starts, the depth in the transaction configuration block is set to one. The system does not allow nested transactions, which is why the depth can be only one or zero. With this information, the L1 cache controller can identify if its corresponding core is running a transaction or not. When the execution is transactional, conflict detection (see Section 4.3.1 for more details) will be activated and memory operations performed by the other cores will be monitored for violations of real data dependencies. If no transaction is running, conflict detection is not active.

When detecting a conflict, the abort reason is written to the transaction configuration block, since it is later read by the abort handler. To inform the core that it has to abort its transaction, the cache controller is able to trigger an interrupt. When the core then executes the interrupt service routine, it loads the address of the abort handler of the predefined location in the transaction configuration block. It then jumps to where the abort handler is located and starts to execute it. This ensures that the core's state is restored. The transaction then can either be run again or executed in fallback mode.

Transaction starts, aborts and commits have to be treated as memory fences. If not, cache lines outside of a transaction could be considered as part of the transactional execution and could be mistakenly added to the transactions RS or WS set. This could cause false conflicts and disturbs the transactional execution. For transaction starts, it can be easily ensured by software that all previous memory operations are processed, when starting a transaction (see Section 4.5 for more details). Transaction aborts are more critical and require changes to the L1 cache controllers. This is the case, when a transaction aborts and one or more memory requests are still in flight. When the request cannot be processed by the L1 or LLC it has to be answered by the main memory. This is time-

consuming and could take longer than it takes to abort and re-execute the transaction. If that happens the request will be treated as part of the re-executed transaction and will mistakenly be added to its RS or WS. To solve this issue, the cache controller has to ensure that all memory operations are finished prior to triggering the interrupt, indicating to its corresponding core that it has to abort its transaction. This issue cannot be handled in software and therefore the cache controller has to keep track of all outgoing memory requests. This is realized with a counter. If a coherence request leaves the L1 cache controller the counter is raised by one. An incoming message will lower the counter by one. Only memory requests, which require an answer will raise the counter. Incoming messages, which are considered as an answer to these requests will lower the counter. An interrupt can only be triggered when the counter equals zero. For transaction commits no further actions have to be taken, since all memory operations have already been handled.

If a transaction commits or aborts the cache controller has to clean up the transaction's RS and WS. Therefore, the cache controllers need to offer routines for transaction commits and aborts. When a cache controller notices that a transaction commits or aborts the cache controller has to clear the RS and WS. This is done by setting the bits, indicating that the cache line is part of the WS or RS, to zero. If the transaction commits regularly no further actions have to be taken. For aborting transactions also the cache coherence has to be straightened out: When a cache line was accessed to be written, it is set to the modified state in the L1 cache controller. Because of the coherence protocol, the L1 cache controller is also set as owner for that cache line in the directory. To straighten this out, the L1 cache controller sends a *putMnoData*-message to the LLC controller to inform it that the cache line was invalidated. This cannot be performed atomically which is why a follow up situation can occur, which cannot be handled by the regular protocol and therefore has to be considered: If the *putMnoData*-message (see Section 4.4.2 for more details) was sent to the LLC controller, but has not yet reached it, the L1 cache controller will still be considered as owner of that cache line. Therefore, read and write requests are still redirected to it. If the cache controller receives such a request, it has to now detect that it cannot handle the request and has to redirect the request to the LLC controller. The L1 cache controller is able to detect this situation, because it observes that no transaction is running, and the data requested is invalid. This can only occur in a transactional execution, because the cache line does not devolve in an intermediate state when it is removed from the WS. Functionality allowing this could be added, but currently is not required, since the problematic situation can be identified as described. Because the redirect message cannot be the original request, the *redirectRWreq*-message type is sent, when such a situation occurs. The LLC controller then meets this request by also sending the newly added message type *sendData*, only sending the data of the cache line (see Section 4.4.2 for more details). Figure 4.3 depicts and explains a detailed example of such a situation.

If a cache line, part of the RS or WS, is evicted, the cache controller has to trigger an interrupt indicating its corresponding core to abort its transaction. In case the cache line

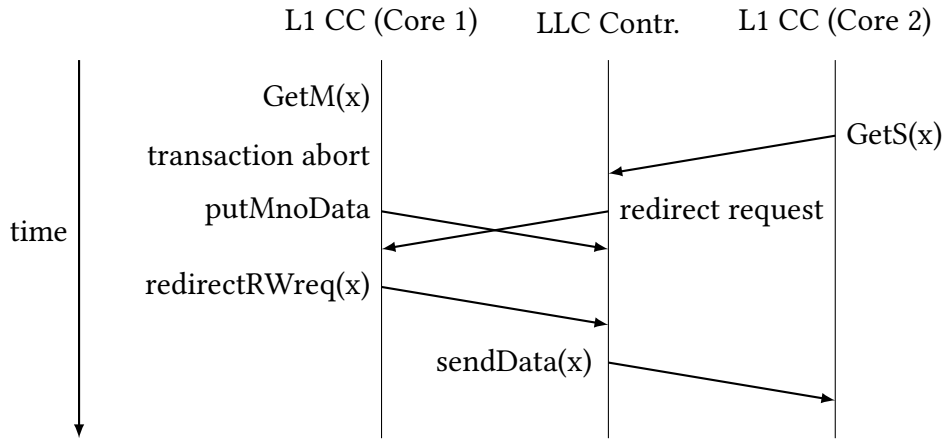

Figure 4.3: In this example core 2 tries to read the cache line x. Since the corresponding cache controller (L1 CC (Core 2)) cannot find a copy in the L1 cache, it redirects the request to the LLC controller (LLC Contr.). The LLC controller detects that the cache line is owned by the L1 cache controller of core 1 (L1 CC (Core 1)) and redirects the request there. Meanwhile core 1 aborted its transaction and sends a *putMnoData*-message for cache line x, which was part of the transactions WS, to straighten out cache coherence. Shortly after the redirected request arrives at the L1 cache controller of core 1. The L1 cache controller of core 1 then detects that its version of cache line x is invalid and redirects the request to the LLC controller by sending a *redirectRWreq*-message. Because the message was redirected, the LLC controller answers the request, by sending a *sendData*-message to the L1 cache controller of core 2.

is part of the WS the L1 cache controller also has to send a *putMnoData*-message (see Section 4.4.2 for more details) to the LLC cache controller, to ensure that cache coherence remains consistent. This has to be done immediately and not in the abort routine, since the cache line is evicted from the L1 cache and therefore cannot be handled later.

A modified cache line contains the most recent data. If a transaction is started and adds this cache line to its WS, it has to be backed up. Otherwise, the most recently written data could be lost, in case the transaction has to be aborted. The L1 cache controller was adapted in the way that it detects if a transaction wants to add a cache line in the modify state. This is easily possible, because only the state has to be checked before adding the cache line to the WS. When the state of the cache line is modified the L1 cache controller sends an *updateMO*-message (see 4.4.2 for more details), to update the cache line in the LLC. Because this is on the critical path concerning execution time, the cache line is then added to the WS, without waiting for an acknowledgment message from the LLC controller. A similar situation occurs, when a cache line was modified by a core not running a transaction and now is requested by another core running a transaction trying to add the cache line to its WS. Here the L1 cache controller was adapted that it detects that the cache line came from another core instead of the LLC. This can be easily

done, because of the underlying cache coherence protocol and the provided message types. When detecting such a situation an update of the data is initiated by the L1 cache controller, since the data would be lost, if the transaction later aborts. Figure 4.4 depicts and describes a detailed example of such a situation.

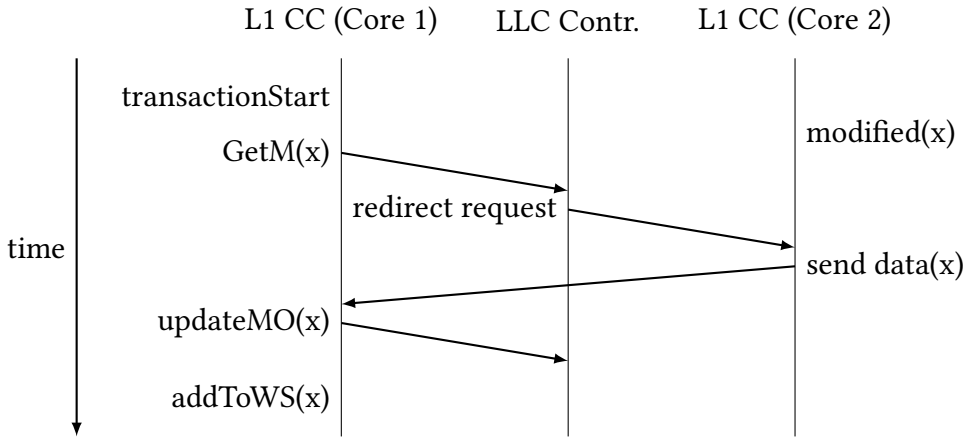

Figure 4.4: Core 1 wants to add cache line x to its WS after starting a transaction. Since the cache line is not located in the L1 cache, the request is forwarded to the LLC controller (LLC Contr.). The LLC controller identifies the L1 cache controller belonging to core 2 (L1 CC (Core 2)) as owner of the cache line and redirects the request to it. The L1 cache controller of core 2 then sends the cache line to the L1 cache controller of core 1 (L1 CC (Core 1)). It identifies that the cache line was not sent from the LLC controller and sends an *updateMO*-message to the LLC controller, before adding it to its WS.

4.5 Interface

The baseline HTM has two possibilities (transaction start and commit) for a user to interact with it. Before being able to use the baseline HTM an initialize function has to be called by a user. Afterwards a user can actively start or commit transactions. The system does not offer an explicit abort function, since it was not required to execute the evaluated benchmarks. It could be added easily, if required. The interface is provided via software, which is why no changes to the cores themselves had to be made. To be able to use the interface, hardware changes to L1 cache controller were required further described in Section 4.4.3. The following paragraphs give an overview of how the interface is implemented.

Initializing baseline HTM: Before using the baseline HTM an initialization function has to be called. This function uses the unix syscall *mmap* to map an address range into memory. These addresses are then exclusively reserved to be used for transactional

operations. For the HTM an addresses range starting at 0xf0000000 until 0xf0200000 is mapped. Due to debug and experimental reasons an address space, covering 2MB of storage is reserved. Note that the size could be drastically reduced, since only 208 bytes have to be addressed to enable the complete functionality.

Transaction start: A transaction is started by writing the address to the abort handler to address 0xf0000004. Then the registers are backed up. This is done in the address range of 0xf0000064 - 0xf0000244. Finally, the transaction depth is written to the address 0xf0000000. This access is intentionally performed last, since it is also used to detect if a transaction is going to commit. Then a Data Memory Barrier (DMB) call is performed, which acts as a memory barrier. It ensures that all subsequent memory requests are definitely handled after. This is important because otherwise, memory operations belonging to a transaction might be missed. If the transaction started correctly, it returns 0 as a return value.

Abort handler: When an L1 cache controller detects a conflict, the transaction running on the corresponding core has to be aborted (see Section 4.3.1 for more details). This is indicated by an interrupt. Since the address to the abort handler is stored in the transaction configuration block it can be accessed within the interrupt service routine. The interrupt service routine then sets the program counter (PC) to the address of the abort handler and execution continues. The handler is provided in software. First, a zero is written to address 0xf0000010 indicating to the L1 cache controller that the transaction is aborted. After that, a DMB is called since the L1 cache controller cleared the RS and WS (see Section 4.4.3 for more details), which involves memory accesses. Next all registers are restored by reloading them from memory range 0xf0000064 - 0xf0000244. Finally, the abort reason is returned. This indicates why the transaction aborted (conflict, capacity or other), which is read from address 0xf0000008.

Transaction commit: If a transaction finished successfully, it has to commit. For this the transaction depth is set back. This is done by writing zero to address 0xf0000000. The controller detects this access and triggers the commit routine within the L1 cache controller (see Section 4.4.3 for more details).

4.6 Evaluation

In this section the performance of the implemented HTM is evaluated. The baseline HTM is compared to an execution with POSIX synchronization. Before discussing the benchmarks, we will give a short overview on the hardware costs of our approach. Then the settings of the gem5 simulator and the configuration of the benchmarks are shown. At the end of this chapter the results are analyzed and explained.

4.6.1 Estimation of Hardware Costs

Since we use a simulator for our work, we are not able to estimate which actual impact our requirements would have in real hardware concerning size and complexity. Nevertheless, we are aware that our requirements definitively impact the underlying hardware. Therefore, we will present the modifications which we believe impact the hardware in terms of size and complexity when augmenting a system towards featuring an HTM.

Every cache line has to be equipped with at least two extra bits. The two extra bits are needed to determine if a cache line is part of the RS or WS. For the extra bits to be set extra hardware is required which can identify that a transaction is currently running. Depending on the current operation (read or write request) it has to set the corresponding bits to indicate if a cache line was added to the RS or WS.

To identify conflicts additional logic is required which results in extra hardware. The extra hardware has to be able to identify if incoming requests conflict with a cache line which is part of the RS or WS. Therefore, the hardware has to be able to check the additional bits indicating if a cache line is part of the RS or WS. Depending on the type of access and the status of the cache line the hardware has to ensure that the correct measures are performed (e.g., an abort).

The cache controllers as well as the LLC controller have to be augmented to support new messages (see Section 4.4.2 for more information). Since new message types are used additional logic has to be supplied to provide the functionality which is required to feature the new requests.

The cache controllers have to be adapted in the way that they support the interface we use. This causes additional logic since the cache controllers have to be adapted in the way that they are able to identify certain requests (e.g., transaction start, transaction commit).

Once a transaction aborts or commits we need hardware functionality which allows us to clear the RS and WS. Therefore, the cache controllers have to be augmented with logic which is able to clear the status of the cache lines.

4.6.2 Benchmarks

For the evaluation, the STAMP benchmark suite [13] was used. The applications used in the suite are explicitly chosen to evaluate TM systems. The STAMP benchmarks are highly portable and used to evaluate a wide variety of TM systems including hardware,

software and hybrid approaches. Since the benchmarks are also used for the evaluation in Chapter 5, a general detailed description of the suite is given in Section 2.4.

4.6.3 Methodology

To evaluate the performance of the baseline HTM the STAMP benchmarks were executed. They were executed by the gem5 simulator in syscall emulation mode in which system services are directly provided by the simulator [37]. The exact configuration of the gem5 simulator can be found in Table 4.2.

Table 4.2: System Configuration

Num CPUs	{1,2,4,8,16}
Microarchitecture	ARM Cortex-A15
L1 data cache	32KB
L1 data cache assoc.	8
LLC cache	2MB
LLC assoc.	16
Cache Coherence	directory-based
Coherence Protocol	MOSI

The STAMP benchmarks were evaluated with the small input set. This is recommended by the authors of [13] when using a simulator. If a transaction exceeds a certain amount of tries it is executed in fallback mode. The fallback path executes the same code as the transaction. The difference is that instead of starting a transaction a mutex is locked when entering the critical section. This assures that the thread is the only one accessing the critical path. After execution finished, the mutex is unlocked and other threads can enter the critical section using transactions or a mutex. The exact launch configurations and the maximum number of attempts for a transaction to commit are stated in Table 4.3.

4.6.4 Analysis

Figure 4.5 depicts the speedup graphs of all of the eight STAMP benchmarks. Within the speedup graphs the execution of the baseline HTM is compared to the execution with POSIX Threads. The x-axes relate to the number of cores the execution was performed with and the y-axes refers to the speedups achieved. The legend entry *pthreads* refers to the execution ran with POSIX Threads and the entry *baselineHTM* to the execution performed on the baseline HTM. The speedups are calculated as shown by Equation

Table 4.3: Configuration of STAMP Benchmarks

Benchmark	Parameters
bayes	-v32 -r1024 -n2 -p20 -s0 -i2 -e2
genome	-g256 -s16 -n16384
intruder	-a10 -l4 -n2038 -s1
kmeans	-m40 -n40 -t0.05 -i inputs/random2048-d16-c16.txt
labyrinth	-i inputs/random-x32-y32-z3-n96.txt
ssca2	-s13 -i1.0 -u1.0 -l3 -p3
vacation	-n2 -q90 -u98 -r16384 -t4096
yada	-a20 -i inputs/633.2
fallback exec.	after 10 failed attempts or capacity issue

(4.1). The reference execution time refers to the run of the benchmark with one core and no synchronization. It is marked as solid horizontal line in every speedup graph.

$$\text{speedup} = \frac{\text{reference execution time}}{\text{examined execution time}} \tag{4.1}$$

For the evaluation we look at the parts of the execution which are executed in parallel. Otherwise, potential improvements achieved could be concealed. The time taken lasts from calling the first parallel operation (e.g., transaction start or locking a mutex) until to the last (e.g., transaction end or unlocking a mutex). The reason for this is that some benchmarks spend a long time preparing the critical part of the execution. The benchmark *bayes* is a prime example for this behavior. Most of its execution time the benchmark spends generating structures, which are used to learn the structure of a bayesian network. This is not relevant for the evaluation, which is why looking at the entire execution time is not reasonable.

The executions of the benchmarks can be assigned to three categories. The first category contains all benchmarks where the POSIX synchronization outperforms the baseline HTM. Note that the benchmarks of the first category in general perform poorly. This is also the case when executed with POSIX Threads. The benchmarks *labyrinth* and *yada* belong to this category. The speedup graphs for these benchmarks are marked with a dark gray background in Figure 4.5.

The benchmark *labyrinth* suffers from quite a lot of capacity conflicts. These let a transaction execute fairly long until it aborts, since it can take a while until a transaction exceeds the capacity of its RS or WS. Therefore, the execution on the baseline HTM with one core suffers a major slowdown compared to the execution on one core executed with POSIX Threads. When the core count increases, contention gets higher, and transactions are executed in fallback mode much earlier. If a transaction aborted ten times and was

not able to commit it will be executed in fallback mode. This speeds up execution for the benchmark, since the capacity conflicts are resolved much faster the higher contention gets. Note that overall execution still stays below the reference execution (one core no synchronization) since execution is more or less serialized.

The benchmark *yada* also suffers from capacity conflicts but also of a very high contention when executed on the baseline HTM. The increased contention does not help to resolve the capacity conflicts for the execution performed on the baseline HTM with two cores. Therefore, performance here decreases. The effect that contention is beneficial for performance is noticeable after the benchmark started on three cores. Then capacity conflicts are resolved beforehand due to other conflicts which is beneficial for performance. Here also the execution cannot be brought above the reference execution.

The second category contains the benchmarks where the HTM performs significantly better than the POSIX Thread synchronization. This category contains the benchmarks *genome*, *ssca2* and *vacation*. The speedup graphs of these benchmarks have a white background in Figure 4.5. For these benchmarks optimistic synchronization is beneficial leading to a higher speedup compared to the POSIX Thread synchronization. Since the critical sections seem to have only a few dependencies, the benchmarks allow multiple transactions to run in parallel. The POSIX Thread synchronization suffers from being serialized, since the critical section is accessed exclusively by only one thread.

For the benchmark *genome* performance continues to rise until eight cores. With 16 cores contention significantly increases and therefore performance decreases. Performance even drops below the speedup achieved on eight cores.

The benchmark *ssca2* also suffers from increasing contention when executed on the baseline HTM with higher core counts. Therefore, performance does not scale and only marginally increases for higher core counts. In general speedup values are good and the POSIX Thread synchronization is clearly outperformed.

When executing the benchmark *vacation* the baseline HTM achieves good speedup values. It is the only benchmark in this category for which the performance can be significantly improved when executed by the baseline HTM using 16 cores.

The third category contains the benchmarks where the executions more or less match or could slightly be improved by the baseline HTM. The benchmarks *bayes*, *intruder* and *kmeans* belong to this category. The speedup graphs of these benchmarks have a light grey background in Figure 4.5.

For the benchmark *bayes* all potential benefits are crushed by high contention and capacity conflicts. Therefore, the execution of both synchronization methods are very close to the reference execution.

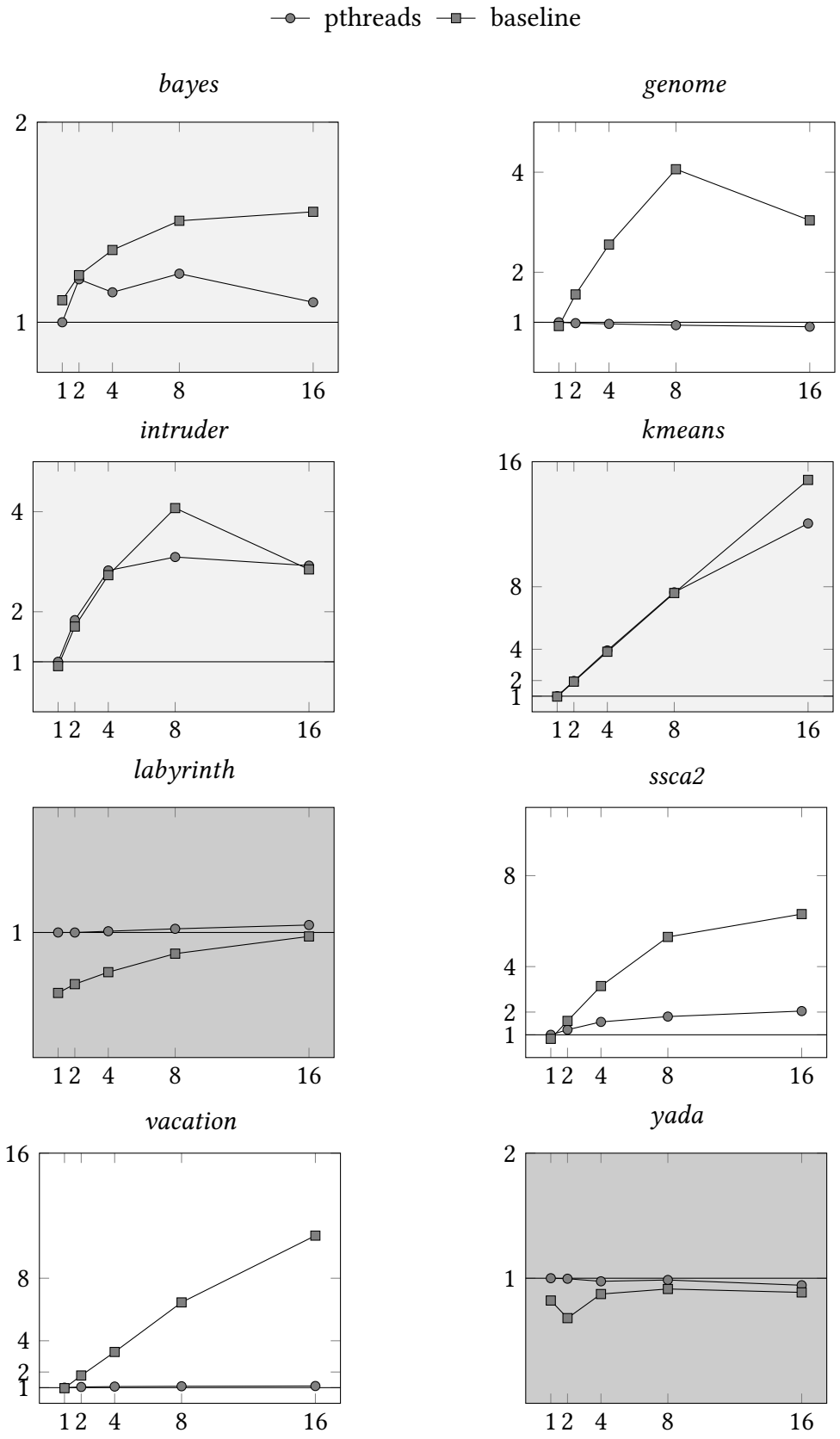

Figure 4.5: Comparison of the performance values achieved with POSIX Thread synchronization and the baseline HTM when executing the STAMP benchmark suite. The y-axes refer to the speedup values compared to the reference execution (one core, no synchronization). The x-axes refer to the number of cores the execution was performed with.

The benchmark *intruder* has small gaps between the critical sections. The POSIX synchronization can exploit this and therefore achieves a speedup. The transactional execution suffers from an overall higher contention which is why the benchmark *intruder* generates low speedups.

The benchmark *kmeans* performs very well and nearly reaches perfect speedup (x16) for the transactional execution. This is because the benchmark has almost no conflicts compared to the number of transactions started. The reason for this are gaps between the critical sections. This means that after every transaction, code is executed which is not critical. This is also beneficial for the execution with POSIX synchronization, because this execution model does not cause serialization. Therefore, speedups are almost as good as for the execution with the baseline HTM, since small delays due to blocking of the mutexes have almost no effect on the speedups. Only when contention increases at a high core count, serialization becomes an issue as shown by a lower speedup for the execution using POSIX Thread synchronization with 16 cores in Figure 4.5.

4.7 Summary

This chapter describes the implementation of an HTM into a baseline system, provided by the gem5 simulator. The chapter starts by laying out the baseline system. Many parts of the baseline system, offered by the gem5 simulator, were reused with small adaptations. Because cache coherence could not be utilized but is a central functionality to implement an HTM, it was redesigned. To implement MOSI cache coherence the cache controllers had to be reimplemented. Also new message types had to be integrated to provide the necessary communication. After the implementation of the cache coherence the baseline system was completed. On top of the baseline system the HTM integration started. Beforehand, some design choices had to be made concerning conflict detection, conflict resolution and versioning. To implement the HTM the cache controllers, the caches, the directory and the communication had to be adjusted. For usage of the HTM an interface is offered. The interface is provided in software, so no changes to the cores had to be made. At the end of the chapter, the HTM was evaluated and compared to an execution with POSIX synchronization. The HTM performed as expected and is therefore used as foundation and baseline of the work described in Chapter 5.

