Karlsruher Institut für Technologie

Institut für Technische Informatik
Lehrstuhl für Rechnerarchitektur und Parallelverarbeitung
Prof. Dr. rer. nat. Wolfgang Karl

Exploiting program phases in an FPGA-based Hybrid Transactional Memory system

Diplomarbeit
von

Philipp Kirchhofer

an der Fakultät für Informatik

Tag der Anmeldung: 01.03.2013
Tag der Fertigstellung: 30.08.2013

Aufgabensteller:
Prof. Dr. rer. nat. Wolfgang Karl

Betreuer:
Dr. Ing. Martin Schindewolf

Ich versichere hiermit wahrheitsgemäß, die Arbeit bis auf die dem Aufgabensteller bereits bekannte Hilfe selbständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderung entnommen wurde.

Karlsruhe, den 30.08.2013

Philipp Kirchhofer

Zusammenfassung

Über lange Zeit wurden Geschwindigkeitssteigerungen bei Prozessoren im Wesentlichen durch die Erhöhung der Taktfrequenz und durch die Optimierung der Mikroarchitektur erreicht. Dieser bislang beschrittene Weg ist nicht mehr wie im bisherigen Maße gangbar. Weitere Geschwindigkeitssteigerungen sind jedoch durch den Einsatz von Mehrkernarchitekturen erreichbar. Die Verteilung der Arbeitslast einer Anwendung auf parallel rechnende Kerne, das heißt die Parallelisierung der Anwendung, wird damit essentiell für einen hohen Durchsatz. Traditionelle Verfahren zur Programmierung von mehrfädigen Anwendungen sind schwierig zu erlernen, aufwendig in der Anwendung und eine bedeutende Quelle für Programmierfehler. Ein Programmier-Konzept für solche Anwendungen sollte deshalb im Interesse der Fehlerfreiheit einfach zu nutzen sein und eine hohe Rechengeschwindigkeit ermöglichen. Transactional Memory ist ein solches Konzept, mit dem diese Ziele für mehrfädige Anwendungen auf Mehrkernsystemen erreicht werden können.

Neuere Forschung hat gezeigt, dass einige Transactional Memory Anwendungen aus verschiedenen Phasen mit unterschiedlichen Charakteristiken (z.B. Verhältnis von abgebrochenen zu erfolgreichen Transaktionen) bestehen. Die vorliegende Diplomarbeit setzt an diesem Punkt an und zeigt auf, wie die Laufzeit solcher Transactional Memory Anwendungen durch die Anwendung von verschiedenen Transactional Memory Strategien verringert werden kann. Der vorgestellte adaptive Optimierungsprozess erlaubt eine dynamische Optimierung von Transactional Memory Anwendungen mit Programmphasen. Die Umschaltung der Strategie erfolgt dynamisch während der Laufzeit der Anwendung. Weiterhin wird eine Beobachtungsinfrastruktur entworfen, die die für die Analyse der Programmphasen einer Anwendung nötigen Informationen sammelt und für eine Auswertung, z.B. für die oben genannte dynamische Umschaltung der Transactional Memory Strategien, zur Verfügung stellt. Desweiteren wird die Umsetzung des vorgestellten Systems auf einem FPGA Board vorgestellt, Kriterien für den Entwurf der beteiligten Hardware-Komponenten erläutert sowie experimentell ermittelte Messergebnisse diskutiert.

Contents

List of Tables

List of Figures

1. Introduction

This chapter contains the motivation, the aims and the outline of this diploma thesis.

1.1. Motivation

Transactional Memory is a new paradigm for programming parallel applications, which tries to fulfill the promises of being easy to use for programmers while delivering good scalability and high performance. It tries to keep the additional complexity added by programming parallel applications low by providing advanced data access semantics to application programmers.

Recent research by Rossbach et. al. in *"Is transactional programming actually easier?"* [1] and by Pankratius et al. in *"A study of transactional memory vs. locks in practice"* [2] shows that using Transactional Memory simplifies the programming of parallel applications. A Transactional Memory runtime provides Transactional Memory semantics for an application. The runtime is needed to execute Transactional Memory applications. To get high performance and scalability for each application a programmer currently has to manually set the different strategies and settings for a Transactional Memory system. A strategy in a Transactional Memory system is, for instance, how and when to detect conflicts between transactions. A state-of-the-art mechanism is commit time locking (CTL), where conflict checking between transactions is deferred until commit time. The strategy operates under the optimistic assumption that two transactions will not conflict during runtime or at least not often. Another more pessimistic strategy is encounter time locking (ETL), which checks for conflicts before the transaction tries to commit by acquiring locks and holding them until commit time. CTL provides advantages in application phases with a low amount of contention between threads, whereas ETL is more suitable for phases with high contention.

An application programmer has to specify the value of the settings and which strategies are used before executing his Transactional Memory application. It is difficult to select a set of strategies and settings beforehand without further insight into the behavior of the application. The application programmer therefore has to select a set of strategies and settings though typically not having any knowledge about the transactional behavior of the application. Furthermore it is time intensive to select the best-suited settings and it requires

a thorough understanding of the interaction between an application and the Transactional Memory implementation. This contradicts the simplicity of the Transactional Memory programming model. Furthermore, some Transactional Memory applications exhibit a phased behavior, where transactional behavior, characterized by, for example, the level of contention, changes strongly during execution of the application. The periods of time with stable transactional characteristics are called program phases. Even if an optimal set of strategies and settings for the first program phase is selected at the very start of the application it can lead to decreased performance in program phases with a differing transaction behavior. As a consequence selecting the settings statically at compile time before executing the application comes with the disadvantage that the settings may not suit all program phases. This may lead to a suboptimal or even poor performance as a static strategy can not adapt to changing transactional characteristics.

1.2. Aims

The aims of this diploma thesis in order to contribute to the state-of-the-art research in this field are the following: First to show that some transactional memory applications exhibit program phases with differing transactional characteristics. Further to allow the exploiting of program phases in an FPGA-based Hybrid Transactional Memory system by designing and implementing an appropriate software and hardware framework. Additionally an adaptive process is designed and implemented, which allows to use the framework for dynamic optimization during runtime by switching on-the-fly between different Transactional Memory strategies. A further contribution in this thesis is to provide experimental results that show that the performance of phased transactional memory applications can be improved by using the adaptive process mentioned above.

1.3. Outline

This diploma thesis is structured as follows: Chapter 2 contains a short introduction to concurrent programming and Transactional Memory. It also shows related work on Transactional Memory, tracing and adaptive systems. The novel ideas of this diploma thesis are also explained there. The following chapter 3 introduces a state-of-the art Transactional Memory system. Chapter 4 presents the design for an adaptive Hybrid Transactional Memory System. Chapter 5 is focused on the implementation of the proposed design. Chapter 6 shows the results originating from this diploma thesis. The thesis ends with Chapter 7 by summarizing the results of this diploma thesis and presenting possible future extensions. The appendix contains additional information, a glossary and the bibliography of referenced papers.

2. Fundamentals and related work

The chapter introduces concurrent programming and commonly used techniques for programming parallel applications. Since Transactional Memory is used as the fundamental programming paradigm for parallel applications in this diploma thesis an introduction to it is included in this chapter. Further sections include an related work and a summary of the novel ideas in this thesis.

2.1. Introduction to concurrent programming

Until some years ago, the performance increase of mainstream processors was mainly achieved by increasing the processor frequency and, by a lesser degree, with micro-architecture optimizations. Increasing power consumption and declining performance advances between processor architecture steps made this approach infeasible to continue. For this reason current desktop processors have adopted a multi-core type architecture, where multiple processor cores are connected using an on-chip system interconnect. Ongoing industry expectations currently reach a 30 times performance increase in the next 10 years using this approach (for more information see Borkar et al. *"The Future of Microprocessors"* [3]), as the expectation is that the number of cores per chip will rise considerably in the future. This also means that sequential algorithms will not perform much faster in the future.

To obtain full performance on state-of-the-art multi-core architectures it is a requirement to transform a sequential algorithm into a parallel algorithm, where parts of the algorithm run concurrently. This is done by discovering which parts of the original algorithm can be executed concurrently and modifying them so that they can be run simultaneously on multiple processor cores. This is usually done by using threads. In an ideal case this approach can increase the throughput linearly by the number of used processor cores.

Common properties of parallel algorithms

Each thread has two data types, which are required for executing a parallel algorithm: private and shared data. Information for a generic algorithm can be usually divided into

input data, result or output data and state data, which stores the current progress in the algorithm execution. In combination both types contain the information required for the execution of the algorithm. Private data is information which is specific to a single thread and is not shared with other threads. Input data is usually read-only and distributed as private or shared read-only data to threads. Normal shared data on the other hand is, as its name implies, shared with other threads and used to communicate with other threads. The communication process is carried out by reading and writing the shared data from several threads.

A major distinguishing property of shared data in comparison to private data is that shared data can be read and modified simultaneously by different threads, as it is shared between simultaneously running threads. It is important to ensure a correct computation process by coordinating the access and modification of shared data in a responsible way. If this is not ensured situations could occur, where the output of an computation is dependent on the timing of other computations, and not on the flow of data, as originally intended by the programmer. These situations are called race conditions and generally lead to undesired non-deterministic results. They are caused when critical sections are not properly handled. A critical section is a part of an algorithm, that accesses and possibly modifies shared data that must not be concurrently accessed by more than one thread. This problem was first identified by Dijkstra in *"Solution of a problem in concurrent programming control"* [4].

Locks

A common approach to prevent race conditions is to use locks. Locks are a synchronization mechanism to restrict the access to shared resources. Locks can be used for all sorts of resources, like devices or in memory/on disk data. Before accessing a shared resource the associated lock has to be acquired, i.e. it is ensured that no other thread has currently locked it and the current thread is the only thread being in the process of acquiring the specific lock. After successfully acquiring the lock the requesting thread can continue accessing and modifying the data. Afterwards the thread releases access to the shared resource by unlocking the previously acquired lock and continues the computation process. A concurrent thread, which also wishes to access the same shared data which another thread has currently locked, runs through the same lock acquire process and is prevented from accessing the shared data. The access prevention is usually done by either running continuously in a loop until the access to the shared resource is possible again (spin-lock) or by blocking further computations, giving processor control back to the operating system or another thread and waiting for a signal to continue computation later on when access to the shared resource is free to acquire again.

But traditional programming of parallel applications using locks is complex and error-prone, as shown in *"Is transactional programming actually easier?"* by Rossbach et al. [1] and *"A study of transactional memory vs. locks in practice"* by Pankratius et al. [2].

A short excursion about why it is complex and error-prone: The use of locks can easily lead to undesirable situations like dead locks, where two threads need to acquire access to a resource the respectively other thread has already access to. As a result both threads can not progress further and the computation process involving these threads comes to a halt. This problem can be solved, as suggested by Dijkstra in [4], by ordering the shared resources and establishing a rule that all resources have to be acquired in this particular order. This commonly used approach has some limitations: Firstly it's application is limited, as the number of locks has to be known during design time and it also has to be fixed value. Secondly it is a complex procedure and therefore has difficult to implement correctly. This example shows that using locks forces to choose a trade-off between flexibility, complexity, scalability and performance.

Using locks also adds complexity to the design and implementation steps of programming concurrent applications. For instance during design a decision has to be made whether to use coarse- or fine-granular locks, or even a mixture of both. Coarse-granular locks protect complex compounded data structures as a whole, where as fine-grained locks protect the individual data fields, of which the data structure is composed. Coarse-grained locks reduce the number of locks needed and therefore reduces design complexity. However it also does increase contention, i.e. the frequency of acquiring a specific lock, and therefore decreases scalability and performance. Fine-grained locks on the other hand can be used to protect disjoint data fields in a data structure and allow threads to simultaneously access and modify these unrelated data sets. This approach reduces contention and provides better scalability, but it also increases the number of locks and thus the design complexity.

As a short summary it is safe to say that the complexity of an application directly correlates to the level of difficulty to correctly implement this application. Also to ensure that a complex parallel system works correct is inherently difficult due to its concurrency. To increase this difficulty by adding even more complexity (locks, correct use of locks) is certainly not an easy and promising approach.

Read-copy update

Special parallel algorithms working on shared data without employing traditional synchronization mechanism, so called lock-less or non-blocking algorithms [5], can be used in special corner cases. One of the most widely used algorithms of this type is read-copy update (RCU), as described by McKenney in *"Structured deferral: synchronization via procrastination"* [6] and Desnoyers et al. in *"User-Level Implementations of Read-Copy Update"* [7]. It is used in the network stack and memory management subsystem of newer versions of the Linux operating system kernel. RCU is used to protect very low contented mainly read shared data in linked lists by creating a separate copy ("new version") of a data structures whenever a modifying thread tries to make changes to the contents of the data structure. Simultaneously running threads reading from the same data structure

("old version") are therefore not affected by the modification and can continue unhindered. When the updating thread has finished its work the new version is put in place in the linked list by modifying the pointer from the previous node to the following node. The pointer originally pointed to the old version and after updating now points to the new version. This works correctly as RCU makes use of the fact that writes to aligned pointers are atomic in modern processor architectures. The now old version of the data structure is finally freed after all reading threads have finished access to it. This approach allows threads to read and modify shared data simultaneously without blocking any reader thread.

Transactional Memory

As seen in the previous paragraphs about locks it is hard for programmers to design and implement applications using locks correctly. A programming paradigm should therefore be easy to use for programmers, have a good scalability and deliver high performance. Transactional Memory, as proposed by Herlihy et al. in *"Transactional memory: architectural support for lock-free data structures"* [8], is a new paradigm trying to fulfill these promises. It tries to keep the additional complexity added by programming parallel applications low by providing enhanced semantics for data access. A central advantage of Transactional Memory is that the programmer specifies what should be done with shared data, rather than having him to specify exactly how the problem of concurrent access to shared data is handled. This approach relieves a programmer from the previously mentioned problem of trade-off selection and increased application complexity.

Transactional Memory introduces the concept of atomic blocks. These blocks guarantee atomicity, isolation and consistency. Changes on shared data are done at the end of an atomic block in an all-or-nothing fashion through implicit commit or abort operations (atomicity). A specific instance of execution of a atomic block is called a transaction. It is ensured that each atomic block has “seen” a consistent set of shared data during its lifetime (consistency) and is not allowed to modify the data of another concurrently running atomic block (isolation). Data read or written by an transaction is recorded in a read and write-set. A special handling procedure is invoked if these conditions are violated, e.g. an atomic block has operated on inconsistent data or it has modified data shared with another concurrently running atomic block. The procedure usually undoes the changes made by the atomic block and restarts the execution of the atomic block. This conflict detection treatment is transparent for the algorithm executing in the atomic block and is called an abort. The occurrence of aborts is expected during normal system operations. In contrast an commit is done when an atomic block finishes running without violating its conditions. Transactional Memory is an optimistic approach to parallel programming, as atomic blocks theoretically modifying the same shared data can be executed in parallel, in contrast to a implementation using locks. The atomicity, consistency and isolation guarantees needed for a correct application execution must be handled only if the atomic

blocks actually do modify the same data. This is usually done by aborting all except one of the conflicting atomic blocks and committing the remaining one.

The Transactional Memory semantics (atomic blocks) are usually provided by Transactional Memory framework libraries interfacing with an application. The framework libraries are independent from the algorithms employed in an application. This eases the implementation and testing of these libraries. It also allows applications requiring Transactional Memory semantics to rely on well-proven libraries.

Using Transactional Memory in applications

Traditional Locking

```
lock lock_a, lock_b;
[...]
lock(lock_a);
lock(lock_b);
a->cnt = b->cnt;
b->cnt++;
unlock(lock_a);
unlock(lock_b);
```

Transactional Memory

```
atomic {
    a->cnt = b->cnt;
    b->cnt++;
}
```

Table 2.1.: Comparison of traditional locking and Transactional Memory

Table 2.1 shows a comparison between implementing a critical section using locks and Transactional Memory. The shown program manipulates two objects. The implementation using locks has to handle several lock variables whereas the Transactional Memory implementation is very concise. The application programmer using locks has to handle an increased application complexity when compared to programming using Transactional Memory, as he has to use the locks in the right way (e.g. locking in a consistent order), because otherwise a deadlock can occur.

Software-, Hardware- and Hybrid Transactional Memory

Transactional Memory framework libraries can be implemented completely in software. This type of Transactional Memory is therefore called Software Transactional Memory (STM). In this case the employed algorithms for providing Transactional Memory semantics for an application are written to run using standard general-purpose processors.

Transactional Memory semantics can also be provided by hardware, usually done by extending the processor instruction set architecture (ISA). Special instructions are used

to indicate the begin and end of atomic blocks to the processor. Conflict detection and transactional reads and writes are done directly in hardware. Only a thin software layer is needed for better usability by an application. This type of a Transactional Memory system is called Hardware Transactional Memory (HTM). The execution speed of Transactional Memory applications is usually increased by this type of system when compared to an implemention solely in software (STM). Unbounded Hardware Transactional Memory systems allow the execution of arbitrary transactions, where as bounded Hardware Transactional Memory systems impose certain restrictions on the characteristics of transactions. They can have capacity contraints, e.g. the hardware can handle only transactions with a certain maximum read and write set size, or capability constraints, e.g. transactions can not call I/O operations. Transactions which are not supported by hardware therefore cannot successfully run in Hardware Transactional Memory mode and must be handled by other means.

Software Transactional Memory has the advantage of a flexible execution of transactions, where as Hardware Transactional Memory executes transactions faster. The advantages of both Software- and Hardware Transactional Memory can be utilized together by combining Software- and Hardware Transactional Memory. Such a system type is called Hybrid Transactional Memory.

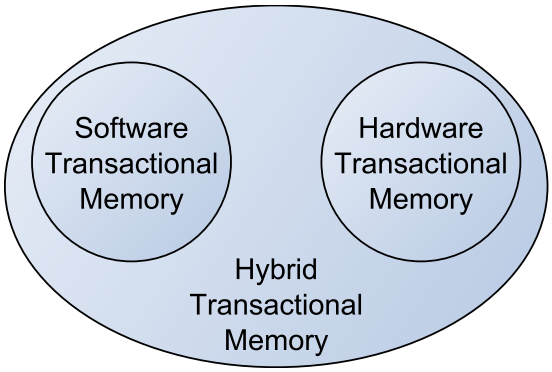


Figure 2.1.: Software-, Hardware- and Hybrid Transactional Memory

Figure 2.1 summarizes the dependency between the different types of Transactional Memory systems. Software- and (Pure) Hardware Transactional Memory systems can work standalone, where as a Hybrid Transactional Memory system depends on a implementation of both Software- and Hardware Transactional Memory.

2.2. Related work

This section summarizes current state-of-the-art research in Transactional Memory and related areas.

Transactional Memory on General Purpose Central Processing Units

Transactional Memory applications can be executed either either through Software or Hardware Transactional Memory support. There are generally two feasible approaches for Hardware Transactional Memory support: A light-weight approach adds special instructions to the processor ISA for a more efficient execution of Software Transactional Memory systems. This approach can be summarized as Hardware-assisted Transactional Memory. A more intrusive approach, in terms of structural changes to the processor design, adds new execution units and memory dedicated to Transactional Memory support directly to the processor core and consequently uses more hardware resources (Transistors, Logic routing, etc.). The main advantage of the second approach is to allow the fast execution of some Transactional Memory transactions (Hardware Transactional Memory mode) by providing Transactional Memory semantics directly in hardware.

Several proposals have been published for Transactional Memory support in next-generation processor architectures: AMD proposes the "Advanced Synchronization Facility" (see *"ASF: AMD64 Extension for Lock-Free Data Structures and Transactional Memory"* [9] by Chung et al.), an AMD64 hardware extension for lock-free data structures and Transactional Memory. Cache lines can be locked using specific instructions to facilitate the running of a fast ASF-STM system. An evaluation by Christie et al. in *"Evaluation of AMD's advanced synchronization facility within a complete transactional memory stack"* [10] observed that ASF-based Transactional Memory systems show very good scalability and much better performance than purely Software Transactional Memory based systems for the applications in the STAMP benchmark suite [11].

Intel's design "Hardware assisted Software Transactional Memory" (HASTM) (see *"Architectural Support for Software Transactional Memory"* [12] by Saha et al.) also takes the same approach by proposing changes in the processor ISA to speed up the execution of Software Transactional Memory runtime systems. This light-weight approach allows for a relatively non intrusive implementation in current processor cores, but also limits the possible acceleration.

The Transactional Memory implementation in Sun's Rock processor, as described by Dice. et al. in *"Early experience with a commercial hardware transactional memory implementation"* [13], takes on a hybrid approach by implementing the parts, which allow to accelerate the common case behavior of Transactional Memory applications, in hardware while at the same time supporting advanced Transactional Memory features in software. The design of this Transactional Memory implementation allows to take advantage of future processor architecture generations, where on each iteration a successively higher level of Hardware Transactional Memory support can be achieved.

The TMbox system, as presented by Sonmez et al. in *"TMbox: A Flexible and Reconfigurable 16-Core Hybrid Transactional Memory System"* [14], follows a different, more

heavy-weight approach. Entire transactions can be executed directly in hardware in a best-effort way. This means that certain restrictions of transactional characteristics (like size of read-/write-set, no I/O operations) have to be satisfied to allow a successful execution. The advantages are fast execution and, on the software side, decreased complexity because a Software Transactional Memory runtime is not necessarily needed. The design of the TMbox system is used as the underlying platform in this diploma thesis.

Some of these proposed changes are currently being implemented in commercially available processors, as published by Jacobi et al. *"Transactional Memory Architecture and Implementation for IBM System Z"* [15] for IBM System Z and by Wang et al. in *"Evaluation of Blue Gene/Q hardware support for transactional memories"* [16] for Blue Gene/Q. The ongoing research on Transactional Memory by nearly all major microprocessor companys indicates a certain possibility of seeing it in more future CPU architectures.

The Transactional Synchronization Extensions, as implemented by Intel in the current state-of-the-art Haswell processor family[1], supports a concept similar to Transactional Memory called Hardware Lock Elision [17]. Lock elision allows a thread to elide the acquisition of a lock by optimistically assuming that no other thread will use the lock. If later on the assumption proves to be wrong the thread is restarted at the lock-eliding instruction and a normal sequence of locking and unlocking takes place. Another ISA extension implemented by Intel called Restricted Transactional Memory looks similar to a stripped down subset of Hardware Transactional Memory, but a lot of care must be taken to get good performance out of this new technique, as shown by Wang et al. in *"Opportunities and pitfalls of multi-core scaling using hardware transaction memory"* [18].

Transactional Memory on Graphic Processing Units

An increasingly interesting new runtime environment for computation-intensive applications are state-of-the-art graphic processing units (GPUs) through the use of General Purpose Computation on Graphics Processing Unit (GPGPU) techniques. These GPUs use SIMD and massively multi-threaded execution to provide a high raw computing power. Recent non-graphic oriented programming APIs like OpenCL, DirectCompute and CUDA allow an adaption of applications to the special requirements of GPUs. But the conversion of applications using shared data to the specific features and requirements of an GPU is difficult: Barrier synchronization does slow down the system a lot, while the use of fine-grained locks is very difficult to implement correctly for more than 10,000 scheduled hardware threads.

Fung et al. address these issues in *"Hardware Transactional Memory for GPU Architectures"* [19] by proposing and simulating a GPU with Hardware Transactional Memory

[1] Intel: *Architecture Instruction Set Extensions Programming Reference*, pages 506 ff., http://download-software.intel.com/sites/default/files/m/3/2/1/0/b/41417-319433-012.pdf

support. They show that Hardware Transactional Memory on GPUs performs well for applications with low contention. Their proposed Transactional Memory design "KILO Transactional Memory" captures 59 % of the performance of an GPGPU programmed with fine-grained lockings and has an estimated hardware overhead of about 0.5 %.

Cederman et al. show a related feasibility study in *"Towards a Software Transactional Memory for Graphics Processors"* [20]: They use the unmodified hardware of a Nvidia GPU to run two variants of a Software Transactional Memory runtime environment. One variant is a simple, easy to implement Software Transactional Memory with low resource requirements, specifically designed for use in GPUs. The other Software Transactional Memory variant uses a more complex design oriented for general purpose multiprocessors. The results show increased performance and reduced abort rates when using the complex design.

The cooperation of CPU and GPU oriented Transactional Memory runtime environments remains an developing area: Future GPU architectures are going to acquire some high-level semantics from standard CPU architectures like virtual memory support and memory protection.

In a new development AMD currently brings system designs both based on a new shared memory architecture for CPUs and GPUs (hUMA[2]) and a standard for tight integration of heterogeneous processors (HSA[3]) into the consumer market. The hardware units called Accelerated Processing Units (APU) are going to be delivered in PCs and upcoming video game consoles (PlayStation 4 and Xbox One). This high-volume influx of ubiquitous heterogeneous multi-cores will surely be an attractive field of application for Transactional Memory research.

