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

Enhancing an HTM System with HW Monitoring Capabilities

Studienarbeit
von

Philipp Kirchhofer

an der Fakultät für Informatik

Tag der Anmeldung: 01.08.2011
Tag der Fertigstellung: 30.12.2011

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

Betreuer:
Dipl.-Inform. 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 30.12.2011

Philipp Kirchhofer

Abstract

Bis vor einigen Jahren wurden Geschwindigkeitssteigerungen bei x86 Prozessoren hauptsächlich durch die Erhöhung der Taktfrequenz und, in einem geringeren Maße, durch Optimierungen der Mikroarchitektur erreicht. Der steigende Stromverbrauch und immer kleiner werdende Fortschritte beim Übergang auf neue Prozessorarchitekturen führten dazu, dass dieser Prozess nicht mehr fortgesetzt werden kann. Aktuelle Desktop- und Server-Prozessoren verwenden deshalb typischerweise Mehrkernarchitekturen. Erwartet wird durch dieses neue Vorgehen ein 30-facher Geschwindigkeitszuwachs in den nächsten 10 Jahren.

Die Verteilung der Arbeitslast einer Anwendung auf parallel rechnende Bereiche ist essentiell für die volle Ausnutzung von Mehrkernarchitekturen. Traditionelle Verfahren zur Programmierung von mehrfädigen Anwendungen mit Hilfe von Locks sind schwierig zu erlernen und deshalb eine große Quelle für Programmierfehler bezüglich Performance und Fehlerfreiheit. Ein neues Programmier-Konzept sollte deshalb vom Programmierer einfach zu nutzen sein, gut skalieren und eine hohe Rechengeschwindigkeit liefern können. Transactional Memory (TM) ist ein solches Konzept, mit dem die gewünschten Eigenschaften erfüllt werden können.

TM ist ein neuartiges Konzept, mit dem die Programmierung von mehrfädigen Anwendungen auf Mehrprozessorsystemen vereinfacht wird. Neuere Forschung hat gezeigt, dass der Einsatz von TM im Vergleich zu bisherigen Programmiermodellen für Mehrprozessorsysteme einfacher umzusetzen ist und deshalb zu einer verringerten Programmierfehlerrate bei den so erstellten Anwendungen führt.

Für eine breite Anwendbarkeit von TM ist es wichtig, dass eine hohe Rechengeschwindigkeit und gute Skalierbarkeit erreicht wird. Auf aktuellen Hardware basierten TM Systemen (HTM) kann die Unkenntnis der Interaktion zwischen Anwendung und TM Laufzeitumgebung zu Anwendungen mit sub-optimaler Geschwindigkeit und Skalierbarkeit führen. Die vorliegende Studienarbeit setzt an diesem Punkt an und erweitert ein bestehendes HTM System (TMbox) um eine Monitoringinfrastruktur, die die für die Analyse von HTM Anwendungen nötigen Daten erhebt und für eine spätere Verarbeitung speichert. Kriterien für den Entwurf dieser Monitoringinfrastruktur werden erläutert und Messergebnisse diskutiert.

Über diese Aufgabenstellung hinausgehend wurde die Monitoringinfrastruktur um weitere Komponenten erweitert, mit denen die durch die Monitoringinfrastruktur erhobenen Daten weiterverarbeitet und für eine Visualisierung und Analyse des Anwendungsverhaltens genutzt werden. Damit wird in dieser Arbeit gezeigt, wie man durch den Einsatz dieses neu entworfenen Überwachungs-, Visualisierungs- und Analysesystems das Laufzeitverhalten einer gegebenen HTM Anwendung visualisieren, analysieren und einer Optimierung zugänglich machen kann.

Contents

List of Tables

List of Figures

1 Introduction

1.1 Motivation

Transactional Memory (TM) is supposed to simplify parallel programming. Recent research shows that current state-of-the-art TM implementations are on a good way to achieve this goal. Previous work [1] shows that programming with TM semantics exhibits a much smaller error rate compared to programming with traditional fine-grained explicit locking.

But another issue remains: Performance and scalability are both important for a successful adoption of TM. The 90/10 law, originating from software engineering, states that about 90% of application runtime is spent in 10% of code. It is, according to this law, important to identify the parts of code where the bottlenecks are, to allow the creation of scalable and fast applications.

A study at Universität Karlsruhe [2] confirmed that TM applications have the potential to run faster than lock-based applications. But the study also showed that performance tuning of TM applications is currently a difficult task. The programmer, who uses current Hardware Transactional Memory (HTM) systems, is often unaware of an application's behavior. This makes optimizing the application a trial-and-error process. As a consequence the TM applications do not run as efficiently as possible. A solution is to generate event logs with software. This method captures and preserves the dependencies between transactions that occur during run time. This approach typically comes with runtime overhead and may influence the application runtime characteristics.