5

Hardware Transactional Memory for Embedded Systems

Contents

Developing an HTM for embedded systems implies better control over the transactional execution. Therefore, we propose an approach which allows a better and more extensive management of the transactional execution. To achieve this we extended the HTM we developed in Chapter 4 to gain further functionality. The main changes concern the cache controllers, since they have to manage all necessary information. To give a holistic explanation of our work, a more detailed description follows. First, we motivate our work by describing common problems of COTS HTMs. Then we will describe the technical details of how we adapted the baseline HTM. After highlighting the changes to the interface, we introduce a technique to reduce aborts. To provide better control we

then explain which contention management strategies we provide and how we implement unbounded transactions. Since estimating executions times is important for embedded systems, we show how calculations for estimating the worst case execution time (WCET) can be simplified for our approach directly after. Before the extended HTM and its techniques are evaluated and analyzed in detail, we present a technique to reduce false conflicts. We close this chapter by providing a summary.

5.1 Motivation

COTS HTMs work very well, if the code they are executing is highly parallel. In that case no conflicts occur, and optimistic synchronization shows its full potential. Unfortunately, real world software often lacks parallelism and suffers from poor programming. The evaluation of the baseline HTM which is modeled to resemble a COTS HTM shows exactly that behavior. The scenarios in which COTS HTMs do not perform well are explained in the following paragraphs.

To handle conflicts COTS HTMs usually apply static rules. A wide spread, because easy to implement, conflict resolution technique is to abort the transaction, mapped to the hardware detecting the conflict. This means that a transaction running on core 1 is aborted if the L1 cache controller of core 1 detects a conflict. Since following static rules as described could leave the execution with an arbitrary abort sequence, this is problematic. An arbitrary abort sequence could lead to a live lock. In a live lock no transaction is able to commit, because the transactions continuously abort each other before being able to commit. Figure 5.1 depicts such a scenario, in which the entire system is blocked, and no progress is made. Another consequence of this behavior is that a lot of work is discarded, resulting in a high amount of wasted cycles. Wasted cycles occur, when instructions performed by a core do not benefit the execution in any way. Instead, the instructions cost energy and computation time, which go to waste.

If a situation depicted by Figure 5.1 occurs in a system utilizing an HTM, it would not be able to complete execution. In such a case a fallback strategy has to be provided. Most commercial HTMs like Intel TSX [16] require the provision of of a fallback execution path. Usually, the fallback path is an replica of the critical section of code using different synchronization mechanisms to ensure the correct parallel execution. A transaction is executed in fallback mode once it exceeds a user defined value of failed attempts to commit a transaction. This adds complexity to the programming when a providing code for an HTM.

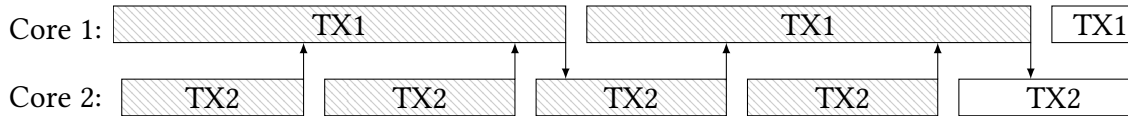

Figure 5.1: Core 1 and core 2 try to execute a transaction. Transaction 1 (TX1) runs on core 1 and transaction 2 (TX2) is executed on core 2. For this execution we assume that the HTM aborts the transaction which detects a conflict. A conflict is depicted by an arrow going from TX1 to TX2 or vice versa. If the arrow goes from TX2 to TX1 this means that the hardware TX2 is mapped to, detects a conflict. Because of the static conflict resolution strategy, it has to abort and restart its execution. Neither transaction 1 nor transaction 2 are able to commit, because throughout the execution repeatably aborted each other. The reason for this is the alternation of which transaction detects a conflict and the lack of prioritization.

A widely spread method to create the fallback path is to copy the critical section of code and to protect it by locking it with a POSIX mutex. This can be done by replacing the begin statement of a transaction by lock(mutex) and the end statement of a transaction with unlock(mutex). This often results in coarse grained locking and serialization but allows the HTM to recover itself from a situation depicted by Figure 5.1. Figure 5.2 shows how the livelock can be resolved by executing the fallback path. Note that during the execution of the fallback path no other thread can enter the critical section which highly prohibits parallelism compared to the transactional execution.

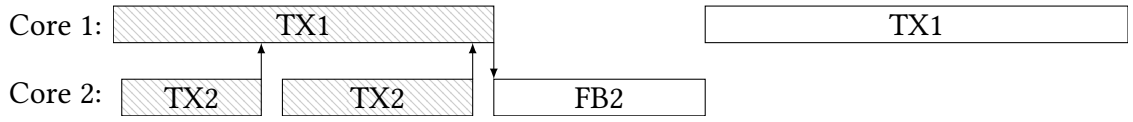

Figure 5.2: When transaction 2 (TX2) reaches its limit of attempts to commit it is executed in fallback mode. After the fallback execution (FB2) finished, Transaction 1 (TX1) is re-executed and can try again to commit. This mechanism can lead to serialization, preventing parallelism, since FB2 has to be executed exclusively preventing transaction 1 to be executed. Since the original scenario (see Figure 5.1) prevented any progress, it still makes sense to utilize the fallback path.

Static rules for conflict resolution can also result in an abort distribution which can be bad for performance. This is the case if a transaction is continuously disadvantaged and is not able to commit due to conflicts with other transactions. Note that even more sophisticated contention management can suffer from this problem. Figure 5.3 shows such an exemplary execution in more detail.

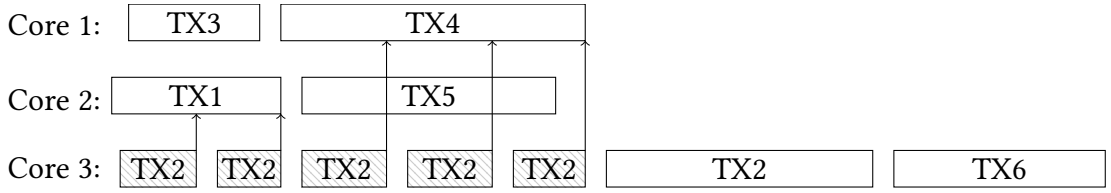

Figure 5.3: Cores 1 and 2 are able to run their transactions without any conflicts, transaction 2 (TX2) is aborted multiple times, due to conflicts with transaction 1 (TX1) and transaction 4 (TX4). In this example every abort is executed on Core 3, resulting in a high delay for Core 3 until it can finish the execution of transaction 2 (TX2).

Transactional conflicts are not the only reason for transactions to be executed in the fallback mode. Another limiting factor is the size and associativity of the L1 cache. If a transaction gets too big for the L1 cache and a cache line, which is part of its RS or WS is evicted, the transaction also has to be aborted. The reason for this is that a transaction saves speculative data in the L1 cache, which cannot be written back to a higher cache hierarchy. If a transaction would be able to write back speculative data, it would be visible to other cores and could be accessed. This hurts the isolation and atomicity criteria of transactions (see Section 2.1 for more details) and would lead to a faulty execution. A similar issue concerns interrupts. If an interrupt occurs within a transaction, it also has to be aborted. The reason for this is that the speculative data, saved in the cache, could be accessed or evicted by the code executed by the interrupt service routine. In the worst case scenario, the interrupt service routine would consume and manipulate speculative data from the transaction. This would most likely end in a faulty interrupt execution and hurts the isolation and atomicity criteria of the transaction.

With the extended HTM, we want to provide an HTM for embedded systems which is able to apply a variety of contention management strategies. Through them the extended HTM is able to guarantee progress, fairness or other features. Additionally, we want to enable unbounded transactions to be able to survive capacity conflicts without having to utilize the fallback path. In combination the necessity for providing a fallback path should be omitted which would greatly simplify programming. Furthermore, we want to try to minimize aborts to reduce the number of wasted cycles. Compared to the HTM presented in Chapter 4 the extended HTM should offer the following improvements:

- support of a variety of contention management strategies
- support of unbounded transactions
- no necessity for providing a fallback path
- mechanisms to reduce the number of aborts

5.2 Adapting the baseline Hardware Transactional Memory

For the extended HTM the same baseline multi-core system of up to 16 cores as described and depicted in Section 4.1 is considered. Furthermore, the baseline HTM implemented in Chapter 4 was used as foundation for the work described in this Chapter.

One of our goals is to provide several sophisticated contention management strategies. This requires to accessing the meta data of the conflicting transactions for the decision-making process concerning the abort. In the following we discuss the three most promising ways to do this in terms of feasibility:

1. The transactional meta data, the RSs and the WSs are managed by the L1 cache controllers. Also, the decision-making process is handled by the L1 cache controller.

2. The transactional meta data is managed by the LLC controller. Therefore, the decision-making process for aborts is moved to LLC controller. The RSs and WSs are managed locally by the L1 cache controller.

3. The transactional meta data, the RSs and the WSs are accessible to the LLC controller. The decision-making process concerning aborts is also moved to the LLC controllers.

If the first approach were chosen, the decision-making process, if a transaction has to abort, would be performed by the L1 cache controllers. Since we want to apply more sophisticated conflict resolution policies an L1 cache controller cannot just abort the transaction running on its corresponding core when it detects a conflict. Instead, it has to inform its opponent that a conflict occurred. Additionally, the data upon which the decision-making progress is based on would have to be transferred. Therefore, a lot of communication would be necessary to resolve a conflict. This would most likely cause long blocking times which would have to be taken into account. Although the conflict is directly detected it takes a lot of communication until execution can continue.

The second approach is a hybrid approach. The WSs and RSs would still be managed by the L1 cache controller. The LLC controller would manage the relevant transactional meta data. Since the meta data contains the information where transactions are currently running potential conflicts could be detected by the LLC. The LLC controller would have to re-ensure that it really detected a conflict by sending a message to the L1 cache controller running the conflicting transaction requesting it to check its RS and WS. This also requires a lot of communication which ultimately lengthens the average execution time for transactional read and write requests.

For the third approach all relevant data is accessible to the LLC controller. In contrast to the second approach also the RSs and WSs can be accessed by the LLC controller. Also, the decision-making process concerning transactional aborts is moved to the LLC controller. Since all relevant data has been transferred to the LLC beforehand, abort decisions can be made very fast. Compared to the other approaches this is a big advantage, since they always cause some sort of waiting or blocking. Therefore, this approach was chosen to extend the baseline HTM. To shift the management of the relevant data and the abort-decision process to the LLC controller, the baseline HTM was completely revised.

The remainder of this section is organized as follows: First the changes to the management of the RS and WS are pointed out. Before the changes to conflict detection and resolution are described the additional message types needed are characterized. Then the adaptations made to the cache controllers are explained.

5.2.1 Managing the Read- and Write-Set

Shifting the conflict detection and resolution to the LLC cache controller also implies keeping track of the RS and WS at that level. To detect conflicts and to be able to allow more complex abort decisions double bookkeeping of the RSs and WSs was implemented. This means that the RSs and WSs are managed locally by the L1 cache controllers and globally by the LLC cache controller. Every L1 cache controller keeps track of the RS and WS of the transaction running on the corresponding core. Additionally, the RSs and WSs are also managed globally by the LLC cache controller. The LLC controller keeps track of the RSs and WSs of all transactions running on the system.

The reason why the RSs and WSs also have to be managed locally is that the LLC controller needs to know which cache lines have to be removed from the RS and WS in case of an abort. Meaning the L1 cache controller has to inform the LLC controller which cache lines have to be removed from the WS or RS. Since the LLC controller would have to iterate over the entire LLC whenever a transaction aborts or commits to identify the lines contained in the corresponding transaction, the LLC controller needs that support. Because of the size of the LLC iterating over it would not be reasonable and would take too long. The information for the local RS and WS is maintained as described in Section 4.4.1.

The global RS and WS allows more complex abort decisions, because every in a conflict involved transaction can be directly identified. Therefore, it can also be directly addressed through its corresponding cache controller (e.g., to trigger an abort). The information regarding the RS and WS is stored by the directory. Similar to storing the sharers of a cache line the directory is augmented to manage an RS and WS for a cache

line. For this the LLC controller adds an L1 cache controller as sharer to the RS when a transactional read and to the WS when a transactional write occurs.

5.2.2 Additional Coherence Messages

Since the extended HTM requires more functionality than the baseline HTM new message types are required. To shift the conflict detection and resolution to the LLC controller in total fourteen new message types were added to the baseline HTM system. Table 5.1 lists all of them and gives a brief description of their functionality. The additional message types added in Section 4.4 are not used by the extended HTM.

The LLC controller needs to manage transactional meta data. Therefore, the L1 cache controllers have to inform the LLC controller when a transaction started, wants to commit or in case a transaction aborted when it finished the abort routine. To do this, three new message types were added. The new message type *transactionStart* is used to inform the LLC controller that a transaction started on the corresponding core of the L1 cache controller which sent the message.

In case a transaction aborted the corresponding L1 cache controller indicates the LLC controller that it finished the abort routine by sending the new message type *transactionEnd*. Since cache coherence has to be maintained no request concerning the RS or WS of the aborting transaction can be redirected to the L1 cache controller currently aborting that transaction. Instead, the LLC controller has to detect that the L1 cache controller is currently aborting a transaction, which it can since all aborts are triggered by the LLC controller. Therefore, the L1 cache controller is in a transient state in which it is still running a transaction, but it is marked as aborted. Requests concerning the RS or WS of that transaction have to be serviced directly by the LLC. As soon as the *transactionEnd* message hits the LLC controller it knows that the L1 cache controller completed the abort of the transaction and can set the meta data accordingly.

If a transaction wants to commit the new message type *transactionCommitReq* is utilized to inform the LLC controller. A commit of a transaction cannot occur atomically. This means that at the time an L1 cache controller is informed to commit a transaction the LLC controller can still detect a conflict and send an abort request. Therefore, the L1 cache controller sends the *transactionCommitReq* to inform the LLC controller that it wants to commit. After the LLC controller receives that message it sends a message acknowledging the commit. The message is only sent if meanwhile no conflict was detected causing the LLC controller to abort the transaction. To acknowledge the commit the new message type *commitAck* is sent. Figure 5.4 depicts and describes the problem when no acknowledgment message is implemented. Figure 5.5 shows the same scenario and how it is resolved by implementing a request as well as an acknowledgment message.

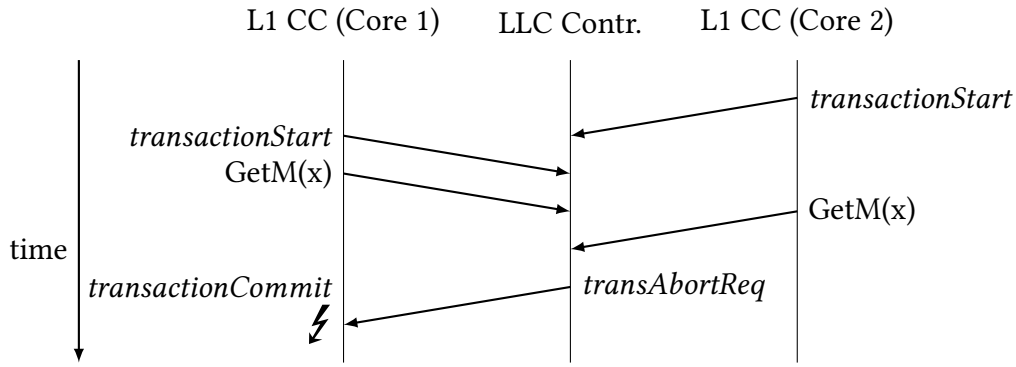

Figure 5.4: Core 1 and core 2 are both running a transaction. Therefore, their corresponding L1 cache controllers (L1 CC (Core 1) and L1 CC (Core 2)) send a *transactionStart* message to the LLC controller (LLC Contr.) to inform it. First the transaction running on core 1 and later the transaction running on core 2 add cache line x to their WS. Again the LLC controller is informed. Therefore, the LLC controller detects a conflict. Meanwhile core 1 commits its transaction and its corresponding L1 cache controller starts clearing the RS and WS. Shortly after an abort request reaches the L1 cache controller. Since the L1 cache controller already removed entries from the RS and WS it cannot abort the transaction running on its corresponding core. Due to the information loss this is not possible anymore. Another option would be to abort the other transaction. This would involve communication, violate the conflict resolution strategy, and could also fail. Therefore, we provide a different mechanism depicted in Figure 5.5.

Since we implemented double bookkeeping of the RS and WS the management of the RS and WS is also done by the LLC controller. Therefore, two new message types had to be implemented. The first new message type is called *addToRS*. It allows the L1 cache controller to inform the LLC controller that the cache line the message refers to was added to the RS of the currently running transaction. The second new message type, named *addToWS*, informs the LLC controller that the cache line the message refers to was added to the WS of the currently running transaction.

To maintain the RS and WS correctly the LLC controller also has to be informed when a cache line is removed from the RS or WS. Otherwise, false conflicts could arise. In contrast to adding cache lines to the RS or WS managed by the LLC controller this is done with three new message types, since a cache line can simultaneously be contained in the RS and the WS. The new message types *removeFromRS* and *removeFromWS* indicate the LLC to remove the L1 cache controller from the RS or WS of the cache line the message refers to. If an L1 cache controller is contained in the RS and WS and has to be removed the new message type *removeFromRSWS* is used.

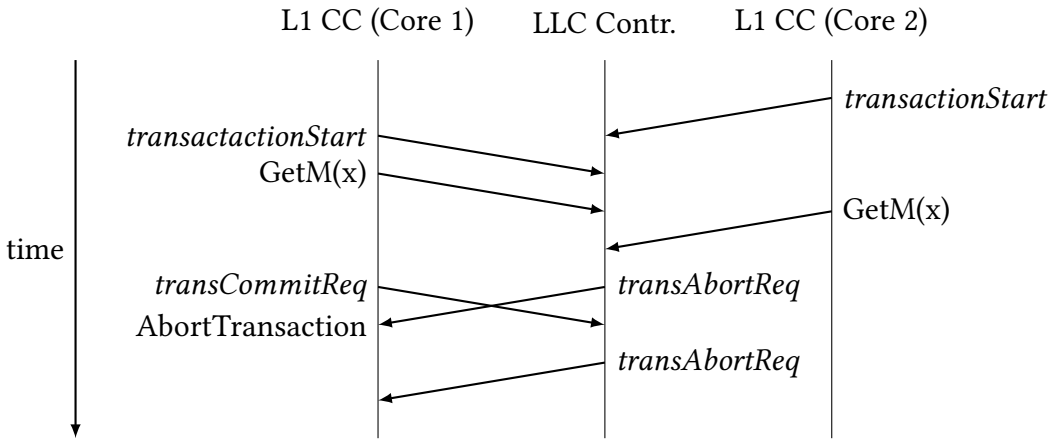

Figure 5.5: This example shows the exact same scenario as depicted in Figure 5.5. The difference is that the transaction running on core 1 will not commit until it receives approval from the LLC. Due to this the abort request (*transAbortReq*) is sent to the L1 cache controller of core 1 (L1 CC (Core 1)) after the LLC detects the conflict caused by cache line x. It can now abort the transaction since it has not started to clear the RS and WS. The commit request (*transCommit*) is denied by sending a second transaction abort request (*transAbortReq*). The L1 cache controller of core 1 will detect that the transaction has already aborted and continues to abort the transaction.

Conflict resolution shifted to the LLC controller. Therefore, we implemented a new message type indicating to the L1 cache controllers that they have to abort the transaction running on their corresponding core. To do this the new message type *transactionAbortReq* was added. This message type is sent, if the LLC controller detects a conflict and one or more L1 cache controllers have to be notified that the transactions running on their corresponding cores have to be aborted.

In the transactional execution it can be necessary that data held by an L1 cache has to be written back (see Section 5.2.5 for more details). For this purpose, we added the new message type *writeBackReq*. In contrast to the *invalidate* message type, provided by the MOSI cache coherence protocol, it can be used for a wider spectrum of states and does not send any acknowledgment messages (see Section 2.2 for more details). The message type is only triggered by the LLC controller and causes the L1 cache controller it was sent to only initiate a write back of the concerning cache line.

To be able to detect all conflicts it can be necessary that a cache line is added to a transactions RS managed by the LLC controller before the cache line is added locally to the RS managed by the L1 cache controller (see Section 5.2.3 for more details). For this purpose we added the new message type *addToRSinDir*. To confirm the cache line was added to the RS managed by the LLC controller an acknowledgment message is triggered by the LLC controller. Here the new message type *addedToRS* is utilized.

In the extended HTM we can allow an abort-aware execution which can be beneficial for energy consumption (see Section 5.4 for more details). For it to work cores are stalled. To wake them up again, we implemented the message type *canContinue*. The message type informs the L1 cache controller to continue execution.

Message type:	**Brief description:**
transactionStart	Indicates the LLC controller that a transaction started on the corresponding core of the L1 cache controller which sent the message.
transactionAbortReq	Indicates the L1 cache controller that its corresponding core has to abort its transaction.
transactionEnd	Indicates the LLC that the abort of the currently running transaction is completed.
transactionCommitReq	Indicates the LLC that the transaction running on the corresponding core of the L1 cache controller which sent the message wants to commit its transaction.
commitAck	Indicates the L1 cache controller that the transaction running on the corresponding core can be committed.
addToRS	Informing the LLC controller to add cache line to RS of currently running transaction.
addToWS	Informing the LLC controller to add cache line to WS of currently running transaction.
removeFromRS	Removes a cache line from the RS managed by the LLC controller.
removeFromWS	Removes a cache line from the WS managed by the LLC controller.
removeFromRSWS	Removes a cache line from the RS and WS managed by the LLC controller.
writeBackReq	Writes back the cache line referred to by the message by evicting the cache line form the L1 cache.
addToRSinDir	Indicates the LLC controller to add a cache line to the RS managed by the LLC controller.
addedToRS	LLC sends this message, if the cache line the *addToRSinDir* request refers to was successfully added to RS managed by LLC
canContinue	Indicates the L1 cache controller that it can continue to service request from its corresponding core.

Table 5.1: New message types for the baseline HTM

5.2.3 Conflict Detection

Just as in the baseline HTM conflicts are detected eager. In contrast to the baseline HTM the conflicts are not detected by the L1 cache controllers. Instead, the LLC controller performs conflict detection. Detecting conflicts at the level of the LLC controller allows more complex abort decisions. To do this, we rely on the implementation of the MOSI cache coherence protocol. Since the MOSI cache coherence protocol can be operated without silent state changes as e.g., performed by the MESI cache coherence protocol, it allows the LLC controller to directly identify conflicts reducing blocking times. Silent state changes are changes of the cache line states, which are performed without notifying the LLC controller. Figure 5.6 shows the for our approach problematic silent state changes performed by the MESI protocol and compares it to the same procedure performed by the MOSI cache coherence protocol. Silent cache changes are problematic, since the LLC controller cannot determine directly if the concerning cache line only has to be added to the RS or also to the WS. Since it is possible that the LLC controller is not informed about a state change of the cache line the LLC controller cannot be certain if it detected a conflict or not. This means that whenever a potential conflict is detected the LLC controller would have to check if a state change of the cache line were performed by the L1 cache controller exclusively holding the cache line. In such a case extra communication would be needed to determine the actual state of the cache line.

A conflict is detected by the LLC controller, if a transactional write request, signaled to the LLC controller by a *getM* request, concerns a cache line which is contained in the RS or WS of another transaction. Also, if a transactional read, identified by a *getS* request, concerns a cache line contained in the WS of another transaction a conflict is detected. Since the directory entry also stores the information to which other RSs or WSs a cache line belongs, this is possible.

If a transaction exceeds the size for the RS or WS, which is still tied to the size and associativity of the L1 cache, the L1 cache controller sends the corresponding put-message (see Section 2.2 for more details) to the LLC controller. The LLC controller then detects if the cache line is evicted from a transactions RS or WS. If the cache line is part of the RS or WS it sends an abort request to the corresponding L1 cache controller.

Conflicts are exclusively detected by the LLC controller. The L1 cache controllers do not detect conflicts. Furthermore, the extended HTM is built in such a way that no conflicting requests will reach the L1 cache controllers.