Characterization of Transactional Memory applications

All of these previously mentioned proposals show different environments for running Hardware Transactional Memory and Software Transactional Memory applications. To get a high computing performance it is essential to characterize Transactional Memory application behavior and adjust the internal parameters and algorithms of a Transactional Memory runtime environment accordingly. Multiple papers have been published about the characterization of Software Transactional Memory applications. Ansari et al. ported some applications from the STAMP benchmark suite to DSTM2, a Java-based Software Transactional Memory implementation with profiling features. The results are published in *"Profiling Transactional Memory Applications"* [21]. They used some well-known

[2] AMD Heterogeneous Uniform Memory Access, http://www.amd.com/us/products/technologies/hsa/Pages/hsa.aspx#3

[3] AMD Heterogeneous Systems Architecture, http://www.amd.com/us/products/technologies/hsa/Pages/hsa.aspx#2

metrics like speed up, wasted work and time in transaction to characterize the behavior of these applications. Some of the presented metrics can also be used as input for the decision making process, as proposed in this diploma thesis. Chung et al. present a comprehensive characterization study of the common case behavior of 35 multi-threaded applications in *"The Common Case Transactional Behavior of Multithreaded Programs"* [22]. The applications mostly originate from computational sciences and use a wide range of programming languages. Tracing markers were added to the applications and a trace with all executed instructions and tracing markers was collected for each application. The results show an interesting insight into the common case behavior of real world applications not directly designed for Transactional Memory. The Software Transactional Memory monitoring techniques and the metrics presented in these papers are, in general, transferable to other Transactional Memory variants, but the specific implementation of a monitoring infrastructure is different on Hardware Transactional Memory systems. One specific different aspect is the difference in processing speed of a Transactional Memory application running on a system with enabled or disabled monitoring. The processing speed of Transactional Memory applications running on a Software Transactional Memory runtime environment with enabled monitoring support is always slowed down due to the increased amount of computations done by the Transactional Memory system (e.g. generation and saving of traces). Monitoring support for an Hardware Transactional Memory system can, on the other hand, be implemented with low overhead, as shown in an related work by the author in [23].

The PhD thesis of Ferad Zyulkyarov *"Programming, Debugging, Profiling and Optimizing Transactional Memory Programs"* [24] does include an extensive introduction to various Transactional Memory runtime design patterns, functionalities and optimization opportunities. Topics also include debugging, profiling and optimization techniques. The profiling framework is based on the Bartok-STM system, an ahead-of-time C# compiler with Transactional Memory support. The aim of the developed techniques were to combine profiling work with the already existing C# garbage collector. The garbage collector runs at dynamic and non-deterministic time points during the application runtime. Application threads must be synchronized at these points. This behavior, inherent to managed programming languages with a garbage collector, changes the applications transactional behavior and characteristics when compared to an implementation in an unmanaged language with static memory management. The dynamic behavior also makes accurate monitoring and optimization harder. The Transactional Memory tracing techniques in the PhD thesis are therefore integrated into the garbage collector to allow a parallel execution of memory management and tracing algorithms and to prevent further transactional behavior changes. This helps to reduce the probe effect (i.e. the change of application behavior when enabling or disabling the generation of traces).

The monitoring techniques used in this diploma thesis are in some parts comparable to the Transactional Application Profiling Environment, as presented by Chafi et al. in *"TAPE: A transactional application profiling environment"* [25]. The TAPE system was simulated

using an execution-driven simulator, where as the system proposed in this thesis can both be simulated by software using a Hardware Description Language simulator and run in hardware (with a much higher speed) using an FPGA chip.

The transactional behavior of the application is gathered during runtime using an enhanced version of a low overhead profiling framework covering both Software and Hardware Transactional Memory modes, as originally described by Arcas and the author et al. in *"A low-overhead profiling and visualization framework for Hybrid Transactional Memory"* [23] and in the study thesis by the author *"Enhancing an HTM system with Hardware monitoring capabilities"* [26].

Adaptive systems

Current research by Payer et al. in *"Performance evaluation of adaptivity in software transactional memory"* [27] shows the benefits of having adaptivity in Software Transactional Memory runtime environments. Compared to the current state of the art this diploma thesis enhances the scope by running on a Hybrid Transactional Memory system, additionally accounting for both changing Software and Hardware Transactional Memory behavior.

Lev et al. describe a Hybrid Transactional Memory system in *"PhTM: Phased Transactional Memory"* [28], which analyzes the effectiveness of the Hardware Transactional Memory unit during runtime and falls back to a permanent software mode if it detects decreased performance through using the hardware unit. Their system was tested using a simulator and, in comparison to the adaptive system propose in this diploma thesis, does not adapt the strategies and settings of the Software Transactional Memory system.

Felber et al. describe dynamic tuning for the TinySTM Software Transactional Memory library in *"Dynamic Performance Tuning of Word-Based Software Transactional Memory"* [29]. They describe a dynamic adaption of various tuning parameters that affect the transactional throughput. The three described parameters are:

1. The hash function to map a memory location to a lock. TinySTM right-shifts the address and computes the rest modulo the size of the lock array. The number of right shifts allows controlling how many contiguous addresses will be mapped to the same lock. This parameter allows exploiting the spatial locality of the data structures used by an application.
2. The number of entries in the lock array. A smaller value will map more addresses to the same lock and, in turn, decrease the size of read sets. It can also increase the abort rate due to false sharing.
3. The size of the array used for hierarchical locking. A higher value will increase the number of atomic operations but reduce the validation overhead and potential contention on the arrays elements.

They use a hill climbing algorithm to randomly change one parameter at a time and measure the resulting throughput over a period of time. If the throughput increases the parameter is varied in the same direction at the start of the next period. When encountering decreased throughput the algorithm chooses a previously seen best configuration as a new base and restarts from there by choosing a new parameter to vary. This algorithm works unlike the adaption process presented in this diploma thesis without previously obtained knowledge on the exact effect of the adapted parameters on transactional throughput. The algorithm by Felber et al. works, on the other hand, only with applications having static unchanging transactional characteristics, as it provides no facility of detecting a major change in transactional characteristics and, in response, restarting the adaption process from scratch. These facilities are proposed in the design of the adaption process in this diploma thesis.

Other related work

Gottschlich et al. present a transactional memory profiler in *"Visualizing Transactional Memory"* [30]. They base their profiler on three visualization principles. The principles are the precise graphical representation of transaction interactions including cross-correlated information and source code, visualized soft real-time playback of concurrently executing transactions and dynamic visualizations of multiple executions. They note *"[...] that a TM profiler should be primarily visual, as graphical representation is the best way to convey complex interactions that unfold over time"*. The presented visualization post-processing steps of the event-based tracing framework for Hybrid Transactional Memory, as proposed in this diploma thesis, follow similar principles, but are based on prior work by the author, as published in [23, 26], preceding the publication of Gottschlich et al.

2.3. Novel ideas

The following novel ideas distinguish the work done in this diploma thesis from previous research:

- This thesis proposes a systematic approach for enabling the dynamic adaption of strategies and settings in a Hybrid Transactional Memory system. This approach detects and exploits program phases and improves the performance of Transactional Memory applications. The phase detection and switching algorithms are designed in a modular way providing high flexibility and exchangeability.
- An event-based tracing framework suitable for dynamically selecting appropriate Transactional Memory strategies during runtime depending on the current program phase is presented. Hardware units are used to ensure zero overhead when tracing transactions using Hardware Transactional Memory and a one cycle overhead per state change when tracing transactions using Software Transactional Memory.

3. TMbox: A Hybrid Transactional Memory System

The TMbox system, designed at the Barcelona Supercomputing Center (BSC), is used as the base implementation of a Hybrid Transactional Memory system for this thesis. It is an multiprocessor system on chip design and implementation built to explore trade-offs in multicore design space and to evaluate parallel programming methods like Transactional Memory. The system uses ring buses to connect a configurable number of MIPS R3000-compatible soft-core processors. The interconnect is based on a 2-way ring bus with an unidirectional data lane in each direction. This interconnect design offers the space and the flexibility to add, synthesize and determine the impact of new hardware components on application performance.

The following chapter provides an introduction to the TMbox design and summarizes key characteristics. Additional information about the TMbox design is available in *"TMbox: A Flexible and Reconfigurable 16-Core Hybrid Transactional Memory System"* [14] by Sonmez et al. and *"Resource-bounded multicore emulation using Beefarm"* [31] by Arcas et al.

3.1. System schematics

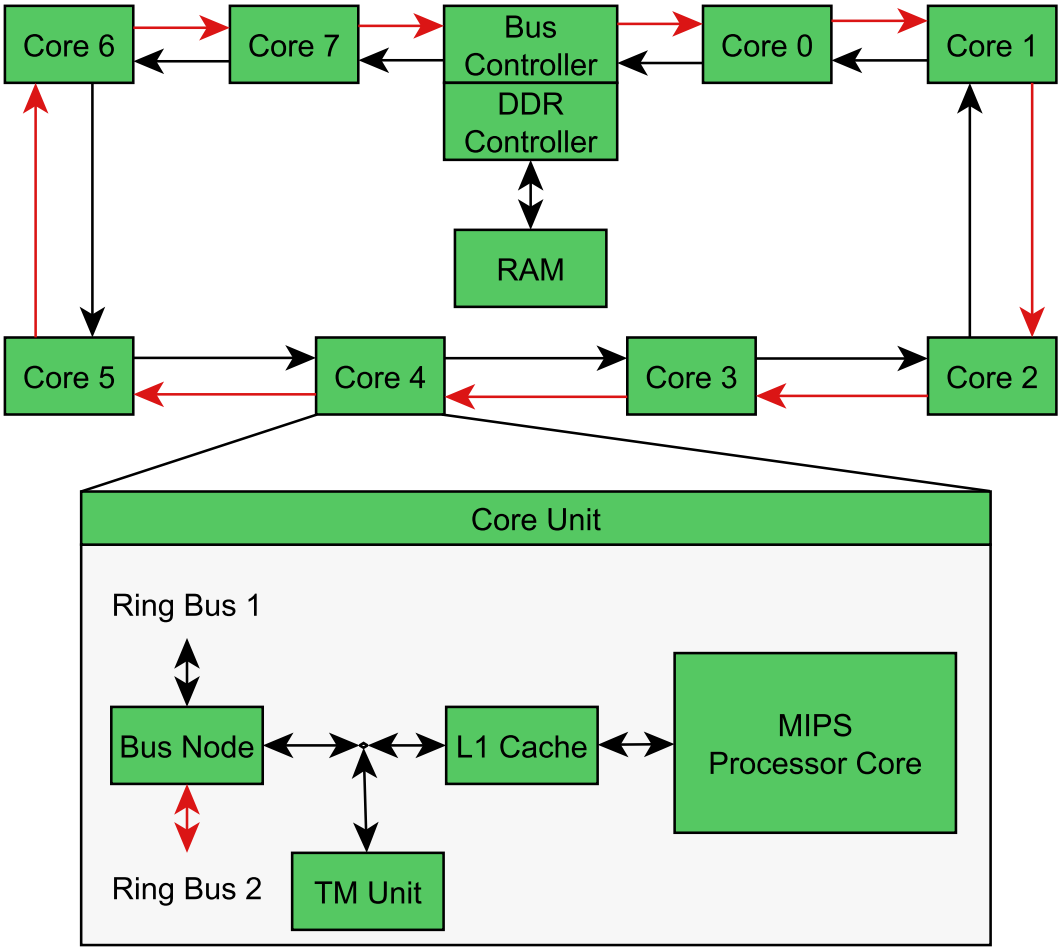


Figure 3.1.: 8 Core TMbox system block diagram

Figure 3.1 shows a high-level overview of the TMbox hardware components. The black ring bus transfers memory read/write requests and responses while the red ring bus transfers invalidation and event messages. The invalidation messages are used to coordinate cache and memory coherency between the participating processor cores.

The following paragraphs describe the units which were re-used from the TMbox system.

Bus Node

The bus node unit connects the processor core, L1 unit, TM unit and log unit to two ring buses. One ring bus transmits memory related messages, whereas the other ring bus

transmits invalidations and (added in this study thesis) events created by the monitoring infrastructure.

TM Unit

The Transactional Memory unit is necessary for supporting Hardware Transactional Memory. It contains the read- and write-set of the currently running transaction. Some Transactional Memory related parameters like read-/write-set size can be changed before synthesizing the system.

Bus Controller Unit

The bus controller unit forwards memory related messages received via the ring bus to the DDR controller for further processing. It also receives requested memory data from the DDR controller and sends it via the ring bus to the requesting core unit.

Core Unit

The processor core and associated units comprise a core unit. Every neighbouring core units is connected by the two ring buses. The first and the last core unit is connected to the bus controller unit. The number of core units in the TMbox system is variable.

3.2. Transactional Memory design and implementation characteristics

The following section reviews the design and characteristics of the software and hardware components of the TMbox system, which are involved when running Transactional Memory applications on the system. The properties of these components have a major influence on the achievable level of adaptivity and it is therefore important to accurately assess the impact of these properties on the general performance of Transactional Memory applications. This section discusses the impact of the component design on a design-level, where as section 6.1 determines the impact in an experimental way by proposing the use of a Transactional Memory benchmark application.

As introduced previously the support of Transactional Memory semantics can be provided via software, hardware or a combination of both. The following sections consequently describe all three options:

Software Transactional Memory (TinySTM)

The software, which provides Software Transactional Memory semantics for the TMbox system, is called TinySTM. It is an efficient word-based Software Transactional Memory implementation developed at the Universities of Dresden and Neuchâtel. The general design principles of TinySTM are shown by Felber et al. in *"Time-Based Software Transactional Memory"* [32].

Transactional Memory granularity

The granularity of Transactional Memory implementations can be either object- or word-granular. In the high grained case of object granularity a previously defined arbitrary object is accessed and modified in its entirety in a transactional and therefore atomic way. This means that each change to a field within the object marks the whole object as changed and prevents other threads from simultaneously making concurrent modifications to any field of the same object. This approach works fine in an application where fields, which are often modified concurrently, are densely packed in different objects. This ensures a low level of contention on these objects. The underlying approach has been published firstly by C.A.R. Hoare in *"Monitors: an operating system structuring concept"* [33].

The other case of low grained Transactional Memory granularity detects changes to fields of an object or a structure (in non OOP languages) on a word based granularity. This means that threads can successfully concurrently change fields of an object if the accessed fields are mutually exclusive.

Many data structures exhibit this behavior of changes to mutually exclusive internal fields when doing operations on different elements of the same data structure. For example when looking at a standard double-linked list a concurrent change of the left hand node and a change of the right hand node both modify fields of the same (middle) node, but both can also be run simultaneously when having word based granularity, which is not possible when having object granularity. For the sake of optimization the strict word based granularity is often weakened by combining multiple successive words into a region with an atomic behavior.

Transactional Memory snapshots

The set of fields read by a transaction is its read set and similarly the set of fields it writes is its write set. Invisible reads is a strategy, where reads of a transaction are tracked in its read-set, but not visible to other transactions. This improves the performance of a Transactional Memory environment, but special care has to be taken to prevent the reading of inconsistent data by concurrent transactions.

The TinySTM implementation uses a time-based approach called Lazy Snapshot Algorithm (LSA) to construct snapshots of the fields accessed by a transaction. The snapshot remains consistent throughout the whole lifetime of the transaction. The consistency is verified by checking the validity interval for snapshots and comparing them to the modification time stamp of accessed fields. The modification time stamp of a field is changed on each modification of the value of a corresponding field and is obtained from a time base, which is globally accessed by all threads. As shown in [32] this allows to efficiently verify the consistency of snapshots on each object access.

Transactional Memory strategies

Commonly used Transactional Memory related strategies can be grouped by when they do conflict detection and how to handle memory updates caused by transactional progress: Lazy and early conflict detection and buffered and non-buffered transactional updates.

TinySTM implements three different combinations of conflict detection and data versioning strategies, called designs: Write-back using commit-time locking (WB-CTL), write-back using encounter-time locking (WB-ETL) and write-through using encounter-time locking (WT).

Strategies using lazy conflict detection (also known as commit-time locking) defer the detection of conflicts between transactions to the end of a transaction. This implies that in this mode transactions always execute until a commit is requested (unless they are forcefully aborted from outside, for instance by a contention manager). The Transactional Memory implementation then checks at commit time if a conflict has happened during the transaction runtime and then responds by either committing a transaction in the case of no conflict or aborts the transaction when a conflict has happened. This type of lazy conflict detection keeps transactional overhead low, as a possible lengthy conflict detection algorithm has to be run only once in the entire transaction lifetime. A disadvantage of this setting is that the wasted work done by an aborted transaction is higher than when using early conflict detection, because the transaction always finishes doing work and then either commits or aborts, increasing the amount of wasted work and the time it takes to undo it.

Early conflict detection (also known as encounter-time locking) checks for conflicts during transaction lifetime, usually multiple times before a transaction requests a commit. A conflict is detected earlier than when using lazy conflict detection and therefore causes a faster abort-restart cycle. The amount of wasted work is decreased, as the transaction is aborted directly when a conflict is encountered and further useless work in the transaction, which would always be discarded as the result of the conflict, is prevented.

The data versioning strategies write-back and write-through differ in the way changes to data are written to memory. The updated data is stored in a redo log and written to memory

Name	Conflict detection	Update buffering	Amount of aborts	Wasted work in aborted Tx
WB-CTL	Lazy	Yes	Low	High
WB-ETL	Early	Yes	High	Low
WT	Early	No	High	Low

Table 3.1.: Summary of Transactional Memory strategies

upon commit when using write-back. With write-through updates are written directly to memory and the previous data values are stored in an undo log. The original data values are restored to memory in the case of an abort. Write-back has a lower abort overhead, as in the abort case no data must be written back to memory. Write-through has on the other hand a lower commit-time overhead, as data is directly written to memory during the runtime of a transaction. The commit phase is fast as no changed data has to be written to memory in this step.

The control and data flow when using these three commonly used designs are shown in the appendix in figures A.1 (Write-back using commit-time locking), A.2 (Write-back using encounter-time locking) and A.3 (Write-through using encounter-time locking). The control flow is denoted by solid lines in the figures, whereas data flow is denoted by dashed lines. Nodes representing actions related to transactional memory are drawn in a lighter color.

In general, we can say that both lazy conflict detection and non-buffered transactional updates are optimistic methods, which optimize for the case of a successful transaction commit. These methods are especially applicable when an application exhibits high parallelism and a small rate of conflicts. Early conflict detection and buffered transactional updates are, on the other hand, ideal for pessimistic cases, where a high rate of aborts slows down transactional progress. Applying these methods keeps the transactional overhead low when having a high level of contention. The characteristics of the three different strategies are summarized shortly in Table 3.1.

Hardware Transactional Memory (BeeTM)

TMbox supports Hardware Transactional Memory through the addition of special instructions to the processor ISA. These instructions indicate a transaction start or commit to the Transactional Memory hardware unit, which is located in each processor. The software using this type of Transactional Memory transactions has to use special read and write instructions. These instructions automatically update the read and write set and check for conflicts during the runtime of a transaction. The read and write set of a transaction is stored in dedicated hardware memory units, directly located in the Transactional Memory

hardware unit. A transaction in Hardware Transactional Memory mode is aborted either implicit, whenever the Transactional Memory hardware unit detects a conflict with another concurrently running transaction, or explicit by executing an abort instruction. The strategy used by the Transactional Memory hardware unit is write-back with encounter-time locking.

A thin software layer called BeeTM allows to execute transactions in pure Hardware Transactional Memory mode. No Software Transactional Memory implementation is needed is this case. Another mode of operation is Hybrid Transactional Memory. This mode combines Software and Hardware Transactional Memory and is explained in the following paragraphs.

Hybrid Transactional Memory (HyTM)

A Hybrid Transactional Memory runtime, which provides Transactional Memory semantics to applications by utilizing both software and hardware components, has been designed and implemented in the scope of the VELOX project [34]. The VELOX Hybrid Transaction Memory implementation is based on an old version of TinySTM. The old version has, in its unmodified VELOX variant, the restriction that one Transactional Memory design has to be picked at compile time. The picked design can not be exchanged later on during runtime. This prevents the dynamic adaption of Transactional Memory strategies depending on application behavior during runtime, which is a major goal of this thesis. A proof-of-concept modification of TinySTM, which removed this constraint, was developed at the beginning of this diploma thesis and the preliminary results were presented at the Euro-TM Workshop on Transactional Memory (WTM 2013) in Prague, Czech Republic. A newer version of TinySTM was released during approximately the same time frame. The code base of TinySTM had been majorly refactored and simplified in this new version, also removing the one design restriction. The released version had unfortunately no support for Hardware Transactional Memory and as a consequence no support for Hybrid Transactional Memory and was therefore not applicable for further research in the scope of this diploma thesis.

To be able to continue research a plan was made to tackle this problem: The Hardware and Hybrid Transactional Memory mode enabling changes, which were applied to the old VELOX TinySTM version, were identified, extracted and cleanly ported forward to the newest TinySTM version, changing and adapting the implementation whenever necessary. The resulting merged version was adopted as the base for further experimentation.

Summary of characteristics

A clear determination of the correlation between design and influence on Transactional Memory performance is often difficult to make beforehand when looking only at the design of a Transactional Memory system. The design of the involved components is often a result of a trade-off between different design choices. A different approach is to implement the chosen design and then evaluate the influence of the design choices on Transactional Memory performance in an experimental way. Such an approach is shown later on in section 6.1.

4. Design

This chapter contains the design of an adaptive process, which is the foundation for optimizing the performance of Transactional Memory applications by exploiting program phases, as shown later on in chapter 6. Accompanying hardware units are also presented in this chapter.

4.1. Design goals

The Transactional Memory subsystem in a computing system contains several settings affecting the performance and scalability of Transactional Memory applications, as shown in the previous chapters 2 and 3. The goal of this diploma thesis is to improve the performance of Transactional Memory applications by automatically adapting the settings of the Transactional Memory subsystem. An adaptive system, in general, contains an adaptive process, which continuously monitors the behavior of the underlying system, summarizes the current behavior in metrics, processes them using a particular algorithm and reacts by changing parameters of the underlying system in response.

The execution and data flow of such an adaptive process is shown in Figure 4.1. The adaption algorithm works on an input data set, in this case the system metrics, and outputs another set of data, a new set of settings for the Transactional Memory subsystem. Section 4.3 of this chapter describes the metrics available in a Hybrid Transactional Memory system and which settings are suitable for dynamic adaption during runtime.

Some Transactional Memory applications exhibit phased execution, i.e. their behavior during runtime can be decomposed into phases (segments) with a different transactional behavior. This means that the transactional behavior of such an applications is not static and changes during runtime. The transactional behavior of such a phase is characterized by several metrics, which are described in section 4.3. Section 4.2 proposes a novel design for an adaptive process on an Hybrid Transactional Memory system, which reacts to this phased behavior and automatically optimizes the settings of the Transctional Memory subsystem in response.

Furthermore a design is proposed for an event-based tracing framework implemented in hardware, which allows a low overhead tracing of Transactional Memory applications

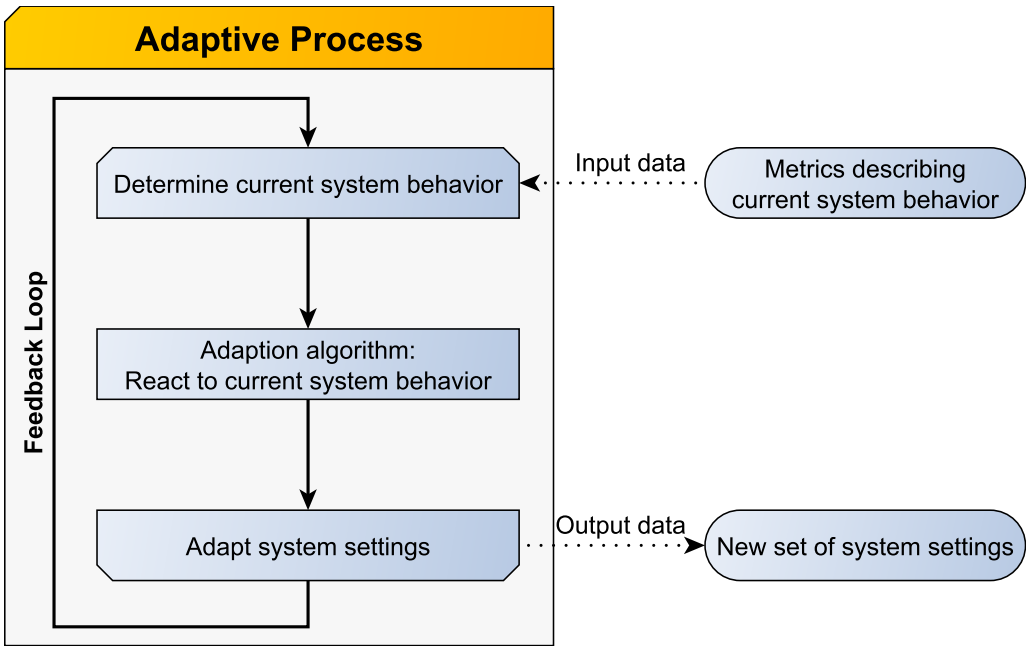


Figure 4.1.: An adaptive process

with a small probe effect. The probe effect describes the fact that tracing an application changes the behavior, performance and scalability of the application when compared to an execution run of the application without tracing. To get a high quality insight into system behavior it is therefore fundamentally important to have a tracing process with a low probe effect. The tracing framework provides the necessary metrics for the adaptive process.