To get a suitable overview of HTM behavior it is vital to have a monitoring infrastructure with no impact on application runtime characteristics and application behavior. The optimization hints gathered could otherwise be influenced in some way and cause a misguided optimization attempt. For HTM systems a separate hardware monitor is therefore the method of choice to non-intrusively gather and preserve run time information.

The purpose of this study thesis is to address these shortcomings and develop a monitoring and visualization infrastructure for an existing HTM system. This allows the programmer to get insights into the interaction between application and HTM system, to detect bottlenecks in the program flow and to optimize the application for the underlying HTM system. The key design aspects of the monitoring infrastructure include multi-core-scalability, high extensibility, zero runtime overhead and therefore no influence on application runtime characteristics. The system should also be easily usable by an application developer.

1.2 Outline

This study thesis is structured as follows: Chapter 2 contains a short introduction to Transactional Memory (TM) and shows related work on TM, profiling and monitoring. The novel ideas of this study thesis are also explained. The following chapter 3 contains both an overview and an in-depth walkthrough of all parts of the designed monitoring infrastructure. The chapter also explains the design decisions made. Chapter 4 does focus on the implementation of the infrastructure. Chapter 5 shows the results originating from this study thesis. The thesis ends with Chapter 6 by summarizing possible future extensions of the project and potential applications to other fields of computer science. The appendix contains a glossary and the bibliography of referenced papers and web sites.

2 Transactional Memory and Related Work

2.1 An introduction to TM

Until some years ago, the performance increase of mainstream x86 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. Current desktop processors have, for this reason, adopted a multi-core type architecture. Ongoing industry expectations currently reach a 30 times performance increase in the next 10 years using this approach (for more information see *"The Future of Microprocessors"*[3]).

Parallel programming is essential to obtain the full performance of multi-core architectures. But traditional parallel programming using locks is hard and error-prone, as shown in *"Is transactional programming actually easier?"* by Rossbach et al. [1] and *"Does transactional memory keep its promises?"* by Pankratius et al. [2]. A programming paradigm should be easy to use for programmers, have a good scalability and deliver high performance. Transactional Memory [4] is a new paradigm trying to fulfill these promises.

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). Each atomic block "sees" a consistent set of shared data (consistency) and is not allowed to modify the data of another concurrently running atomic block (isolation).

The execution speed of TM applications can be increased by adding support for TM semantics and instructions directly to the processor ISA. The resulting system is called a Hardware Transactional Memory (HTM) system. This type of TM system is usually bound by some sort of capacity contraints, e.g. the hardware can handle a specific fixed read-/write-set size. A transaction's read set contains all data locations read during the transaction, where as a transactions's write set contains all data locations written during the transacation. Transactions with a larger read-/write-set size than supported by hardware cannot run in HTM mode and must be handled by other means (for instance execution in software mode).

2.2 Related work

The speed of TM applications can be increased through STM or HTM hardware support. There are generally two feasible approaches: A light-weight approach adds special instructions to the processor ISA for a more efficient execution of STM systems. These instructions mostly deal with the locking of certain already existing memory areas in the processor core. A more intrusive approach adds new execution units and memory dedicated to TM support to the processor core and therefore uses more hardware ressources. The main advantage of this approach is to allow the fast execution of some TM transactions directly in hardware (HTM mode).

Several proposals have been published for TM support in next-generation processor architectures. AMD proposes the Advanced Synchronization Facility (ASF) [5], an AMD64 hardware extension for lock-free data structures and TM. Cache lines can be locked using specific instructions to facilitate the running of a fast ASF-STM system. An evaluation [6] observed that ASF-based TMs show very good scalability and much better performance than pure STM for the applications in the STAMP [7] benchmark suite. Intel's Hardware assisted Software Transactional Memory (HASTM) [8] also takes the same approach by proposing changes in the processor ISA to speed up the execution of STM runtime systems. This light-weight approach allows for a relatively non intrusive implementation in current processor cores, but also limits the accelerating opportunities. The TM implementation in Sun's Rock processor [9] takes on a hybrid approach by implementing the parts, which allow to accelerate the common case behavior of TM applications, in hardware while supporting advanced TM features in software. The design of this TM implementation allows to take advantage of future processor architecture generations, where a successively higher level of HTM support can be achieved. The TMbox system [10], used as the underlying platform in this study thesis, follows a different, more heavy-weight approach. Entire transactions can be executed directly in hardware on a best-effort base. This means that certain restrictions of transactional characteristics (like size of read-/write-set, no I/O operations) have to be satisfied to guarantee a successful execution. The advantages are fast execution and, on the software side, decreased complexity because a STM runtime is not needed.

None of these proposed changes are currently used in commercially available processors. The ongoing research on TM by nearly all major microprocessor companys does however indicate a certain interest and a possibility of an implementation in future, currently unannounced, CPU architectures.