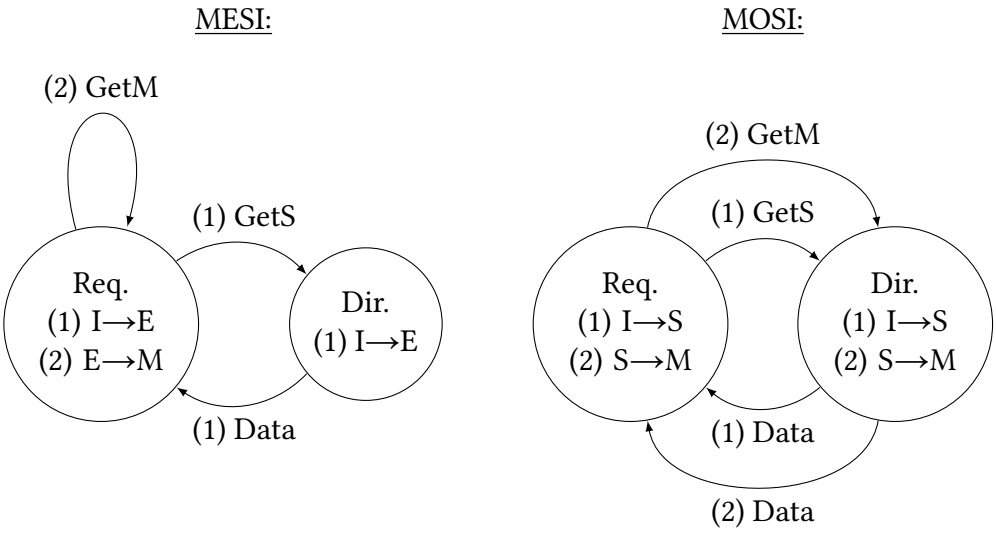

Figure 5.6: The left side of this figure shows the scenario in which a silent evict can occur in the MESI cache coherence protocol. If a cache line is read it first sends a GetS request. When the cache controller is identified to be the only sharer the state of the cache line is set to exclusive. Next the cache is written. Therefore, the cache controller triggers a GetM request. Since the cache line is in exclusive state the request does not have to be sent to the LLC controller. Instead, it is directly changed to modified. The MOSI protocol, depicted on the right side, always sends a request to the LLC controller when it wants to change the state of a cache line. It therefore sends the GetM request to the LLC before changing the state of the cache line. Graphic is inspired by [57, S. 163]

5.2.4 Conflict Resolution

The system we propose allows a wide varity of conflict resolution techniques. Detecting conflicts with the LLC controller brings some advantages when resolving a conflict. When detecting a conflict all conflicting parties can be identified by the LLC controller. In contrast to the baseline HTM the LLC controller can now make a more sophisticated abort decision which can include the consideration of transactional meta data (e.g., transaction start time, priorities, number of commits, etc.). We propose several contention management strategies described in more detail by Section 5.6.

5.2.5 Modifying the Cache Controllers

Shifting the conflict detection and resolution to the LLC controllers requires further adaptations to the cache controllers. Although a lot of functionality could be adopted

from the baseline HTM some major changes had to be made which are described in the following paragraphs.

Starting a transaction is performed as implemented in the baseline HTM (see Section 4.4.3 for more details). The only change made is that the L1 cache controller now sends a message to the LLC controller. By sending a message with the new message type *transactionStart* the L1 cache controller informs the LLC controller that its corresponding core started a transaction. The LLC controller stores this information and uses it for conflict detection.

Since the information concerning the transactional execution cannot be lost, the LLC controller has to be able to save and alternate the transactional meta data. Therefore, we augmented the LLC controller to be able to save the transactional meta data in a transaction config data block comparable to the way data is saved by the L1 cache controllers (see Section 4.4.3 for more details). In contrast to the information stored in the L1 cache controllers, which only relates to one transaction, the data managed by LLC controller relates to all transactions running. Since one core can run one transaction at the time the space needed varies depending on how many cores the system offers. All data saved in this block relates to the transactional execution. The data saved heavily relies on the contention management strategy applied and is listed in Section 5.6.

Although many things had to be adjusted for the commit and abort routine, the baseline functionalities were adopted (e.g., clearing the local RS and WS) from the implementation of the baseline HTM (see Section 4.4.3 for more details). In the following we will strongly focus on the differences. Before the L1 cache controller can start to commit the transaction running on its corresponding core, it has to send a commit request to the LLC by sending the new message type *transactionCommitReq*. If meanwhile no conflict occurred, the LLC controller sends a *commitAck* message to the corresponding L1 cache controller. The LLC controller also marks that currently no transaction is running on the corresponding core of the L1 cache controller which sent the message. This is important for conflict detection. Note that the L1 cache controller can still be contained by the RS or WS of several cache lines. Since the LLC controller marked that no transaction is currently running requests concerning these lines can be redirected to the L1 cache controller and no conflict will be detected.

After the L1 cache controller receives the *commitAck* message it can start to execute the commit routine. The reason why the commit request has to be acknowledged is that once the commit routine started it cannot be stopped. It has to look like it was performed atomically. Therefore, the LLC controller has to be informed beforehand so that subsequent request will not cause an abort request (see Section 5.2.2 for more details). Within the commit routine the L1 cache controller starts to clear the RS and WS (see Section 4.4.3 for more details). When removing a cache line from the RS or WS it sends one of the new message types *romoveFromRS*, *removeFromWS* or *removeFromRSWS* (see Section 5.2.2 for more details)) to the LLC controller. When receiving such a message the LLC

controller removes the L1 cache controller from the RS, WS or both. If a transaction commits only the RS and WS has to be cleared. Cache line states remain untouched. Note that because of the strict ordering of the messages it is guaranteed that the RS and WS is cleared before a new transaction starts.

In contrast to the baseline HTM a transaction abort is never initiated by the L1 cache controllers. The request to abort a transaction always comes from the LLC controller. Once the L1 cache controller receives a *transactionAbortReq* the L1 cache controller will no longer serve read or write request of its corresponding core. When all messages in flight are serviced the L1 cache controller triggers an interrupt (see Section 4.4.3 for more details). Just like when committing the L1 cache controller starts to clear the RS and WS. The abort routine works like the one described in Section 4.4.3. Here also the states of the cache lines are reset. In contrast to the implementation of the baseline HTM the cache controllers of the extended HTM also straightens out the RS and WS managed by the LLC controller. For this it sends the new message types *removeFromRS*, *removeFromWS* or *removeFromRSWS*. If the coherence state has not already changed the messages are also used to straighten out the state of the cache line stored by the directory entry. Table 5.2 shows how the states are set back.

State saved by directory:	**Is set to:**
shared	*invalid or remains shared*
modified	*invalid*

Table 5.2: Restoring coherence states after a transaction abort

Cache states not only have to be straightened out in the directory, it can also be necessary to adapt them in the L1 cache before the abort routine is executed. Note that within the abort routine cache states are set back as decribed by Section 4.4.3. When an L1 cache controller tries to add a cache line to the RS or WS it sends the according messages to the LLC controller. If the attempt fails because a conflict was detected, and the conflict is resolved in favor of an opponent transaction, measures have to be taken by the L1 cache controller to straighten out the state of the affected cache lines. To explain this in more detail we give the following example: If a cache line is marked as invalid and should be added to the RS of a transaction the corresponding L1 cache controller triggers a *getS* request to the LLC controller. Meanwhile, the state of the cache line is set to *isd* (see Section 2.2 for more details). When the request reaches the LLC controller it checks if it can service the request. In this example we assume that the cache line is part of the WS of another transaction and a conflict is detected causing the transaction trying to add the cache line to its RS to abort. The LLC then sends an *transactionAbortReq* message to the L1 cache controller which sent the *getS* request. Since the *transactionAbortReq* message does not only indicate the L1 cache controller to initiate an abort but also contains the address of the cache line for which the request got sent, the L1 cache controller is able to restore the coherence state. This is necessary since the intermediate state of the cache

line is likely to cause a deadlock if not set back. The cache line set to *isd* is set back to invalid. All possible intermediate states and to which state they are reset are listed by Table 5.3.

Intermediate State:	**Is set to:**
isd	*invalid*
imad	*invalid*
smad	*shared*

Table 5.3: Restoring coherence states in case of rejected requests

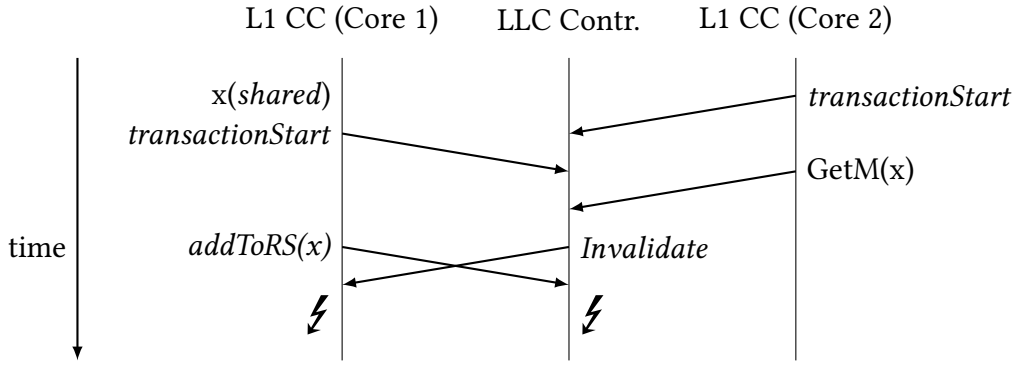

Figure 5.7: In this example cache line x is contained in the L1 cache of core 1. The state of the cache line is shared. Core 1 then starts a transaction. Therefore, the L1 cache controller of core 1 (L1 CC (Core 1)) sends a *transactionStart* message to the LLC controller. Meanwhile core 2 also starts a transaction. The L1 cache controller of core 2 (L1 CC (Core 2)) therefore also sends a *transactionStart* message to the LLC controller. The transaction running on core 2 tries to write cache line x and sends a *getM* request to the LLC controller. Since the LLC controller lacks the information that the transaction running on core 1 added cache line x to its write set no conflict is detected. Furthermore the LLC sends an invalidate request to the corresponding L1 cache controller concerning cache line x. As soon as the *invalidate* request hits the L1 cache controller of core 1 it detects the conflicting access. Meanwhile the message informing the LLC controller that the transaction running on core 1 added cache line x to its RS reached the LLC controller. To resolve this conflict to act as demanded by an applied contention management strategy is now very hard, since a lot of communication would be necessary.

Adding a cache line to the WS or RS works completely different as in the baseline HTM. In the baseline HTM the cache lines are directly added to the RS or WS. The only criteria which has to be fulfilled is that a transaction is currently running on the corresponding core of the L1 cache controller adding a cache line to the RS or WS. In the extended HTM

cache lines which are not yet part of an RS or WS in general have to be treated more carefully, since they have to be added to RS or WS managed by the LLC controller before they can be added to the local RS or WS.

Since conflict detection shifted to the LLC controller only it should initiate aborts. Otherwise, it is not possible to enforce conflict resolution policies. Therefore, a cache contained in the L1 cache in shared state cannot just be added to the local RS. If added without further measures an erroneous state can arise. In some cases, conflicts cannot be detected as intended and might slip which can lead to a faulty execution. Figure 5.7 shows one problematic scenario if no precautions are taken.

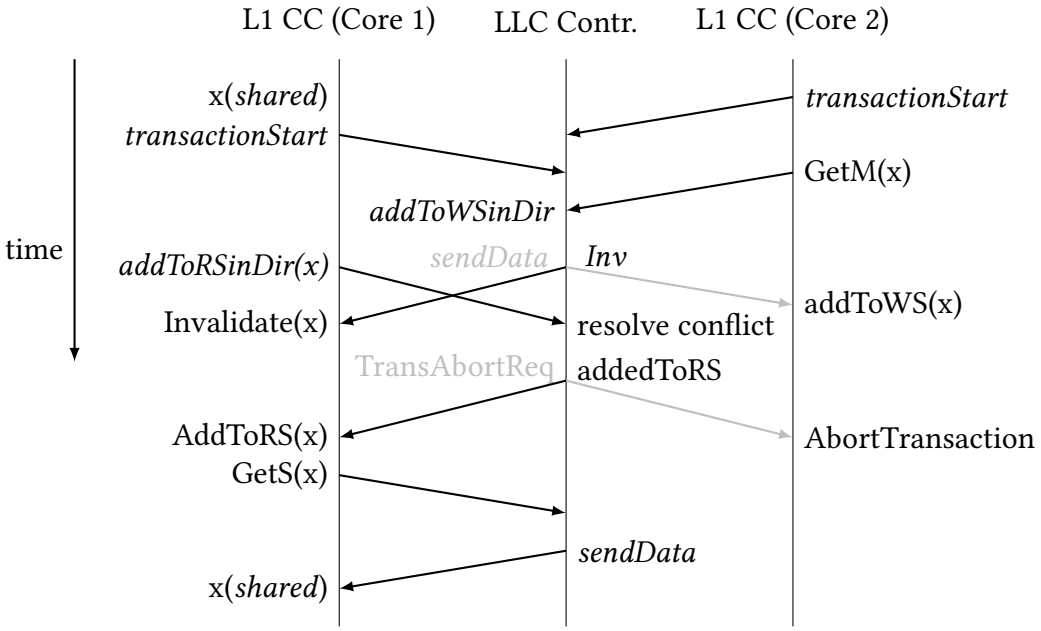

Figure 5.8: In contrast to Figure 5.7 cache line x is added to the RS managed by the LLC controller before it is added to the local RS managed by the L1 cache controller. Now the LLC controller is able to detect the conflict directly once the *addToRSinDir* message, triggered by L1 cache controller of core 1 (L1 CC (Core 1)), hits the LLC controller. This brings the advantage that the conflict can be resolved according to the applied conflict management strategy. The drawback of adding the cache line to the local RS this way is that although cache line x is present the execution is blocked until the L1 cache controller of core 1 (L1 CC (Core 1)) receives the *addedToRS* message from the LLC controller. Additionally, cache line x has to be reread since it was evicted due to the write request of core 2. Note that the cache line only has to be added to the RS managed by the LLC controller once. A second time is not necessary, since the cache entry although the state is set to invalid still remains and can be added to the local RS.

To solve the issue depicted by Figure 5.7 we implemented that a cache line always has to be added to the RS managed by the LLC controller before it is added to the RS managed by the L1 cache controller. Figure 5.8 shows exemplary how this is done. Note that this procedure only has to be performed if the cache line is not yet part of the RS or WS.

Cache lines which are in modified state but not yet part of the RS or the WS have to also be treated specially. For these cache lines the L1 cache controller has to initiate a write back of the data, since the most recent data has to be written back to the LLC. Otherwise, data loss might occur in case the transaction aborts. In contrast to the implementation of the baseline HTM the write back of a cache line and adding it to the RS or WS cannot be initialized simultaneously. Instead, the data first has to be written back. Then it can be added to the WS or RS. For this no new message type is required, since it can be written back with a *putM* message provided by the MOSI cache coherence protocol. The subsequent read or write is also performed by message types offered by the MOSI cache coherence protocol.

Another situation which has to be considered occurs when an L1 cache controller sends a request to the LLC controller indicating that it wants to add a cache line to its RS or WS. If the LLC controller detects that the cache line is in modified state but not part of an RS or WS it sends a message with the new message type *writeBackReq* to the L1 cache controller registered as owner. This will force the L1 cache controller to initiate a write back of the concerning cache line.

If the measures described in the previous two paragraphs are not applied a situation could arise which would lead the system into an undefined and possibly erroneous state. A detailed description of what goes wrong in such a situation is given by Figure 5.9. Note that especially for higher core counts it is important to be able to restore a system state which is correct and follows the applied conflict resolution policy.

Figure 5.10 shows the same initial situation as Figure 5.9. Here the described measures are applied and no erroneous state is reached. The conflict can be detected and resolved according to an applied conflict resolution policy.

In contrast to the baseline HTM some functionality had to be added to provide cache coherence. For this purpose, we exploit the behaviour of the the MOSI cache coherence protocol in several ways:

If the system detects a conflict and the cache line due to which the conflict arose is in modified state, this is critical. Especially if the transaction running on the corresponding core of the L1 cache controller registered as owner is identified to be aborted. Since no blocking should occur, the state of the cache line has to be adapted immediately. Therefore, the state of the cache line is set to invalid, and the owner of the cache line is cleared. This ensures that the correct data is sent to the continued transaction and causes the LLC controller to send the data stored in the LLC. The state is then also changed

accordingly and if needed a new owner of the cache line can be set. Note that this works for read or write accesses. This mechanism is especially important for non-transactional accesses. Non-transactional accesses always win over transactional accesses causing a conflicting transaction to abort.

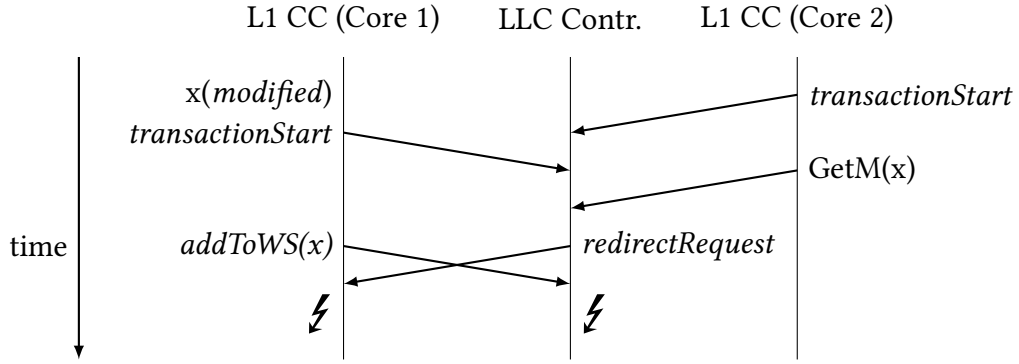

Figure 5.9: In this example the cache line x is contained in the L1 cache of core 1. The state of the cache line is modified. Core 2 starts a transaction, which is why its corresponding L1 cache controller (L1 CC (Core 2)) informs the LLC controller (LLC Contr.) by sending a *transactionStart* message. Also core 1 starts a transaction. Therefore, its L1 cache controller (L1 CC (Core1)) also sends a *transactionStart* message to the LLC controller. As depicted the transaction running on core 2 wants to add the cache line x to its WS by sending a getM request. Because the cache line is not part of any other RS or WS no conflict is detected, and the request is redirected to the L1 cache controller of core 1. Meanwhile, the transaction on core 1 also added cache line x to its WS and triggered an backup of the cache line. When the L1 cache controller of core 1 receives the redirect request it cannot forward the cache line since it is in a speculative state. As soon as the LLC controller receives the message that cache line x was added to the WS of the transaction running on core 1 a conflict is detected. Since the ownership of the cache line changed because of the request of the L1 cache controller of core 2 it can be problematic to restore correct ownership.

If a transaction has to abort the LLC controller marks it accordingly. Since the transaction cannot empty its RS or WS atomically it has to be continued in an intermediate state. If a read or write request concerning a cache line owned by an L1 cache controller which corresponding core is currently aborting its transaction the LLC controller detects that and sets the state to invalid and removes the owner. Otherwise, a request could be redirected to that L1 cache controller although it would not be able to service it. Note that read requests to a shared cache line is no problem because a cache line can be read by multiple transactions. A write request to a shared cache line will cause a conflict but requires no changes to the state of the cache line.

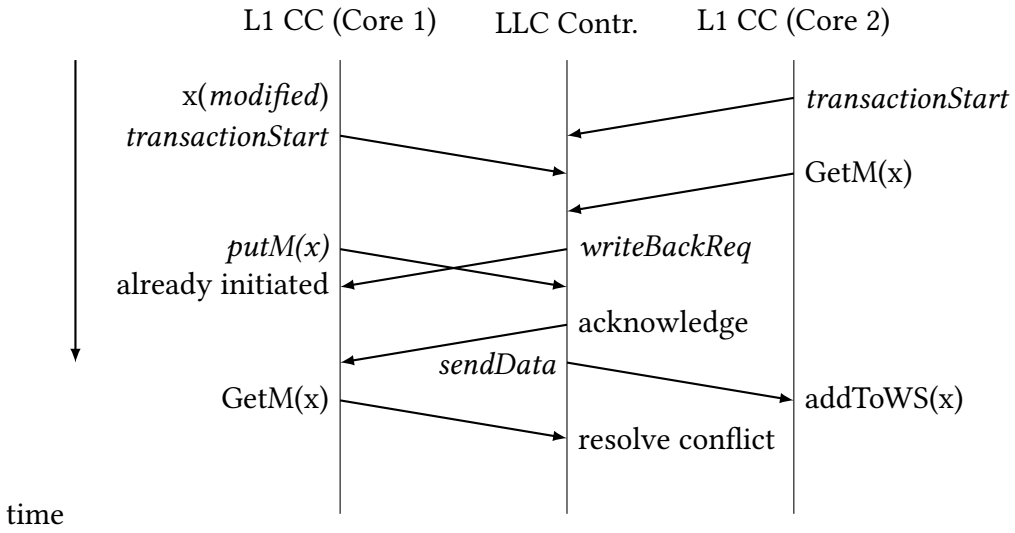

Figure 5.10: In contrast to Figure 5.9 cache line x is not added to the WS of the transaction running on core 1. Instead, the L1 cache controller of core 1 (L1 CC Contr.) initiates a write back of cache line x. Another difference is that the request of the L1 cache controller of core 2 (L2 CC Contr.) trying to add cache line x to the WS of the transaction running on core 2, does not cause a redirect message. Instead, a *writeBackReq* is triggered (see Section 5.2.2 for more Details). When the L1 cache controller of core 1 receives that request it can be ignored since the intermediate state (ima) indicates that the data is already written back. The L1 cache controller of core 1 sends a *getM* requests for cache line x to the LLC after receiving the acknowledgment which indicates that the write back of cache line x is completed. This causes a conflict which can be resolved without having to send redirect messages as it would be necessary in the example of Figure 5.9. Since the most recent data is now saved by the LLC either the transaction running on core 1 or core 2 can be aborted. Depending on which transaction aborts, the ownership can remain or has to be changed.

5.3 Interface

The interface basically remains as described in the baseline HTM (see Section 4.5 for more details). We added the functionality that a potential user can set a priority for every transaction which is started. For this purpose, the interface was changed. Starting a transaction now allows setting a priority. Therefore, the transaction start method was adapted so the user can pass a priority. The priority is then written to address 0xf0000014.

5.4 Abort-Aware Transactional Execution

Since detecting and resolving conflicts is managed by the LLC controller an optimization concerning the transactional execution can be applied. The optimization yields to improve energy consumption and lower abort rates. When a transaction conflicts with another transaction, the transaction which has to abort is very likely to again conflict with the transaction which continued. This does not affect the performance but certainly affects energy consumption since the core executes a transaction repeatedly without committing it. Since we are developing an HTM for embedded systems it makes sense to consider energy consumption. Therefore, we propose a mechanism with which a transaction is only executed if there is a chance that it will commit.

When a transaction aborts because of a conflict the transaction will not be restarted after the abort handler was executed. Instead, the L1 cache controller will no longer service requests from its corresponding core until it is signaled to let execution continue. This can be applied because the L1 cache controller exactly knows when to block subsequent requests. The *transactionEnd* message sent to the LLC controller once the abort handler finished indicates that moment. After this message, no requests of the corresponding core are handled, and it can be put into power saving mode. The execution continues when the L1 cache controller is signaled to continue by the LLC controller.

For this the LLC controller has to manage a stall list for every core provided by the system. A core (core 1) is added to the stall list of another core (core 2) when the transaction hosted by core 1 is aborted. If the transaction running on core 2 aborts or commits, the LLC controller sends a *canContinue* message to all corresponding L1 cache controllers of the cores saved in the stall list of the core aborting its transaction.

Once a stalled L1 cache controller is notified it will continue servicing the requests from the corresponding core.

5.5 Unbounded Transactions

In conventional COTS HTM, after which our baseline HTM is modeled, a transaction has to abort if the size of the RS or WS exceeds the size or associativity of the corresponding L1 cache. Since the information if a cache line is contained in the RS or WS is missing if it was evicted, a system like our baseline HTM cannot assure to detect all conflicts. This is problematic, since other threads could then access speculatively written cache lines which can lead to erroneous behavior. To avoid transaction aborts we implemented a form of unbounded transactions [3] to the extended HTM. This technique

allows transactions to survive evicts of cache lines from the L1 cache contained in the RS or WS.

One difficulty of implementing unbounded transactions is keeping track of the RS and WS. To be more precise the problem is in maintaining the RS and WS after an unbounded transaction is completed. Starting an unbounded transaction implies that a cache line contained in the RS or WS of a transaction was evicted by the L1 cache controller hosting the transaction. Therefore, the information that was contained in an RS or WS gets lost and the LLC controller will not receive a message to remove that cache line from the RS or WS managed by the LLC when the transaction commits. Since the LLC controller would have to iterated over every cache line contained in the LLC, the RS and WS cannot be cleared by the LLC controller because this would be too costly concerning time. To keep the RS and WS up to date the cache line also has to be removed from the RS or WS managed by the LLC controller as soon as the corresponding put request hits the LLC. Otherwise, the cache line remains in the RS or WS. This would be problematic, since the information will be used for future transactions which would cause false conflicts. Fortunately, the extended HTM allows a novel approach towards unbounded transactions, which ensures the correct continuation of the transactional execution.

In our execution model only one unbounded transaction at the time is allowed. Since unbounded transactions use the entire memory hierarchy speculative values are also written to the L2 cache. In consequence no versioning for the evicted cache lines exists which is why unbounded transactions cannot be rolled back. An unbounded transaction always has the highest priority, and every conflict is resolved in favor of the unbounded transaction. When an unbounded transaction is executed, no interrupts can occur for the core running the unbounded transaction. Therefore, preemption has to be deactivated for the core during the executing of the unbounded transaction.

An unbounded transaction is started when a cache line is evicted from the L1 cache contained in the RS or WS of the transaction running on the corresponding core. The LLC controller detects that a cache line is removed from the RS or WS and starts an unbounded transaction. Additionally, the requests of all other L1 cache controllers are not serviced for the time of the unbounded transaction. As soon as the unbounded transaction commits the LLC continues to service the requests of the other L1 cache controllers. Note that conflict detection is active for the other transaction and that they can be aborted in case a conflict is detected. Furthermore, the other transactions continue execution. As long as they work on data contained in the L1 cache execution continues. If a miss occurs the transaction can continue once the unbounded transaction is committed.