The design of both the proposed adaptive process and the tracing framework are agnostic of the design and implementation of the underlying Hybrid Transactional Memory system.

4.2. Providing adaptivity for a Hybrid Transactional Memory system

Current state of the art Hybrid Transactional Memory systems have a large number of strategies and settings and most of them influence the performance of Transactional Memory applications, as shown in section *"Characteristics of the TMbox Hybrid Transactional Memory implementation"* (3.2). Before running an application an programmer has to specify the to be used strategies and the value of the settings. It is very difficult to decide a set of strategies and settings before hand without further insight into the behavior of the application. Furthermore, some Transactional Memory applications exhibit a phased behavior, where the characteristics of the transactions changes during runtime. Transactions

can be characterized by the rate of conflicts they cause, the transaction runtime, the size of the read and write set, if they can be effectively executed using the Hardware Transactional Memory unit etc. The periods of time with stable transactional characteristics are called program phases. Even if a optimal set of strategies and settings for the first program phase is picked at the start of the application it can lead to decreased performance in program phases with a differing transaction behavior. Selecting the settings statically at compile time before executing the application therefore comes with the disadvantage, that the settings may not suit all program phases. As a consequence, a phased behavior of the application leads to suboptimal or even poor performance.

This section describes a decision making process, which detects program phases and responds to phase changes by adapting the Transactional Memory settings. This is the foundation of how the performance of Transactional Memory applications can be improved by matching program phases to appropriately picked sets of Transactional Memory settings and strategies.

Three main stages have been identified for the decision making process. The stages are described in the following paragraphs and also visualized in the corresponding figures 4.2 and 4.3.

- **Stage 1** - Determine when to switch.

  A decision making unit continuously evaluates received Transaction Memory metrics and interprets them using a given phase detection algorithm. The metrics are computed from the events received from the Transactional Memory tracing framework. The Transactional Memory statistics hardware unit sums up each event type and provides a history of previous tracing periods. The phase detection algorithm uses this information to try to detect when a program phase with one set of characteristics ends and another program phase starts.

  The process proceeds to stage 2 every time a phase change is detected.

- **Stage 2** - Determine set of new settings and strategies.

  Decide which settings to switch and to which values. This process is called decision making and is done by a switching algorithm. The switching algorithm maps an set of Transactional Memory strategies and settings to each occurring program phase. Simple algorithms pick a set of Transactional Memory settings and strategies by looking at the current value of metrics. More advanced algorithms can maintain a history of previously seen application phases and decide based on this broader data base. The overhead of switching between sets of settings also has to be kept in mind.

- **Stage 3** - Switch to new set of settings.

  A new set of Transactional Memory settings and corresponding strategies has been picked by the switching algorithm. This new set can now be activated on a global

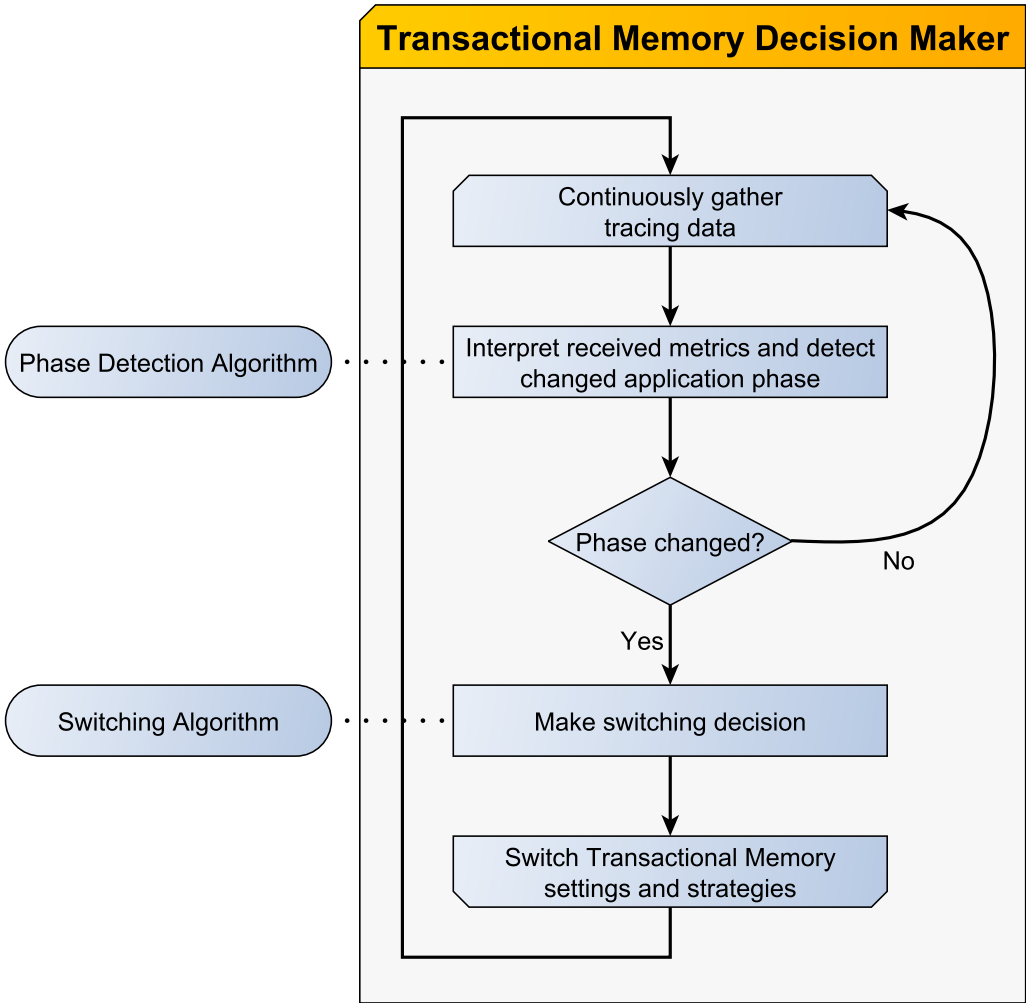


Figure 4.2.: Transactional Memory decision making process

(system-level) or on a local (per-core) scale. This addresses the issue that sometimes threads running at the same time differ in Transactional Memory behavior when compared to each other (locally different behavior) when at other times all threads in the system change behavior (globally different behavior).

This stage has 2 cases, a complex and a simpler one, as shown in Figure 4.3. The chosen path depends on which settings should be changed:

> **Case 1**: A switch of some settings requires an idle system, from a Transactional Memory point of view. For example an increase of the number of locks in the TinySTM lock array requires a shutdown and a subsequent restart of the TinySTM runtime. The switch of these parameters is accomplished by using the quiescent support of the TinySTM runtime, which ensures that at a future

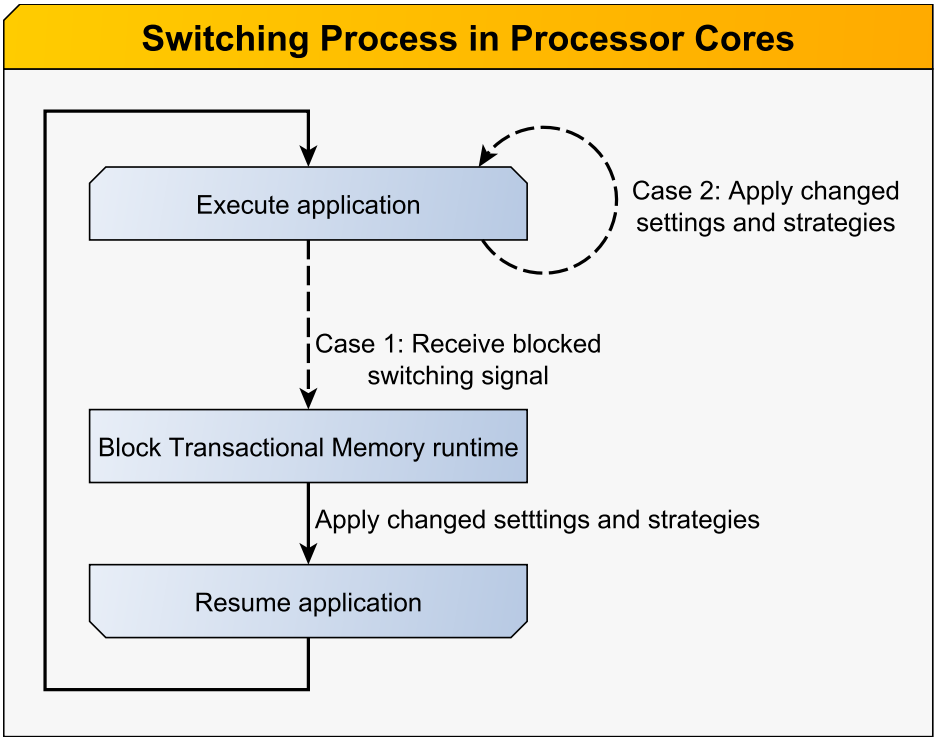


Figure 4.3.: Switching process in processor cores

point in time no processor runs transactions by blocking the activation of new transactions. A blocked switching signal is sent to all processor core, application activity is halted in response, the Transactional Memory runtime is restarted with changed settings and the application is unblocked.

Case 2: Most of the settings and strategies can be changed on-the-fly during normal Transactional Memory activity. The changes are usually picked up by the application threads when starting a new transaction. There is no need to block and unblock transactional activity.

The whole process is, in both cases, transparent and non-disruptive from an application point of view.

4.3. Design Space for an adaptive Hybrid Transactional Memory system

The following paragraphs detail the options that are available when designing and implementing the phase detection and switching algorithms. To keep this project in the scope of

a diploma thesis a subset of the options presented here have been picked and implemented, as shown later in chapters 5 and 6. The other non-implemented options may be researched in the future as further work.

Suitable metrics for the phase detection algorithm

The following paragraph describes metrics, which can be computed by utilizing an appropriate implementation of the tracing framework. These metrics are then available as a source for the decision making process.

Metrics: Contention, transaction length, transaction size, Hardware Transactional Memory effectiveness, switching overhead.

The contention level is calculated as the ratio between the number of aborts and commits. Transaction length as a portion of time can be calculated as the time between a start of a transaction and the end of it through either a commit or an abort. The size of a transaction is the number of entries in its read and write set. There is a correlation with the effectiveness of the Hardware Transactional Memory unit, as on a bounded Hardware Transactional Memory implementations transactions exceeding a certain predetermined transaction size cannot execute using the hardware unit and must be executed solely in software using a Software Transactional Memory library. The switching overhead is the delay caused by deciding which settings to switch and afterwards actually switching between different sets of strategies. The switching overhead is an interesting metric, as it can vary when employing complex phase detection and switching algorithms which exhibit a variable runtime.

As the underlying Transactional Memory system supports both Software- and Hardware Transactional Memory the switching decision can be based not only on the usual metrics like contention, transaction length and transaction size but on a broader base also on some additional ones like the effectiveness of the Hardware Transactional Memory unit.

Types of phase detection and switching algorithms

The phase detection algorithm can run on a per-core (local) or system (global) level. The system level view is generated by aggregating the detected behavior of each processor core. When deciding on a system level view all settings are set globally at the same time for all processor cores. The other case of deciding on a per-core level allows to set the settings on a fine-grained per-core level, selecting an optimal set of strategies for each core. To ease the implementation settings can also be switched on a global level. When some cores exhibit a different or diverging behavior (program phase) than other cores either a majority or consensus decision has to be made. A majority decision switches to a set of strategies

that is optimal for the majority of cores, where as a consensus decision type algorithm does not switch in the diverging case and instead switches later on when all cores exhibit the same program phase again.

Strategy selection

Software Transactional Memory libraries support several optimistic and pessimistic sets of data versioning and conflict detection strategies. Optimistic strategies are more suitable than pessimistic strategies for low contention program phases and vice versa for high contention program phases. The strategies write-through and commit time locking are optimistic, where as write-back and encounter time locking are pessimistic strategies.

In a Hybrid Transactional Memory system transactions are usually started utilizing the Hardware Transactional Memory unit. The transactions are marked as running in hardware mode. A transaction has to fall back to execution by a Software Transactional Memory library if it exceeds the capabilities of the Hardware Transactional Memory unit. The transaction is then marked as running in software mode. This fall back can also be requested voluntarily, for example when transactions abort repeatedly in hardware mode. The Hardware Transactional Memory unit can handle aborting a transaction in fewer ways than when handling the aborting of an transaction running in software mode.

4.4. Application tracing

A goal of this diploma thesis is to improve Transactional Memory performance by adapting Transactional Memory parameters dynamically during runtime based on the detection of application phases. To achieve this goal application behavior has to be traced constantly during runtime and fed to a decision making process, which then interprets the traced information and adapts the parameters based on the gained insights.

To get a suitable overview of Transactional Memory behavior it is vital to have a tracing system with ideally no impact on application runtime characteristics and application behavior. The tracing data gathered could otherwise be influenced in some sort and cause a misguided optimization attempt. For Hardware Transactional Memory systems a separate hardware monitor is therefore the method of choice to non-intrusively gather and preserve run time information.

This section describes an event-based tracing framework, which later on is used to gain information for a decision making process. A ring bus interconnect is especially suited for the transmission of the generated events to a central unit evaluating the events. But the described design can also fundamentally be applied to other types of interconnect, like a switched bus network with dedicated lines between connected nodes.

Design of event-based tracing

Both Software and Hardware Transactional Memory behavior is split into a stream of small events containing information about Transactional Memory state changes. The events are later used as is for input into a central event processing unit. This processing unit contains a decision making process, which decides how to do an dynamic adaption to changing application behavior. The events can also be later on recomposed into states to enable the visualization of application behavior. This design allows to run the monitoring infrastructure with low transfer bandwidth needs. Data concerning events is transported as low-priority traffic: The data is sent on the system interconnect only during phases where the bus is not transferring high-priority data. This procedure therefore does not influence the application behavior and its runtime characteristics. An alternative would be to collect and send the complete Transactional Memory state each time it changes. This approach needs more bandwidth than the chosen approach, as on each state change all data describing the state must be transferred on the interconnect.

Event format

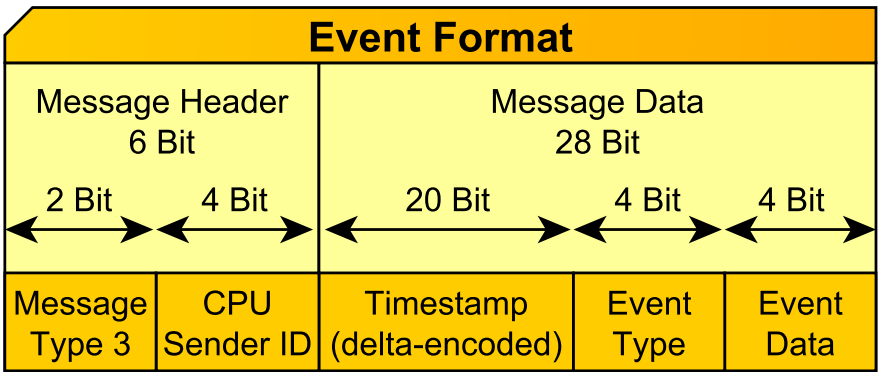


Figure 4.4.: Format of an event

Figure 4.4 shows the format of an event. The data in the message header contains metadata about the event: The message type field is used to distinguish tracing events from other data being transmitted on the system interconnect. This allows to transfer the tracing events with a lower priority than the other data. The system interconnect can thus be shared with other communications without creating a probe effect. The sender ID is used to determine the processor number, on which a particular event happened.

The timestamp, i.e. the time when an event occurred, is delta-encoded. This means only the difference between consecutive event timestamps is saved. This space efficient encoding allows to determine the moment an event occurred with an accuracy of 1 cycle. A temporal

distance of more than one million cycles between two events occurring on one processor does on the other hand lead to an overflow of the timestamp field. An exemplary system with a clock frequency of 50 MHz would therefore have a timestamp overflow every 20 milliseconds. Not handling this overflow would cause the “real” event time (during application runtime) and the reconstructed event time after post-processing to diverge. A timestamp overflow is however an unusual case: Events are created during normal system operation with reasonable Transactional Memory activity with a much higher frequency than required by this technical limitation. An easy solution to rule out timestamp overflows is to add the generation of a special no-operation event, whenever a timestamp overflow would have normally occurred. This special event can be ignored later on by the central event processing unit.

The event type field stores which event occurred. Up to 16 different event types can be defined, allowing an easy and flexible addition of new event types whenever necessary.

The event data field stores additional data concerning a given event, for instance the cause of an abort of a transaction. Aborts can be caused either by a software request (software induced), by exceeding a hardware constraint (capacity abort) or in most cases by detecting a conflict with a committing transaction.

Event types

The event types defined for Transactional Memory transactions are:

Name	Description
Start	Transaction in software, hardware or hybrid mode has started.
Commit	Transaction was successfully committed.
Abort	Transaction was aborted. The cause is stored in the event.
Overflow	A timestamp overflow occurred.

Table 4.1.: Event types for software, hardware and hybrid mode

The following event types are used only for transactions executing in hardware or hybrid mode, as they can occur only in states related to Hardware Transactional Memory:

The generation and capturing of these event types allows to rebuild all Transactional Memory states during post-processing. Additionally subsets of events can be selected later during analysis, allowing a focus on specific types of transactions (for example only committed transactions).

Name	Description
Invalidation	A write has occurred to a memory location. This event is used for the detection of transactional conflicts by the Hardware Transactional Memory unit.
Try Lock	Try locking the system interconnect for commit: A transaction has finished computing. The Hardware Transactional Memory subsystem tries to prepare the commit phase by exclusively locking the system interconnect.
Lock Success	Succeeded locking system interconnect: Lock was acquired, transactional data is being stored into main memory.

Table 4.2.: Event types for hardware and hybrid mode

Generated event stream

Event Stream

INV	ADDR	ID	EVENT	DATA	TIMESTAMP
3	1398450	2	1	0	39845
3	1398db0	0	1	0	398db
3	139a210	3	1	0	39a21
3	139aaf0	1	1	0	39aaf
3	5006e50	2	5	0	006e5
3	6000050	2	6	0	00005
3	2000100	2	2	0	00010
3	5006aa0	0	5	0	006aa
3	6000030	0	6	0	00003
3	2000090	0	2	0	00009
3	5006cd0	1	5	0	006cd
3	6000040	1	6	0	00004

Figure 4.5.: Monitoring infrastructure event stream

The generated event stream can be easily transferred, processed and saved. Figure 4.5 shows a short example of such an event stream.

Every row shows one specific event. The value 3 in column “INV” indicates a stream of events. The “ADDR” column contains encoded values of the four columns to the right. The “ID” columns contains the number of the processor core which generated the event. The “DATA” column contains additional information about the event. The last column “TIMESTAMP” contains the delta-encoded event time (i.e. the difference between the time

during the generation of an event and the time of the previously generated event on the same processor core).

The event stream is fed into the statistics unit for further processing. It can also be sent to a Host PC for further visualization and analysis, as shown later on in chapter 6.

4.5. Tracing units

The following paragraphs describe the design of the hardware units of the tracing framework. As the design is language-agnostic, it can be implemented using an arbitrary hardware description language, such as VHDL [35], Verilog and Bluespec System Verilog [36].

Event generation unit

The event generation unit is connected to the Hardware Transactional Memory unit and monitors the state of it, generating events whenever the state changes. The generated events cover all of the state changes occurring during runtime. They are augmented with additional data that is useful later on for behavior analysis. This additional data includes for instance the number of a processor core, which caused the abort of a transaction. This type of event generation adds no additional overhead when enabling tracing, as it can be implemented entirely in hardware units running in parallel to normal system operation. The tracing of Hardware Transactional Memory is thus possible without any probe effect on the proposed design.

Event generation can also be requested by issuing special instructions in software. This hardware-assisted event generation is useful when tracing transactions running in Software Transactional Memory mode. These transactions are processed entirely by software and do not change the state of the Hardware Transactional Memory unit. Software Transactional Memory state changes are traced by adding the event-emitting instructions to corresponding functions in the Software Transactional Memory runtime library. This mode of operation introduces a one cycle overhead for each state change, which is small when compared to a software-only tracing framework (also see section 6.4 for further information on this topic).

Log unit

The log unit captures and saves events sent by the event generation unit located in the processor core. These events are timestamped and saved using delta encoding in memory

blocks located in each core unit. The events are later transferred via the system interconnect to a central event processing unit. The transfer is only done whenever the interconnect is idle. This approach prevents a disturbance of application timing behavior (no probe effect). A buffer is used to save the arriving events. The size of the buffer is dependent on the rate of arriving events and the worst case bandwidth of the system interconnect available for low-priority traffic.

Location of event generation and log unit

The location of the event generation and log unit influences the scope of the available tracing data. The complexity of the necessary design changes needed for connecting these units to the to be traced units also needs to be considered. Two possible locations have been identified:

1. Tightly integrated into processor core

   The internal processor state can be easily monitored by embedding the units directly in the processor core. The biggest disadvantage is the necessity to make major design changes in the processor core to connect the various processor core buses and signals to the units.

2. Between processor core and interconnect

   This method allows a great extensibility of the tracing framework by allowing access to a broad amount of available traceable data. It is also still relatively easy to non-intrusively connect the event generation and log unit to the rest of the system.

The second location has been chosen to ease a later implementation and still deliver a broad amount of logging data. This also keeps the complexity of necessary changes to the design of a underlying Transactional Memory system at a reasonable level.

Statistics unit

The statistics unit is the central event processing unit. It is directly attached to the system interconnect and counts how many events occurred in a time period (i.e. a fixed time span). The sampling period can be changed to increase or decrease the sampling frequency. It can be changed by reconfiguring the statistic unit during runtime.

The decision process, which manages adaptivity on the proposed design and is introduced shortly in section 4.2, has some specific requirements: It has to gather insight into the system state at discrete points in time. These points in time correspond to the execution phases of the phase detection algorithm. The advantage of the proposed event based tracing

framework (high accuracy and low overhead) can be combined with the requirements of the decision making process (provide insights into the system state at discrete points in time) by counting the events created from the tracing framework and summarizing them in a regular fashion (sampling). The data flow of the statistics unit is shown in the next Figure 4.6.

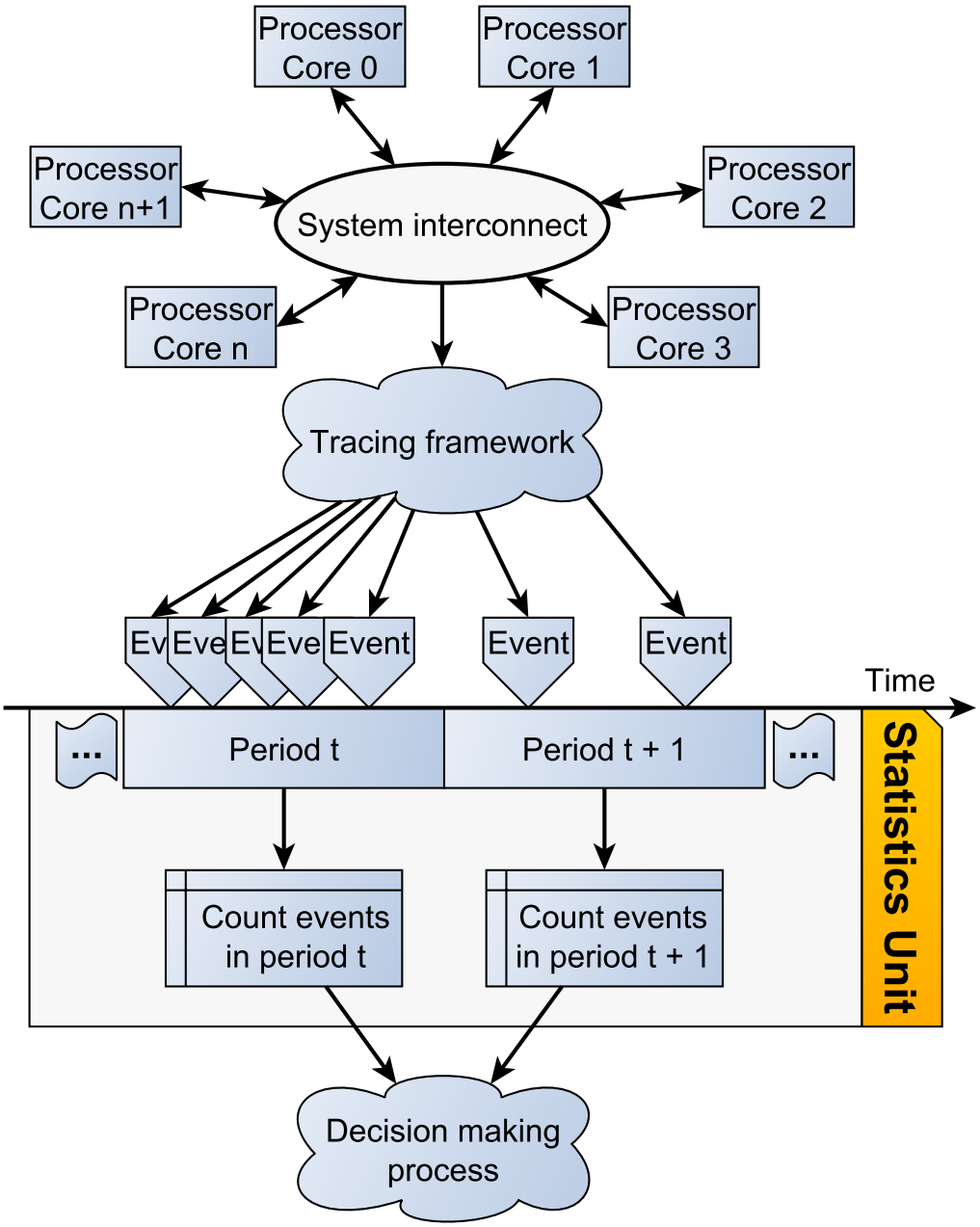


Figure 4.6.: Data flow of statistics unit

5. Implementation

Chapter 4 showed the design of an adaptive process for a Hybrid Transactional Memory system. This is the foundation for reaching the goal of this diploma thesis: Optimizing the performance of Transactional Memory applications by exploiting program phases. The following chapter details the implementation of the hardware and software units of the proposed adaptive process, which was done during this diploma thesis. The section also tells about the substantial porting efforts, which were required to run the system on an FPGA board available at KIT. The core parts of the implementation (ring bus, processor cores) are based on an initial implementation of the TMbox system for a different FPGA board.