Theoretically the extended HTM would also provide a more parallel approach for the unbounded transactions. This means that the LLC would not necessarily have to quit servicing requests for the other L1 cache controllers. Note that this approach exceeds the scope of this thesis and is considered future work. Since conflicts for cache lines

contained in the RS or WS managed by the L1 cache controller could still be detected, conflict detection works for all transactions running in regular mode. To maintain conflict detection with an unbounded transaction, false conflicts have to be taken into account. Here the problematic case is if a regular transaction tries to add a cache line to its RS or WS which has already been read or written by the unbounded transaction. Since the LLC controller cannot rely on the RS and WS for conflict detection with an unbounded transaction conflict detection is pessimistic. Therefore, a potential conflict is detected when a regular transaction tries to add a cache line to its RS or WS for which the L1 cache controller, hosting the unbounded transaction, is not registered as owner or sharer. Only *modified* cache lines owned by a different L1 cache controller do not cause a conflict even though the L1 cache controller hosting the unbounded transaction, is not registered as owner or sharer. Note that conflicting accesses are not resolved by instantly aborting the regular transaction. The LLC controller would stop servicing the requests of the L1 cache controller hosting the potentially conflicting transaction. This allows us to obtain the progress already made. Also accesses from non-transactional threads can be problematic. If a potential conflict is detected with a non-transactional access the LLC immediately stops servicing requests of the corresponding L1 cache controller until the unbounded transaction committed.

Another problem to consider is that if a contention management strategy is applied which does not guarantee progress and therefore relies on the possibility to execute transactions in fallback mode, an alternative fallback execution has to be offered. In a regular transactional execution, the fallback execution of a transaction aborts all conflicting transactions. Since mutexes, used to protect critical sections, use a completely different underlying mechanism to provide mutual exclusion, a user has to assure that every transaction, operating in the critical section a mutex is protecting, is aborted as soon as the mutex is locked. Otherwise, the thread locking the mutex would not have exclusive access to the critical section. This is done by adding the mutex to the RS of a transaction. As soon as a transaction is executed in fallback mode the mutex is set. The transaction then detects a conflict with a non-transactional execution and aborts. When running unbounded transactions this cannot happen, since an unbounded transaction cannot be rolled back. Therefore, we provide a fallback mode which allows the fallback execution only when all transactions finished the current run of their execution.

5.6 Contention Management Strategies

In the following we describe which contention management strategies were implemented to the system. Note that this is only a selection. Our system allows to implement a wide variety of strategies. Apart from the first strategy *passive* we chose to implement policies which allow to achieve one of the following goals:

1. guarantees progress
2. guarantees fairness
3. offers good performance (best possible speedup)

The first strategy we implemented is named ***passive*** [29, S. 51]. It was implemented mainly for debugging purposes since it resembles the conflict resolution policy of the baseline HTM. Although ***passive*** also aborts the transaction which detects the conflict just like the contention management strategy implemented in the baseline HTM the two do not behave the same way. In the baseline HTM two transactions can abort simultaneously if two conflicting accesses are performed at the same time. This is not possible in the extended HTM, since one conflict will occur before the other. Once that conflict is resolved the other conflicting request will either be serviced or be ignored since it belongs to an aborted transaction.

An effective way to guarantee progress and to prevent live locks (see 5.1) is to implement a conflict resolving mechanism, which automatically prioritizes a certain transaction preventing it from aborting. Depending on the mechanism meta data has to be evaluated, when a conflict occurs. In the following we will explain which contention management strategies we implemented.

To guarantee progress, we first implemented a simple but effective strategy named ***timestamp*** [53]. This strategy allows the transaction, which started earliest, to continue in case a conflict with another transaction arises. To implement the strategy a timestamp is saved for when the transaction started. The timestamp is saved in the transaction configuration block managed by the LLC controller (see Section 5.2.5 for more details). In case of a conflict the LLC controller resolves the conflict by comparing the start times of the conflicting transactions. The transaction which started first is allowed to continue. The other transaction has to be aborted. Note that this strategy also works if the transaction conflicts with more than one other transaction. If the transaction causing the conflict to be resolved has the oldest timestamp the other transactions have to abort. If one of the other transactions has an older timestamp, the transaction causing the conflict to be resolved is sent an abort request.

Our target systems are embedded systems. Therefore, we offer a management strategy which allows the prioritization of transactions which is named ***priority***. The priority for every transaction can be set manually. If no priority is set manually the priority is set to the core Id, it is running on. The higher the core id the higher the priority meaning that a transaction running on core 2 has a higher priority as a transaction running on core 1. For the strategy to work the LLC saves a priority for every transaction running. If a conflict arises it is used to resolve the conflict as described.

Since we are not able to completely avoid aborts, because of real data dependencies, another goal of our work is to fairly distribute the aborts among the participating cores. Figure 5.3 from the motivation of the chapter shows an execution which guarantees progress, since it behaves as if the *timestamp* conflict resolution policy was applied. The problem is that the execution cannot be considered as optimal in terms of fairness since all aborts are handled by core 3. Therefore, we offer a hybrid conflict resolution policy called ***commit***. It prioritizes the core which performed the least amount of transaction commits. If the conflicting transactions all committed the same number of transactions the ***timestamp*** strategy is applied. The conflict resolution policy will resolve the conflicts as depicted by Figure 5.11. Since core 3 in Figure 5.11 has not yet committed a transaction, it gets a higher priority than core 1 and therefore TX4 is aborted. In the example provided by Figure 5.11 the policy *commit* is beneficial for performance and ensures that not only one or a few cores suffer from the majority of aborts.

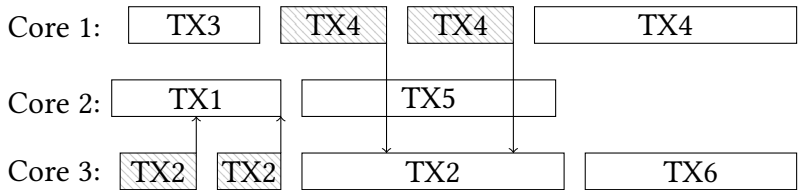

Figure 5.11: In contrast to Figure 5.3, Transaction 2 (TX2) now aborts transaction (TX4), which results in a more distributed handling of the aborts. The delay of the aborts is not significantly reduced, but, and this is beneficial for the overall performance, the delay is divided up onto two cores (core 1 and core 3). This extends the execution time of core 1 but reduces the execution time of core 2 which leads to an overall reduced execution time compared to the execution depicted by Figure 5.3.

Another conflict resolution policy we offer is based on the idea of prioritizing a transaction which already aborted other transactions. The name of the contention management strategy is ***abort***. If this strategy is applied a transaction is prioritized over other if it already aborted more transactions than the conflicting one. The underlying assumption of this conflict resolution policy is that a transaction which already aborted many other transactions is very likely to again be responsible for a lot of conflicts once it was aborted and restarted. Therefore, one is well advised to allow the transaction to continue and commit so it cannot further disturb execution. In order to work the LLC saves the number of how many transactions were aborted by a transaction. If a conflict arises this number is taken into account to resolve the conflict. If conflicting transactions aborted the same number of transactions the transaction detected the conflict wins.

Table 5.4 lists the different contention management strategies and shows which information has to be saved by the LLC controller for the contention management strategy to work.

Contention Management Strategy:	**Data Saved:**
passive	nothing
timestamp	timestamp of when transaction started
priority	priority per core
commit	timestamp and number of commits
abort	how many transactions a core already aborted

Table 5.4: Data saved depending on the contention management strategy

5.7 Estimating Execution Times for Extended HTM

Since we developed an extended HTM for the use in high performance embedded systems, we are aware that estimating execution times is an important feature. Unfortunately, our extended HTM does not allow to calculate an accurate WCET but the properties of the extended HTM allow some simplifications when calculating execution times.

The term WCET is highly overloaded, since it is used for a lot of different things. Therefore, we will give a brief explanation on how we use the term. With the term WCET we refer to the maximum time it took a system, offering one or more cores, to execute all transactions. In the following we refer to this time as WCET_{HTM}. For our work, we assume that the transactions all execute the same section of code, although this does not imply they have to execute the exact same code. Note that we assume that the execution time of all transactions could be estimated beforehand. In the following we show how the extended HTM is beneficial to calculate the WCET_{HTM}:

Calculating the WCET_{HTM} for the baseline HTM is not reasonable because most likely the WCET_{HTM} will be highly overestimated. Formula (5.1) shows how the WCET_{HTM} is calculated. The main factors for the overestimation are that the potentially failed transactions ($(f - 1) * T_i$) and the blocking time B_i have to be taken into account. Blocking time B_i is caused by transactions executed in fallback mode. Estimating the time of the blocking time is difficult since it is arbitrary and can strongly vary. Additionally, it is dependent on the execution performed on the other cores. Therefore, a very pessimistic value has to be assumed which has to be considered for every execution of a transaction.

$$\mathrm{WCET}_{\mathrm{HTM}} = \max_{x \in E}\left(\sum_{i=1}^{N_x}(T_i + (f-1) * T_i + B_i)\right) \tag{5.1}$$

E number of cores
N_x number of transactions executed on core x
T_i execution time of transaction i
f number of tries until transaction has to be executed in fallback mode
B_i blocking time

For the extended HTM the $\mathrm{WCET}_{\mathrm{HTM}}$ is still overestimated but is theoretically bounded by the execution time achieved on a single core executing the transactions one after the other. Since sophisticated contention management strategies (e.g., *timestamp*) in combination with unbounded transactions allow an execution where no fallback path is needed, the blocking time B does not have to be considered when calculating the $\mathrm{WCET}_{\mathrm{HTM}}$ for the extended HTM. Note that to bound execution time only the transaction with the oldest timestamp is able to execute in unbounded mode. Throughout the execution, interrupts have to be deactivated. Otherwise, a gapless execution cannot be guaranteed. Furthermore, the contention management strategy (e.g. *timestamp*) has to guarantee progress which means that always one transaction is running. Therefore, failed restarts ($(f-1) * T_i$ in Formula 5.1) can also be neglected. Calculating the $\mathrm{WCET}_{\mathrm{HTM}}$ for the extended HTM can therefore be simplified as shown by Formula 5.2.

$$\mathrm{WCET}_{HTM} = \sum_{i=1}^{N} T_i \tag{5.2}$$

N total number of transactions
T_i execution time of transaction i

5.8 Reducing False Conflicts

False conflicts are caused if a thread executing non-transactional code causes a conflict. The conflict occurs if the thread accesses data, which is part of a cache line contained in the RS or WS of one or more transactions, in a conflicting way. This is detected as a conflicting access although no data dependencies are really hurt.

With the extended HTM we are able to identify problematic accesses like this. If a non-transactional access violates a data dependency of a transaction the LLC can detect this. Since conflict detection is shifted to the LLC controller it knows exactly which cache lines are contained in the RS and WS of the transactions. Furthermore, it also has the information which core is currently running a transaction. Therefore, accesses of a non-

transactional execution can be deferred. This can be beneficial for performance. Figure 5.12 depicts an example of the described scenario.

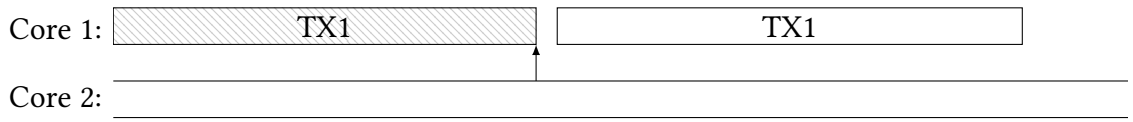

Figure 5.12: Core 1 executes transaction 1 (TX1). Core 2 executes non-transactional code. The code running on core 2 accesses a memory location, which is part of the RS or WS of TX1, in a conflicting way. Therefore, TX1 has to be aborted and re-executed.

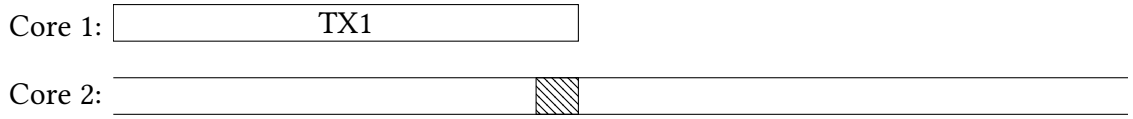

Figure 5.13: Figure 5.13 shows the same situation as Figure 5.12. In contrast the memory access of core 2 is deferred, which allows the transaction running on core 1 to finish without having to be aborted.

Figure 5.13 shows how the situation can be solved by the extended HTM. Here the system detects the false conflict and defers the memory access. As shown, this can be beneficial for overall execution.

This mechanism only works in combination with unbounded transactions and a contention management strategy guaranteeing progress. The idea behind this is that no transaction will be executed in fallback mode. Otherwise, false conflicts are detected if a core has to execute a transaction in fallback mode. Here the core will then try to write the mutex, used to alternatively secure the critical section. Since all other currently running transactions contain the cache line which holds the data of the mutex in their RS a false conflict would occur. The system detects that and defers the accesses. This is not the purpose of this mechanism. It should only be used on real false conflicts, since otherwise execution could be affected negatively.

Please note that the mechanism can only reduce the number of false conflicts. It is possible that the mechanism is not able to eliminate all false conflicts, since the properties of the MOSI cache coherence protocol can cause a deadlock like situation, when deferring memory accesses. Therefore, an access is only deferred a certain number of times. If the memory access cannot be serviced by then, the conflicting transaction is aborted.

5.9 Evaluation

In this section we are going to evaluate the extended HTM. For the evaluation we compare different setups of the extended HTM to the baseline HTM, described in Chapter 4, and to other configurations of the extended HTM. What exactly is compared is described in more detail in the following subsections. With the evaluation we are able to show the following achievements:

1. The extended HTM is able to provide different contention management strategies.
2. The extended HTM is able to reduce the number of aborts for the executions.
3. The extended HTM offers acceptable performance and is even able to increase performance for some benchmarks.
4. The extended HTM is able to reduce the number of false conflicts.
5. The extended HTM is able to guarantee progress and eliminates the need of a fallback path.

Since we use the STAMP benchmark suite [13] for our evaluation we show that our measures positively impact real world software. We can therefore do without synthetic benchmarks tailored to a specific problem that can be solved well by the extended HTM.

The rest of this section is structured as follows. First, we will give a brief overview of the hardware costs of our approach. Next, we will briefly describe the methodology and the benchmarks. Then, we will discuss where the extended HTM can potentially build up overhead compared to the baseline HTM. Before looking at the effects of unbounded transactions, we offer the evaluation of the contention management strategies described in Section 5.6. Since interesting results are achieved when combining the priority contention management strategy with unbounded transactions, we next provide the evaluation of the combination. After looking at the effects of the abort-aware execution we will complete this section by showing that the extended HTM is able to reduce the number of false conflicts.

5.9.1 Estimation of Hardware Costs

As we already stated in Section 4.6.1 we are not able to precisely estimate which impact our extensions have on real hardware concerning size and complexity. Therefore, we will here also point out where we believe additional hardware is necessary. Hereby we

look at the impact on the hardware when changing a system already providing an HTM as described in Chapter 4.

The logic used by the cache controllers in the baseline HTM to detect conflicts is not necessary for the extended HTM since conflicts are detected at the LLC level. The other functionalities are still required. Therefore, the other extensions described in Section 4.6.1 remain necessary and their impact on the hardware here also has to be considered. Additionally, the cache controllers as well as the LLC controller have to be further augmented to handle more new message types which adds to the hardware costs.

In our opinion the biggest impact to the hardware compared to the baseline implementation concerns the changes of the LLC controller and the directory. Since conflict detection was moved to the LLC level the system has to offer the possibility to globally add cache lines to the RS or WS. Therefore, the directory was augmented to hold the readers and writers of a cache line currently running a transaction. The extra lists are implemented like the list of sharers for a cache line and add to the size of a directory entry. Additionally, a stall list per core is required for the abort-aware execution. This adds complexity to the setup of the directory itself since more data has to be handled efficiently.

Since conflict detection is performed by the LLC controller additional logic is needed to detect conflicts at that level. Here the LLC controller had to be changed in the way that it checks if a cache line is part of an RS or WS. Depending on the incoming message, it has to then take the correct action (e.g. trigger an abort). Therefore, additional hardware is required, increasing the hardware costs.

The extended HTM offers different contention management policies. Therefore, the logic for them has to be provided by the LLC controller. Also, the registers which save counters and timestamps have to be considered when investigating the hardware costs.

To clear the RS and WS at LLC level also the logic for an appropriate mechanism has to be provided. It can be considered very similar to the mechanism of removing a sharer from a cache line. Nevertheless, this also adds to the complexity and costs of the hardware setup.

5.9.2 Methodology

Just as in the evaluation of the baseline HTM we used the STAMP benchmark suite (see Section 2.4 for more details) to evaluate our extended HTM. The execution was performed in syscall emulation mode allowing a bare metal execution of programs by emulating syscalls. The exact configuration of how the extended HTM was executed

matches the configuration used for the baseline HTM. For the sake of completeness, the exact parameters are listed in Table 5.5.

Table 5.5: System Configuration

Num CPUs	{1,2,4,8,16}
Microarchitecture	ARM Cortex-A15
L1 data cache	32KB
L1 data cache assoc.	8
LLC cache	2MB
LLC assoc.	16
Cache Coherence	directory-based
Coherence Protocol	MOSI

5.9.3 Benchmarks

Table 5.6: Configuration of STAMP Benchmarks

Benchmark	Parameters
bayes	-v32 -r1024 -n2 -p20 -s0 -i2 -e2
genome	-g256 -s16 -n16384
intruder	-a10 -l4 -n2038 -s1
kmeans	-m40 -n40 -t0.05 -i inputs/random2048-d16-c16.txt
labyrinth	-i inputs/random-x32-y32-z3-n96.txt
ssca2	-s13 -i1.0 -u1.0 -l3 -p3
vacation	-n2 -q90 -u98 -r16384 -t4096
yada	-a20 -i inputs/633.2
fallback exec.	defined in subsections but always ten for baseline HTM

The STAMP benchmarks are highly portable and are often used to examine TMs. Therefore, the STAMP benchmarks were used to evaluate a wide variety of TMs. This includes software, hardware, and hybrid approaches. Since the Benchmarks are also used to evaluate the baseline HTM a detailed description of the STAMP benchmarks can be found in Section 2.4.

The STAMP benchmarks were evaluated with the small input set, since this is recommended by the authors of [13] when using a simulator. The launch configuration also matches the launch configuration used to evaluate the baseline HTM and is given by Table 5.6. Please note that some measures only show an effect when alternating the number of attempts until a transaction is executed in fallback mode. Therefore, the number of attempts varies and is defined in the subsections. The number of attempts is always ten

for the baseline HTM. Independent of where a transaction was executed (on the baseline or the extended HTM) a capacity conflict is always resolved by directly executing the transaction in fallback mode.

5.9.4 Overhead of extended HTM

Compared to the baseline HTM extended HTM generates an overhead when adding a cache line to the RS or WS. How big the overhead is depends on three different factors:

1. In which state is the cache line currently?
2. Is cache line dirty and needs to be written back?
3. Is the cache line already part of the RS or WS?

When a cache line, which is going to be added to the RS or WS, is in a state (e.g., invalid) which causes the L1 cache controller to request data from the LLC controller, the overhead of adding a cache line to the RS or WS is comparable to the overhead caused by the baseline HTM. Since a request to the LLC controller is triggered anyway the information that the cache line was added to the RS or WS can be transferred with no extra costs.

If the state of that cache line indicates that it was only read (e.g., shared) the L1 cache controller has to send an coherence message to the LLC informing it that the cache line is going to be added to the RS. The L1 cache controller will not service the original request until the LLC controller acknowledges the adding of the cache line to the RS. Note that adding the cache line to the WS would not cause any extra overhead since a request to change the cache lines state to modified would have to be triggered anyway.

If a cache line is already part of the RS or WS the overhead is as big as an extra coherence message. Compared to the baseline HTM an extra message to the LLC controller has to be triggered informing it about that the cache line was added to the RS or WS. Sending the extra message is not on the critical path of the execution of the transaction's and therefore is not critical for performance. Note that these messages certainly cause contention on the bus, which has an indirect effect on the overall performance since it takes longer to service requests.

The biggest overhead is generated when a cache line in modified or owned state is added to the RS or WS. Prior to being able to add the cache line to the RS or WS it has to be backed up. Otherwise, data might be lost (see Section 5.2.5 for more details). After the write back of the cache line is acknowledged by the LLC controller the L1 cache can reload the data. With that request the cache line is added to the RS or WS managed by the LLC controller.

Another source of overhead concerns owned cache lines. If the corresponding core of the cache controller, marked as owner of the cache line, runs a transaction a read or write request cannot just be redirected, since conflicts could slip (see Section 5.2.5 for more details). Therefore, the L1 cache controller is forced to write back the cache line. The original request will be serviced as soon as the data is written back.

In general, adding cache lines, which are currently present in one of the local L1 caches, to the RS or WS is quite costly. The overhead then ranges between one or two cache misses. Therefore, the average size of the transactions has increased.

5.9.5 Contention Management Strategy: timestamp

One of our goals is to get rid of the fallback path. Transactions executed in fallback mode inhibit parallelism. Adding code for the fallback path also complicates the programming of the application. Therefore, we evaluate how beneficial the *timestamp* strategy is to reduce the number of transactions executed in fallback mode. To see how effective the *timestamp* strategy is we launched the benchmarks without offering a fallback path when applying the *timestamp* strategy. This means that a transaction can fail without limitation until it commits. The only exception made is when the RS or the WS of a transaction exceeds the maximum capacity. If such a capacity conflict is detected the transaction is directly executed in fallback mode. Note that without this measure transactions exceeding the capacity of the RS or the WS would be re-executed infinitely often. This would cause a livelock situation in which no progress is made. In the following we will first examine if the number of transactions executed in fallback mode could be lowered. Then we will investigate how the number of aborts changed. To give a holistic evaluation we will also look at the performance compared to the execution on the baseline HTM.

Figure 5.14 shows bar charts for all eight STAMP benchmarks. The bar charts allow the comparison of the number of transactions executed in fallback mode for the baseline HTM and the extended HTM applying the *timestamp* strategy. The x-axis of the bar charts represents the number of cores with which the execution was performed. The y-axis indicates how many transactions were executed in fallback mode. The number on top of the bars is an exact representation of how many transactions were executed in fallback mode. The legend entry baseline refers to the execution performed on the baseline HTM. Here a transaction is executed in fallback mode after ten failed attempts or due to capacity issues. The legend entry *timestamp* refers to the execution performed on the extended HTM applying the *timestamp* contention management strategy.

First it is important to note that all executions performed on the extended HTM were completed. This means that the *timestamp* contention management strategy is an effective way to ensure progress. Since always one transaction is executed because it is prioritized over the others, livelock situations are avoided.

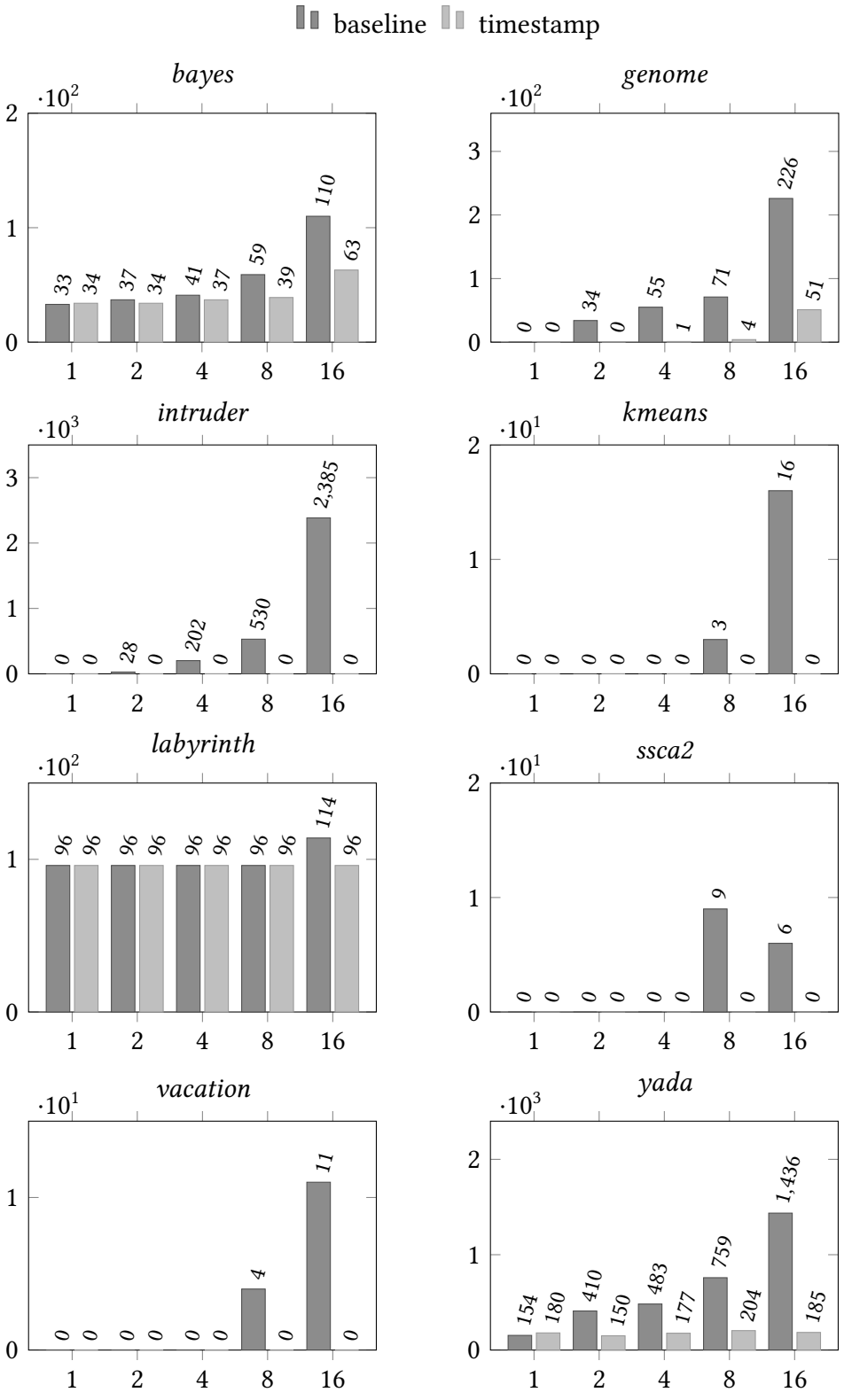

Figure 5.14: The bar charts compare how often the fallback path was taken when the STAMP benchmarks are executed by the baseline HTM and the extended HTM applying the *timestamp* strategy. The y-axis of the bar charts relates to the number of transactions executed in fallback mode. The x-axis relates to the number of cores. For every execution the number of transactions executed in fallback mode was reduced or stayed the same, when applying the *timestamp* strategy.

As expected, the number of transactions which had to be executed in fallback mode could be reduced for the runs performed on the extended HTM. The greatest reduction is achieved by the benchmark *intruder* when executed on 16 cores. The benchmark has no capacity related fallback executions, which is why the number could be lowered to zero when applying the *timestamp* strategy. Figure 5.14 shows that when applying the *timestamp* strategy four benchmarks theoretically could have been executed without offering a fallback path. This concerns the benchmarks *intruder*, *kmeans*, *ssca2* and *vacation* since none of these benchmarks executed a transaction in fallback mode. In theory therefore no fallback path would have to be provided for these benchmarks. In reality providing the progress guaranteeing strategy *timestamp* is not sufficient enough to not provide a fallback path. Only minor changes in the behavior of the caches could mean that a transaction exceeds the size of the RS or WS which would make a fallback path necessary again.

The number of transactions executed in fallback mode for the benchmarks *bayes*, *genome*, *labyrinth* and *yada* stayed the same or were reduced. The remaining transactions executed in fallback mode, where the *timestamp* strategy was applied, all relate to capacity conflicts. Note that for the benchmarks *genome* and *yada*, which not only suffer from capacity related fallback executions, the numbers of transactions executed in fallback mode could be reduced.

Since the transactions are re-executed infinitely often until they commit, when the *timestamp* strategy is applied, we also look at how the transaction aborts behave. Therefore, the number of aborts obtained during the execution on the extended HTM is compared to the number of aborts obtained during the runs on the baseline HTM. The comparison is depicted by Figure 5.15.

Figure 5.15 shows a bar chart for every benchmark contained in the STAMP benchmark suite. The bar chart compares how many transactions were aborted for the baseline HTM and the extended HTM applying the *timestamp* contention management strategy. The x-axis of the bar charts represents the core count with which the execution was performed. The y-axis indicate how many transactions were aborted. The number on top of the bars is an exact representation of how many transactions were aborted. The legend entry *baseline* refers to the execution performed on the baseline HTM. Note that for the baseline HTM conflicts are resolved following a simple contention management strategy (see Section 4.3.1 for more details). The legend entry timestamp refers to the execution performed on the extended HTM applying the *timestamp* strategy.

The number of aborts could be lowered or stayed about the same for four (*genome*, *kmeans*, *ssca2* and *vacation*) of the eight benchmarks. Therefore, the *timestamp* strategy can be beneficial to lower abort rates. Since the *timestamp* strategy prevents that two transactions continuously abort each other one source, which increases the number of aborts, is eliminated. If a benchmark is vulnerable to this the *timestamp* strategy is beneficial.

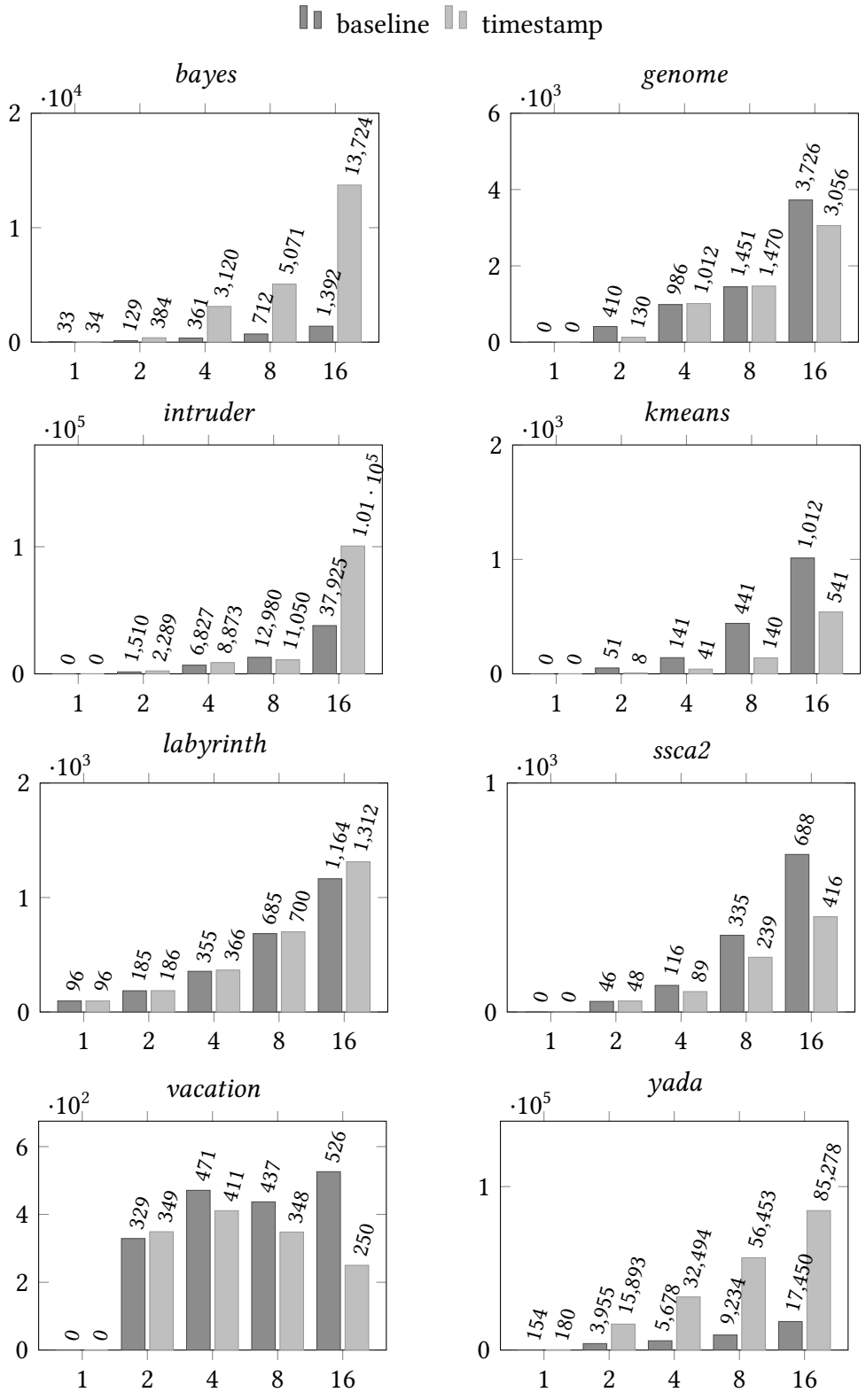

Figure 5.15: This figure compares the number of aborts when the STAMP benchmarks are executed by the baseline HTM and the extended HTM applying the *timestamp* strategy. The y-axis of the bar charts relates to the number of aborts. The x-axis relates to the number of cores. The benchmarks *bayes*, *intruder* and *yada* stick out because here the numbers of aborts are very high for the extended HTM applying the *timestamp* contention management strategy.

For the other benchmarks (*bayes*, *intruder*, *labyrinth* and *yada*) the number of aborts increased. In particular for the benchmarks *bayes*, *intruder* and *yada*. Especially with higher core counts the contention for these benchmarks is extremely high. Therefore, the number of conflicts explodes for the executions where the *timestamp* strategy is applied. Since the number of failed attempts is not limited the number of aborts can get very high. Performance is not affected as we will describe later.

Although the aborts of the benchmark *labyrinth* almost entirely relate to capacity conflicts the numbers of aborts are quite high. Since a transaction in fallback mode is the only one allowed in the critical section, it will always abort all currently running transactions. The conflict which then occurs is identified as a regular conflict and does not indicate that the transaction will abort due to a capacity conflict. Therefore, the execution does not know that its transaction will fail due to capacity issues. Instead, it aborts and re-executes the transaction. Since the baseline execution will execute a transaction in fallback mode after ten failed attempts the number of aborts is lower as for the execution applying the *timestamp* strategy. It will continue to abort and re-execute the transaction until it commits.

At the end of this subsection, we look at the performance of the extended HTM applying the *timestamp* contention management strategy and compare it to the performance of the baseline HTM. Therefore, Figure 5.16 depicts eight speedup graphs. Every speedup graph relates to one of the eight benchmarks of the STAMP benchmark suite. The x-axes represent the number of cores used for the execution. The y-axes indicate the speedup compared to the reference execution time, which refers to the run of the benchmarks with one core and no synchronization. It is marked as solid horizontal line in every speedup graph. The speedups for the benchmarks depicted in Figure 5.16 are calculated as shown by Equation (5.3). This is the same equation also used in Chapter 4 to calculate the speedups.

$$\text{speedup} = \frac{\text{reference execution time}}{\text{examined execution time}} \tag{5.3}$$

Figure 5.16 compares the performance of the baseline HTM to the extended HTM. The legend entry *baseline* refers to the executions performed on the baseline HTM. The legend entry *timestamp* refers to the execution performed on the extended HTM applying the *timestamp* contention management strategy.

For the benchmark *bayes* performance could not be improved instead performance slightly deteriorated. Although the extended HTM was able to reduce the number of transactions which had to be executed in fallback mode by over 40%, performance could not be improved. Since the benchmark suffers from a lot of contention the execution is serialized for great parts, which explains the overall poor performance. This brings some advantages for the execution with the baseline HTM. Due to the high contention and capacity

conflicts, the extended HTM is slowed down. Since transactions are prioritized the execution time of a transaction, causing a capacity conflict, can be quite long. Meanwhile the other transactions are aborted. In contrast to the baseline HTM that has no effect since no limit for failed attempts exists. Therefore, capacity conflicts slow down the entire execution in comparison to the baseline HTM, since here the transactions are executed in fallback mode much faster. In some cases, as for the execution of the benchmark *bayes* this is beneficial for performance. Please note that the decrease is very low and overall performance also for the baseline HTM is poor.

Also, for the benchmark *genome* performance decreases slightly when executed with the extended HTM applying the *timestamp* contention management strategy. Especially for eight cores the performance drop is noticeable. The reason for this is that the benefits for the execution cannot make up for the overheads which are generated by the extended HTM. This changes when the core count and contention increases. For 16 cores the benefits are higher and the gap between the executions gets smaller.

For the benchmark *intruder* the execution times for one, two, four and eight cores are almost the same. The execution with 16 cores differs. Although the baseline execution as well as the execution applying the *timestamp* strategy both decrease performance compared to the execution with eight cores, the drop of the execution of the extended HTM is more significant. Since the contention between the transactions is very high, indicated by the high number of aborts in Figure 5.15, the execution of the transactions takes longer. Due to a high amount of traffic on the bus connecting the L1 cache controller and the LLC controller (see Section 4.1 for more details) the execution times of the transactions are extended. The high amount of traffic on the bus is a result of the high contention and that the RS and WS managed by the LLC has to be cleared (see Section 5.2.5 for more details). Since a lot of aborts occur during the execution the traffic on the bus is high. Therefore, it is beneficial to execute the transactions in fallback mode as done by the baseline HTM after 10 failed attempts.

The benchmark *kmeans* also scales very well for the execution with the extended HTM. The small deterioration of the speedup compared to the execution of the baseline HTM can be explained by the overhead discussed in section 5.9.4. The overhead lets the short transaction slightly grow and therefore increases the overall execution time.

Many transactions of the benchmark *labyrinth* abort due to capacity conflicts. Therefore, the parallel execution of this benchmark is only possible to a limited extent and overall performance is below the reference execution. Since the execution slightly profits from being executed in parallel, a performance increase can be observed when executing the benchmark on more than one core. Therefore, execution time decreases with higher core counts. Overall, the speedup still stays below the reference execution.

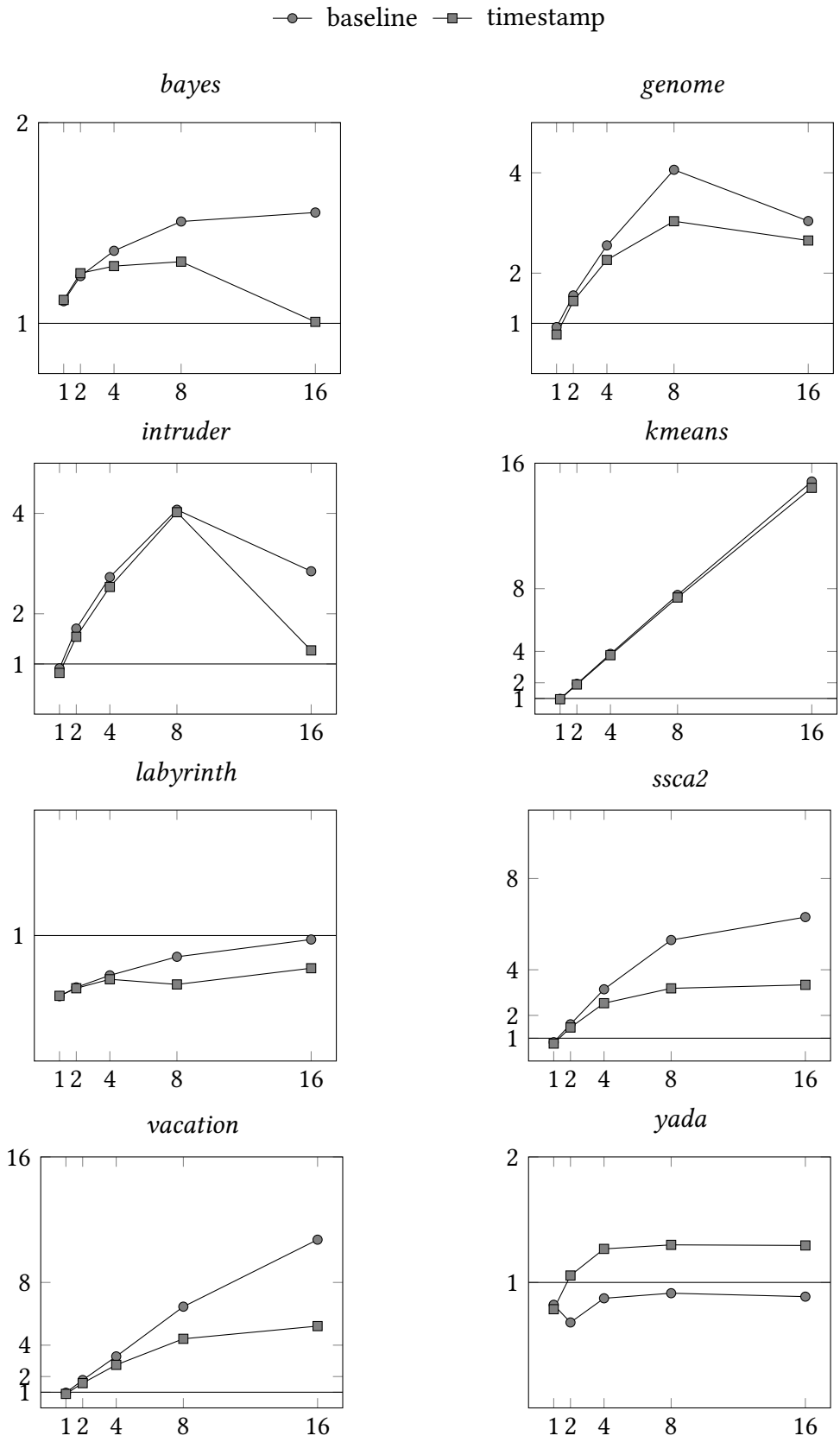

Figure 5.16: Comparison of the performance values achieved by the baseline HTM and the extended HTM when executing the STAMP benchmark suite. The y-axes refer to the speedup values compared to the reference execution (one core, no synchronization). The x-axes refer to the number of cores the execution was performed with.

The benchmarks *ssca2* and *vacation* hardly execute any transactions in fallback mode when executed by the baseline HTM. This means that the benchmarks both are suited to be executed highly parallel. Therefore, they are a prime example of benchmarks which can be very well executed by transactional memories. Since the benchmarks are obviously also programmed to make progress the extended HTM has no attack points to improve performance. Instead, performance is lower due to the overhead the extended HTM generates. Overall, the speedups for the extended HTM are still acceptable and it makes sense to provide as many cores as possible, since performance continuously increases. The overhead becomes higher the higher the core count gets, since the benchmarks launch many small transactions touching many different cache lines.

The last benchmark is the benchmark *yada*. This is the only benchmark of the benchmark suite for which the extended HTM achieved a better speedup than the baseline HTM. Since the benchmark has many conflicts the baseline execution has to execute a lot of transactions in fallback mode (see Figure 5.14). Transactions executed in fallback mode prohibit parallelism which is bad for performance. The extended HTM gets along without offering a fallback mode for conflicting transactions which is why no parallelism is prohibited. This produces gains in performance since the execution is not blocked.

5.9.6 Contention Management Strategy: commit

Although the *commit* contention management strategy aims at fairly distributing commits and aborts the strategy does not show any significant effects compared to the *timestamp* strategy. This concerns the number of aborts, the number of transactions having to be executed in fallback mode and performance. Therefore, we refrain from a detailed analysis.

For the sake of completeness, we provide the data which lead to our findings through three figures (Figure 5.17, 5.18 and 5.19). All figures contain the data of the execution performed when applying the *commit* or the *timestamp* contention management strategy to the extended HTM. The legend entries *timestamp* and *commit* hereby refer to the corresponding executions. Additionally, the figures also provide the data generated by the execution of the eight STAMP benchmarks on the baseline HTM to which the legend entry *baseline* refers.

The y-axes in Figure 5.17 hereby provide the number of aborts. The y-axes in Figure 5.18 depict how often the fallback path was executed and the y-axes in Figure 5.19 refer to the achieved speedups calculated by Formula 5.3. The x-axes in all three Figures refer to the number of cores.

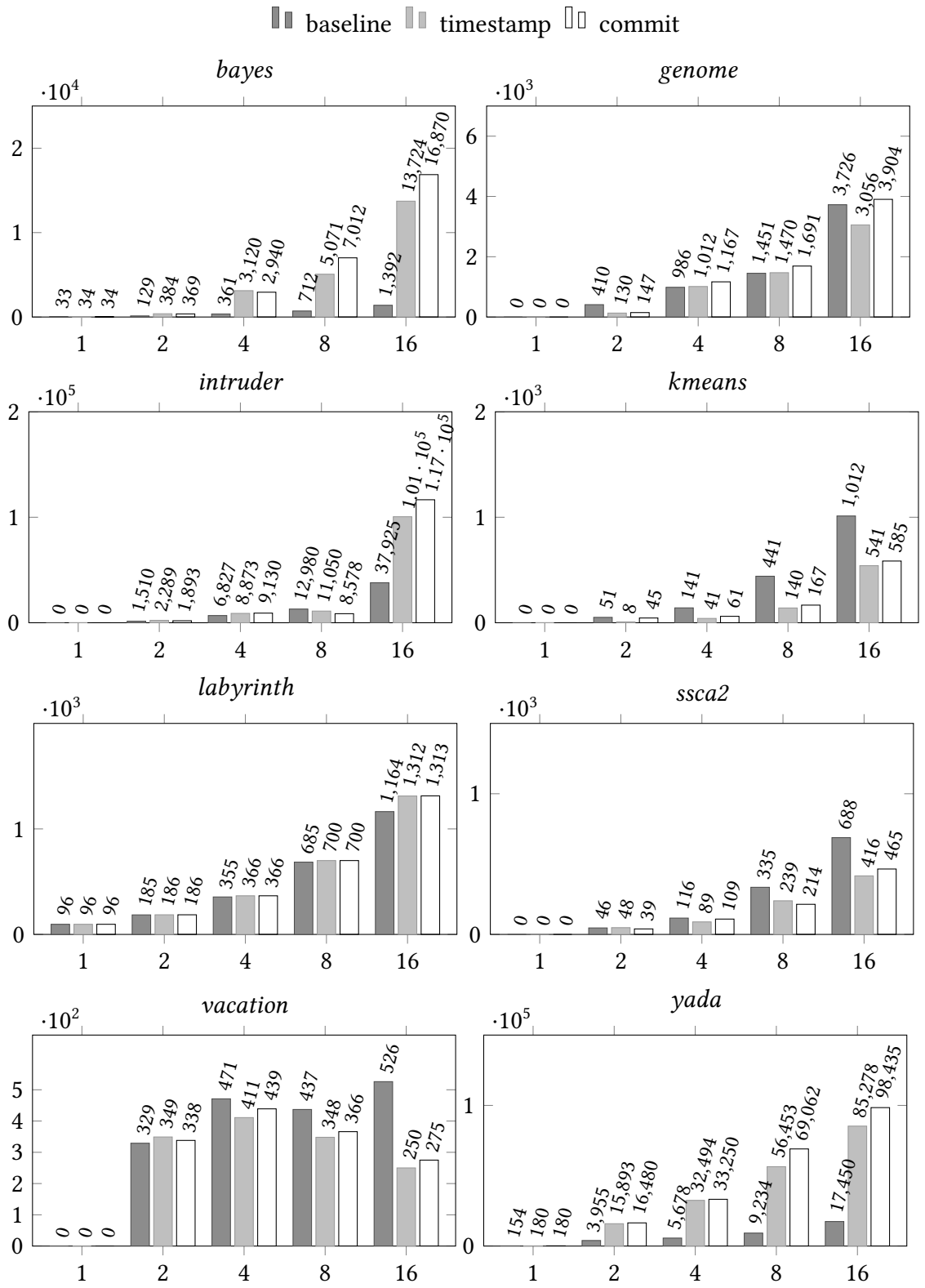

Figure 5.17: In general, the number of aborts generated when applying the *commit* contention management strategy, are very similar to the number of aborts achieved with the *timestamp* contention management strategy. Additionally, we also depict the number of aborts achieved by the baseline system.

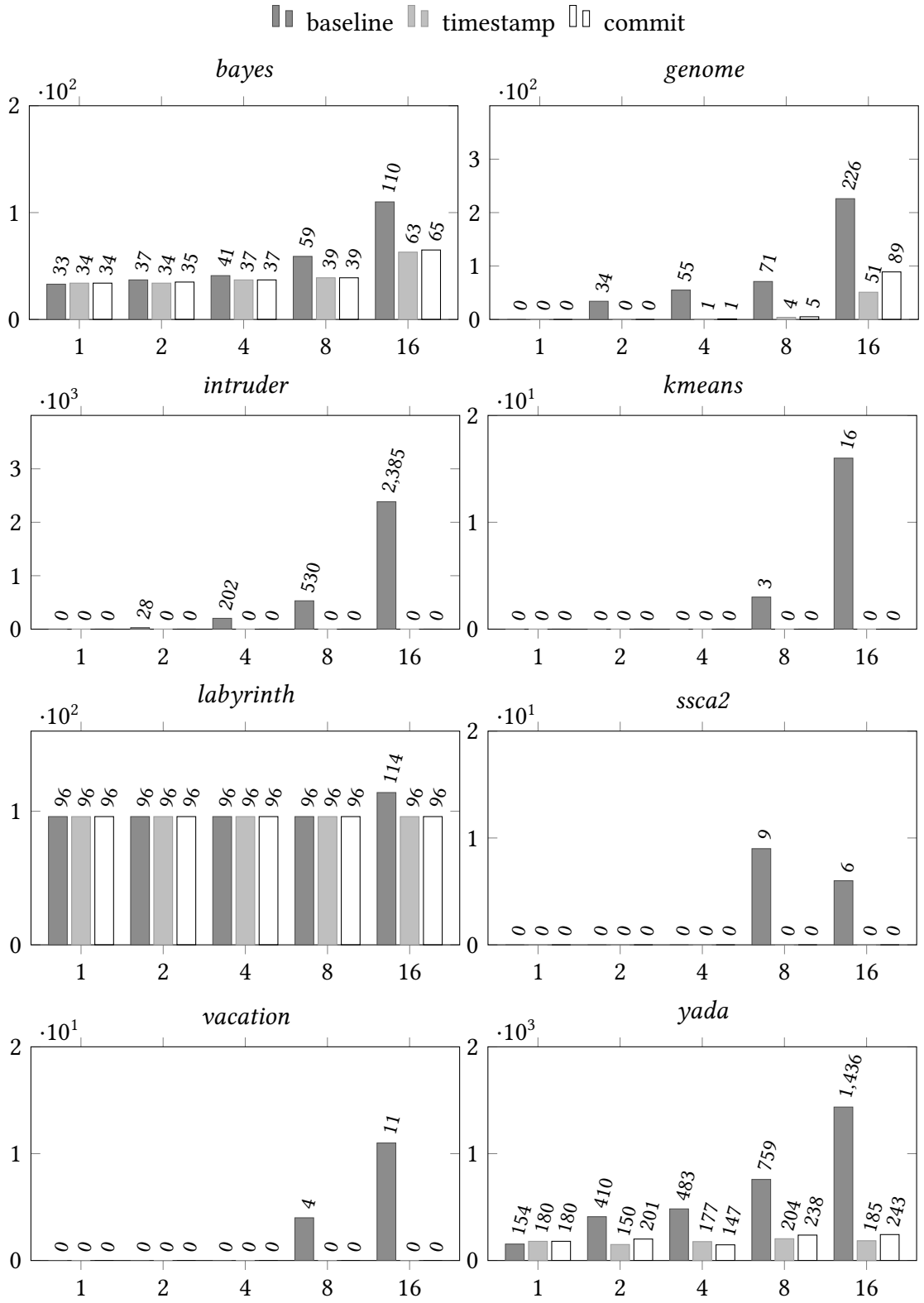

Figure 5.18: The bar charts show that the number of fallbacks generated when applying the *commit* contention management strategy, are very similar to the number of aborts achieved with the *timestamp* contention management strategy. For reasons of comparison, we also depict the number of fallback executions performed when the benchmarks are executed by the baseline HTM.