5.1. The BEE3 FPGA Board

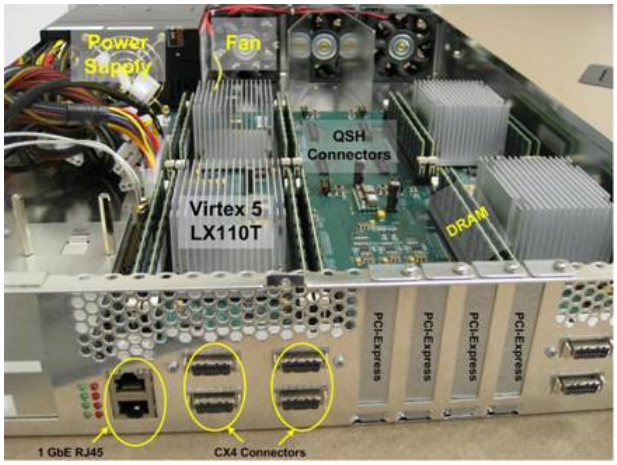


Figure 5.1.: BEE3 board

The TMbox system was originally implemented for the BEE3 research platform. The BEE3 (Berkeley Emulation Engine, version 3) is a multi-FPGA system with up to 64 GB of DRAM, as described by Davis et. al. in *"BEE3: Revitalizing Computer Architecture"* [37]. It is equipped with Xilinx Virtex 5 Series FPGAs and built into a 2U high chassis. The BEE3 board uses the fourth-largest FPGA available in the Xilinx Virtex 5 series: A

LX155T FPGA contains 97,280 LUT-flipflop pairs, 212 36K-bit Block RAMs for a total of 954 allocatable kilobytes of Onboard-RAM and allows up to 16 MIPS compatible processor cores to be fitted in one FPGA chip. A picture of the components comprising the BEE3 system can be seen in Figure 5.1. For simplicity reasons only one out of the four FPGAs is used on the BEE3 board, all hardware components of the TMbox system accordingly sit on one FPGA. The ring bus could be extended to the other FPGAs and form a multi-FPGA many-core system with up to 64 processor cores.

5.2. The XUPv5 FPGA Board

The XUPv5 board mainly features a Xilinx Virtex 5 XC5VLX110T FPGA. Additionally two Xilinx XCF32P Platform Flash PROMs are used for storing the synthesized system and a 256 MByte DDR2 SO-DIMM RAM module is used as main memory.

The board communicates with a host PC using the Universal Asynchronous Receiver/Transmitter (UART) unit in processor core 0, a series of pins on the XUPv5 board providing the required connection for the signals and a custom built Low Voltage Transistor Transistor Logic (LV-TTL) to Universal Serial Bus (USB) converter. Data is transferred in intermittent packets of one byte. The UART receives and sends bytes of data from and to the processor core sequentially by transmitting and receiving one bit at a time over the serial connection. The serial connection consists of a TX signal for sending bits, a RX signal for receiving bits and a GND signal for providing a common ground voltage level. The bit values 0 and 1 are transmitted and received by varying the voltage on the signals between 0 and 3.3 volts, respectively, as per LV-TTL specification. There is no need for a clock signal, as the UART communicates asynchronously by starting each transmission with a start and stop bit and each side of the connection uses a fixed pre-set symbol transmission rate (also called baud rate). A symbol consists of the payload data, the data bits, and the auxiliary start, stop and parity bits.

The LV-TTL to USB converter was previously used for connecting a mobile phone to a PC. The proprietary connector was cut off and replaced by soldered on pin headers for the UART signals. These UART signals are then connected to the board headers using wires.

The XUPv5 board can be plugged into a PC using the on board PCI-Express connection as an alternative to the much slower UART interface. The PCI-Express connection is currently inactive, as there are no hardware units handling the FPGA-side of the connection on the board. Developing PCI-Express software units for the FPGA is outside the scope of this diploma thesis. Further work may use the PCI-Express connection to transfer events from the Transactional Memory tracing framework to the PC for further analysis and visualization.

5.3. Implementation of the proposed design

The design for an adaptive Hybrid Transactional Memory system, which is proposed in chapter 4, was implemented in the scope of this diploma thesis. This section describes the implementation of the hardware units, which were written in the hardware description language VHDL.

Event-based Transactional Memory tracing framework

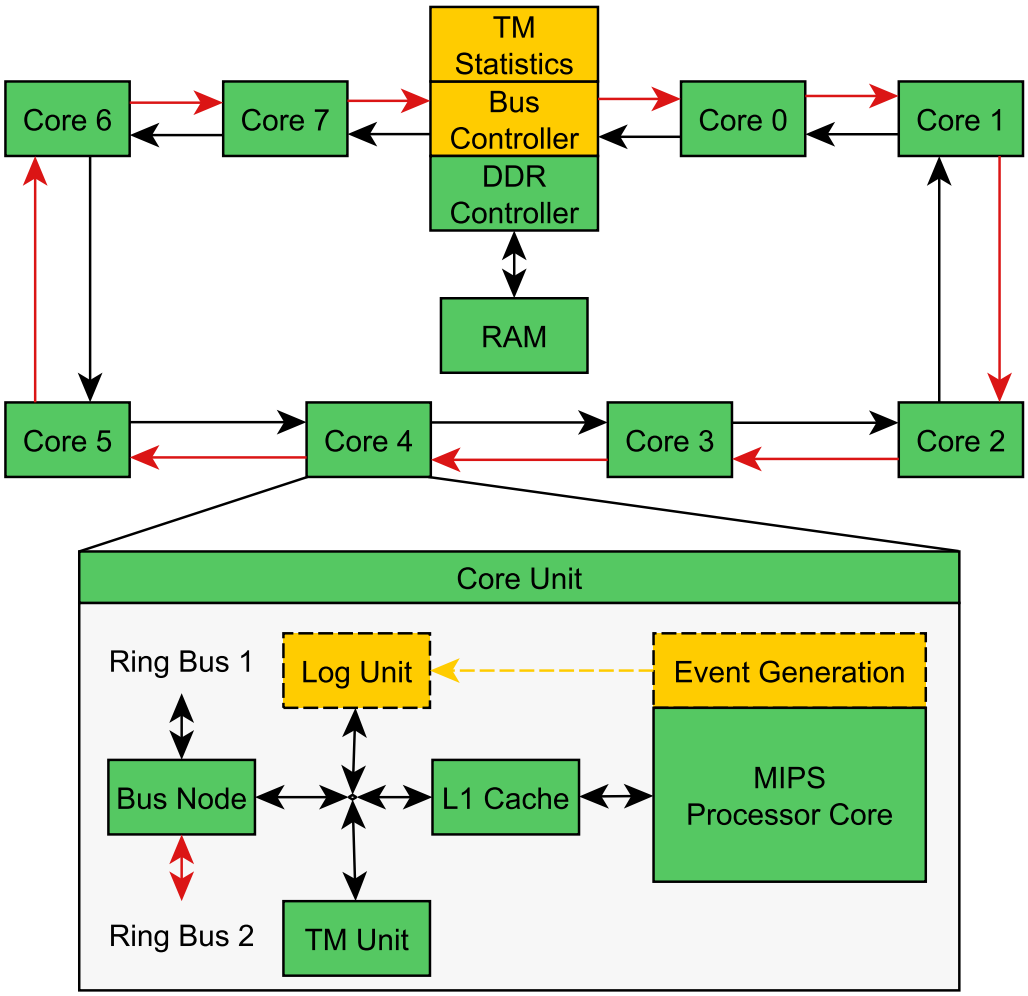


Figure 5.2.: 8 Core TMbox system block diagram (with event-based tracing framework)

The design of the proposed event-based Transactional Memory tracing framework was implemented during this diploma thesis. The implementation is based on a previous

implementation by the author, which is described in [26]. Figure 5.2 shows the tracing hardware units in yellow color.

The log unit buffers events, if other higher priority traffic is detected on the ring bus. This approach prevents a probe effect through by tracing, as the generated events do not preempt traffic on the ring bus initiated by an application. Experiments did show that a buffer size of 32 entries for the log unit is large enough to prevent a buffer overflow and a following loss of events during application runtime. A specially designed assembler program, which produces a very high rate of generated events, was used to determine the proper threshold experimentally.

Hardware Transactional Memory tracing

The TMbox system uses a finite-state machine to manage the internal states of each processor core. This finite-state machine contains the current state of the processor cache and reacts to memory requests and answers coming from the ring bus. Figure 5.3 shows, in a flow chart, a simplified image of this finite-state machine at the end of the chapter. The full TMbox finite-state machine contains 11 states and 131 transitions. For simplicity only the four states relevant to Transactional Memory operations are shown in Figure 5.3. Tracing functionality for Hardware Transactional Memory was implemented by adding event emitting codes to the finite-state machine. The event emitting code parts are marked in red color. The event emitting code was implemented as unintrusive as possible, the changes to the finite-state machine were in fact accomplished by adding about 30 lines of VHDL code. This is a small change compared to the sum of lines of code in the main unit of the processor (1533 lines). By embedding the event generation into hardware there is no probe effect when enabling tracing for Hardware Transactional Memory.

Software Transactional Memory tracing

Special instructions have been added to the processor ISA to trace the execution of the Software Transactional Memory library. These instructions, called *xevent1* through *xevent4*, allow to generate events, similar to the generation of events in the Hardware Transactional Memory case. As a difference these events can instead be generated from software. The TinySTM functions handling the start, abort and commit of transactions use the tracing instructions to generate an event on each transactional state change. This approach allows to enable the tracing of Software Transactional Memory with a overhead of one cycle per transactional state change.

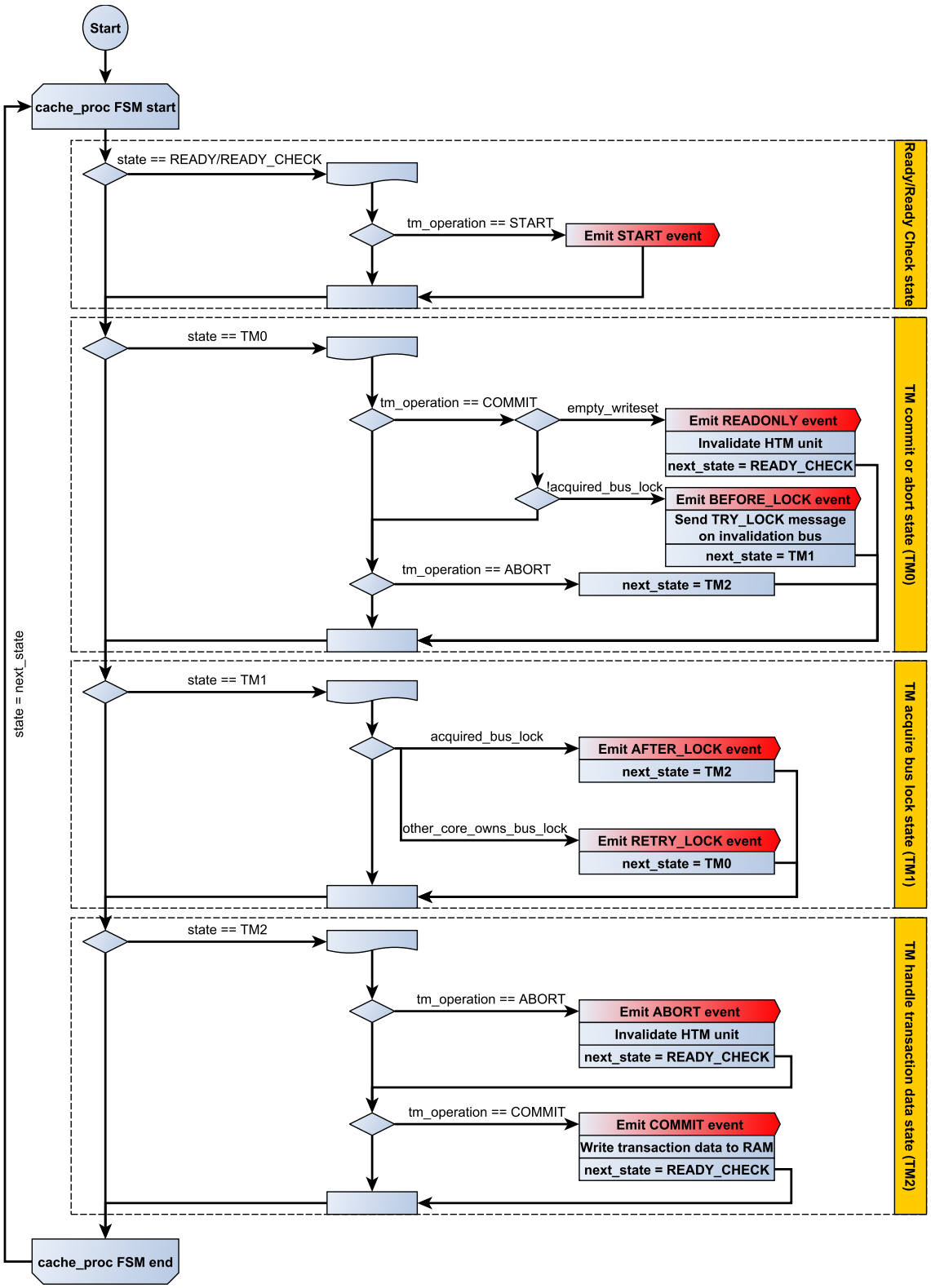


Figure 5.3.: Processor core cache state finite-state machine with Hardware Transactional Memory tracing extensions

Statistics unit

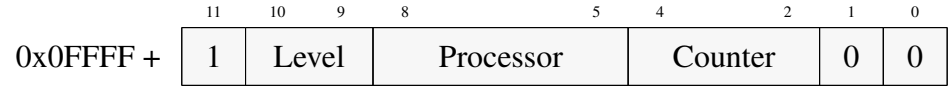
The statistics unit counts the events generated by the tracing framework and summarizes them periodically. The events are counted with an counter array, where every event type is connected to one counter. Each counter is 32-bit wide. A counter is incremented by one whenever an event of the corresponding type arrives. A counter field consists of a counter array for each processor core. This allows to analyze the incoming events on a local (per-processor) or a global (system) level. The global system level view is built by adding the counter values of all processors. This is done for each event type.

The runtime of the system is divided into uniform periods of time. Each period has the length (in cycles) of the sampling period. The sampling period is configurable by writing to the corresponding configuration register and initiating a reset of the statistics unit. The events received from the tracing framework are summarized and stored for the current and the previous period.

The counter values of previous periods are retained to enable the phase detection and switching algorithm to analyze the behavior of previous periods of time.

Each group of counter fields is called a level: By default there are three levels of counter fields available. The special level 0 contains the number of events for each event type and processor that occurred after the last reset of the statistics unit. Level 1 contains the number of events that occurred in the current sampling period whereas level 2 contains the number of events that occurred in the previous sampling period. The counters in level 2 are frozen until the current sampling period ends. The value of the counters of the now finished sampling period in level 1 are moved to level 2 at that time, replacing the values of the now penultimate sampling period. The values of the level 1 counters are set to zero and the new sampling period starts. The number of retained previous periods can be increased by changing the number of levels before synthesis.

The statistics unit is memory-mapped at the top of the system memory (also see Figure A.5). The unit is accessed by using standard memory read and write instructions. This allows an easy access from the phase detection and switching algorithm. The memory-mapped area is divided into two main regions: The first region contains configuration and debugging registers, while the second region contains a counter arrays for each level. All registers and counter arrays are read-only, unless noted otherwise. Figures 5.4 and 5.5 show how to access the statistics unit counter arrays, configuration and debug registers from software.

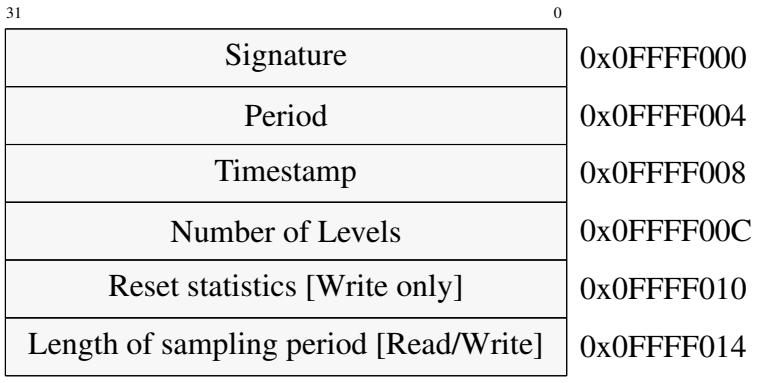


Level
- 00 Events since last reset
- 01 Events in current period
- 10 Events in previous period

Processor Select counter array for specified processor core

Counter
- 000 Number of transaction starts in hardware mode
- 001 Number of transaction commits in hardware mode
- 010 Number of transaction aborts in hardware mode
- 011 Number of transaction starts in software mode
- 100 Number of transaction commits in software mode
- 101 Number of transaction aborts in software mode

Figure 5.4.: Statistics unit counter selection



Signature Fixed signature, used for debug purposes

Period Current sampling period number, used to detect period changes

Timestamp Number of elapsed clock cycles in current period

Number of Levels Number of counter arrays, set during synthesis

Reset statistics A write to this register sets all counter values to zero

Length of sampling period Get and set the length of a sampling period (in clock cycles)

Figure 5.5.: Statistics unit configuration and debug registers

5.4. Porting the TMbox system

The existing implementation of the TMbox system is tightly linked to the BEE3 [37] board. As a consequence the implementation had to be ported to the XUPv5[4] board, which is available at KIT.

The core parts of the initial implementation of TMbox, the ring bus and the processor cores, was straight forward, as these units were written in a board-agnostic way. Porting the other hardware units, which connect the core units to other hardware units outside the FPGA, was a more complex task. The XUPv5 board has a different hardware setup than the BEE3 board. The RAM controller and the Host PC connection interface had to be replaced as the original units were tied to specific hardware characteristics on the BEE3 board. The PCI-Express interface, which acted as the interconnect between the initial implementation of TMbox and a Host PC, had to be replaced with a serial interconnect.

Also additional hardware units had to be designed, implemented and tested specifically for the new board. These new units are the infrastructure, reset management, clock generation, clock domain crossing control, DDR2 RAM controller and top level units.

The bus controller unit, which manages the interface to the DDR2 RAM controller, was completely rewritten, because the signals and timing differed highly from the original microprocessor controlled main RAM controller used in the initial implementation of the TMbox system for the BEE3 board.

The initial implementation of the TMbox system for the BEE3 board originally contained a small microprocessor, used for calibration and control of the on board DDR2 RAM chips. The program running on the microprocessor is designed specifically for the BEE3 platform and did not work on the XUPv5 board. The implementation for the XUPv5 board uses a dedicated DDR2 RAM controller unit implemented in hardware in place of the microprocessor.

Infrastructure unit

The infrastructure unit provides a basic hardware environment, on which the other hardware units can depend on after hardware power on and during reset and normal system operation. The unit performs the reset management, clock stabilization and generation and generally connects the other hardware units.

[4] *Xilinx University Program XUPV5-LX110T Development System*
http://www.xilinx.com/univ/xupv5-lx110t.htm

Reset management

Reset management is a delicate matter, as three separate reset signals in three different clock domains have to be synchronized and released in a specific order. To further complicate matters the reset signals assert each other in a given way.

The reset signals in order of priority are "PLL reset" (highest priority level), "infrastructure stage 1 reset" and "infrastructure stage 2 reset" (lowest priority level). The PLL reset is asserted during power on and released after the PLL clock is stable. Infrastructure stage 1 reset connects to the DDR2 RAM reset. The DDR2 RAM controller calibrates and configures memory access and releases the reset signal afterwards. Infrastructure level 2 reset connects the reset signals of the remaining units, mainly the ring bus and processor cores. Each signals has to assert whenever a reset signal of a higher priority level is being asserted. The release process follows the same order: Reset signals of a higher priority level are always released before the release of reset signals of a lower priority level occurs. At power on all reset signals are asserted. The system releases the reset signal step by step afterwards until all components are correctly initialized and are working nominal.

Clock Domain Crossing

As the DDR2 RAM controller and the rest of the system are part of different clock domains the problem of clock domain crossing has to be considered. Signals and data buses connected between the two clock domains have to be appropriately synchronized. For example a signal which is asserted for one cycle and arrives from the partition with a lower frequency also has to be asserted exactly one cycle on the other side of the partition (the side with a higher frequency). This means that the number of clock cycles a signal is asserted should be the same on both sides. As a consequence of the different clock frequencies in both clock domains the time (in seconds) the signal is asserted is not the same. Another problem occuring during clock domain crossing is that a signal arriving from one clock domain can appear asynchronous on the other clock domain, which disturbs the operation of synchronous logic circuits. Signals therefore have to be re-synchronized using either a FIFO or another appropriate synchronization technique, as shown by Sharif et al. in *"Quantitative analysis of State-of-the-Art synchronizers: Clock domain crossing perspective"* [38].

Two dual-ported asynchronous FIFOs with different clocks for the read and write port are used to accomplish synchronization. The reason for needing the FIFOs is two-fold: First when memory data arrives from the DDR2 RAM controller it is received in two cycles at the rate of two memory words (64 bit) per cycle with a frequency of 200 MHz. This data is forwarded to the bus controller, which receives the data with the same width and a lower frequency of 50 MHz. The clock domain crossing unit handles the resulting bandwidth difference by buffering the incoming and sent data in FIFOs. The other case, where data

is stored in main memory and therefore sent from the bus controller to the DDR2 RAM controller, is handled accordingly by using a second set of FIFOs.

Clock generation

The shown system needs several periodic signals for the clocking of its logic. These signals need to have particular properties. For instance they need to have a fixed and stable frequency without drift and a fixed phase relative to a particular input clock. The clock generation unit controls the generation of the system clocks. The unit works in tight cooperation with the clock domain crossing unit, as the synthesized hardware units of the system are partitioned in two distinct areas with a different main clock frequency (clock domains). Signals crossing a partition have to be handled in a particular fashion, as described later.

The XUPv5 board contains a 100 MHz crystal oscillator as a source for a stable ciruit logic input clock. The signal is connected to an FPGA pin and forwarded using a dedicated clock distribution network to the clock generation unit.

Clock generation in the system is done by using a phase-locked loop (PLL), which is integrated directly in the FPGA hardware. A phase-locked loop is a control system that generates one or several output signals whose phase is related to the phase of an input signal. The circuit consists of a variable frequency oscillator and a phase detector. The oscillator generates a periodic signal and the phase detector compares the phase of that signal with the phase of the input periodic signal and adjusts the oscillator to keep the phases matched. The phase-locked loop is used here to generate frequencies that have a multiple of the input frequency or whose phase is shifted by a fixed degree. The PLL is unstable directly after power on, the PLL generated clocks are therefore not usable until the PLL has reached nominal operation parameters (PLL lock is acquired). The reset management unit handles this case by switching between external and internal (PLL generated clocks) on the fly. The remaining system is also not activated until the PLL is locked.