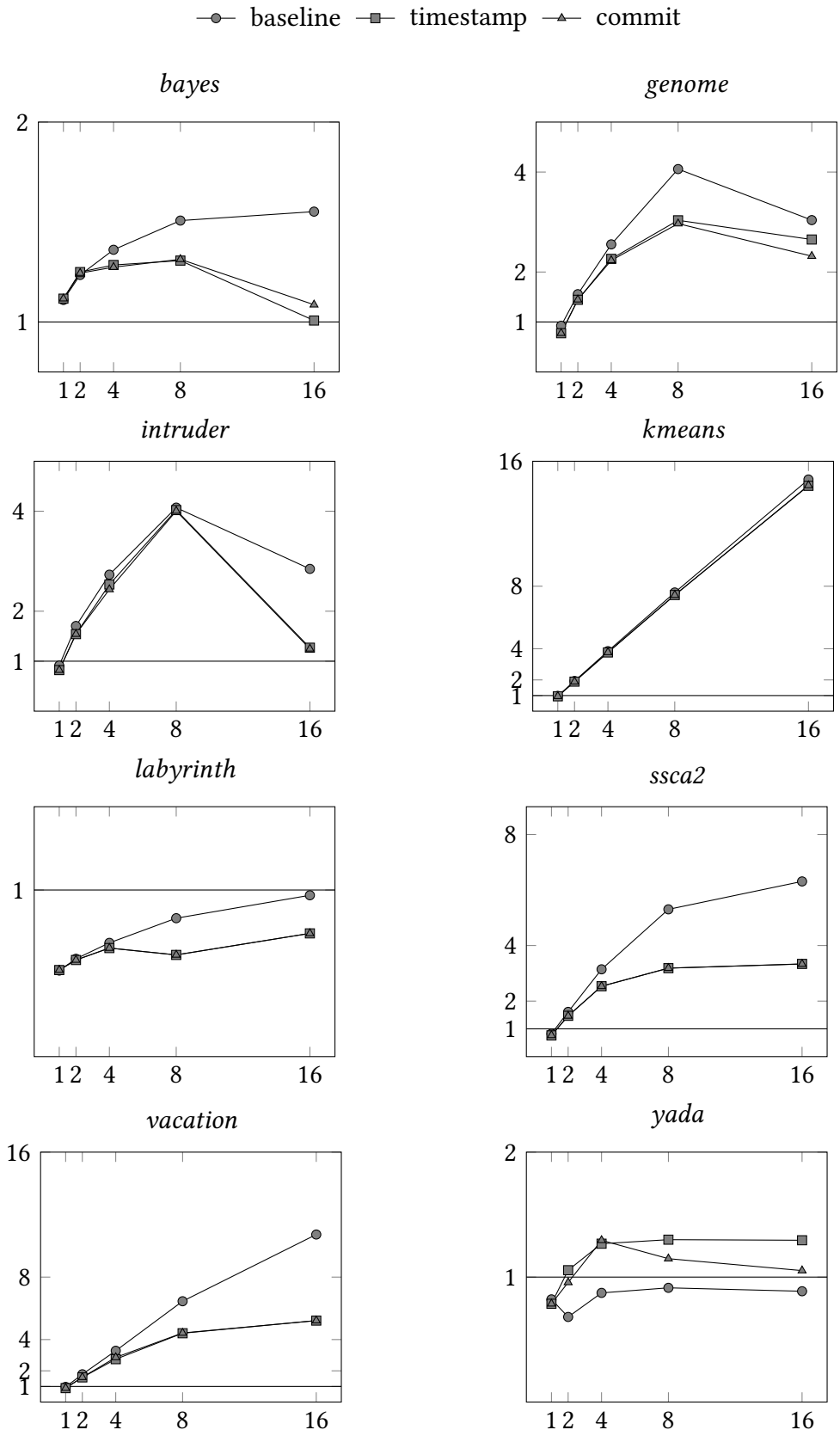

Figure 5.19: The speedups of the execution where the *commit* contention management strategy is applied, behaves very similar to the speedups achieved when the *timestamp* contention management is applied. For a better classification we also depict the speedups achieved by the execution with the baseline HTM.

For the *commit* strategy to show its full potential, applying a contention management strategy like the *timestamp* would have to cause a core to be systematically disadvantaged. The STAMP benchmarks do not offer such a benchmark. Additionally, conflict management strategies like *timestamp* also seem to be quite fair in terms of distributing commits and aborts.

Achieving fairness seems to be a very attractive goal but it can also become problematic. Especially if cores execute a different number of transactions, which they do for almost every benchmark in the STAMP benchmark suite. Then applying the *commit* strategy can be a disadvantage: If a transaction repeatably aborts because of another transaction which runs on a core executing fewer transactions in total and therefore has not yet committed as many as the core which executes a lot of transactions, this can be counterproductive. If this happens the core which has to execute many transactions has to wait until the transaction committed on the core executing fewer transactions in total until it can continue, even though it has a greater workload.

Since the *commit* contention management strategy is a hybrid approach also taking the transaction start time into account if two cores committed the same number of transactions, the contention management strategies *commit* and *timestamp* seem to behave quite similar. Advantages achieved by the *commit* strategy seem to be very small and dissolved by a disadvantage caused by the alternated schedule of the transactions.

5.9.7 Contention Management Strategy: abort

The *abort* strategy has greater effects on the execution than the *commit* strategy. For three of the eight STAMP benchmarks the numbers of aborts could be reduced significantly compared to when applying the *timestamp* strategy.

Figure 5.20 shows eight bar charts comparing the number of aborts. The bar charts relate to the executions of the eight benchmarks of the STAMP benchmark suite. We focus our analysis on the benchmarks *bayes*, *intruder*, and *yada*, since the other benchmarks showed no significant effects and more or less achieved the same results concerning the number of aborts compared to the execution of the extended HTM which applied the *timestamp* strategy. One run in Figure 5.20 refers to the execution where *timestamp* contention management strategy is applied and the other one to where the *abort* strategy is applied. The legend entry *timestamp* refers to the execution of the extended HTM applying the *timestamp* strategy and the legend entry *abort* refers to the run on the extended HTM applying the *abort* strategy. For better comparability we also added the number of aborts, accumulated by the execution of the benchmarks on the baseline HTM. The baseline run is referred to as *baseline* in the legend. The y-axis of the bar charts indicate how many aborts occurred during the executions. The x-axis gives the information on

how many cores an execution was performed on. The number on top of the bars is an exact representation of how many aborts were accumulated.

For the benchmarks *bayes*, *intruder* and *yada* (except for the run with 4 cores) the numbers of aborts were reduced or stayed the same. This is especially the case for the higher core counts. For these benchmarks, the *abort* strategy therefore is able to reduce contention.

To show which impact the lowered number of aborts have on the speedup of the benchmarks, Figure 5.21 shows the achieved speedups. Therefore, the figure presents eight speedups graphs which relate to the eight benchmarks of the STAMP benchmark suite. The x-axes hereby represent the number of cores used for the execution. The y-axes indicate the speedup compared to the reference execution. The reference execution time refers to the run of the benchmarks with one core and no synchronization. It is marked as solid horizontal line in every speedup graph. The speedups for the benchmarks depicted in the Figure are calculated as shown by Equation 5.3. Just as in Figure 5.20 the legend entries indicate to which execution the lines belong.

The speedups of the benchmarks *bayes*, *intruder* and *yada* are very similar and more or less match the speedups where the *timestamp* strategy is applied. The benchmarks cannot profit in terms of speedup although the abort rates and therefore contention was lowered. Reducing the number of aborts did not affect the critical path. Therefore, the overall execution time of the benchmarks could not be shortened. Execution time can be lowered if the reduction of accumulated aborts leads to an increase of parallelism e.g., if the critical path of the execution could be lowered by a better distribution of e.g., transactional aborts. Parallelism can also be increased if the number of transactions executed in fallback mode is decreased since the execution of a transaction in fallback mode restricts parallelism. The longer the transactions the greater the effects. Since both strategies (*timestamp* and *abort*) only execute transactions with capacity issues in fallback mode the number of transactions executed in fallback mode is not lowered by reducing the number of aborts.

Figure 5.22 depicts how often the eight benchmarks of the STAMP benchmark suite executed the fallback path. The bar charts hereby show the baseline execution which refers to the execution of the benchmarks performed on the baseline HTM. Additionally, the bar charts also show two more executions which show the results of the execution of the benchmarks performed on the extended HTM applying the *timestamp* and *abort* contention management strategy. Here the legend entry *timestamp* refers to the execution which applied the *timestamp* strategy and the entry *abort* to the execution which applied the *abort* strategy. The x-axis of the bar charts indicate the number of fallback paths executed. The y-axis refers to the number of cores.

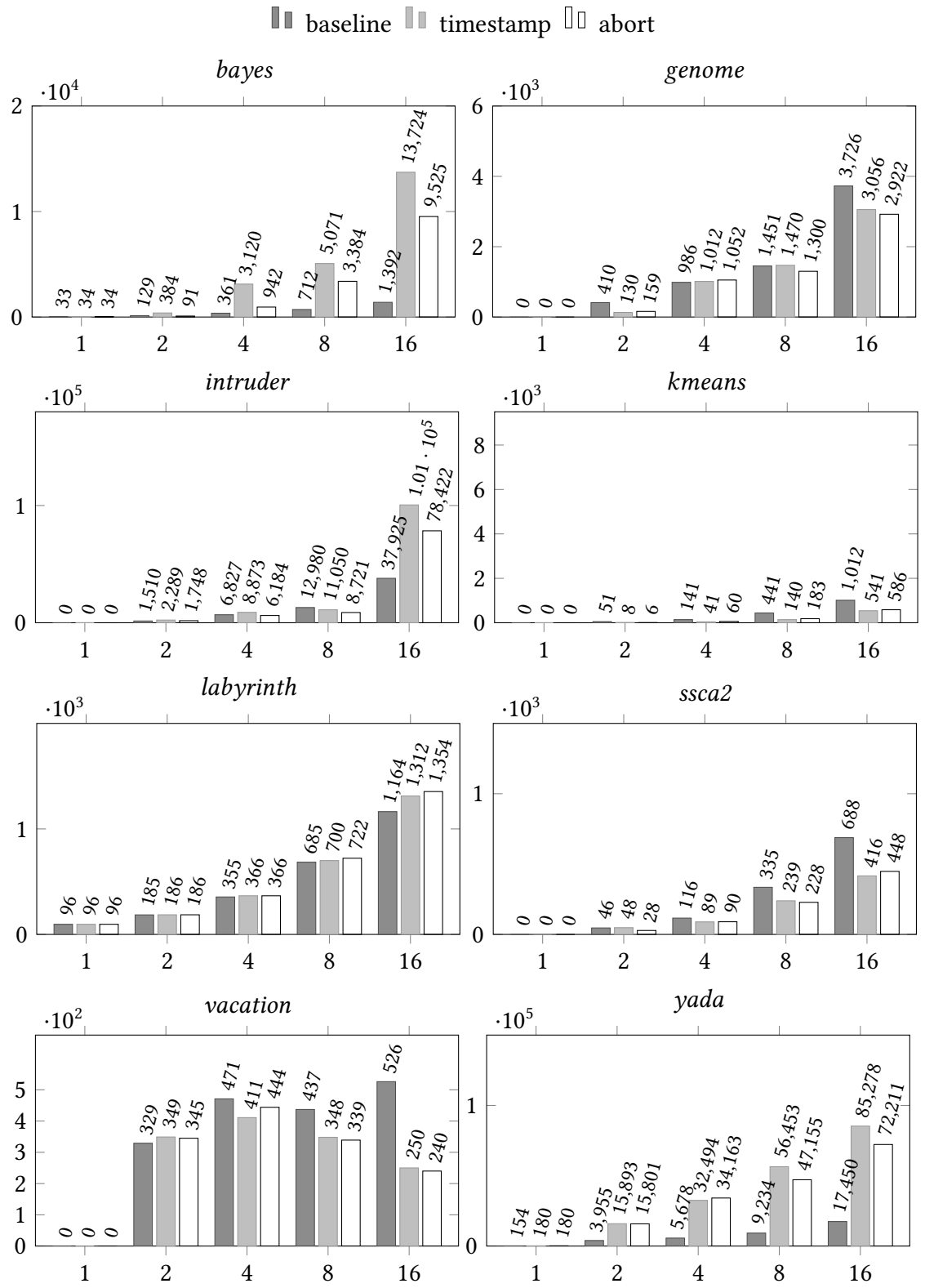

Figure 5.20: The number of aborts could be reduced for three benchmarks (*bayes*, *intruder* and *yada*) when applying the *abort* contention management strategy compared to the *timestamp* strategy. The benchmark *kmeans* suffers from applying the *abort* strategy indicated by high numbers of aborts.

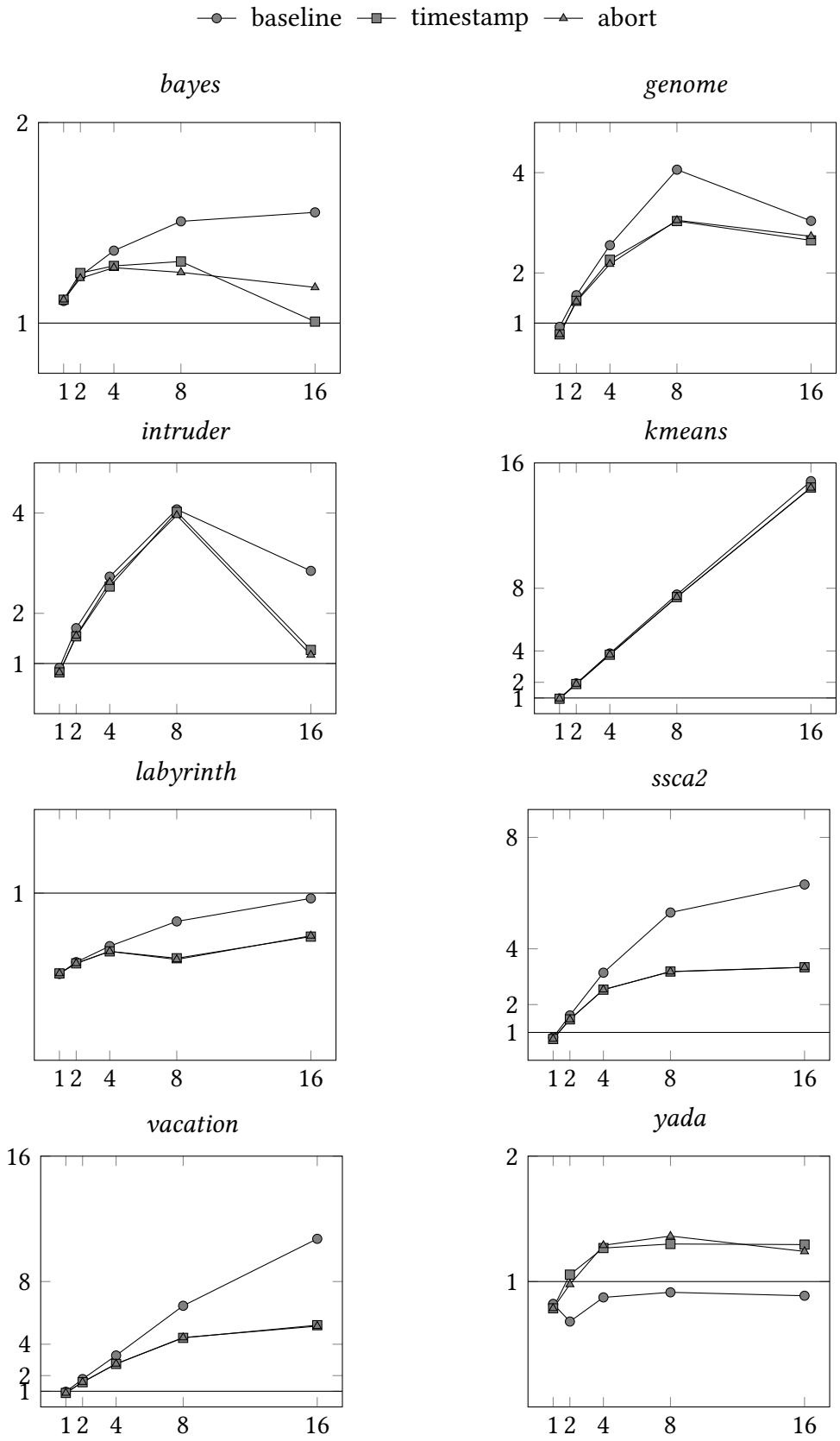

Figure 5.21: Comparison of the speedup values achieved by the baseline HTM and the extended HTM applying the *timestamp* and *abort* strategy when executing the STAMP benchmark suite.

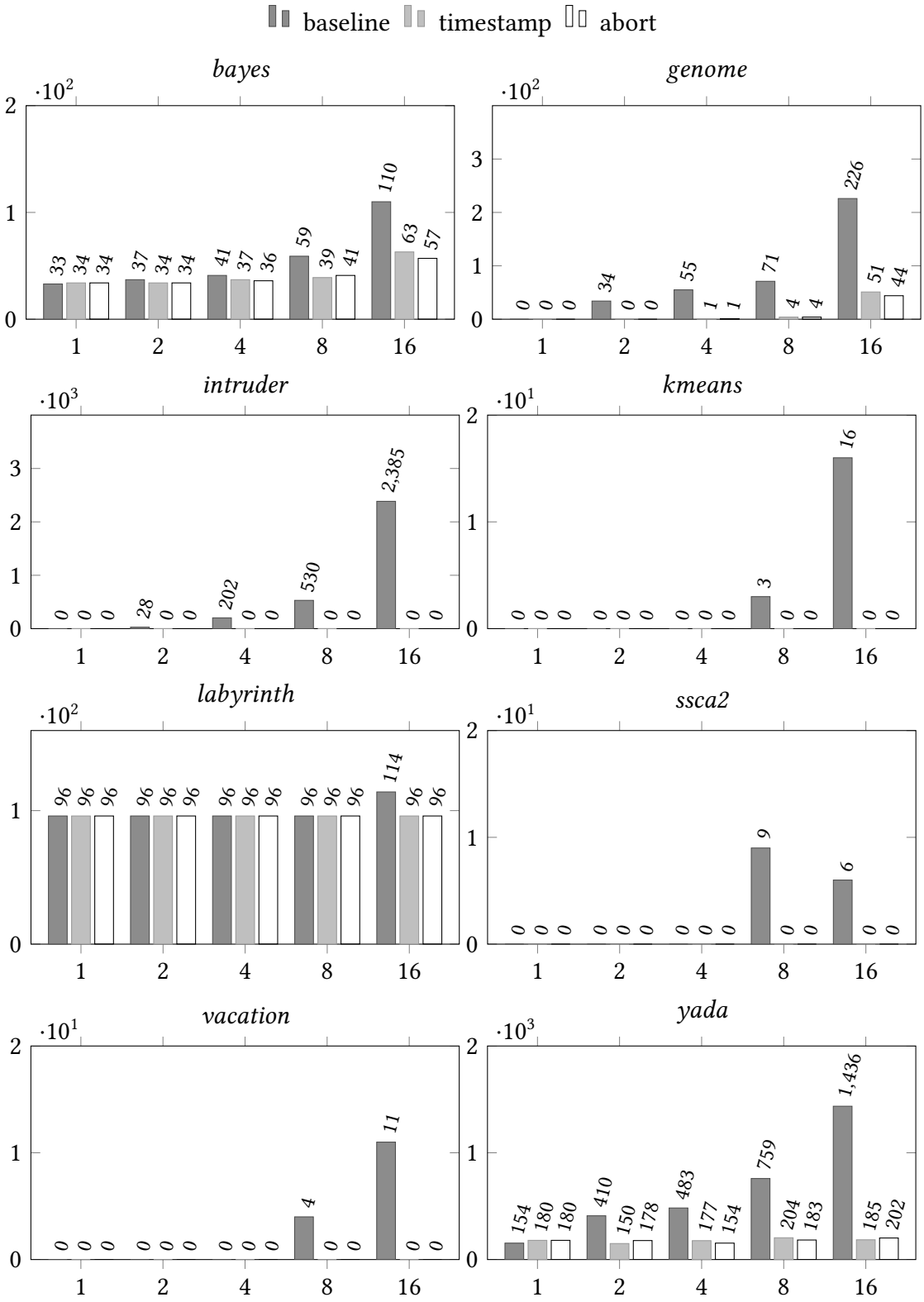

Figure 5.22: The bar charts show that the number of fallbacks generated when applying the *abort* contention management strategy, are very similar to the number of aborts achieved with the *timestamp* contention management strategy. For reasons of comparison, we also depict the number of fallback executions performed when the benchmarks are executed by the baseline HTM.

The number of fallback executions between the strategies *timestamp* and *abort* is quite similar. Even though the numbers are only slightly different the effects can be observed in Figure 5.21 since a variation of transactions executed in fallback mode for the execution performed on the extended HTM causes a slight speedup increase or decrease. The lower the number of transactions executed in fallback mode the better the speedup. The number of fallback executions is affected by the strategies (*abort* and *timestamp*) since they effect in which order transactions are executed and therefore have an effect on the content of the cache.

5.9.8 Unbounded Transactions

Unbounded transactions are an effective way to deal with transactions which exceed their capacity concerning the RS and WS managed by the L1 cache. We utilize unbounded transactions to eliminate the need to provide a fallback path. Eliminating the necessity of the fallback path makes code development easier and potentially allows better speedups, since transactions running in fallback mode prohibit parallelism. Eliminating the fallback path from the benchmarks only works in combination with a contention management strategy which guarantees progress. For this evaluation we chose the contention management strategy *timestamp*. As shown in Section 5.9.5 applying the *timestamp* strategy eliminated the fallback executions related to non-capacity conflicts.

Figure 5.23 shows bar charts for the eight STAMP benchmarks. The bar charts allow the comparison of the number of transactions executed in fallback mode for the extended HTM applying the *timestamp* strategy and the extended HTM applying the *timestamp* strategy as well as unbounded transactions. For better comparability we also added the number of fallback executions which were accumulated by the benchmarks when executed on the baseline HTM. The x-axis of the bar charts represents the number of cores. The y-axis indicates how many transactions were executed in fallback mode. The legend entry *baseline* refers to the execution performed on the baseline HTM. The legend entries *timestamp* and *timestampUnbounded* refer to the execution performed on the extended HTM applying the *timestamp* strategy. The execution referred to by timestampUnbounded also activates unbounded transactions.

By combining the *timestamp* contention management strategy with unbounded transactions we were able to eliminate the need of providing a fallback path. This is indicated by Figure 5.23 since no fallback execution has to be performed when using the extended HTM in combination with the *timestamp* strategy and unbounded transactions. Since the *timestamp* strategy guarantees progress and the unbounded transactions can handle transactions which exceed the size of the RS and WS of the L1 cache, every need of having to execute a transaction in fallback mode is handled by the extended HTM.