DDR2 RAM controller

The DDR2 RAM controller block, which is provided by Xilinx, was also modified to run on the XUPv5 board. The original Xilinx implementation uses a dedicated clock generation unit to generate the various clocks needed for the DDR2 RAM controller. This unit was combined with the system-level clock generation unit, which also provides clock signals for the other units of the system.

The DDR2 memory module used in the XUPv5 board is a dynamic random-access memory. The generic DDR2 RAM controller implementation, which is provided by Xilinx, has been modified to run on the XUPv5 board and handles memory refresh and access. Memory access has to be done in a particular fashion and has to adhere to strict timing constraints, which are pre-determined by the used memory module. The frequency of signal changes is, for instance, limited to the range of 200 to 266 MHz for the memory module used in the XUPv5 board. The timing of other control signals can also be varied only in a small range. The memory controller is written in VHDL as a soft-core IP block and is synthesized together with the rest of the system. The strict timing constraints imposed by the DDR2 RAM controller are ensured by specifying mapping and routing constraints for the synthesis, mapping and routing process. The DDR2 RAM controller has the tightest timing constraints in the implementation, as the ring bus and the processor cores run at a much lower frequency and can have a more relaxed timing.

The memory controller needs several clock signals, whose frequencies determine the speed of memory access. These clocks are used for memory calibration and I/O and internal state machine clocking.

The main clock frequency of the DDR2 RAM controller is directly fed as the input clock to the memory module. The memory module frequency has to be at least 200 MHz. The DDR2 RAM memory controller main clock frequency therefore also has to run at least at the same frequency. This requirement of a higher memory controller frequency actually decreases the latency of memory accesses when compared to a simpler system implementation with an uniform frequency of 50 MHz: A memory access usually has a latency of > 20 memory cycles. The latency in the current non-uniform split-frequency implementation is lower, as the memory clocks run at a higher frequency than the processor core. The latency, as seen by an application running on a processor core, is therefore only a quarter when compared to a hypothetical uniform system where both the processor core and memory clocks run with 50 MHz.

Bus Controller Unit

The bus controller unit, which sits in the ring bus right next to the first and last processor core, manages memory access and bus lock arbitration (needed by Hardware Transactional Memory). It also maps several auxiliary hardware components into the memory address space. This makes it easy for programs running on the processor cores to access and configure these components by reading and writing using normal memory access instructions to specific areas in memory (memory-mapped hardware access). The components include a loader block ram containing the boot loader, which initializes the software side of the system and loads and controls the execution of an application program (see section 5.4).

Memory read and write requests are delivered to the bus controller via the ring bus interconnect. The requests contain a command word, which differentiates between read and write requests, and an address (32 bit, aligned to processor quad-word). These requests must be executed in a special way, as the DDR2 memory can be accessed only using bursts of several memory words per specification. One burst is 64 bit wide. The burst mode for standard DDR2 memory may be set only to either four or eight word burst. The bursts must be transmitted to and received from the DDR2 RAM controller in pairs of two memory words by specification. For this project a burst of four words was chosen. A two word burst mode, which would allow to directly split the incoming processor quad-word (128 bit) into two bursts and execute them without additional control logic is unfortunately not available for DDR2 memory. The bus controller therefore always has to access four words (two processor quad-words), but receives and sends only two words (one processor quad-word). The matter is further complicated as the addresses of the data words following the first data word are wrapped at the burst boundary. This is handled by aligning the incoming memory address to a burst boundary (two processor quad-words).

Boot Loader

The DDR2 main memory of the system is in an uninitialized state after power up. This means that the memory contains entirely random data and therefore has to be prepared for use by an application by the boot loader.

The boot loader software is located in Block RAMs, which are directly connected to the bus controller. To keep resource utilization low a decision has been made to fit the code, data and stack of the boot loader into a memory area of 8 KB. This was achieved by carefully creating a stripped down program written in MIPS assembler and C using no standard library functions. Library functions could not be used, as it would be otherwise not possible to fit the code, data and stack areas of the boot loader into 8 KB of RAM. The boot loader does general hardware initialization, memory function checks, loads an application program to main memory, checks for correct transmission and as the last action starts and transfers control to the previously loaded application.

The Block RAMs used for storing the boot loader are placed to fixed locations on the FPGA to allow the use of the data2mem program, provided by Xilinx. The data2mem program[5] is used to store the compiled boot loader directly into the right location in the bit file created after the synthesis, map, place and route steps. When hardware units or, in this special case, the pre-set content of BRAMs is changed the normal process is to go through the usual synthesis, map, place and route steps to create a bitfile suitable for configuration

[5] Xilinx: *Data2MEM User Guide*
http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/data2mem.pdf

of the FPGA. As these steps usually take a long time due to their elaborate processing[6], even when doing only small changes to hardware units or the boot loader, there was a need for a faster approach. Using data2mem allows to replace and debug the boot loader easily in a time efficient manner, by skipping the lengthy synthesis, map, place and route steps and directly replacing the content of the Block RAMs in the otherwise unchanged bitfile. This saves much time when debugging the boot loader and initial startup process of the software parts of the system, as the turn around time for testing a new software version is much lower.

Endianess / Byte order

Data has to be handled specially when transferring it between the Host PC and the system, as the endianess of common x86 systems (Little-Endian/LE) differs from the endianess of the MIPS cores used in this implementation (Big-Endian/BE). Endianess affects the order, in which values sized larged than a byte are stored bytewise in memory. In BE mode the most significant byte (MSB) is stored at a lower memory address than the least significant byte (LSB), which is stored at the largest address. The storage order is reversed in LE mode.

Boot Loader Image

The application image contains a cyclic redundancy check (CRC) value, which is used for the detection of spurious transmission errors on the serial connection. The CRC value is calculated by applying the standard CRC32 generator polynom $G(x) = x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$ to the application memory dump area of the image, excluding the header fields. The header field is excluded, as the CRC field in the header field can not be used as part of the input data area for the CRC algorithm. The start of an improperly transmitted application is detected and prevented by the boot loader. The boot loader restarts the loading process whenever a transmission error is detected.

5.5. Running an application

General purpose computer systems usually run an operating system, which manages hardware resources, provides protection by preventing simultaneously running application from interfering with each other and, in general, provides a basic environment with common, usually standardized services like memory management and communication

[6] see Bacon et al.: *"FPGA programming for the masses"* [39]

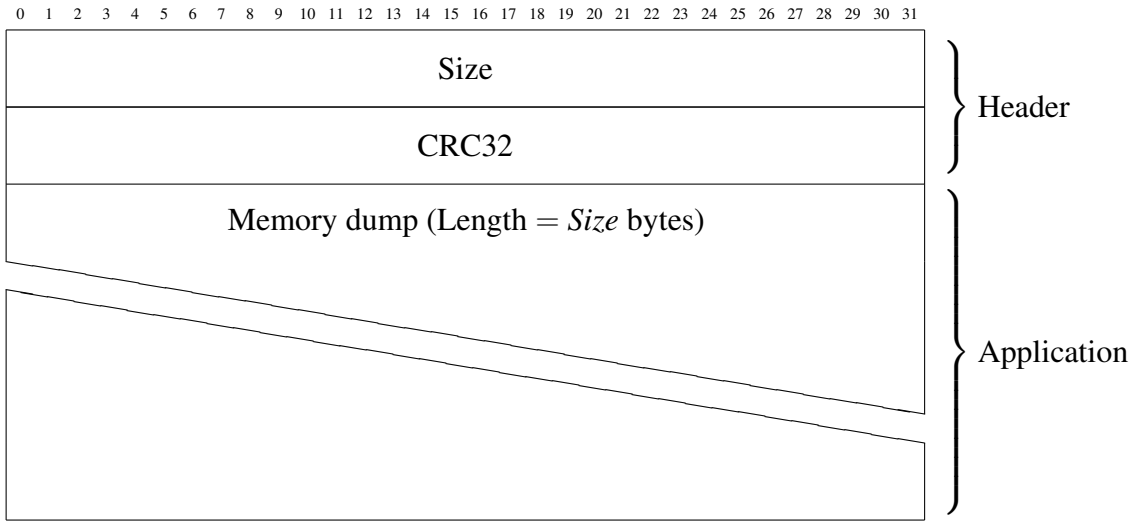


Figure 5.6.: Boot loader image specification

techniques. Application, which run in this environment, hence do not have to be concerned about the concrete implementation of these common services. Since there is no full operating system running on the system standard core system libraries like the GNU C Library (glibc), which provides a standardized interface (syscalls) between operating system services and application, can not be used, as these libraries assume the existence of a full-featured operating system. A set of system libraries called BeelibC are therefore included in the software implementation of the TMbox system. These provide general purpose, memory allocation, I/O and string handling functions. The libraries were enhanced by adding a library, which allows an easy access to the Transactional Memory statistics hardware unit from software.

A MIPS cross-compiler using GCC 4.3.2 compiles applications into object code. The linker included in Binutils 2.19 later on links the application code statically with the system libraries.

The following text details the steps required to run an application:

1. On the first step an application program is compiled using the gcc MIPS cross compiler to an standard ELF executable file.

2. On the second step the file is converted to a plain memory dump by reading the ELF header sections and writing the code and data sections at the appropriate offsets in a raw file.

3. The resulting file can then either be run on the system or simulated using the Xilinx ISIM VHDL Simulator.

a) When running in hardware the application is transferred into DDR2 main memory using the boot loader and an UART connection.

b) For simulation the memory dump file has to be converted one more time into a format suitable for loading into the Xilinx ISIM VHDL Simulator. Using a cycle-accurate simulation the hardware units can be debugged more easily and a faster code-compile turn-around cycle time can also be attained by avoiding the lengthy synthesis, map, place and route steps required to get an bitfile for uploading to the FPGA. These steps usually take a long time in the order of tens of minutes and can reach up to several hours when synthesizing a system with the maximum number of cores enabled. A disadvantage of simulation is the very slow execution speed of the system. When compared to a run in hardware the simulation is much slower.

The Host PC is connected to the implemented system using a standard USB to LV-TTL (3.3 V) UART converter running at a symbol rate of 115.200 baud. The UART connection is also used for application control.

6. Results

6.1. Assessing the influence of transaction characteristics

The Transactional Memory strategies, which are implemented in TMbox, differ in characteristics, as shown previously in Section 3.2. This indicates a difference in behavior and performance in different application phases. This section explores the influence of different Transactional Memory strategies on the general performance of a Transactional Memory application.

General approach

The influence of different Transactional Memory strategies on the performance of a Transactional Memory application has been determined by using the tm-bank application. The application is based on a demonstration application found in the TinySTM distribution with a similar mode of operation, but fewer adjustable application settings. The tm-bank application also models the operation of a bank, where money is transferred between different accounts.

As a first step two random accounts are chosen from the set of all available accounts. These are designated the sending account (i.e. the account to be debited) and the receiving account. A sum of virtual money is afterwards transferred from the sending to the receiving account in two related work parts: The account balance of the sending account is retrieved from the central register of account balances, decreased by the amount of money transferred and stored back into the register. The same process is also done for the receiving account, with the difference of increasing the account balance of the receiving account by the amount of money transferred.

These two work parts must be executed atomically, as otherwise there exists the possibility of a race condition, where the account balances of the involved accounts are set to wrong values by executing the work parts in an interleaved fashion or reading and writing the account values of the same accounts by concurrently running threads.

To avoid these problems the process of sending the money from one account to another is enclosed in a Transactional Memory transaction. It is ensured that the money sending process is executed in the required all-or-nothing atomically fashion. To increase the length of a transaction multiple sending-receiving processes can be combined into the same transaction.

The tm-bank application can be used to simulate transaction phases with differing characteristics. The various settings available allow to parametrize the application in many interesting ways by creating sets of settings (scenarios) and then checking the influence of these scenarios on application performance when using different Transactional Memory conflict detection strategies.

Setting: Number of Accounts (Memory Access Density)

The number of accounts influences the density of memory read- and write-accesses. A smaller number of accounts means that the accesses to the register of account balances, which is modified by each transaction, is clustered in a more compact way. Another consequence is a rise in contention, as the possibility of a concurrent access to a given account rises significantly when decreasing the number of accounts. An increase in the number of accounts decreases the density of memory accessed on the other hand, as the accounts involved in a transaction are spread over a larger memory range. The possibility of a concurrent access decreases with a rise of the available memory range. The possibility of a conflict between threads also decreases and concurrent threads can run better in a parallel fashion as the level of contention decreases.

Setting: Maximum Transaction Length (Transaction Length)

The bank application used in this thesis has been extended by introducing a setting "Max Length", which indicates the number of transfers a transaction can fulfill at most. The exact number of transfers per transaction is determined randomly using a uniform distribution between 1 and "Max Length" during runtime at the begin of each transfer transaction. This means that "Max Length" corresponds to the expected number of transfers done during a transaction and therefore directly relates to the length of a transaction. The length of a transaction can therefore be varied by changing the "Max Length" setting.

Application Runtime

The runtime given is determined by executing a fixed number of transactions per thread. The runtime is determined ten times per parameter set and afterwards averaged using the

arithmetic mean. This accounts for the slightly fluctuating runtime, which occurs due to the non-deterministic nature of the application even when having a fixed set of settings.

Runtime relative to best performing strategy

The runtime relative to the best performing strategy is calculated in the following way:

Let there be

$$rt_1, rt_2, rt_3$$

the runtime rt of three different TM strategies for a given, fixed set of tm-bank settings, respectively, as determined per experimentation. The runtime of the best performing strategy is

$$rt_{best} = min(rt_1, rt_2, rt_3).$$

In succession the runtime relative to the best performing strategy value rr can be calculated for each strategy by

$$rr_x = \frac{rt_x}{rt_{best}} \qquad \forall x \in (1, 2, 3).$$

A rr value of 1.0 marks the best performing strategy for a given set of settings, where as values greater than 1.0 indicate an algorithm performing worse than the best performing strategy. The value is linearly scaled, i.e. an algorithm with a runtime relative to the best performing strategy of 2.0 has double the runtime when compared to the runtime of the best performing strategy. Using this metric simplifies the performance comparison of Transactional Memory strategies.

Relation of program phase, strategy and performance

Program phases can be classified into two types, a high and a low contention type. The following experiment determines the performance of the tm-bank application for each available Transactional Memory strategy. The performance for each strategy is tested using two scenarios, each of which is modeled after the characteristics of a program phase type.

The compared strategies are write-back using encounter time locking (WB-ETL), write-back using commit-time locking (WB-CTL) and write-through using encounter time locking (WT). The insight gained by determining the optimal strategy for each scenario is used later on to parametrize the switching algorithm. The switching algorithm decides which strategy to use during runtime. The decision is based on the detected program phase. The assumption is that the optimal strategy for a high contention program phase, which is detected when running another Transactional Memory application, is the same as that for the high contention scenario and vice versa for the low contention phase type.

Setting	Value
Transfers	1000
Threads	4
Accounts	200
Max Length	8
Runs	10

High contention scenario

Setting	Value
Transfers	1000
Threads	4
Accounts	800
Max Length	2
Runs	10

Low contention scenario

Table 6.1.: tm-bank application settings

The settings of the tm-bank for the two scenarios are shown in table 6.1. The settings differ in the number of accounts and the maximum transaction length. A low number of accounts and a high maximum transaction length yields high contention, where as a high number of accounts combined with a low maximum transaction length results in a low level of contention. All other Transactional Memory settings are set to fixed values and remain unchanged. The high contention scenario exhibits a high rate of aborts and the low contention scenario exhibits a low rate of aborts. The runtime of 10 runs is averaged using the arithmetic mean for each combination of strategy and scenario.

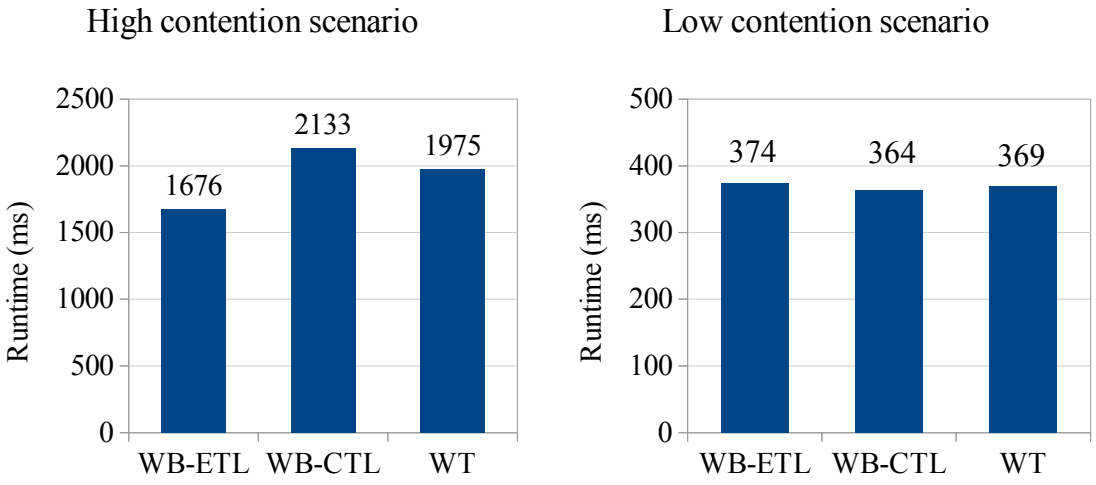


Figure 6.1.: Comparison of tm-bank performance

The experimental results obtained using the XUPv5 board are shown in figures 6.1 and 6.2. They show that the different Transactional Memory strategies exhibit different performance in the two scenarios. An interesting fact is that no single strategy has the best performance

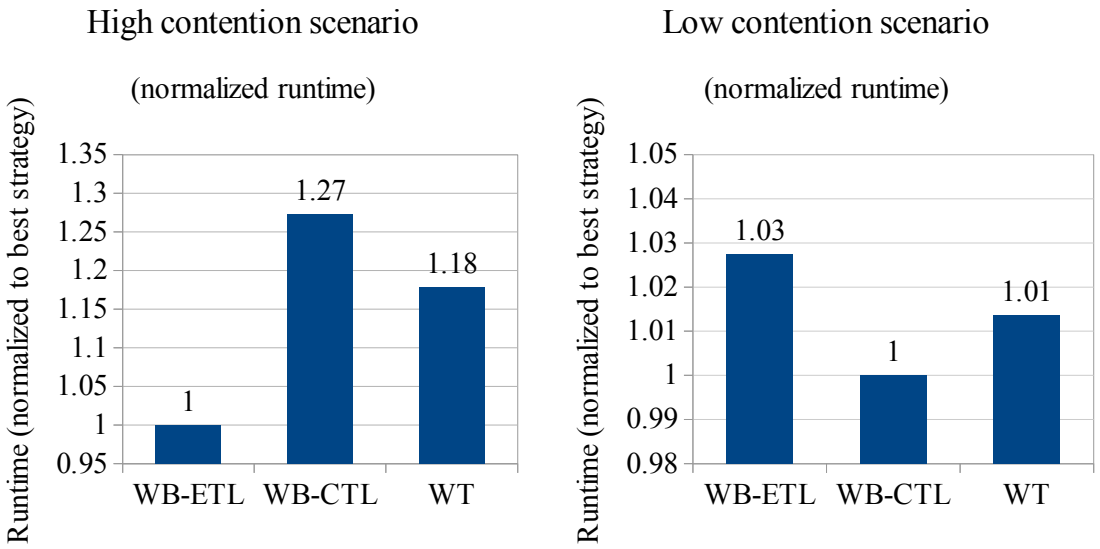


Figure 6.2.: Comparison of tm-bank performance (runtime normalized to best strategy)

for both scenario. There are two winner strategies: WB-CTL exhibits the best performance in the low contention scenario, where as WB-ETL exhibits the best performance in the high contention scenario. These findings mean that the WB-CTL strategy should be used after detecting a low contention phase and correspondingly WB-ETL for a high contention phase.

6.2. Multi-dimensional analysis

This section describes an experiment run on the XUPv5 board. The experiment generalizes the results of the previous section where only two strongly differing scenarios were used. The difference in this experiment is that various scenarios are constructed to model program phases with an intermediate contention level compared to the experiment above.

As shown previously the memory access density and length of transactions can be modeled by varying the number of accounts and the maximum transaction length in the tm-bank application. A set of parameters is composed of a given value for the number of accounts and maximum transaction length. The individual influence on the performance of the three TM design can be determined by benchmarking the tm-bank application in the following way:

A set of parameters is chosen by varying the number of accounts in the range between 200 accounts and 1000 accounts with a step size of 200 and varying the maximum transaction length in the range of 2 transfers and 8 transfers with a step size of 2. The chosen bandwidth

of settings covers scenarios with low, medium and high levels of contention. An increase in the number of accounts and a decrease of the maximum transaction length corresponds to a lower level of contention and vice versa. All other Software and Hardware Transactional Memory parameters are kept fixed in this experiment.

$accounts = \{200, 400, 600, 800, 1000\}$

$max\_tx\_length = [2, 4, 6, 8]$

A matrix of runtime values rt is obtained by running the tm-bank application with this fixed set of parameters for each design:

$$\begin{aligned} rt_{WB-CTL} &= \forall x \in accounts \quad \forall y \in max\_tx\_length \quad \text{tm-bank-wb-ctl(x, y)} \\ rt_{WB-ETL} &= \forall x \in accounts \quad \forall y \in max\_tx\_length \quad \text{tm-bank-wb-etl(x, y)} \\ rt_{WT} &= \forall x \in accounts \quad \forall y \in max\_tx\_length \quad \text{tm-bank-wt(x, y)} \end{aligned}$$

The runtime for each set of parameters is determined ten times and afterwards averaged using the arithmetic mean. This accounts for the slightly fluctuating runtime, which occurs due to the non-deterministic nature of the application even when having a fixed set of settings.

A run for a given design strategy results in a set of data points and takes about 20 minutes to execute on the XUPv5 FPGA board. The collected data points are normalized using the algorithm described in Section 6.1 to allow an easy comparison between the three designs.

A three-dimensional representation of the resulting sets of data points is normally a suitable method of comparing the three design visually. The x- and y-axes represent the changing tm-bank parameters and the z-axis represents the runtime for a given pair of parameters. The plane in the figures is created by connecting each data point with its neighbors. The shape of the surface of the plane represents the changing runtime behavior.

The runtime of the three designs can now be visually compared by combining the shapes of the surfaces into one figure and letting them intersect, as shown in Figure 6.3. The bottom most plane at each data point represents the fastest design for this set of parameters. As can be seen there is no single best design for all sets of parameters. Furthermore it is difficult to determine the best design in this 3D illustration, as the planes conceal each other.

Thus another more suitable visualization had to be found: The following figures 6.4 and 6.5 visualize and compare the runtime of the three Transactional Memory conflict detection strategies by creating quadrilaterals[7] spanning the area between the data points. The value and therefore color of each quadrilaterals is derived by taking the four neighboring data points in each corner of the quadrangle and calculating the mean sum of the values of

[7] A polygon with four edges and four corners.

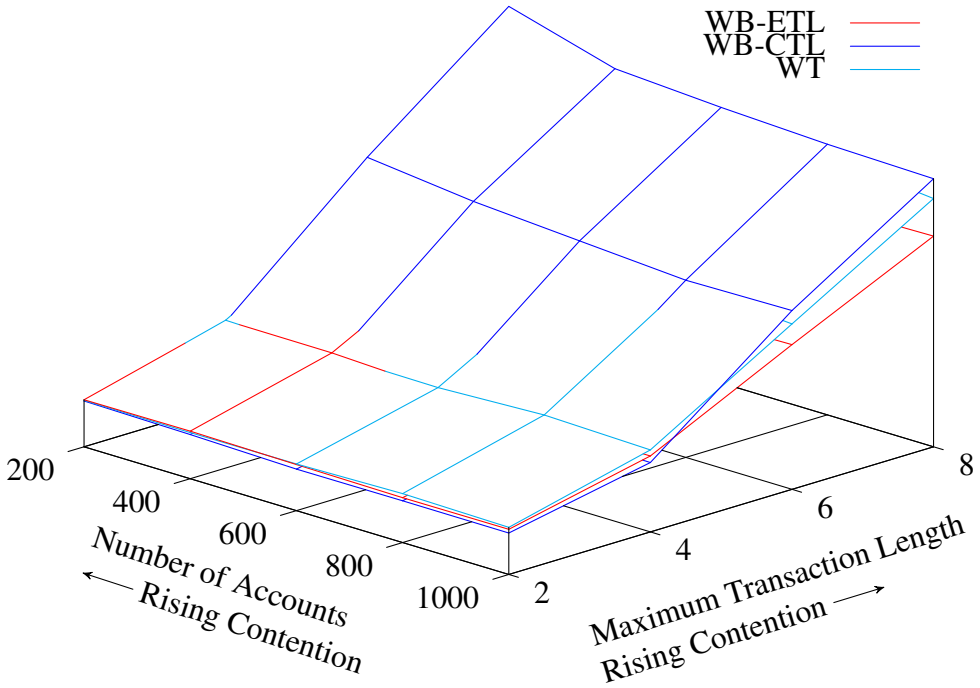


Figure 6.3.: Comparison of Transactional Memory strategies (Average runtime)

the data points. The data points are normalized according to the previously described algorithm. The z-value of 1.00 denotes the best performing algorithm for a given set of parameters, as denoted there. The data point in the top left corner represents the highest contention level and the data point in the lower right corner stands for the lowest contention level of the application. The data points in between are intermediate levels of contention.

The resulting grid is coarse, as seen in Figure 6.4. Decreasing the step size (i.e. the difference between parameters) results in a grid with a finer resolution, but also exponentially increases the time for a run. It is thus not a feasible approach. A good estimation of a run with smaller step sizes can be obtained by using linear interpolation between the data points. In Figure 6.5 a 10 times interpolation was used to get a meaningful visualization. The results show that WB-ETL is a good strategy in the high contention areas, where as WB-CTL is good in the lower contention areas. The performance of WB-CTL is highly dependent on the maximum transaction length. Lower transaction lengths favor the use of the WB-CTL algorithm. WT has a good performance in some of the high contention areas, but otherwise exhibits an inferior relative performance compared to the WB-ETL strategy. The results support the findings of the previous section that WB-CTL should be used for low contention program phases and WB-ETL correspondingly for high contention program phases.

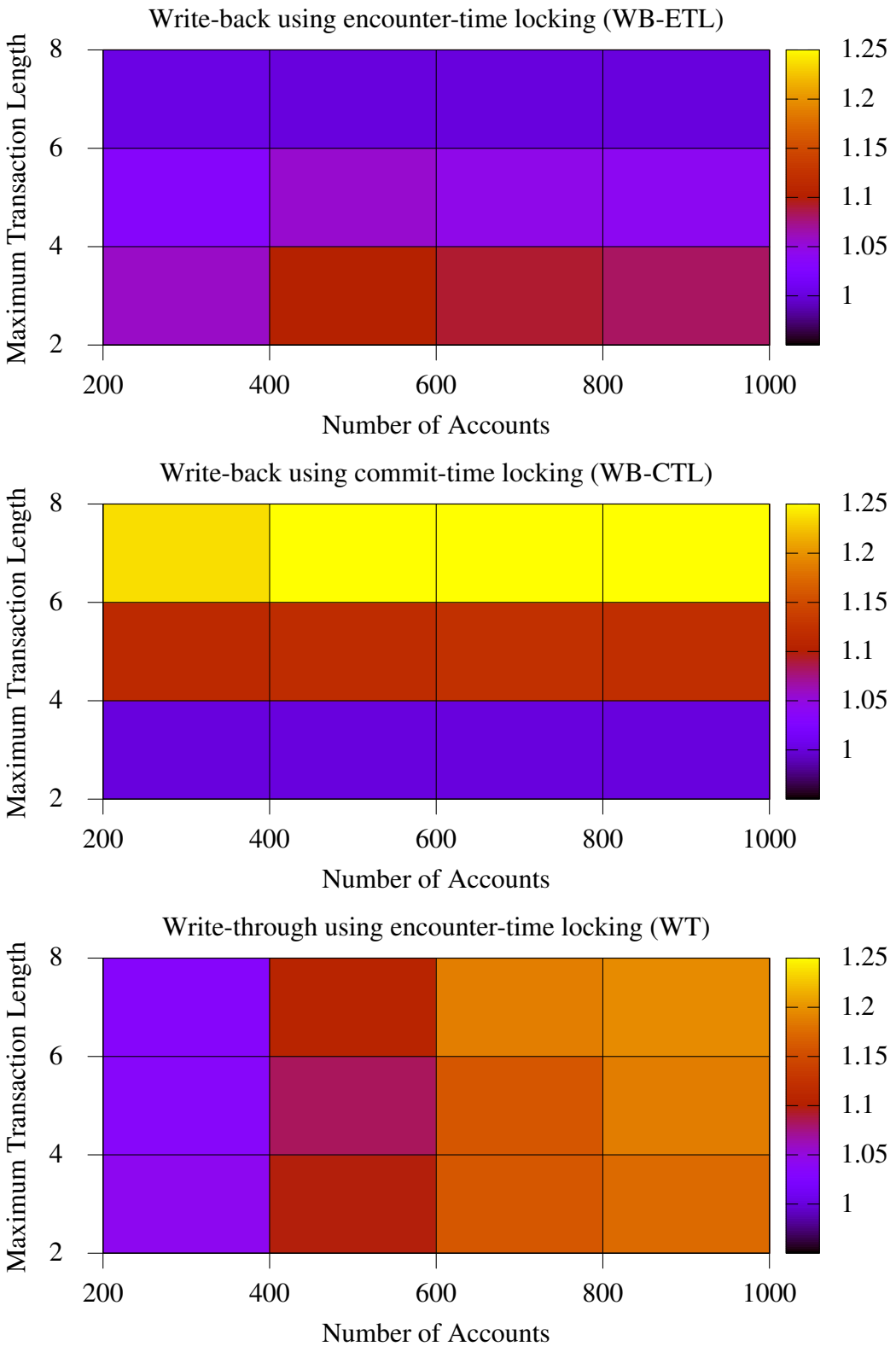


Figure 6.4.: Comparison of WB-ETL, WB-CTL and WT runtime relative to best performing algorithm

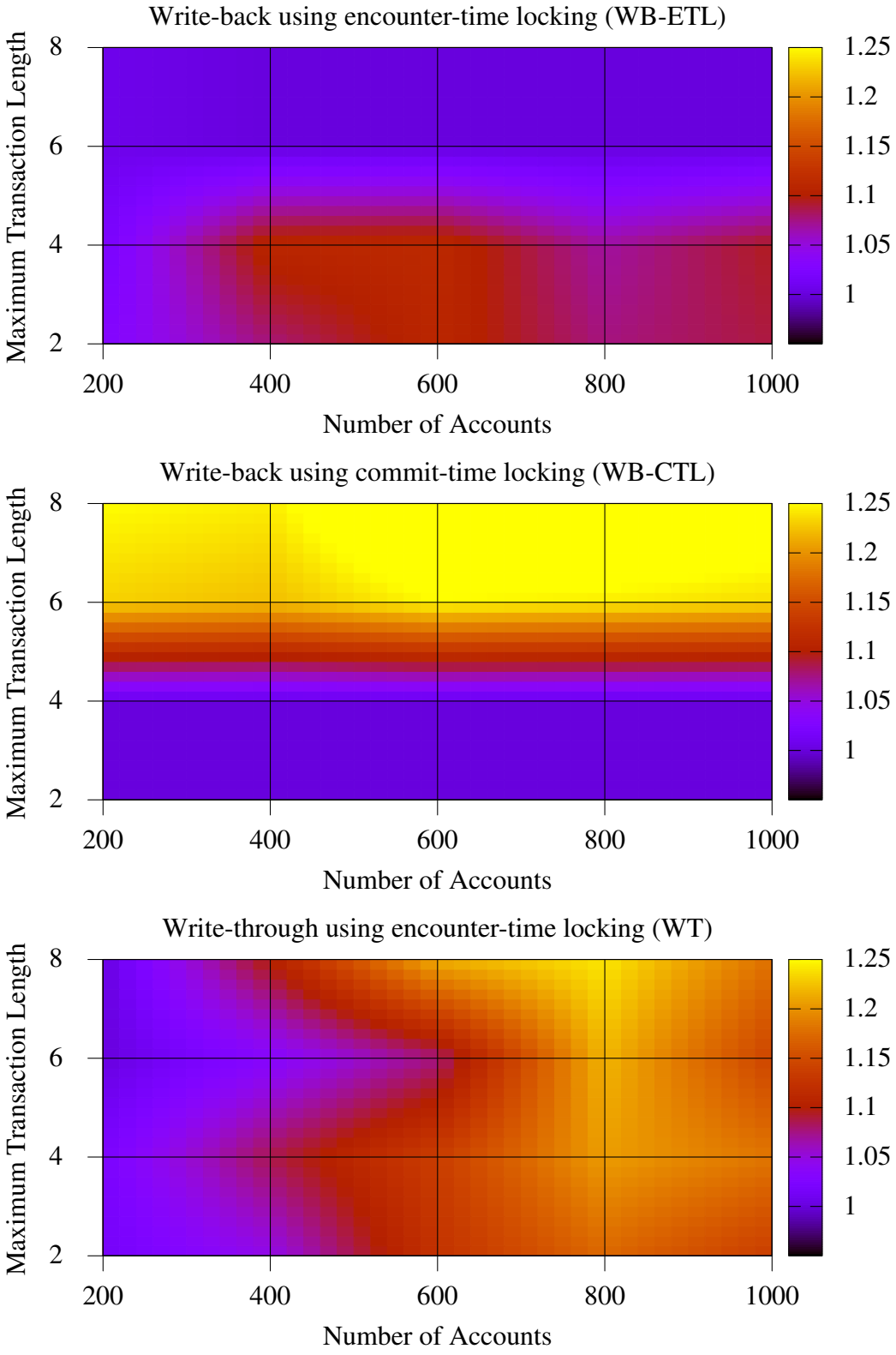


Figure 6.5.: Comparison of WB-ETL, WB-CTL and WT runtime relative to best performing algorithm (interpolation 10x)

6.3. The case for visualization

One of the goals of this diploma thesis is to automatically optimize the performance of Transactional Memory applications with a phased execution. It is thus essential to select Transactional Memory application for optimization, which actually have a phased execution. Visualizing the transactional behavior is a good technique to determine if the Transactional Memory applications has phased execution or not. It helps to gain detailed insight into the different program phases and is therefore an important and essential step in the process of designing and implementing an adaptive system. This insight is used later to parametrize the algorithms necessary for a dynamic optimization process exploiting program phases in Transactional Memory applications. A short introduction to the used visualization tool is shown in the following sections. How visualization can exactly be used to gather knowledge is also showcased with several examples in the following sections.

6.4. Event-based tracing of many-core systems on commodity hardware

The proposed event-based tracing framework is a data source suited for a visualization tool, as it exhibits a low probe effect. Unfortunately no high-bandwidth interface was available on the XUPv5 board. Such a high-bandwidth connection is necessary to transfer the generated events without affecting normal system operations. A buffering of events in main memory would be needed. This slows down the access to memory areas by a running application, thus disturbing normal system operations and creating a large probe effect. It was therefore rejected. Simulating the TMbox system in a VHDL simulator is another possible approach. This approach was also rejected, as the simulation process is very slow. Tracing the run of a normal Transactional Memory application would have taken a long time. A viable approach in getting a suitable data source for the visualization of Transactional Memory applications is to run a tracing framework on commodity hardware. This tracing framework should exhibit a low overhead when enabling the tracing. Such a suitable framework is described by Schindewolf et al. in *"Capturing Transactional Memory Application's Behavior - The Prerequisite for Performance Analysis"* [40]. A set of post-processing tools, which enable the visualization of Software, Hardware and Hybrid Transactional Memory applications, have been adapted to work with the data files generated by this tracing framework. The post-processing and visualization tools can thus work with both the tracing framework proposed by Schindewolf et al. and the tracing framework proposed in this diploma thesis.

The tracing framework by Schindewolf et al. is similar to the event-based tracing framework for Hybrid Transactional Memory in this diploma thesis. A major difference is that the

former tracing framework solely uses software components to implement it's tracing features. It can thus run on commodity state-of-the-art general purpose processors without Hardware Transactional Memory support.

Event tracing for multi-core systems without dedicated hardware units requires a high level of storage bandwidth to store the possibly massive amount of events generated during application runtime. The bandwidth needed can exceed a level of hundreds of megabytes per second. This amount of bandwidth can be easily provided by writing to a RAM-based volatile storage, but as a drawback the time of an application run is severely limited by the amount of RAM dedicated to this storage. Non-volatile storage on hard disk drives (HDD) and solid state drives (SSD) circumvents the drawback of a short application runtime by providing huge amounts of available storage area. But HDDs do not provide enough bandwidth for storing the event traces of a many-core system and SSD-based system are very expensive.

As a solution to this problem the framework, as described in [40], is designed to handle the high level of storage bandwidth needed by compressing the events on-the-fly during runtime using multiple compression threads. The compression scheme employed is the LZO real-time data compression library[8]. The LZO library is optimized to maximize the compression throughput and minimize the time taken for compression, while still providing an adequate compression ratio. In this design each application thread executing transactions is associated with a group of compression threads, each of which takes a set of generated events, compresses them and writes them in a non-linear order to a file system. This approach reduces the bandwidth needed by a factor of approximately 29. It is usually sufficient to have a mapping of one transaction thread to two or three compression threads to achieve an acceptable level of overhead introduced through the use of this proposed compression scheme.

Before visualization the compressed event trace has to be ordered in a linear way in the first place, i.e. the sets of events are ordered in ascending order of generation time. The compressed sets can then be decompressed and written in a linear fashion to non-volatile storage. As this process runs after the to be profiled application has finished execution its runtime is of no great importance.

6.5. Visualization of transactional behavior

Further post-processing is needed for the visualization of the transactional behavior of an application. The input data for the following tools can be an event stream from either the previously mentioned Software Transactional Memory tracing framework or an

[8] Oberhumer: *LZO real-time data compression library*
http://www.oberhumer.com/opensource/lzo/

event stream from the tracing framework, which is presented in this diploma thesis. The event stream from the Software Transactional Memory tracing framework contains only events related to Software Transactional Memory, where as the event stream from tracing framework in this thesis can contain both Software and Hardware Transactional events.

After the supervised application has finished running, a post processing tool called "BusEventConverter" [26] reads and checks the event stream, rebuilds Software, Hardware, Hybrid Transactional Memory and application states, generates statistics and outputs data suitable for later processing with an visualization and analysis tool, explained in the next section. The post-processing tool can also used with an implementation of the low overhead event-based tracing framework on another system on chip design.

The BusEventConverter post-processing tool

The event stream, which is generated by the tracing units, is not directly usable for visualization and analysis. The post processing tool BusEventConverter generates data usable for visualization and analysis. Multiple passes process the input data set step by step. The passes are also called "generators", because a new set of data is emitted in each pass. Each generator uses the input data set and the data generated by previous generator passes, modifies it and generates a new data set for the next generator. Specific finite-state machines (FSM) are used to regenerate the Transactional Memory states depending on whether the event is associated with a transaction in Hardware- or Software Transactional Memory mode. The following two figures 6.6 and 6.7 show these FSMs.

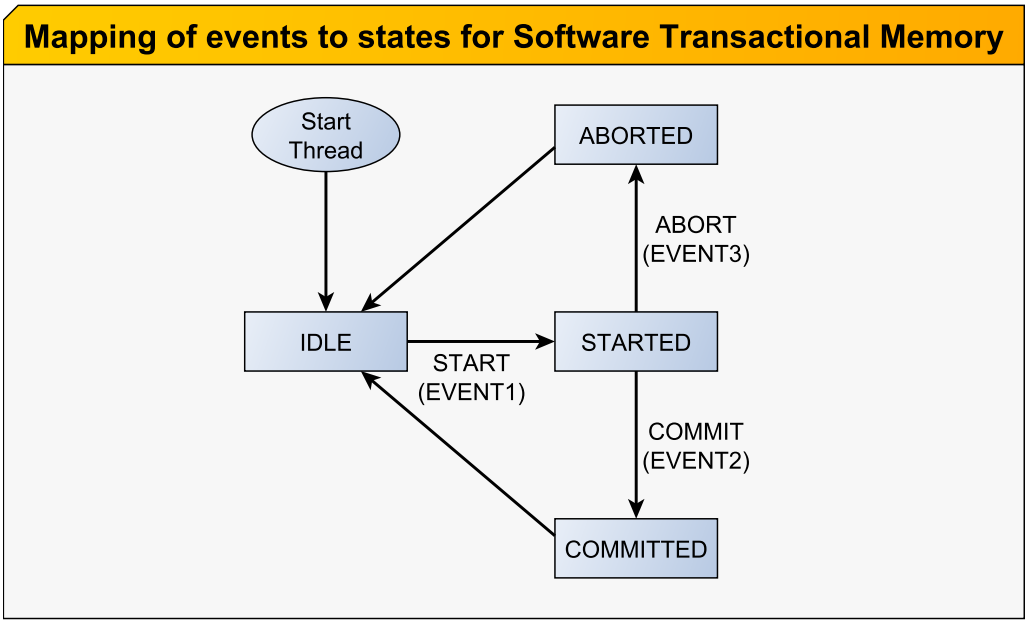


Figure 6.6.: Mapping of Software Transactional Memory events

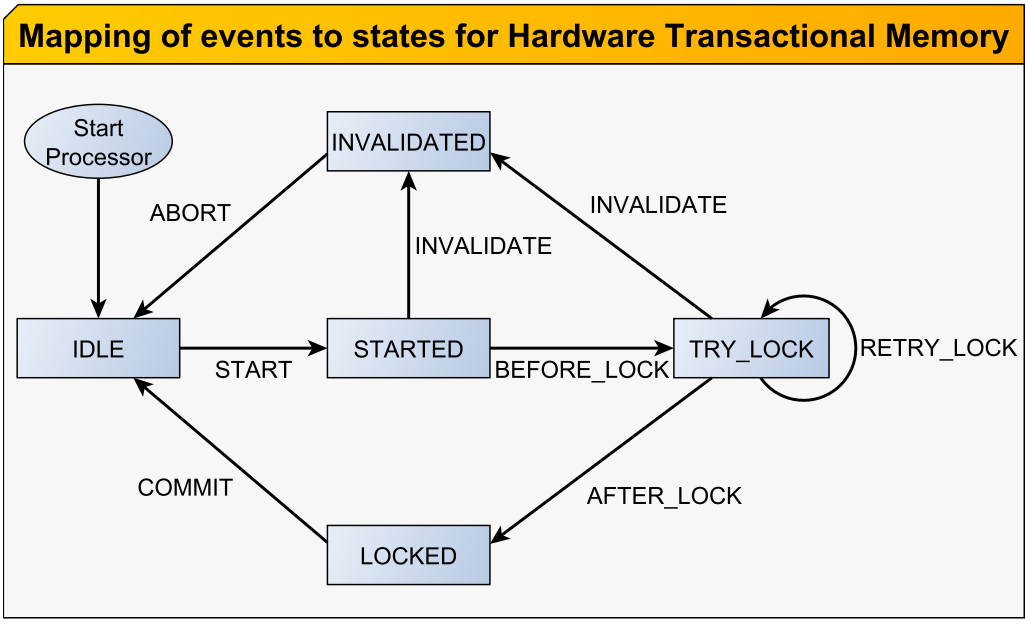


Figure 6.7.: Mapping of Hardware Transactional Memory events

Paraver: A Transactional Memory visualization and analysis program

Paraver[9] is a visualization and analysis program, developed at the Barcelona Supercomputing Center (BSC). It is normally used to analyze MPI and OpenMP programs running on multi-processor and cluster systems. An example of such a cluster is "MareNostrum" [10], one of the most powerful supercomputers in Europe, located at the BSC. Its excellent visualization and data processing capabilities allowed to re-purpose it for Software, Hardware and Hybrid Transactional Memory visualization and analysis in the scope of this diploma thesis. The monitoring of long running applications, which run on many-core systems, creates particularly large traces. Paraver is designed to handle these traces efficiently. The user can freely zoom in and out of traces, displaying only interesting parts of the visualization of a trace.

This section showcases two visual analysis examples, which show how insight is gained about the characteristics of Transactional Memory program phases by using visual analysis.

Paraver structure and features

The Paraver visualization and analysis workflow is shown in Figure 6.8.

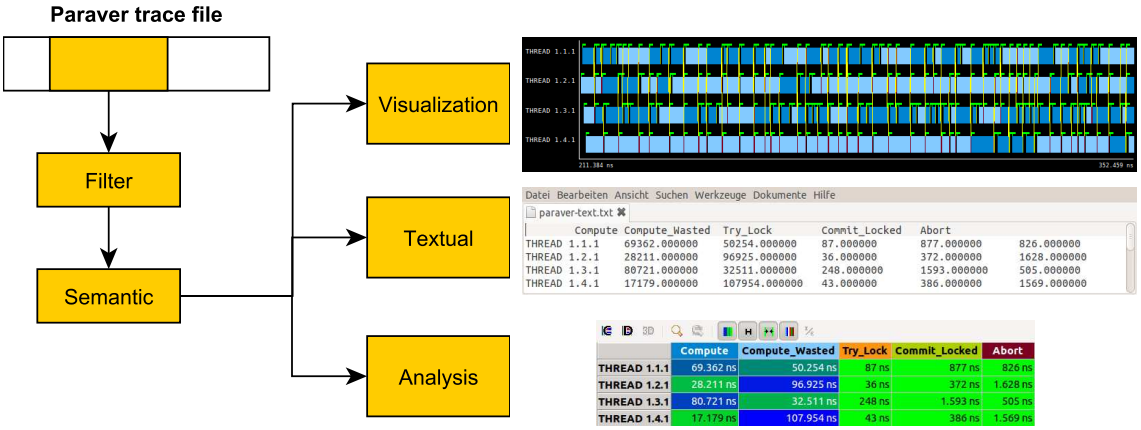


Figure 6.8.: Paraver workflow (Figure derived from Paraver website)

The filter module selects a partial set of records from the trace file. This is useful for the visualization and analysis of a part of the states and events, for instance to analyse only aborted transactions.

[9] http://www.bsc.es/paraver
[10] http://www.bsc.es/plantillaA.php?cat_id=200

The semantic module afterwards assigns a numeric value to each state and event. This can, for instance, be used to compute a thread- or system-level overview.

The visualization, textual and analysis modules comprise the main parts of Paraver. They are used for drawing the time line figures in the follow sections.

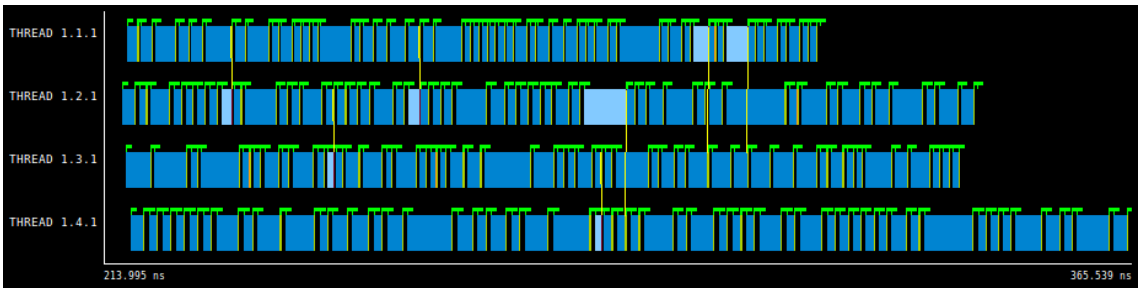
Paraver trace file

Post-processing the event stream with the BusEventConverter tool creates a Paraver trace file, which is the prerequisite for visualization and analysis. A Paraver trace file contains a header and a set of records. There are three record types defined:

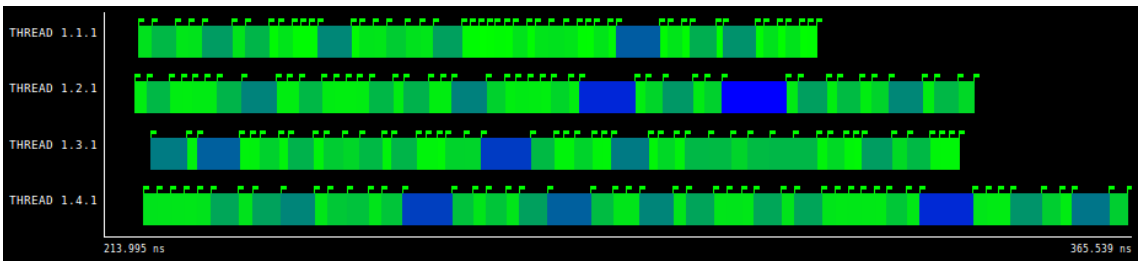
- **State:** Record containing a state value of a thread and its duration. Paraver associates no semantics to the encoding of the state field.
- **Event:** This record represents a punctual event that occurs during the execution of a specific thread. It is encoded into type and value. Paraver associates no semantics to the encoding of these fields.
- **Communication:** Record containing a pair of events and a causal relationship between them.

A trace file contains the Paraver event definitions it's first segment, while the second segment contains the state definitions and the third segment (not shown) contains the communication definitions. Each definition is associated with a processor core number and contains a timestamp. A specification of the Paraver file format can be found in [26].