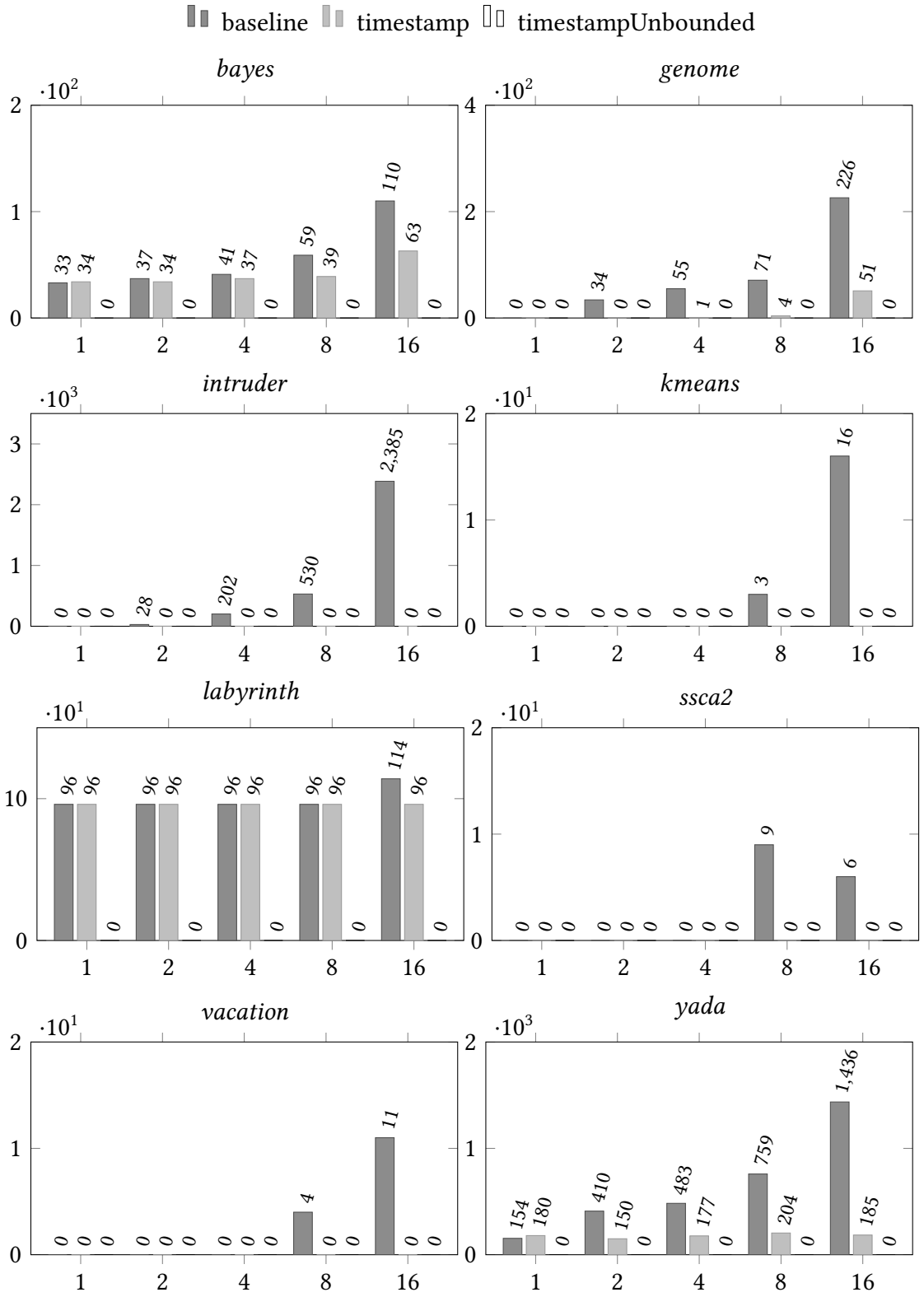

Figure 5.23: This figure compares how often the fallback path was taken when the STAMP benchmarks are executed by the baseline HTM, the extended HTM applying the *timestamp* strategy and the extended HTM applying the *timestamp* strategy as well as unbounded transactions. When combining the *timestamp* strategy with unbounded transaction no fallback path is executed.

Implementing unbounded transactions not only eliminates the need for a fallback path it also allows a performance gain for four of the eight STAMP benchmarks. Figure 5.24 depicts speedup graphs for the STMAP benchmarks. The x-axes represent the number of cores used for the execution. The y-axes indicate the speedup compared to the reference execution. The reference execution time refers to the run of the benchmarks with one core and no synchronization. It is marked as solid horizontal line in every speedup graph. The speedups for the benchmarks depicted in Figure 5.24 are calculated using Equation (5.3).

The benchmark *bayes* profits from applying the functionality of unbounded transactions. An unbounded transaction does not automatically abort the other transactions and also allows more parallelism. Therefore, the benchmark profits in terms of performance compared to the basline execution and the execution only applying the *timestamp* contention management strategy.

Also, the benchmark *genome* profits of from unbounded transactions. For the run with 16 cores executed by the extended HTM, only applying the *timestamp* contention management strategy, the system has to execute quite a lot of transactions in fallback mode compared to the other runs (see Figure 5.14 for more details). The run with 16 cores on the extended HTM applying the *timestamp* strategy and unbounded transactions does not have to execute transactions in fallback mode which is why performance can be increased for the execution with 16 cores. Here also the run on the baseline HTM is outperformed.

The benchmark *labyrinth* could be lifted above the solid line representing the reference execution (one core, no synchronization) for the execution applying the *timestamp* contention management strategy and unbounded transactions. Since almost all transactions abort due to capacity conflicts unbounded transactions are very beneficial for the execution of this benchmark. Unfortunately, also a lot of other conflicts occur which is why no greater performance gains were achieved.

The benchmark *yada* also has a lot of capacity related conflicts (see Figure 5.14 for more details) for which unbounded transactions are beneficial. In contrast to the benchmark *labyrinth* contention between the transactions is not as high which is why better performance values could be achieved.

For the other benchmarks (*intruder*, *kmeans*, *ssca2* and *vacation*) no speedups were observed. Since the number of transactions executed in fallback mode could not be further reduced, because it was already zero when only applying the *timestamp* strategy, no speedups gains could be achieved.

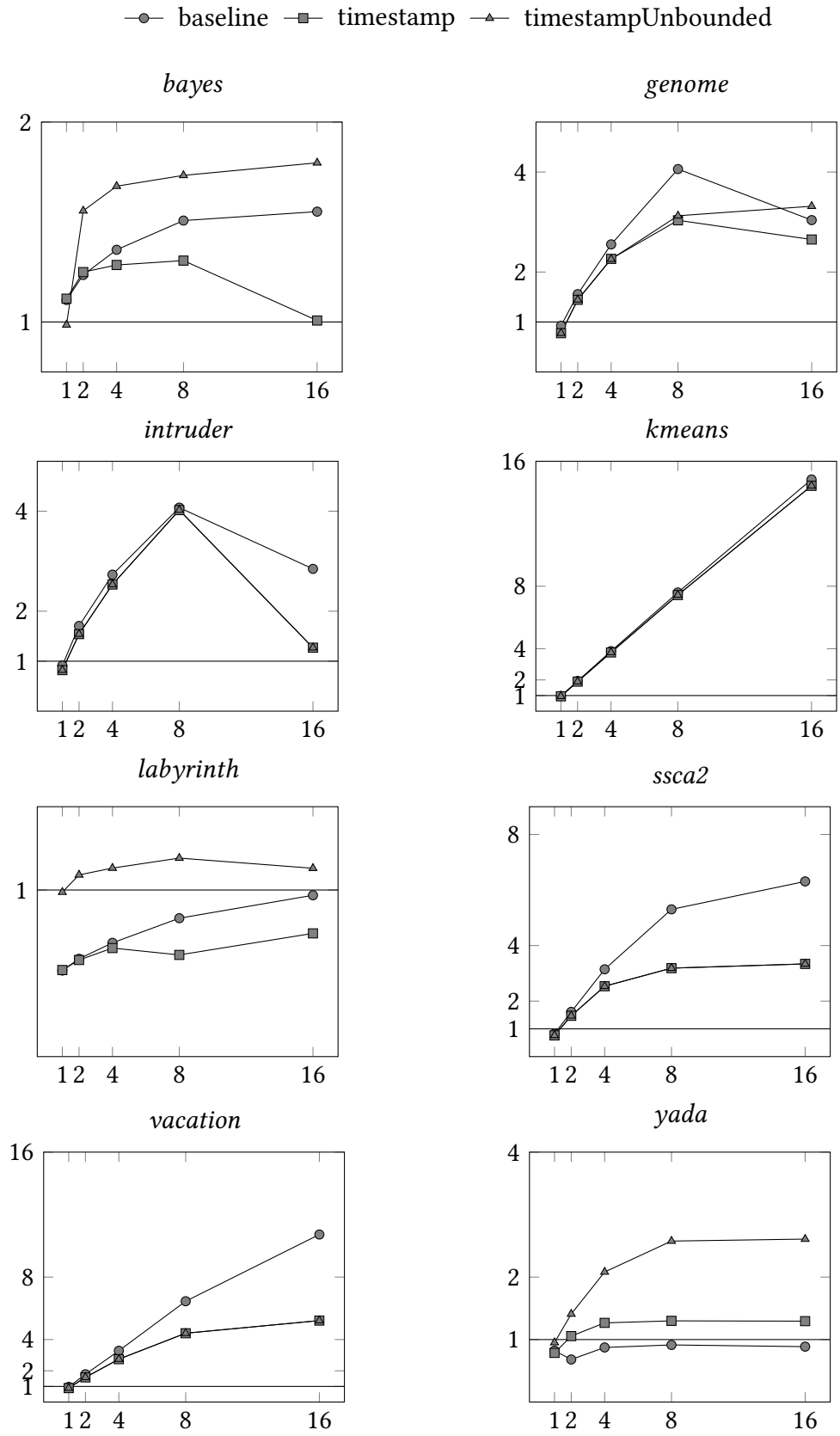

Figure 5.24: For the benchmarks *bayes*, *labyrinth* and *yada* performance was increased when unbounded transactions are activated. For the benchmark *genome* performance could only be improved for the run on 16 cores.

5.9.9 Contention Management Strategy: priority

The contention management strategy *priority* allows to set a priority for a transaction. If a conflict occurs the priority is used to resolve the conflict favoring the transaction with the higher priority. To show that the contention management strategy works as it should we started all eight transactions of the STAMP benchmark suite with priorities. Although it is possible to set priorities manually for every transaction, we tied the priority to the core the transaction is running on. That means the priority a transaction has depends on which core it is executed on. For our experiment core 1 has the lowest and core 16 the highest priority.

Since fallback executions due to capacity conflicts would add a lot of noise to our evaluation, we decided to combine the *priority* strategy with unbounded transactions. Now no transactions are executed in fallback mode. In our opinion this makes sense, since it shows the effects achieved in the best way possible.

Figure 5.25 shows a bar chart for every of the eight benchmarks of the STAMP benchmark suite. The name of the benchmark, to which the bar chart belongs, is written to the left of the bar charts. The y-axis of the bar charts represents the number of aborts. The x-axis of the charts relates to a specific core. Note that the bar charts only represent the execution running with 16 cores. The assumption hereby is that it is sufficient to show that the contention management strategy works by examining the run on 16 cores. If the evaluation were made for every core count (2, 4 and 8) the evaluation overhead would be very high and no additional realizations would be made. Figure 5.25 shows that core with the highest priority (core 16) accumulated the least number of aborted transactions for every benchmark accept vacation. Here the evaluation showed that a non-transactional execution (false conflict) lead to the abort of a transaction running on core 16.

Although one would expect that the number of aborted transactions would increase, the lower the priority of the core is, that does not necessarily occur. The number of aborted transactions highly depends on how many of the transactions a core executes conflict. Therefore, a higher prioritized core can accumulate more transactional aborts than a lower prioritized core. This phenomenon can be observed for all benchmarks, since for every benchmark there exists at least one core with a higher priority than another core which accumulated a higher number of aborts.

Since e.g., the benchmark *labyrinth* executes a lot of unbounded transactions, which have the highest priority, also transactions running on the the highest priority core (core 16) are aborted. Also, false conflicts may cause a transaction running on the core with the highest priority to abort. Therefore, they also accumulate aborts.

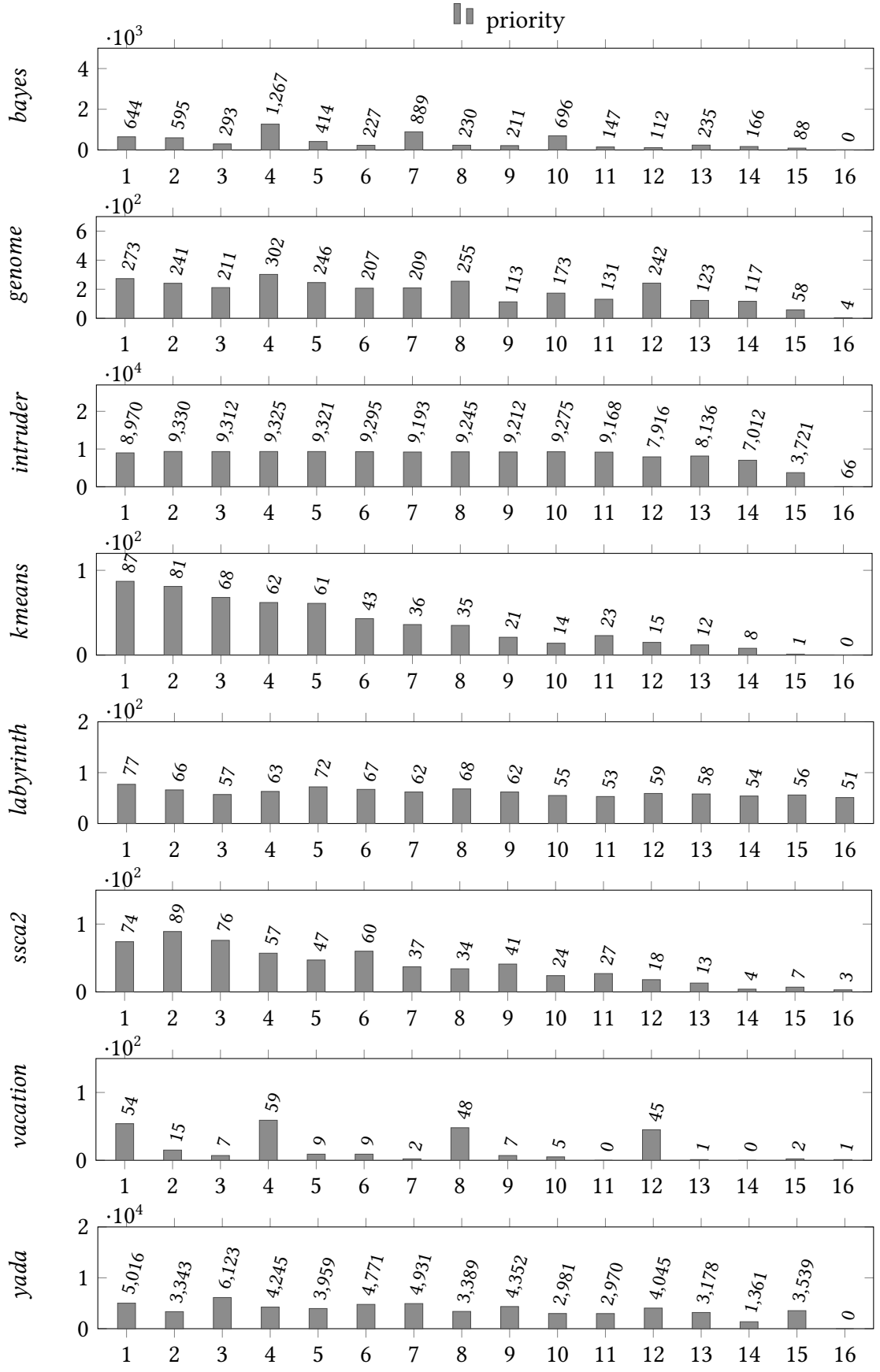

Figure 5.25: Number of aborts per core when executing the extended HTM with 16 cores, applying the *priority* contention management strategy as well as unbounded transactions.

5.9.10 Abort-Aware Execution

For the benchmarks *bayes*, *genome*, *intruder* and *yada* the number of aborted transactions is very high for the execution performed on the extended HTM applying the *timestamp* contention management strategy (see Figure 5.15 for more details). To reduce the numbers we described a mechanism called abort-aware execution which allows to lower the number of aborts (see Section 5.4 for more details). Lowering the number of aborts is beneficial, since a core otherwise has to restart a transaction over and over again. This is bad for energy consumption especially if the core has no chance of committing the transaction.

Figure 5.26 shows bar charts for the eight STAMP benchmarks. The bar charts allow the comparison of the number of aborts for the baseline HTM, the extended HTM applying the *timestamp* strategy and the extended HTM applying the *timestamp* strategy as well as activating the mechanism to reduce aborts. The x-axis of the bar charts represents the number of cores. The y-axis indicates how many transactions were aborted. The legend entry *baseline* refers to the execution performed on the baseline HTM. The legend entries *timestamp* and *timestampReduceAborts* refer to the execution performed on the extended HTM applying the *timestamp* strategy. The execution refered to by *timestampReduceAborts* also activates the mechanism to reduce aborts.

For the three benchmarks *bayes*, *genome* and *yada* the mechanism is effective and the number of aborts can be brought below the values achieved by the execution on the baseline HTM and the extended HTM only applying the *timestamp* strategy. Lowering the number of aborts only has marginal effects on performance. This makes sense, since the execution of aborted transactions is paused because a conflicting transaction is currently running. Note that although the mechanism lowers the number of aborts it does not necessarily lower the overall contention.

The benchmark *intruder* also profits from the mechanism compared to when only the *timestamp* strategy is applied. The problem is that contention still remains so high that the number cannot be brought below the number of aborts generated by the baseline HTM.

For the remaining benchmarks (*kmeans*, *labyrinth*, *ssca2* and *vacation*) the number of aborts matched or could be slightly decreased compared to the number of aborts accumulated by the extended HTM only applying the *timestamp* strategy. For seven executions (*labyrinth* (16 cores), *ssca2* (4 and 16 cores), *kmeans* (4,8 and 16 cores) and *vacation* (4 cores)) the number of aborts slightly increased. Since the order of how the transactions are executed might be affected through the mechanism an execution pattern might occur which causes more aborts. In general though the mechanism is powerful and allows the extended HTM to effectively reduce aborts.

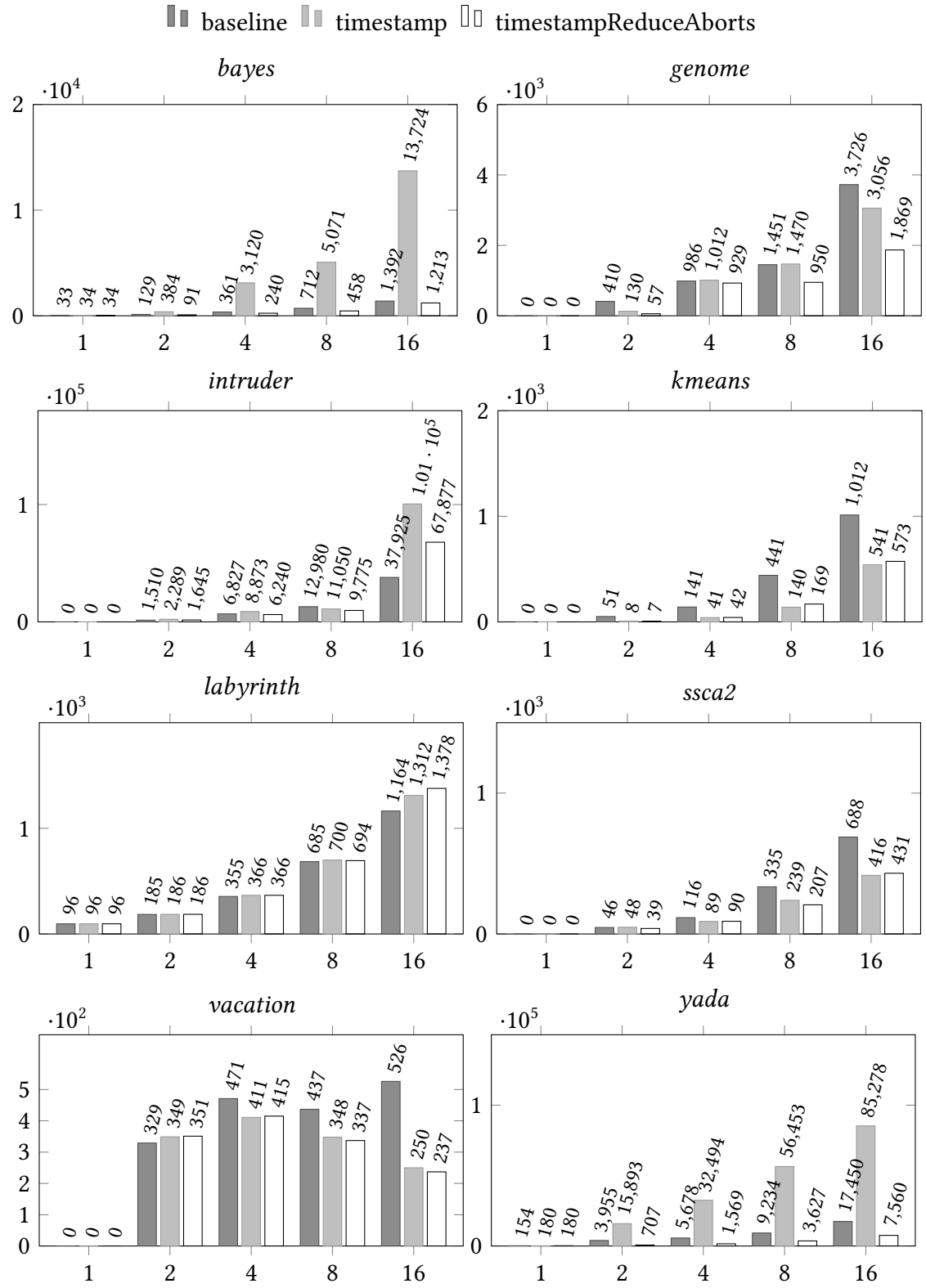

Figure 5.26: The bar charts compare the number of aborts when executing the STAMP benchmarks on the baseline HTM and the extended HTM applying the *timestamp* strategy as well as the mechanism to reduce aborts.

5.9.11 Reducing False Conflicts

With the extended HTM we are able to reduce the number of false conflicts. To show how effective the mechanism is we compare two executions on the extended HTM. For both executions we applied the *timestamp* strategy and unbounded transactions. For one we additionally applied the mechanism to reduce false conflicts. Note that we do not provide a fallback path for the benchmarks executed (see Section 5.8 for more details). The comparison is depicted by Figure 5.27.

Figure 5.27 shows eight bar charts. Every bar chart represents one of the eight STAMP benchmarks. The y-axes refer to the number of false conflicts. The x-axes refer to the number of cores the execution was performed with. The number on top of every bar represents the exact number of false conflicts. The legend entry *timestampUnbounded* refers to the execution where the extended HTM applies the *timestamp* strategy and unbounded transactions. The legend entry *timestampUnboundedRFC* refers to the execution where additionally the mechanism to reduce false conflicts is activated.

For every benchmark, except the benchmarks *labyrinth* and *yada*, we were able to reduce the number of false conflicts. The benchmark *labyrinth* in general does not suffer from any false conflicts. Therefore, the number of false conflicts could here not be reduced. For the benchmark *yada* the number of fallback executions could also not be lowered. Instead, here one additional false conflict was accumulated by our measures (execution with 8 cores) since a shift in the execution of the transactions occurred.

Every other benchmark generates false conflicts which our mechanism can eliminate up to 100 %. The complete reduction of false conflicts for every core count can be observed for the benchmarks *ssca2* and *kmeans*. For the benchmarks *bayes*, *genome*, *intruder*, *kmeans* and *vacation* a significant reduction of false conflicts was achieved. This ranges from 55% to 100% depending on the number of cores the execution was performed with.

The reduction of the false conflicts only had a small impact on the performance, since the number of false conflicts is quite low compared to the overall accesses. Therefore, we do not provide a detailed analysis of the speedups. In general, the mechanism has proven to work and is able to reliably reduce the number of false conflicts.

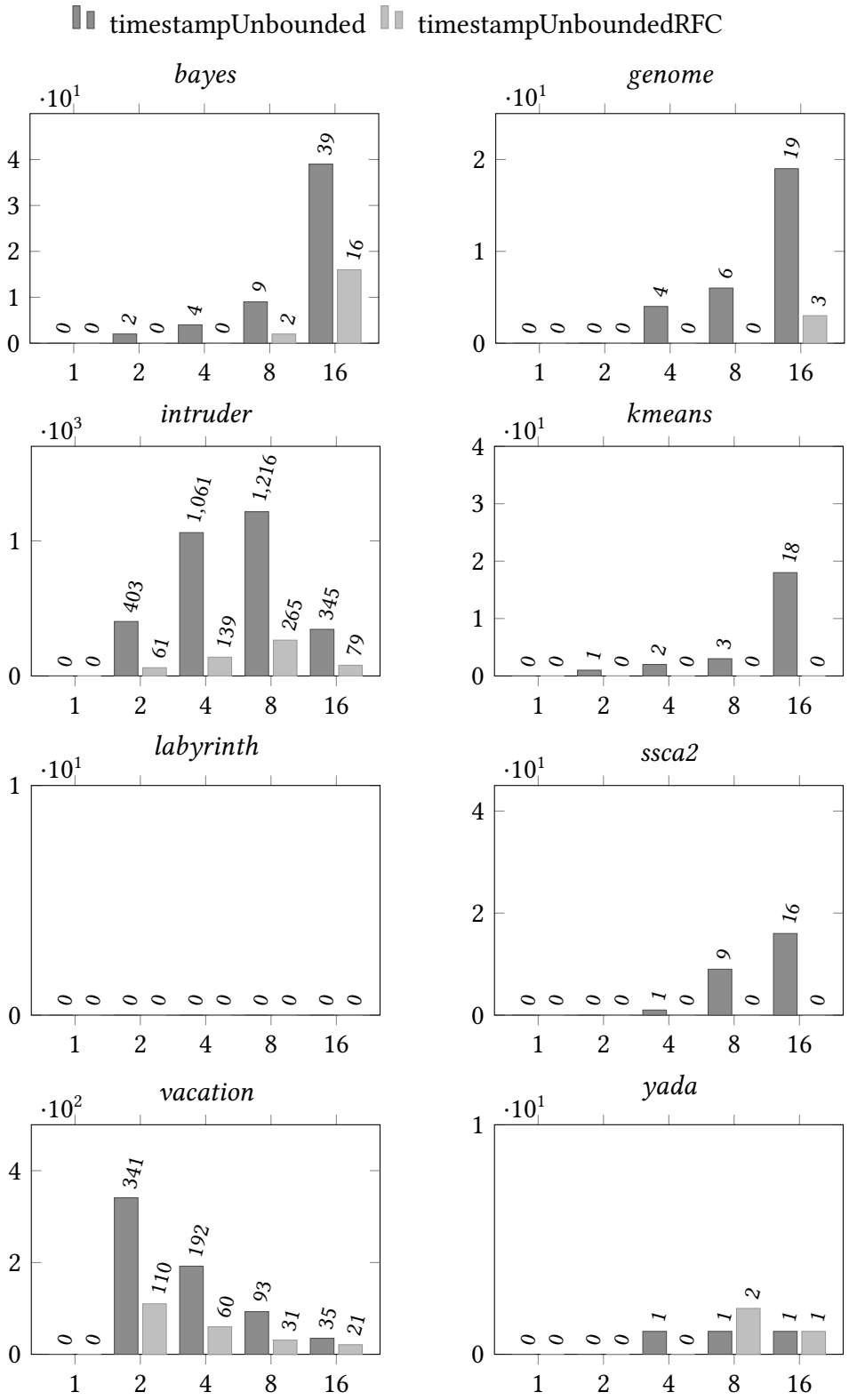

Figure 5.27: The mechanism to reduce false conflicts works effectively. For all benchmarks (except *yada*) contained in the STAMP benchmark suite the number of false conflicts could be reduced.