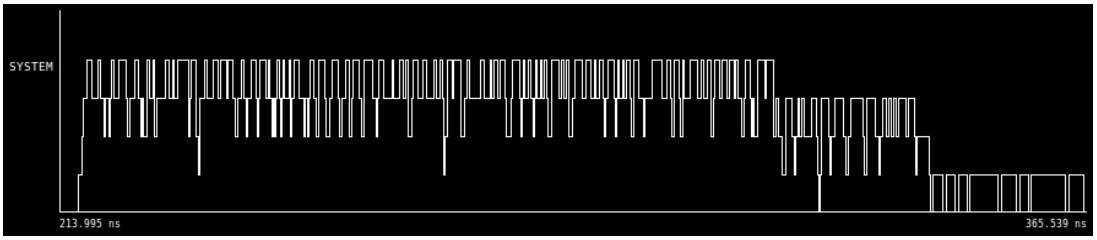
Visual analysis example I - Hardware Transactional Memory usage



(a)



(b)



(c)

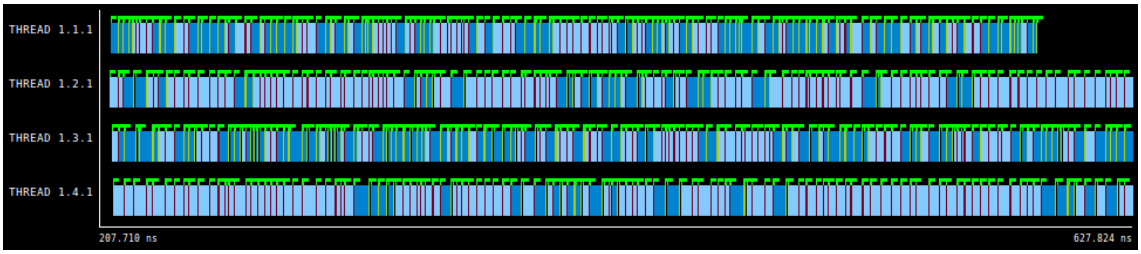
Figure 6.9.: Program trace (a) and corresponding rate of commits (b) and number of used Hardware Transactional Memory units (c)

Interpretation: These traces show an application with a low amount of aborts. The time scale of Figures (a) to (c) is the same.

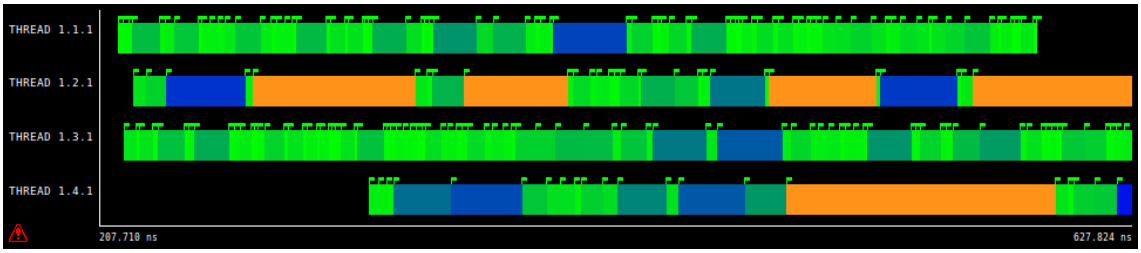
Figure 6.9b shows the rate of commits: Various shades of green correspond to a high rate of commits and a short duration of committing transactions. Blue shades indicate time periods with a low rate of commits and a high duration of committing transactions.

Figure 6.9c has been created using the semantic analysis module of Paraver and shows the number of actively used Hardware Transactional Memory Units on a system level scale over time. During most of the runtime the application uses 2 to 4 Hardware Transactional Memory units. Later after completion of the first thread the usage changes to between 1 and 3 used Hardware Transactional Memory units with an average of 2 used units. Threads 2 and 3 finish computation nearly at the same time. During the last phase of execution only one Hardware Transactional Memory unit is actively used by the last thread.

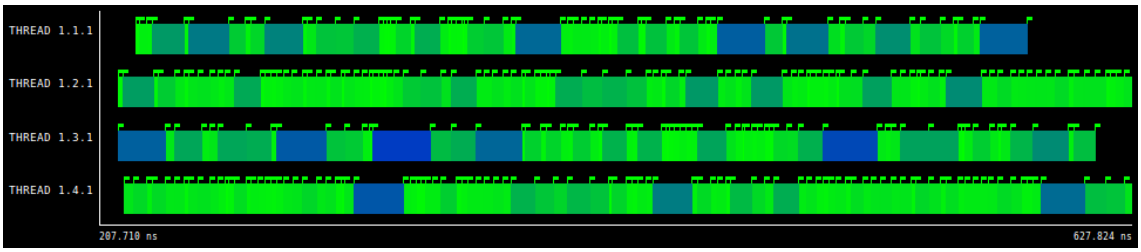
Visual analysis example II - Contention analysis



(a)



(b)



(c)

Figure 6.10.: Starvation of two threads: Program trace (a) and corresponding rates of commits (b) and aborts (c)

Interpretation: This time a trace of an application with a high amount of aborts is shown. Light blue parts in the timeline of Figure 6.10a correspond to wasted work, i.e. work done in aborted transactions. The Figures 6.10b and 6.10c have been created using the filter module of Paraver. These two Figures show a high rate of aborts (bright green parts) and a low rate of commits (blue and yellow parts) on threads 2 and 4. Further analysis showed that threads 1 and 3 were mainly causing a large amount of aborts in threads 2 and 4. The negative effects of the dependencies between these two groups of threads should therefore be optimized.

6.6. Optimizing a Transactional Memory application by exploiting program phases

The following sections show how an Transactional Memory application with program phases can be successfully optimized by utilizing the adaptive process and the units from the framework proposed and implemented in this diploma thesis.

Intruder: A benchmark for Transactional Memory

The Intruder application is a generally accepted benchmark for Transactional Memory performance. It is part of the Stanford Transactional Applications for Multi-Processing (STAMP) benchmark suite. The benchmark suite currently consists of eight transactional memory applications implementing algorithms found in real-life applications. The Intruder application, for example, implements network intrusion detection: Streams of incoming data are received from a network and analyzed for particular attack patterns. The following figures 6.11 and 6.12 show a visualization of the transactional behavior of the Intruder program when running with 4 threads on commodity hardware. The used visualization tool is Paraver.

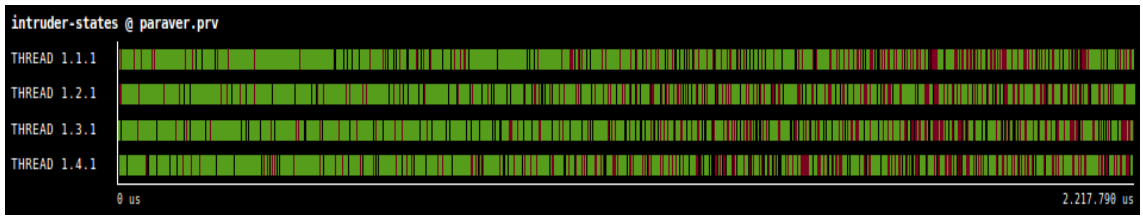
Visually analyzing program phases

The green parts in Figure 6.11a are transactions which successfully commit, where as the red parts contain transactions which abort and subsequently restart. The black parts indicate sections of the application with no transactional activity.

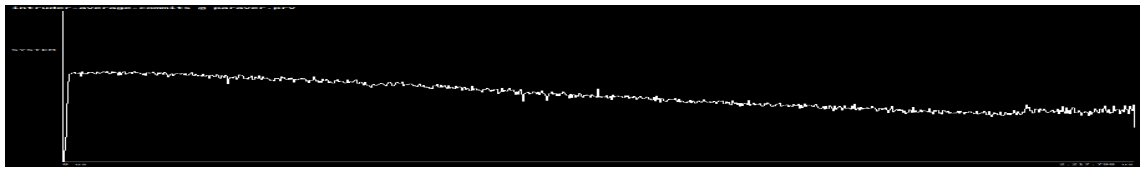
At a first glance we can see visually that the amount of contention, the red parts, rises beginning in the middle of the application runtime. The lower graph shows the ratio of aborts and commits of the same Intruder run. As we can see the amount of contention at the start of the application is low and later on rises steeply. This indicates a change between phases during runtime.

By building a graph of the abort/commit ratio, like in Figure 6.12, it can be easily seen that there are two large program phases. The application starts with a low contention phase and the ratios stabilize at about a third into runtime. The ratio rises again after about half of the runtime indicating the begin of the high contention phase.

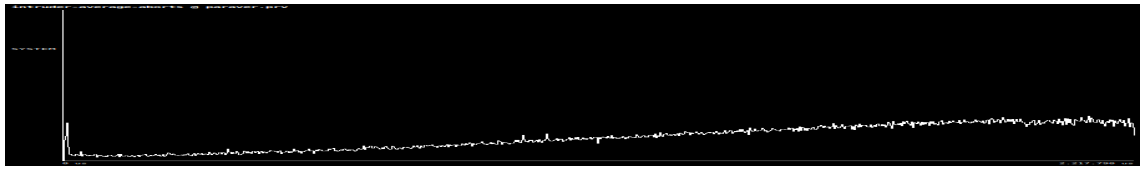
The insight gained by analyzing the program phases visually can now be used to improve the application performance when running on the adaptive implementation of the TMbox Hybrid Transactional Memory system.



(a)



(b)



(c)

Figure 6.11.: Intruder: Visualization of transactional behavior (a), level of commits (b) and level of aborts (c)

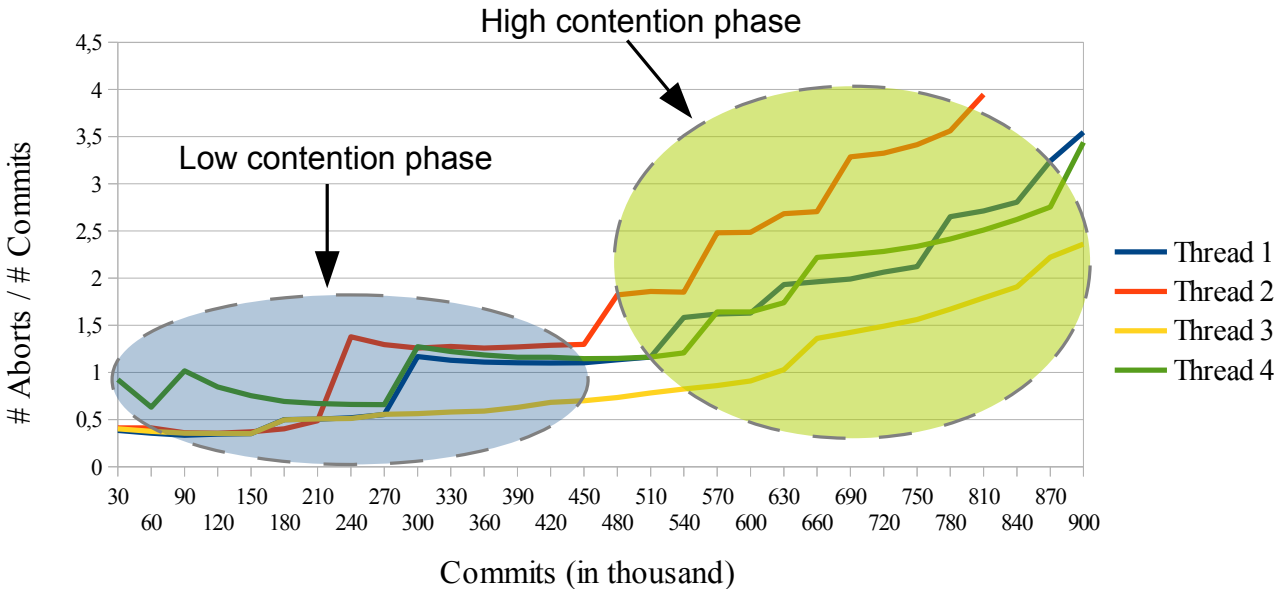


Figure 6.12.: Intruder: Transactional behavior (Ratio of aborts and commits)

Improving performance through the use of adaptivity

The following experiment run on the XUPv5 shows how different switching policies affect the performance of the Intruder application. They also show that the performance of a Transactional Memory application with program phases is increased by choosing an adaptive strategy. The adaptive strategy exploits program phases by matching a set of Transactional Memory strategies to each phase and switching between them dynamically during runtime. The adaptive strategy uses the adaptive process and the hardware units from the framework as proposed in this diploma thesis.

In this experiment a thresholding algorithm is used as a phase detection algorithm. It detects two types of program phases: A low and a high contention phase. This is done by reading the summarized Transactional Memory activity from the statistics hardware unit and computing a ratio between transaction aborts and commits. The algorithm is parametrized using one parameter, the threshold ratio. If the computed ratio is above the threshold ratio (i.e. the detected phase has a ratio higher than the threshold) the phase detection algorithm indicates a high contention phase to the switching algorithm and vice versa for a low contention phase. The value of the threshold ratio is based on the insight gained through the analysis of the visualization of the Intruder application.

Based on the phase type indication from the phase detection algorithm the switching algorithm decides which set of Transactional Memory strategies is optimal for the current application phase. Optimistic strategies are better in low contention phases and pessimistic ones are better in high contention phases. The switching takes place during application runtime.

Three different strategies for the switching algorithm are benchmarked and compared:

- **Static strategy**

  The static strategy selects a Transactional Memory strategy at the very start of the application and leaves it unchanged during runtime. This corresponds to the usual procedure in which Transactional Memory applications are executed: The used strategies are set to fixed values during compile time and can not be varied later on. This means the switching algorithm has a "no operation" function and the adaptivity features are not used.

- **Adaptive strategy 1 (fixed switching point in time)**

  The adaptive strategy 1 uses a fixed switching point in time (after a third of the application's runtime has elapsed). The strategies are switched only once. The switching point has been determined visually by analyzing Figure 6.12.

- **Adaptive strategy 2 (dynamic switching points)**

  This adaptive strategy uses write-back with encounter time locking as the optimal set of strategies for a high contention phase and one of the other strategies for a low contention phase. The used strategy may be switched multiple times during runtime depending on the detected program phase when using this adaptive strategy. The decision to use WB-ETL for the high contention phases is based on the results of sections 6.1 and 6.2.

The results of running the Intruder program on a 4 core system on the XUPv5 board with the three strategies for the switching algorithm are shown in Figure 6.13.

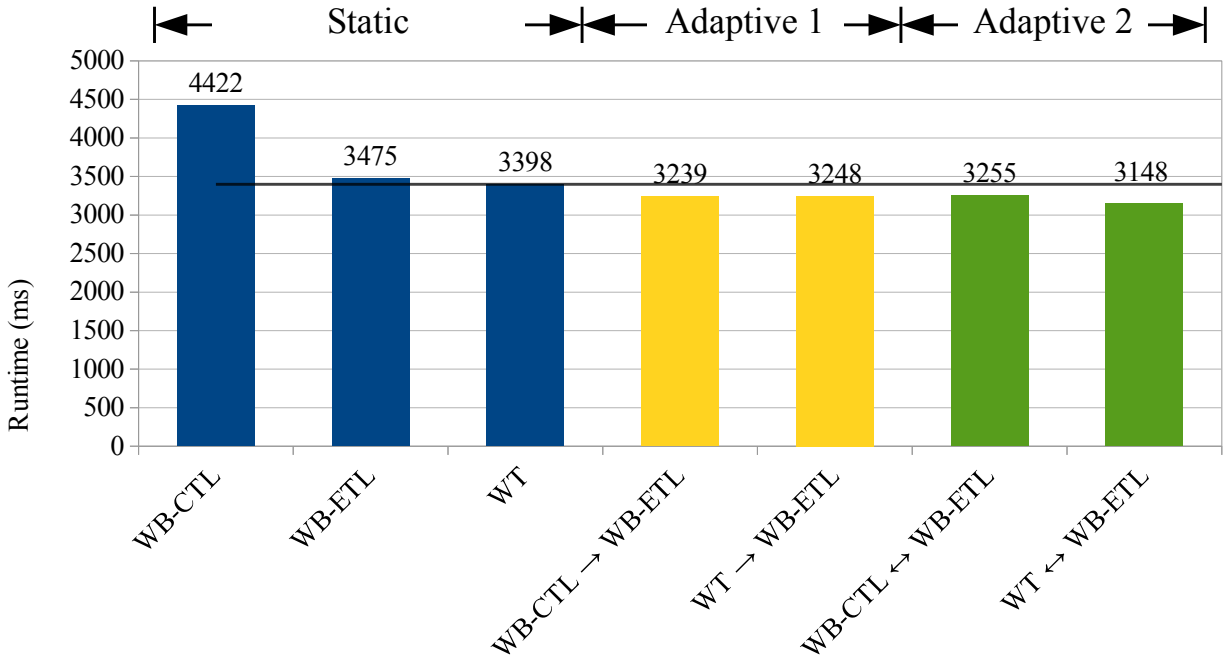


Figure 6.13.: Intruder: Comparison of static and adaptive switching strategies

The blue bars show the results of the static policies. The yellow bars show the results of the dynamic policy with a fixed switching point in time. The green bars show the results of the adaptive strategy with dynamic switching points. The horizontal black line represents the runtime of the best static strategy.

A result is that in each case the adaptive strategies are better than the best static strategy. Using adaptivity here resulted in a relative improvement of up to 7 % when compared to the best static policy. The improvement relative to the other static strategies is up to 28 %. This shows the benefits of using an adaptive system when running Transactional Memory applications with program phases.

7. Summary

7.1. Conclusion

This diploma thesis contributes to the state-of-the-art in the field of Transactional Memory by showing that some transactional memory applications exhibit program phases and how these phases can be detected during runtime by using application tracing. It is also shown that the performance of a Transactional Memory application with program phases can be increased by using an adaptive process which dynamically selects appropriate Transactional Memory strategies. The adaptive process exploits program phases by matching a set of Transactional Memory strategies to each phase and switching between them dynamically during runtime. The hardware units from the framework proposed in this diploma thesis are used for this purpose. The experimental results originate from an implementation on an FPGA-based Hybrid Transactional Memory system.

7.2. Outlook

This diploma thesis reviewed a Hybrid Transactional Memory system, where exactly one application runs on the system at a given time. The application therefore always has uncontested access to all hardware units. An interesting aspect for further research in a currently rather unexplored area is how multiple Transactional Memory applications running simultaneously on a Hybrid Transactional Memory system can compete with each other on the usage of Hardware Transactional Memory units.

Another interesting topic is to look at the other Transactional Memory applications in the STAMP benchmark suite and show if they also exhibit phased execution. If some applications exhibit phases the next step would be to apply the adaptive process to them too.

Different Transactional Memory strategies and parameters can be varied by utilizing FPGA runtime reconfiguration capabilities. This allows to change the hardware constraints dynamically during application runtime. To keep this project in the scope of a diploma thesis a decision has been made to reduce the explorable design space by keeping Hardware Transactional Memory policy and parameters fixed. Based on the results of this project a

future follow-up project could work on determining the feasibility and impact of dynamic Hardware Transactional Memory reconfiguration.

7.3. Acknowledgements

I would like to thank my supervisor Professor Dr. Wolfgang Karl for giving me this interesting topic as a diploma thesis. I'm also grateful for his support of my stay abroad in Barcelona/Spain and when taking part in the EuroTM workshops in Bern/Switzerland and Prague/Czech Republic. I also want to especially thank Dr. Martin Schindewolf for his advice, constructive criticism and encouragement during the development of this thesis. As well I would like to thank Osman Unsal, Adrián Cristal, Oriol Arcas and Nehir Sonmez for their support during my stay at the Barcelona Supercomputing Center. And last but not least I'd like to thank my family for supporting and encouraging me throughout my computer science studies.

Glossary

BEE3 (Berkeley Emulation Engine, version 3) Multi-FPGA system designed to be used to develop and evaluate new computer architectures

BRAM (Block RAM) Dedicated FPGA on-chip memory storage unit

HTM (Hardware Transactional Memory) Special ISA instructions allow to run some parts of a Transactional Memory runtime system directly in hardware; constraint-bound (e.g. capacity constraints: hardware can handle a specific read-/write-set size, larger transactions fail)

HybridTM (Hybrid Transactional Memory) Transactional Memory runtime combining Hardware Transactional Memory and Software Transactional Memory support; transactions run in Hardware Transactional Memory mode and fall back to Software Transactional Memory mode when encountering Hardware Transactional Memory constraints

STM (Software Transactional Memory) Transactional Memory runtime using standard ISA instructions; no modification of hardware necessary; usually slower than Hardware Transactional Memory but with more permissive contraints

TM (Transactional Memory) Programming paradigm, which allows applications to run atomic blocks using shared data concurrently; uses optimistic conflict checking to ensure atomicity and consistency

XUPv5 (Xilinx University Program Development System) The XUPv5 FPGA board is a general purpose evaluation and development platform with on-board memory and industry standard connectivity interfaces. It features the Virtex-5 XC5VLX110T FPGA.

A. Appendix

A.1. Control and data flow of common Transactional Memory strategies

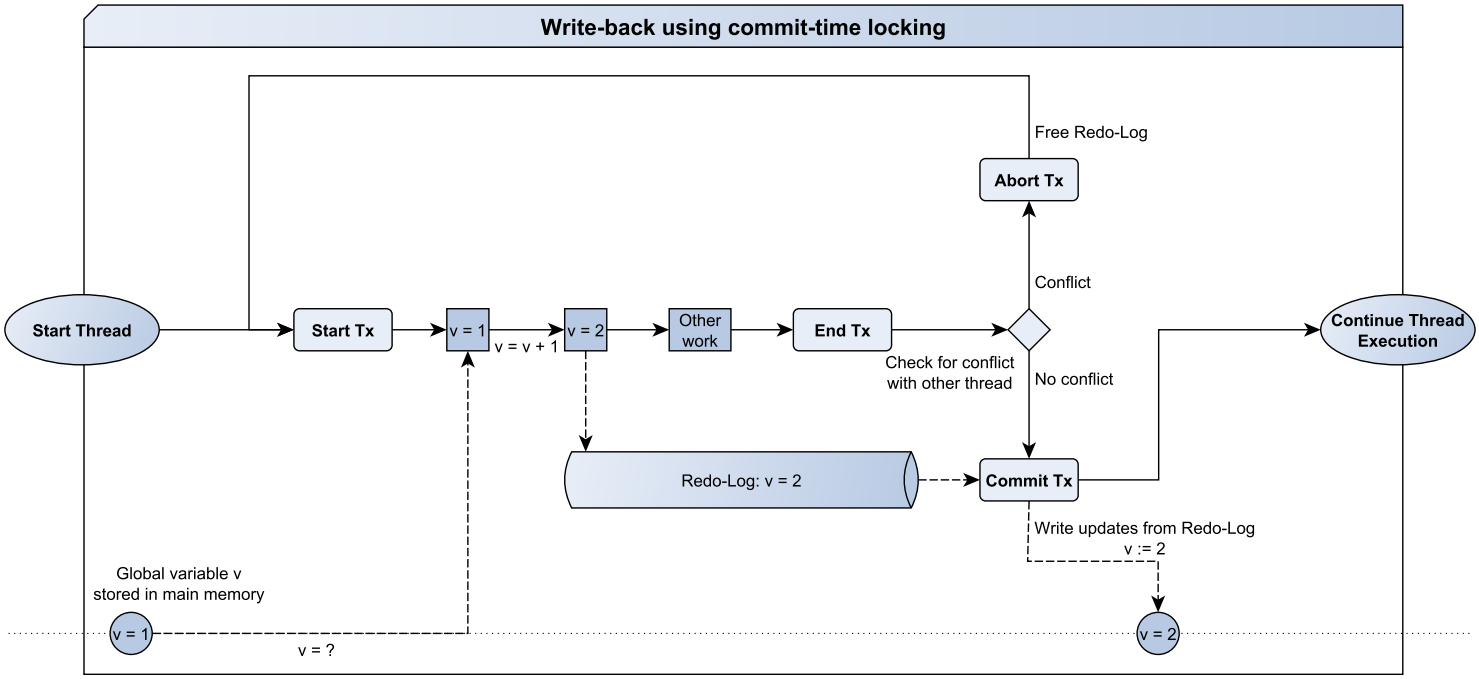


Figure A.1.: Write-back using commit-time locking (WB-CTL)

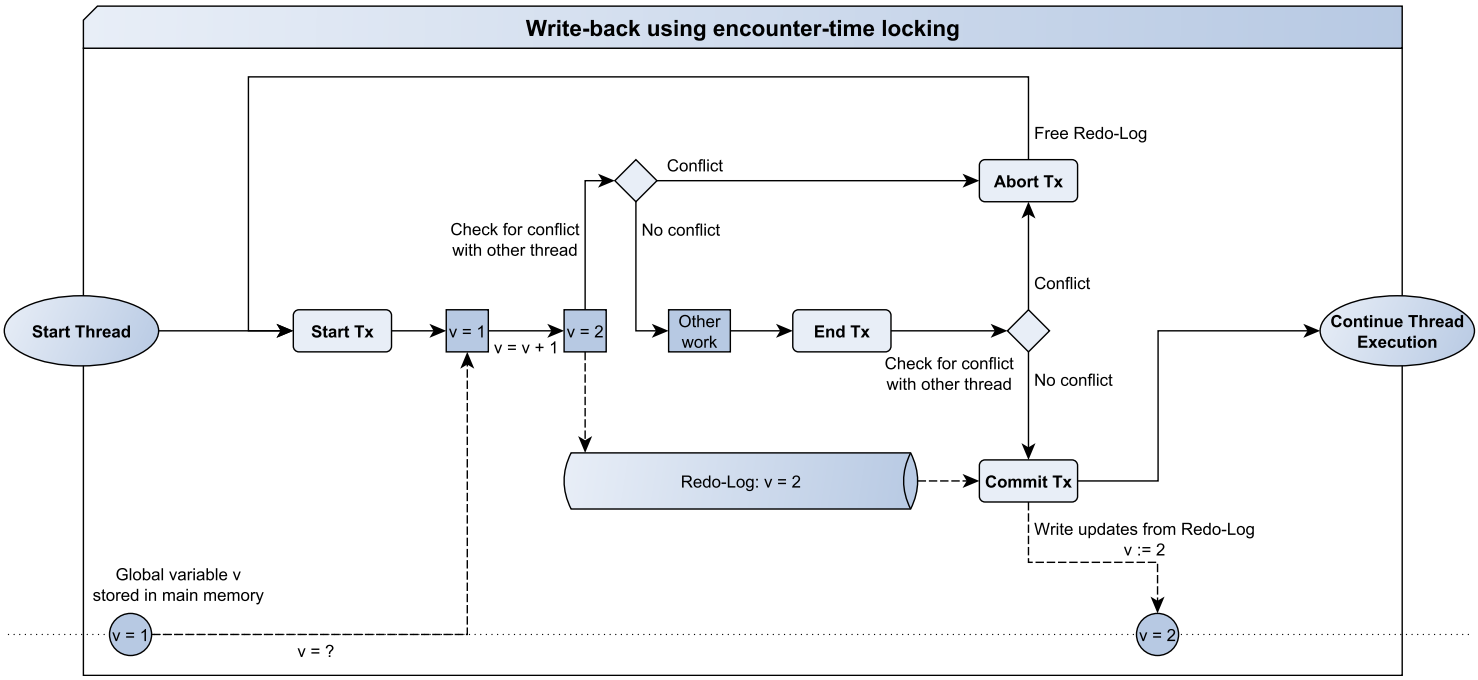


Figure A.2.: Write-back using encounter-time locking (WB-ETL)

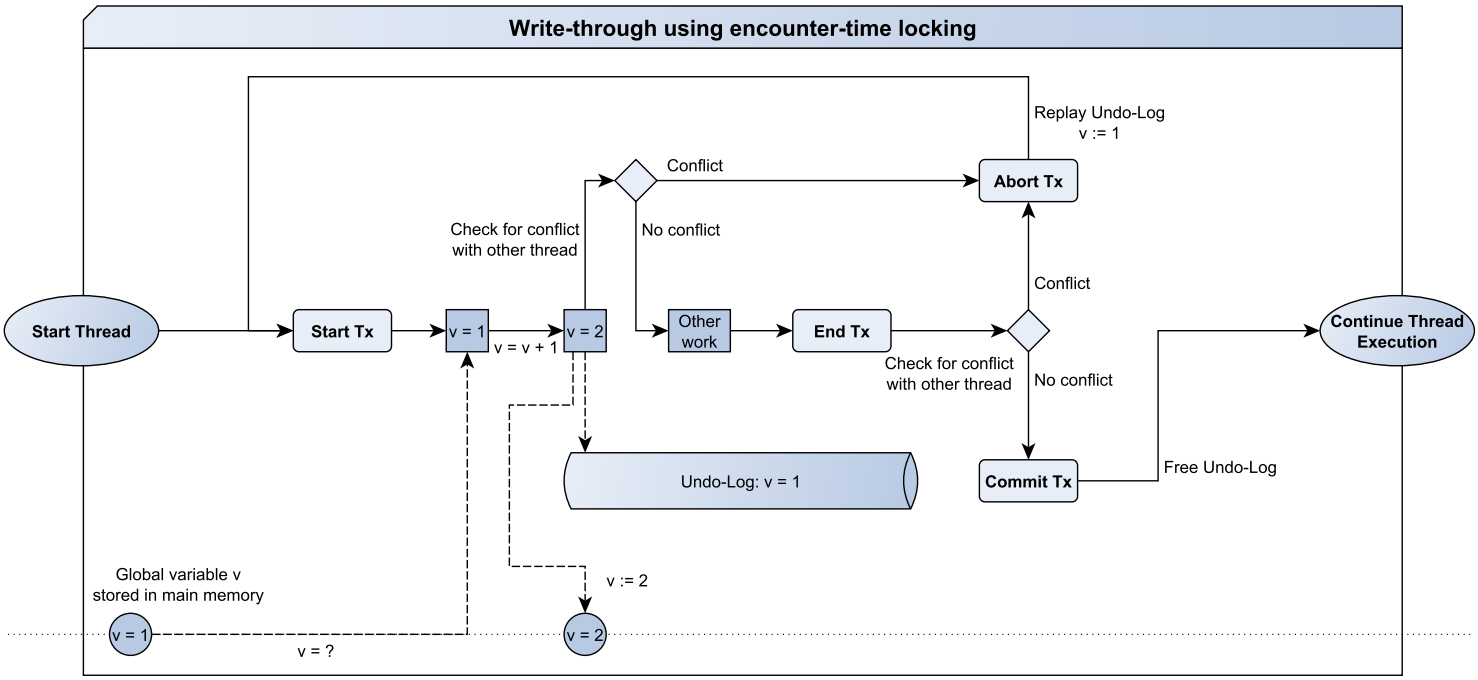


Figure A.3.: Write-through using encounter-time locking (WT)

A.2. Sample run of an application

A sample run of the CoreMark benchmark application is shown in listing A.1. The boot loader output is also shown.

```
Bootloader compiled on Jun 21 2013 18:52:07
Loader initializing ...
Calling main loader function ...
Erasing main memory ...
Reading header and data ...
################################################
################################################
################################################
########################
header.size = 0x00004de8
header.crc = 0x884c24dd
Finished loading!
Computed CRC32 = 0x884c24dd
Returned from main loader function.
Successfull program load.
{0} f000 <-> 19000
{0} CPU 0 is present.
{0} start bench
{0} 2K performance run parameters for coremark.
{0} CoreMark Size : 666
{0} Total ticks : 1126816958
{0} Total time (secs): 22
{0} Iterations/Sec : 45
{0} Iterations  : 1000
{0} Compiler version : GCC4.4.1
{0} Compiler flags : COMPILER_FLAGS
{0} Memory location : STACK
{0} seedcrc     : 0xe9f5
{0} [0]crclist  : 0xe714
{0} [0]crcmatrix : 0x1fd7
{0} [0]crcstate : 0x8e3a
{0} [0]crcfinal : 0xd340
{0} Correct operation validated. See readme.txt for run
   and reporting rules.
{0} Press 'r' to reset...
```

Listing A.1: Boot loader and CoreMark transcript

Lines numbered from 1 to 15 contain the output of the boot loader. Starting at line 16 control is transferred to the application program (CoreMark).

A.3. VHDL interface of bus controller unit

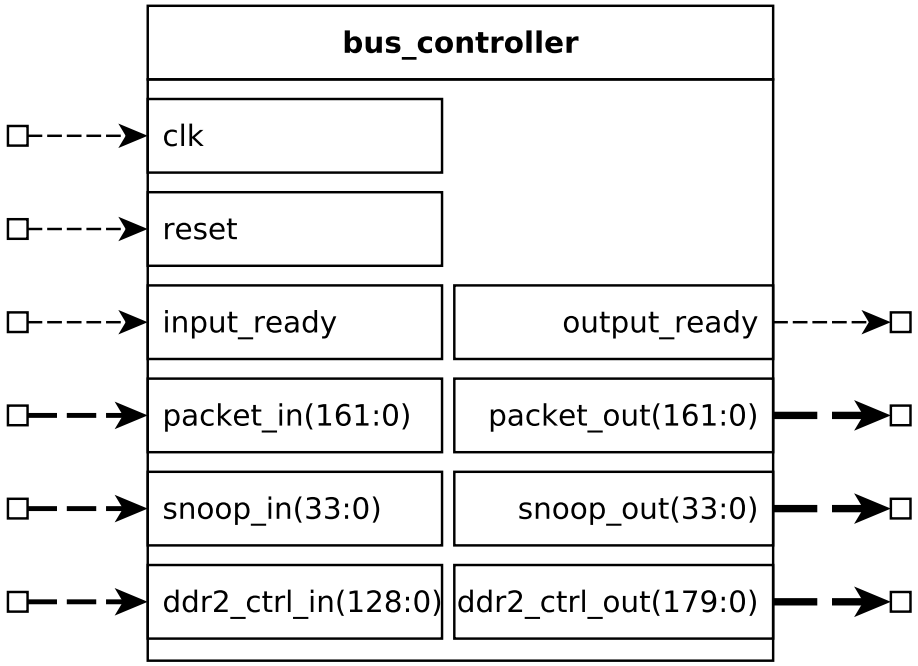


Figure A.4.: Interface of bus controller unit

A.4. Memory regions of the adaptive Hybrid Transactional Memory system

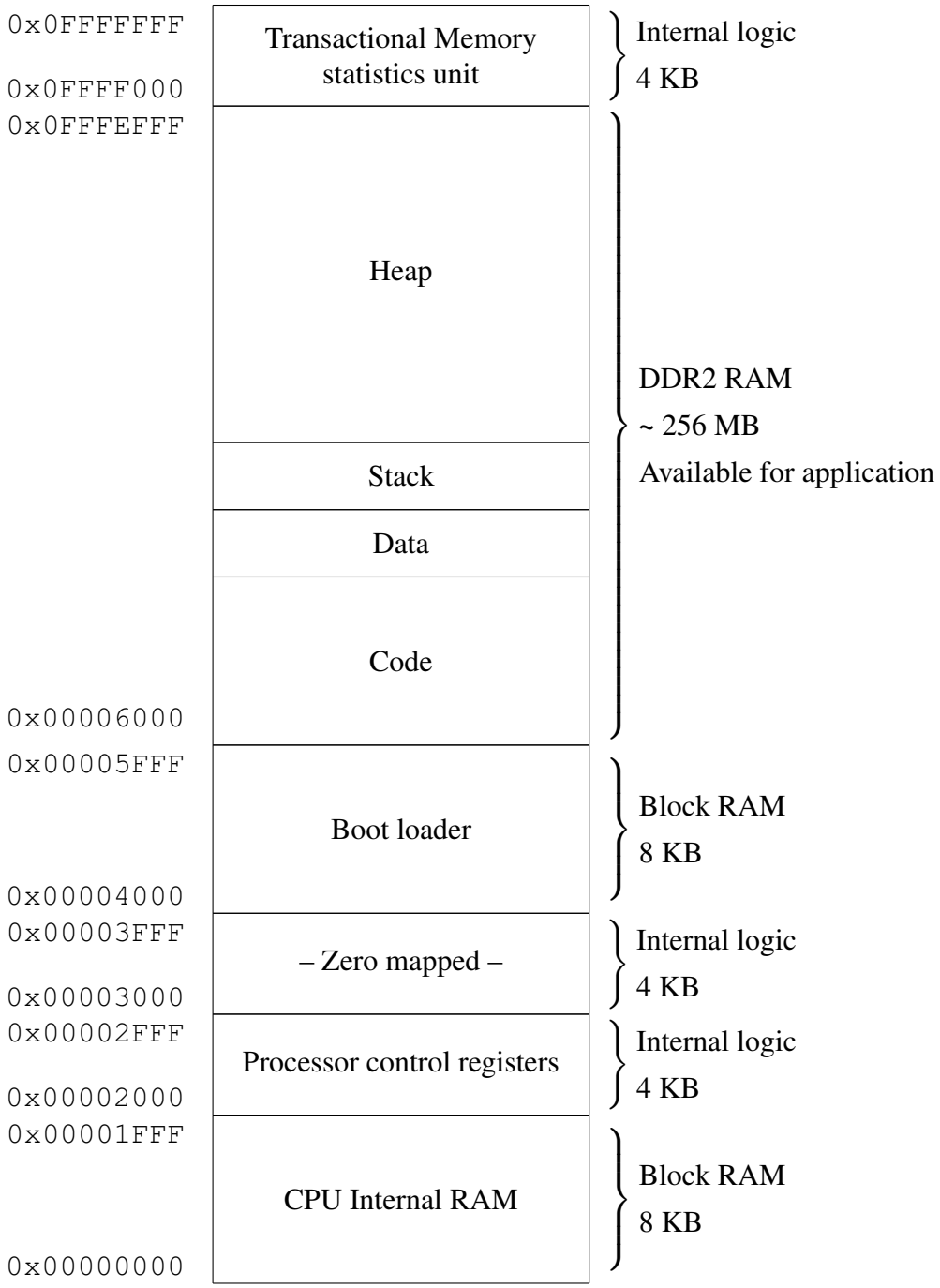


Figure A.5.: Memory regions and corresponding backing of the adaptive Hybrid Transactional Memory system. Note: The figure is not drawn to scale.

A.5. Implementation: Number of lines of code

The files taken from the initial TMbox implementation for the BEE3 board are listed in table A.2 and a short description is given for each file. These hardware units were only slightly modified, generally to fix existing bugs. The units marked with a star indicate the units, which were modified during implementation of the event-based tracing framework.

The units designed, implemented and tested during the scope of this diploma thesis are listed in tables A.1 and A.3.

Lines of code	File name	Description
TMbox_support/ - Test units and Testbenches		
58	bus_checker2.vhd	
63	bus_checker.vhd	
168	bus_controller_tb.vhd	
51	bus_request_generator.vhd	
113	ddr2_bram_tb.vhd	
55	debounce_tb.vhd	
88	mem_test_request_generator.vhd	
64	tmbox_bram_tb.vhd	
37	uart_tb.vhd	
131	uart_top.vhd	
828	Lines of code in directory	

Table A.1.: Lines of code: TMbox_support - Test units

Lines of code	File name	Description
TMbox/		
68	alu.vhd	Arithmetic and Logic Unit
143	bus_mux_vm.vhd	Register selection logic
261	cache.vhd	CPU data and instruction cache
33	common_defines.vhd	Common type definitions
81	common_functions.vhd	Common function definitions
691	control_vm.vhd	Instruction decode stage
506	cpu_vm.vhd	CPU top level unit
1533	honeycomb0.vhd *	Processor core unit 0
1422	honeycomb.vhd *	Other processor core units
120	log_fifo_bram.vhd *	log_fifo (using Block RAMs)
154	log_fifo.vhd *	log_fifo (using distributed memory)
243	mem_ctrl_vm.vhd	Memory write and fetch stage
761	mlite_pack_vm.vhd	Common definitions for CPU core
210	mult.vhd	Multiplication unit
113	pc_next_vm.vhd	Program Counter Unit
150	pipeline_vm.vhd	Processor pipeline control
327	reg_bank_vm.vhd	Register bank
382	ringbus_node.vhd	Ringbus node (connection between processor core and ring bus)
65	shifter.vhd	Shifter unit
271	tmu.vhd	Hardware Transactional Memory Unit
7534	Lines of code in directory	

Table A.2.: Lines of code: TMbox

Lines of code	File name	Description
TMbox_support/		
383	bus_controller.vhd	Bus Controller
71	cdc_ddr2_to_sys.vhd	Clock Domain Crossing DDR2 controller to system clock domain
79	cdc_sys_to_ddr2.vhd	Clock Domain Crossing System to DDR2 controller clock domain
34	clock_divider.vhd	Clock divider
32	clock_enable.vhd	Clock enable
110	ddr2_bram_sim.vhd	DDR2 controller backed by Block RAMs Simulation mode
83	ddr2_bram_synth.vhd	DDR2 controller backed by Block RAMs Synthesis mode
85	ddr2_loader_bram_sim.vhd	DDR2 controller backed by SO-DIMM Simulation mode
232	ddr2_loader_bram.vhd	DDR2 controller backed by SO-DIMM Synthesis mode
48	debounce.vhd	Debounce unit (used for reset signal input)
80	pll_clk_gen_bram.vhd	Clock generation and control (when using Block RAMs as main memory backing)
137	pll_clk_gen_ddr2.vhd	Clock generation and control (when using SO-DIMM as main memory backing)
44	reset_generator.vhd	Controls reset signal generation
184	ringbus.vhd	Ringbus (connection between processor cores and bus controller)
194	tmbox_bram.vhd	Top Unit (BRAM mode)
275	tmbox_ddr2.vhd	Top Unit (DDR2 mode)
59	tm_control.vhd	TM control unit
191	uart.vhd	Universal Asynchronous Receiver/Transmitter
2321	Lines of code in directory	

Table A.3.: Lines of code: TMbox_support

B. Bibliography

[1] Christopher J. Rossbach, Owen S. Hofmann, and Emmett Witchel. Is transactional programming actually easier? In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 47–56, 2010.

[2] Victor Pankratius and Ali-Reza Adl-Tabatabai. A Study of Transactional Memory vs. Locks in Practice. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, SPAA '11, pages 43–52, 2011.

[3] Shekhar Borkar and Andrew A. Chien. The Future of Microprocessors. *Communications of the ACM*, 54(5):67–77, May 2011.

[4] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569 ff., September 1965.

[5] Maged M. Michael. The Balancing Act of Choosing Nonblocking Features. *ACM Queue*, 11(7), July 2013.

[6] Paul E. McKenney. Structured Deferral: Synchronization via Procrastination. *Communications of the ACM*, 56(7):40–49, July 2013.

[7] Mathieu Desnoyers, Paul E. McKenney, Alan S. Stern, Michel R. Dagenais, and Jonathan Walpole. User-Level Implementations of Read-Copy Update. *IEEE Transactions on Parallel and Distributed Systems*, 23(2):375–382, 2012.

[8] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th annual international symposium on computer architecture*, ISCA '93, pages 289–300. ACM, 1993.

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

[10] Dave Christie, Jae-Woong Chung, Stephan Diestelhorst, Michael Hohmuth, Martin Pohlack, Christof Fetzer, Martin Nowack, Torvald Riegel, Pascal Felber, Patrick Marlier, and Etienne Rivière. 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, pages 27–40. ACM, 2010.

[11] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. In *Proceedings of the IEEE International Symposium on Workload Characterization*, IISWC '08, September 2008.

[12] Bratin Saha, Ali-Reza Adl-Tabatabai, and Quinn Jacobson. Architectural Support for Software Transactional Memory. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-39, pages 185–196. IEEE Computer Society, 2006.

[13] Dave Dice, Yossi Lev, Mark Moir, and Daniel Nussbaum. Early Experience with a Commercial Hardware Transactional Memory Implementation. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS '09, pages 157–168. ACM, 2009.

[14] Nehir Sonmez, Oriol Arcas, Otto Pflucker, Osman S. Unsal, Adrian Cristal, Ibrahim Hur, Satnam Singh, and Mateo Valero. TMbox: A Flexible and Reconfigurable 16-Core Hybrid Transactional Memory System. In *Proceedings of the 2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*, FCCM '11, pages 146–153. IEEE Computer Society, 2011.

[15] Christian Jacobi, Timothy Slegel, and Dan Greiner. Transactional Memory Architecture and Implementation for IBM System Z. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '12, pages 25–36. IEEE Computer Society, 2012.

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

[17] Yehuda Afek, Amir Levy, and Adam Morrison. Programming with Hardware Lock Elision. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '13, pages 295–296. ACM, 2013.

[18] Zhaoguo Wang, Hao Qian, Haibo Chen, and Jinyang Li. Opportunities and pitfalls of multi-core scaling using Hardware Transaction Memory. In *Proceedings of the 4th Asia-Pacific Workshop on Systems*, APSys '13. ACM, 2013.

[19] Wilson W. L. Fung, Inderpreet Singh, Andrew Brownsword, and Tor M. Aamodt. Hardware Transactional Memory for GPU Architectures. In *44th IEEE/ACM International Symposium on Microarchitecture*, MICRO '11, 2011.

[20] Daniel Cederman, Philippas Tsigas, and Muhammad Tayyab Chaudhry. Towards a Software Transactional Memory for Graphics Processors. In James P. Ahrens, Kurt Debattista, and Renato Pajarola, editors, *EGPGV*, pages 121–129. Eurographics Association, 2010.

[21] Mohammad Ansari, Kim Jarvis, Christos Kotselidis, Mikel Lujan, Chris Kirkham, and Ian Watson. Profiling Transactional Memory Applications. In *Proceedings of the 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, pages 11–20. IEEE Computer Society, 2009.

[22] Jaewoong Chung, Hassan Chafi, Chi Cao Minh, Austen Mcdonald, Brian D. Carlstrom, Christos Kozyrakis, and Kunle Olukotun. The Common Case Transactional Behavior of Multithreaded Programs. In *Proceedings of the 12th International Conference on High-Performance Computer Architecture*, 2006.

[23] Oriol Arcas, Philipp Kirchhofer, Nehir Sonmez, Martin Schindewolf, Osman S. Unsal, Wolfgang Karl, and Adrian Cristal. A Low-Overhead Profiling and Visualization Framework for Hybrid Transactional Memory. In *Proceedings of the 2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, FCCM '12, pages 1–8. IEEE Computer Society, 2012.

[24] Ferad Zyulkyarov. Programming, Debugging, Profiling and Optimizing Transactional Memory Programs (PhD Thesis). 2011.

[25] Hassan Chafi, Chi Cao Minh, Austen McDonald, Brian D. Carlstrom, JaeWoong Chung, Lance Hammond, Christos Kozyrakis, and Kunle Olukotun. TAPE: A Transactional Application Profiling Environment. In *Proceedings of the 19th annual international conference on Supercomputing*, ICS '05, pages 199–208. ACM, 2005.

[26] Philipp Kirchhofer. Enhancing an HTM System with HW Monitoring Capabilities. *Study Thesis, Department of Informatics, Karlsruhe Institute of Technology*, 2011.

[27] Mathias Payer and Thomas R. Gross. Performance Evaluation of Adaptivity in Software Transactional Memory. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '11, pages 165–174. IEEE Computer Society, 2011.

[28] Yossi Lev, Mark Moir, and Dan Nussbaum. PhTM: Phased Transactional Memory. *Second ACM SIGPLAN Workshop on Transactional Computing*, 2007.

[29] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic Performance Tuning of Word-Based Software Transactional Memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 237–246. ACM, 2008.

[30] Justin E. Gottschlich, Maurice P. Herlihy, Gilles A. Pokam, and Jeremy G. Siek. Visualizing Transactional Memory. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, PACT '12, pages 159–170. ACM, 2012.

[31] Oriol Arcas, Nehir Sonmez, Gokhan Sayilar, Satnam Singh, Osman S. Unsal, Adrian Cristal, Ibrahim Hur, and Mateo Valero. Resource-bounded multicore emulation using Beefarm. *Microprocessors and Microsystems*, 36(8):620–631, 2012.

[32] Pascal Felber, Christof Fetzer, Patrick Marlier, and Torvald Riegel. Time-Based Software Transactional Memory. *IEEE Transactions on Parallel and Distributed Systems*, 21(12):1793–1807, 2010.

[33] C. A. R. Hoare. Monitors: An Operating System Structuring Concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[34] Yehuda Afek, Ulrich Drepper, Pascal Felber, Christof Fetzer, Vincent Gramoli, Michael Hohmuth, Etienne Riviere, Per Stenstrom, Osman Unsal, Walther Maldonado Moreira, Derin Harmanci, Patrick Marlier, Stephan Diestelhorst, Martin Pohlack, Adrian Cristal, Ibrahim Hur, Aleksandar Dragojevic, Rachid Guerraoui, Michal Kapalka, Sasa Tomic, Guy Korland, Nir Shavit, Martin Nowack, and Torvald Riegel. The Velox Transactional Memory Stack. *IEEE Micro*, 30(5):76–87, September 2010.

[35] A. Dewey. VHSIC Hardware Description (VHDL) Development Program. In *20th Conference on Design Automation*, pages 625–628, 1983.

[36] Rishiyur Nikhil. Bluespec System Verilog: efficient, correct RTL from high level specifications. In *Proceedings of the Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design*, MEMOCODE '04, pages 69–70, 2004.

[37] John D. Davis, Charles P. Thacker, and Chen Chang. BEE3: Revitalizing Computer Architecture. *Microsoft Technical Report*, 2009.

[38] N. Sharif, N. Ramzan, F. K. Lodhi, O. Hasan, and S. R. Hasan. Quantitative analysis of State-of-the-Art synchronizers: Clock domain crossing perspective. In *7th International Conference on Emerging Technologies (ICET)*, pages 1–6, 2011.

[39] David F. Bacon, Rodric Rabbah, and Sunil Shukla. FPGA Programming for the Masses. *Communications of the ACM*, 56(4):56–63, April 2013.

[40] Martin Schindewolf and Wolfgang Karl. Capturing Transactional Memory Application's Behavior - The Prerequisite for Performance Analysis. In *Proceedings of the 2012 international conference on Multicore Software Engineering, Performance, and Tools*, MSEPT '12, pages 30–41. Springer-Verlag, 2012.