5.10 Summary

This chapter describes the development of the extended HTM. Since we aim at improving embedded systems, we provide an HTM allowing extensive control over the transactional execution. Therefore, the baseline HTM developed in Chapter 4 was adapted. Here we shifted conflict detection and resolution to the LLC controller. Therefore, mainly the cache controllers had to be adapted. In total 14 new coherence messages had to be added to map the communication caused by the shift. Shifting conflict detection and resolution to the LLC enables the implementation of contention management strategies. In total five strategies are provided by our approach. Additionally, we enabled a form of unbounded transactions allowing transactions to survive capacity conflicts. Our approach also allows other techniques which e.g., lower the number of aborts or false conflicts. Since the extended HTM also provides priorities, the interface was adapted to allow potential users to set a priority for every transaction. The extended HTM also allows simplifying the calculations for estimating the WCET. With the evaluation at the end of this chapter, we show that our approach can reduce abort rates, is able to reduce false conflicts and can eliminate the need for a fallback path. This is all done at acceptable performance. For some benchmarks performance was even increased.

6

Summary and Conclusion

Contents

In the following we summarize the work presented in this thesis. Next, we describe future work. Here we lay out the features and customizations that we would consider useful but are not yet realized in the current version of the HTM for embedded systems. To close this chapter, we summarize the findings of this work and provide an outlook.

6.1 Summary

The thesis starts with an introduction where we offer an overview of how our work is structured. Here we also motivate our work and lay out the objectives of our work. Before summarizing and laying out related work, we provide some background to explain the underlying mechanisms of the work done in this thesis.

For our work we first implemented an HTM into the gem5 simulator exploiting the MOSI cache coherence. We therefore first provide a detailed description of how we consider the setup of the underlying hardware.

After we explain how cache coherence is implemented, we lay out the design choices we made. Here we point out how we realized conflict detection, conflict resolution and Versioning. For our HTM we chose to implement eager conflict detection. The potential

conflicts are hereby detected by one of the L1 cache controllers. A conflict is detected if a transaction's RS or WS overlaps with another transaction's WS. To resolve the conflict, we picked a simple conflict resolution strategy which implies that the transaction running on the corresponding core of the cache controller which detected the conflict has to abort. To allow multiple versions of a cache line, we buffer written cache lines in the L1 cache.

Next, we explain how we actually implemented the HTM. First, we here describe how we extended the caches to manage the RSs and WSs. We continue by laying out how we adapted cache coherence by adding additional messages. Then we give a detailed description of how we adapted the cache controllers, so they offer the necessary functionality to enable the execution of transactions.

After describing the interface, we continue by evaluating our HTM implementation. Here we use the eight benchmarks of the STAMP benchmark suite to investigate how our implementation performs. We here compare our HTM implementation to an execution of the STAMP benchmarks using POSIX Thread synchronization. Overall, we were able to show that the transactional execution performs well since the HTM can achieve a speedup of almost 16 when executing the benchmark *kmeans* with 16 cores. For the benchmarks which benefit from being executed with transactions our HTM also achieves decent speedups, which we discuss in detail. We also lay out why the execution using the HTM was beaten by the POSIX Thread synchronization for some benchmarks. This mainly relates to transactions exceeding resources which are not supported by our HTM.

Next, we focused on how we can adapt and augment the baseline HTM to offer the functionality we believe should be available in embedded systems. The main change performed here is that in the newly designed HTM conflicts are detected at the LLC level. This allows a central point of control and enables us to implement a variety of useful features (e.g., multiple contention management strategies, abort-aware execution, etc.). We continue by laying out how we manage the RSs and WSs. This has to be done locally (L1 cache level) as well as at the LLC level. The RSs and WSs managed on LLC level are used to detect conflicts. The RSs and WSs managed locally are used to prevent the LLC controller of having to walk through the entire LLC to clear the RS and WS of a transaction after a commit or abort. Since shifting the conflict detection to the LLC level requires more communication we added 14 new message types to the coherence protocol. To handle all extra communication, we also extensively adjusted the cache controllers. Especially the LLC controller had to be adapted, since it remained untouched for the implementation of the baseline HTM.

After explaining how the interface works, we proceed to describe how we provide *abort-aware transactional execution*. The feature yields at lowering abort rates to improve energy consumption. The main idea behind this option is to stall a transaction which had a conflict with another transaction and therefore aborted, until the conflicting transaction

finishes. This reduces the number of aborts since a transaction only aborts once due to a concurrent transaction and not multiple times.

Next, we lay out how the mechanism which supports transactions of arbitrary size works. The mechanism supports one unbounded transaction at a time. An unbounded transaction is started as soon as a transaction evicts a cache line which is contained in the currently running transaction's RS or WS. Every conflict of the unbounded transaction with a concurrent transaction is resolved in favor of the unbounded transaction.

To offer a flexible HTM regarding contention management, we offer five conflict resolution policies. Except for one (strategy *passive*: it aborts the transaction which detects the conflict), the strategies we implemented focus on performance, progress as well as fairness. We implemented strategies which consider the start times, the number of commits, the number of aborts and priorities of transactions.

Since we provide an HTM for embedded systems we look at the analyzability of our HTM. Because of the nature of optimistic synchronization, it is quite difficult to analyze an execution which uses an HTM for synchronization. Nevertheless, we were able to show that we can theoretically bound the execution time for our HTM to match the execution as if the transactions were executed sequentially on a single core chip. This is achieved by using unbounded transactions as well as a sufficient contention management strategy such as *timestamp*.

Before concluding the chapter by summarizing it, we extensively evaluate our work. We here first describe where potential overheads come from, which are mainly caused by the additional communication necessary when shifting conflict detection to the LLC level. Next, we evaluate the contention management strategies and compare them to the executions performed on the baseline HTM. Here we are able to show that we were able to significantly reduce the number of fallback executions. When we additionally enable unbounded transactions also performance was improved for some benchmarks. Furthermore, the results show that we are able to completely eliminate the necessity of fallback paths. Since the contention management strategy which is able to handle priorities uses unbounded transactions it is evaluated next. Here we are able to show that the execution behaves as expected since the priority impacts the distribution of aborts. For the *abort-aware execution* we were able to show that it works, since the abort rates could be reduced for the benchmarks *bayes*, *genome*, *intruder* and *yada*. Last, we evaluate the mechanism which was implemented to reduce false conflicts. The evaluation showed that the mechanism works effectively since the number of false conflicts was reduced for every benchmark except one that was affected by false conflicts.

6.2 Future Work

Our HTM cannot allow transactions to survive interrupts. Therefore, a mechanism could be added to our HTM allowing transactions to survive interrupts. This would involve to provide the ability to exactly track the RSs and WSs of the transactions started. Even though transactions may currently not be running conflicts still have to be detected and resolved. Allowing transactions to survive interrupts would further increase usability of our HTM for embedded systems.

It might be possible to further optimize our HTM by reducing the communication needed to assure cache coherence. This would most likely result in a performance gain and would therefore be beneficial for every transactional execution.

State machines providing cache coherence are complex and therefore error prone. To ensure the correctness of the implementation of our cache coherence a model check could be performed.

The mechanism allowing to execute transactions of arbitrary size could be augmented in a way that multiple unbounded transactions could be executed simultaneously. To enable this feature the system has to be adapted in a way, so it is able to abort unbounded transaction. This requires that the system has to be able to keep track of the RS and WS even though some lines may have been evicted from the local caches.

The analyzability of the HTM could be further improved. Here the goal would be to bound the execution time below the time needed for the transactions to be executed sequentially.

Since we did not operate the HTM in combination with an operating system there are some adaptions which have to be performed to allow the HTM to work in combination with an operating system of choice (e.g. an operating system especially developed for embedded systems). Therefore, the HTM could be adjusted, so it can be used in combination with an operating system. The operating system then could be used to interact with the HTM to provide more functionality.

To further investigate how the HTM behaves it could be ported to a Field Gate Programmable Array (FPGA). An FPGA implementation allows a faster execution. Therefore, a more accurate evaluation could be made since the FPGA is one step closer to real hardware. Additionally, an FPGA implementation would help to demonstrate that the underlying concepts of this work can actually be implemented in real hardware.

6.3 Conclusion and Outlook

Offering more control over the transactional execution comes at a price. The increased costs hereby mainly relate to the additional communication necessary to gain the extra control needed to enable the desired functionality. Also, the complexity of the hardware (in our case cache the controllers) increases.

Nevertheless, we were able to show that the features, we believe contribute to embedded systems, could be enabled. Therefore, we provide a mechanism to allow transactions of arbitrary size. To provide more flexibility concerning contention management we implemented five contention management strategies. Even though we had to perform many adjustments our HTM offers acceptable performance. When combining unbounded transactions and contention management strategies we were able to even increase performance for some benchmarks when compared to the HTM developed in Chapter 4. Additionally, the necessity for a fallback path could be eliminated. Our mechanism meant to provide power saving options works since it is able to reduce the number of aborts. Furthermore, we present a mechanism which reduces the number of false conflicts. By providing unbounded transactions and sufficient contention management strategies we were able to bound the execution time for our HTM. Since mainly the memory hierarchy was adapted the cores themselves did not have to be changed which makes the system very portable. Other processors would most likely only need to make small adjustments to be able to use our HTM implementation.

Due to the fact that multi-cores become highly relevant for embedded systems, it becomes more important to write parallel software to exploit the available computational power. For this HTMs should certainly be considered. Especially for workloads which operate on a big amount of data and therefore only have a few sporadic conflicts HTMs perform well. With our work we provide an HTM suitable for embedded systems since it eliminates the typical limitations of conventional HTMs, offers several conflict resolution mechanisms and supplies a feature to conserve energy.

Bibliography

[1] Allon Adir et al. "Verification of Transactional Memory in POWER8". In: *Proceedings of the 51st Annual Design Automation Conference*. DAC '14. 2014, pp. 1–6. DOI: 10.1145/2593069.2593241.

[2] Rico Amslinger. "Loosely-coupled fail-operational execution on embedded heterogeneous multi-cores". doctoralthesis. Universität Augsburg, 2021, p. 193.

[3] C. S. Ananian et al. "Unbounded transactional memory". In: *11th International Symposium on High-Performance Computer Architecture*. Feb. 2005, pp. 316–327. DOI: 10.1109/HPCA.2005.41.

[4] ARM Ltd. *Transactional Memory Extension (TME) intrinsics*. Accessed: 2022-11-28. URL: https://developer.arm.com/documentation/101028/0012/16--Transactional-Memory-Extension--TME--intrinsics.

[5] N.L. Binkert et al. "The M5 Simulator: Modeling Networked Systems". In: *IEEE Micro* 26.4 (2006), pp. 52–60. DOI: 10.1109/MM.2006.82.

[6] Nathan Binkert et al. "The Gem5 Simulator". In: *SIGARCH Comput. Archit. News* 39.2 (2011), pp. 1–7. DOI: 10.1145/2024716.2024718.

[7] Colin Blundell, E. Christopher Lewis, and Milo M.K. Martin. "Subtleties of transactional memory atomicity semantics". In: *IEEE Computer Architecture Letters* 5.2 (2006), pp. 17–17. DOI: 10.1109/L-CA.2006.18.

[8] Harold W. Cain et al. "Robust Architectural Support for Transactional Memory in the Power Architecture". In: *SIGARCH Comput. Archit. News* 41.3 (June 2013), pp. 225–236. DOI: 10.1145/2508148.2485942.

[9] John M. Calandrino et al. "LITMUS^{RT} : A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers". In: *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*. 2006, pp. 111–126. DOI: 10.1109/RTSS.2006.27.

[10] Calin Cascaval et al. "Software Transactional Memory: Why is It Only a Research Toy?" In: *Commun. ACM* 51.11 (Nov. 2008), pp. 40–46. DOI: 10.1145/1400214.1400228.

[11] Luis Ceze et al. "Bulk Disambiguation of Speculative Threads in Multiprocessors". In: *Proceedings of the 33rd Annual International Symposium on Computer Architecture*. ISCA '06. 2006, pp. 227–238. DOI: 10.1109/ISCA.2006.13.

[12] Shailender Chaudhry et al. "Rock: A High-Performance Sparc CMT Processor". In: *IEEE Micro* 29.2 (2009), pp. 6–16. DOI: 10.1109/MM.2009.34.

[13] Chi Cao Minh et al. "STAMP: Stanford Transactional Applications for Multi-Processing". In: *2008 IEEE International Symposium on Workload Characterization*. Sept. 2008, pp. 35–46. DOI: 10.1109/IISWC.2008.4636089.

[14] Dave Christie et al. "Evaluation of AMD's Advanced Synchronization Facility within a Complete Transactional Memory Stack". In: *Proceedings of the 5th European Conference on Computer Systems*. EuroSys '10. 2010, pp. 27–40. DOI: 10.1145/1755913.1755918.

[15] Jaewoong Chung et al. "ASF: AMD64 Extension for Lock-Free Data Structures and Transactional Memory". In: *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. 2010, pp. 39–50. DOI: 10.1109/MICRO.2010.40.

[16] Intel Corperation. *Intel architecture instruction set extensions programming reference. Chapter 8: Intel transactional synchronization extensions*. 2012.

[17] R.H. Dennard et al. "Design of ion-implanted MOSFET's with very small physical dimensions". In: *IEEE Journal of Solid-State Circuits* 9.5 (1974), pp. 256–268. DOI: 10.1109/JSSC.1974.1050511.

[18] Marco Elver and Vijay Nagarajan. "RC3: Consistency Directed Cache Coherence for x86-64 with RC Extensions". In: *2015 International Conference on Parallel Architecture and Compilation (PACT)*. 2015, pp. 292–304. DOI: 10.1109/PACT.2015.37.

[19] Marco Elver and Vijay Nagarajan. "TSO-CC: Consistency directed cache coherence for TSO". In: *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. 2014, pp. 165–176. DOI: 10.1109/HPCA.2014.6835927.

[20] Robert Ennals. *Software transactional memory should not be obstruction-free*. Tech. rep. Technical Report IRC-TR-06-052, Intel Research Cambridge Tech Report, 2003.

[21] Hadi Esmaeilzadeh et al. "Dark Silicon and the End of Multicore Scaling". In: *SIGARCH Comput. Archit. News* 39.3 (June 2011), pp. 365–376. DOI: 10.1145/2024723.2000108.

[22] Cesare Ferri et al. "Embedded-TM: Energy and complexity-effective hardware transactional memory for embedded multicore systems". In: *Journal of Parallel and Distributed Computing* 70.10 (2010), pp. 1042–1052. DOI: 10.1016/j.jpdc.2010.02.003.

[23] Cesare Ferri et al. "Energy Efficient Synchronization Techniques for Embedded Architectures". In: *Proceedings of the 18th ACM Great Lakes Symposium on VLSI*. GLSVLSI '08. 2008, pp. 435–440. DOI: 10.1145/1366110.1366213.

[24] Keir Fraser. *Practical lock-freedom*. Tech. rep. UCAM-CL-TR-579. University of Cambridge, Computer Laboratory, Feb. 2004. DOI: 10.48456/tr-579.

[25] Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. "Toward a Theory of Transactional Contention Managers". In: *Proceedings of the Twenty-Fourth Annual ACM Symposium on Principles of Distributed Computing*. 2005, pp. 258–264. DOI: 10.1145/1073814.1073863.

[26] Florian Haas. "Fault-tolerant Execution of Parallel Applications on x86 Multicore Processors with Hardware Transactional Memory". doctoralthesis. Universität Augsburg, 2019, p. 164.

[27] Per Hammarlund et al. "Haswell: The Fourth-Generation Intel Core Processor". In: *IEEE Micro* 34.2 (2014), pp. 6–20. DOI: 10.1109/MM.2014.10.

[28] Tim Harris and Keir Fraser. "Language Support for Lightweight Transactions". In: *SIGPLAN Not.* 49.4S (July 2014), pp. 64–78. DOI: 10.1145/2641638.2641654.

[29] Tim Harris, James Larus, and Ravi Rajwar. *Transactional Memory, 2nd edition*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2010. ISBN: 1608452352.

[30] Tim Harris et al. "Transactional Memory: An Overview". In: *IEEE Micro* 27.3 (2007), pp. 8–29. DOI: 10.1109/MM.2007.63.

[31] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. The Morgan Kaufmann Series in Computer Architecture and Design. Elsevier Science, 2017. ISBN: 9780128119051.

[32] Maurice Herlihy. "Transactional Memories". In: *Encyclopedia of Parallel Computing*. Ed. by David Padua. Boston, MA: Springer US, 2011, pp. 2079–2086. ISBN: 978-0-387-09766-4. DOI: 10.1007/978-0-387-09766-4_122.

[33] 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*. 1993, pp. 289–300. DOI: 10.1145/165123.165164.

[34] Maurice Herlihy et al. "Software Transactional Memory for Dynamic-Sized Data Structures". In: *Proceedings of the Twenty-Second Annual Symposium on Principles of Distributed Computing*. PODC '03. 2003, pp. 92–101. DOI: 10.1145/872035.872048.

[35] Christian Jacobi, Timothy Slegel, and Dan Greiner. "Transactional Memory Architecture and Implementation for IBM System Z". In: *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. 2012, pp. 25–36. DOI: 10.1109/MICRO.2012.12.

[36] Jason Lowe-Power. *Ruby*. Accessed: 2022-11-28. URL: https://www.gem5.org/documentation/general_docs/ruby/.

[37] Jason Lowe-Power. *Syscall Emulation Mode (SE mode)*. Accessed: 2022-11-28. URL: https://www.gem5.org/documentation/learning_gem5/introduction/.

[38] Eric H Jensen, Gary W Hagensen, and Jeffrey M Broughton. *A new approach to exclusive data access in shared memory multiprocessors*. Tech. rep. Technical Report UCRL-97663, Lawrence Livermore National Laboratory, 1987.

[39] James Larus and Christos Kozyrakis. "Transactional Memory". In: *Commun. ACM* 51.7 (July 2008), pp. 80–88. DOI: 10.1145/1364782.1364800.

[40] Jason Lowe-Power et al. "The gem5 simulator: Version 20.0+". In: *arXiv preprint arXiv:2007.03152* (2020).

[41] Virendra J. Marathe, William N. Scherer, and Michael L. Scott. "Adaptive Software Transactional Memory". In: *Distributed Computing*. Ed. by Pierre Fraigniaud. 2005, pp. 354–368. DOI: 10.1007/11561927_26.

[42] Milo M. K. Martin et al. "Multifacet's General Execution-Driven Multiprocessor Simulator (GEMS) Toolset". In: *SIGARCH Comput. Archit. News* 33.4 (Nov. 2005), pp. 92–99. DOI: 10.1145/1105734.1105747.

[43] K. E. Moore et al. "LogTM: log-based transactional memory". In: *The Twelfth International Symposium on High-Performance Computer Architecture, 2006.* Feb. 2006, pp. 254–265. DOI: 10.1109/HPCA.2006.1598134.

[44] Christian Piatka et al. "Investigating transactional memory for high performance embedded systems". In: *International Conference on Architecture of Computing Systems*. Springer. 2020, pp. 97–108. ISBN: 978-3-030-52794-5.

[45] Fong Pong and Michel Dubois. "Verification Techniques for Cache Coherence Protocols". In: 29.1 (Mar. 1997), pp. 82–126. DOI: 10.1145/248621.248624.

[46] Ricard Quislant, Eladio Gutierrez, and Oscar Zapata Emilio L.and Plata. "Conflict Detection in Hardware Transactional Memory". In: *Transactional Memory. Foundations, Algorithms, Tools, and Applications: COST Action Euro-TM IC1001.* 2015, pp. 127–149. DOI: 10.1007/978-3-319-14720-8_6.

[47] R. Rajwar and J.R. Goodman. "Speculative lock elision: enabling highly concurrent multithreaded execution". In: *Proceedings. 34th ACM/IEEE International Symposium on Microarchitecture. MICRO-34.* 2001, pp. 294–305. DOI: 10.1109/MICRO.2001.991127.

[48] R. Rajwar, M. Herlihy, and K. Lai. "Virtualizing transactional memory". In: *32nd International Symposium on Computer Architecture (ISCA'05).* 2005, pp. 494–505. DOI: 10.1109/ISCA.2005.54.

[49] Ravi Rajwar and James R. Goodman. "Transactional Lock-Free Execution of Lock-Based Programs". In: ASPLOS X. 2002, pp. 5–17. DOI: 10.1145/605397.605399.

[50] J. Ruppert. "A Delaunay Refinement Algorithm for Quality 2-Dimensional Mesh Generation". In: *Journal of Algorithms* 18.3 (1995), pp. 548–585. DOI: https://doi.org/10.1006/jagm.1995.1021.

[51] Toufik Sarni, Audrey Queudet, and Patrick Valduriez. "Real-Time Support for Software Transactional Memory". In: *2009 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications.* 2009, pp. 477–485. DOI: 10.1109/RTCSA.2009.57.

[52] R.R. Schaller. "Moore's law: past, present and future". In: *IEEE Spectrum* 34.6 (1997), pp. 52–59. DOI: 10.1109/6.591665.

[53] William N. Scherer and Michael L. Scott. "Advanced Contention Management for Dynamic Software Transactional Memory". In: *Proceedings of the Twenty-Fourth Annual ACM Symposium on Principles of Distributed Computing*. 2005, pp. 240–248. DOI: 10.1145/1073814.1073861.

[54] William N Scherer and Michael L Scott. "Contention management in dynamic software transactional memory". In: *PODC Workshop on Concurrency and Synchronization in Java programs*. 2004, pp. 70–79. DOI: 10.1145/1073814.1073861.

[55] Martin Schoeberl, Florian Brandner, and Jan Vitek. "RTTM: Real-Time Transactional Memory". In: *Proceedings of the 2010 ACM Symposium on Applied Computing*. SAC '10. 2010, pp. 326–333. DOI: 10.1145/1774088.1774158.

[56] Nir Shavit and Dan Touitou. "Software transactional memory". In: *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*. 1995, pp. 204–213.

[57] Daniel J Sorin, Mark D Hill, and David A Wood. *A primer on memory consistency and cache coherence*. Synthesis lectures on computer architecture. Morgan & Claypool Publishers, 2020. ISBN: 1681737094.

[58] Nigel Stephens. *New Technologies for the Arm A-Profile Architecture*. Accessed: 2022-11-28. URL: https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/new-technologies-for-the-arm-a-profile-architecture.

[59] J.M. Stone et al. "Multiple reservations and the Oklahoma update". In: *IEEE Parallel & Distributed Technology: Systems & Applications* 1.4 (1993), pp. 58–71. DOI: 10.1109/88.260295.

[60] P. Sweazey and A. J. Smith. "A Class of Compatible Cache Consistency Protocols and Their Support by the IEEE Futurebus". In: *SIGARCH Comput. Archit. News* 14.2 (May 1986), pp. 414–423. DOI: 10.1145/17356.17404.

[61] Rubén Titos, Manuel E. Acacio, and José M. García. "Directory-Based Conflict Detection in Hardware Transactional Memory". In: *High Performance Computing - HiPC 2008*. 2008, pp. 541–554. ISBN: 978-3-540-89894-8.

[62] Amy Wang et al. "Evaluation of Blue Gene/Q Hardware Support for Transactional Memories". In: *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*. PACT '12. 2012, pp. 127–136. DOI: 10.1145/2370816.2370836.

[63] Luke Yen et al. "LogTM-SE: Decoupling Hardware Transactional Memory from Caches". In: *2007 IEEE 13th International Symposium on High Performance Computer Architecture*. 2007, pp. 261–272. DOI: 10.1109/HPCA.2007.346204.

List of Figures

List of Tables