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 get right with more than 10,000 scheduled hardware threads. Fung et al. [11] address these issues by proposing and simulating a GPU with HTM support. They show that HTM on GPUs performs well for applications with low contention. Their proposed TM design "KILO TM" 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. [12] show a related feasibility study: They use the unmodified hardware of a Nvidia GPU to run two variants of a STM runtime environment. One variant is a simple, easy to implement STM with lower resource requirements, specifically designed for use in GPUs while the other STM variant uses a more complex design oriented at general purpose multiprocessors. The results show increased performance and reduced abort rates when using the complex design. The cooperation of CPU and GPU oriented TM 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.

All of these previously mentioned proposals show different environments for running HTM and STM applications. To get a high computing performance it is furthermore essential to characterize TM application behavior and adjust the internal parameters and algorithms of a TM runtime environment accordingly. Multiple papers have been published about the characterization of STM applications. Ansari et al. [13] ported some applications from the STAMP benchmark suite to DSTM2, a Java-based STM implementation with profiling features. They used some well-known metrics like speed up, wasted work and time in transaction to characterize the behavior of these applications. Some of the presented metrics are also implemented in this study thesis, as shown in chapter 5, but the implementation of a software profiling framework in Java is quite different than the hardware-based implementation done in this study thesis. Chung et al. [14] present a comprehensive characterization study of the common case behavior of 35 multi-threaded applications. 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 TM. The STM monitoring techniques and the metrics presented in these papers are, in general, transferable to other TM variants, but the specific implementation of a monitoring infrastructure is different on HTM systems. One specific different aspect is, for instance, the difference in processing speed of a TM application running on a system with enabled or disabled monitoring. The processing speed of TM applications running on a STM runtime environment with enabled monitoring support is always slowed down due to the increased amount of computations done by the TM system (e.g. generation and saving of traces). Monitoring support on an HTM system can, on the other hand, be implemented with zero overhead, as shown in this study thesis.

The PhD thesis of Ferad Zyulkyarov [15] 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 TM 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 TM 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). As a contrast, the work done in this study thesis is based on the TMbox system, which uses a C based HTM system (BeeTM) with a thin layer directly above the hardware. The TM runtime on this system does not include a garbage collector and is therefore not susceptible to the previously mentioned probe effect.

The TMbox system is, in general, comparable to previously published research. The programing model of the TMbox system [10] is comparable to the Transactional Coherence and Consistency model [16] and the monitoring techniques used in this study thesis are in some parts comparable to the Transactional Application Profiling Environment [17]. Both were developed at Stanford university. Major differences include the use of multiple ring buses in the TMbox system, while other systems use a switched bus network with different timing characteristics and influence on HTM behavior. The TAPE system was simulated using an execution-driven simulator, where as the TMbox system can both be simulated by software and run in hardware (with a much higher speed) using a FPGA chip.

A related work in nearby fields is the description of an event-based distributed monitoring system for soft- and hardware malfunction detection in manycore architectures [18]. Up to 4 MIPS32 processors and associated units form a cluster. The proposed design is composed of several of these clusters, where each cluster is connected via a global interconnect to the other cluster units. Several monitoring tasks run on each cluster unit: The correct operation of the units inside the cluster is monitored as well as the operation of the monitoring tasks on neighboured clusters. The monitoring tasks communicate by sending event messages to each other. Any deviation from normal operation mode causes a broadcast of a special alert event to all nearby clusters. The whole system then saves application progress, restarts in a safe way and continues the interrupted computation with the remaining functional cluster units.

2.3 Novel ideas

The following novel ideas distinguish the work done here from previous research:

The framework developed in this study thesis allows an in-depth visualization and analysis of HTM applications. The extensibility also allows to add STM tracing support for HybridTM behavior analysis. A comprehensive profiling environment for HybridTM systems has not been published, up to now.

The visualization and analysis capabilities of the existing scalable tool Paraver have been leveraged for the purposes of HTM and HybridTM behavior analysis. This novel feature allows an easy interactive in-depth visualization and analysis of HTM behavior.

Using FPGA technology for the implementation of the monitoring infrastructure allows for fast execution, monitoring and analysis of long and complex TM applications. A simulation-only approach is usually several magnitudes slower. The use of FPGAs has another positive effect as it allows the rapid adaption of the monitoring infrastructure to new requirements and hardware design changes because of short development turn-around cycles.

3 Design

An event-based logging system was designed, tested and implemented during this study thesis for the TMbox HTM subsystem. This chapter starts with an introduction of the TMbox system, whose HTM unit was used as the underlying TM implementation. The modification of TMbox, design decisions and the designed units are also explained in later sections.

3.1 The TMbox system

The TMbox system [10], designed at the Barcelona Supercomputing Center (BSC), was used as the base implementation of an HTM system for this project. It uses two ring-buses to connect the soft-core processors. The ring-bus is especially suited to connect one or more monitoring elements. Moreover, the TMbox makes use of FPGAs and thus offers the space and the flexibility to add and synthesize new hardware components. TMbox runs on the BEE3 [19] multi-FPGA research platform equipped with Xilinx Virtex 5 Series FPGAs. The TMbox system allows up to 16 MIPS-compatible computation cores to be fitted in one FPGA chip. A picture of the BEE3 platform can be seen in Figure 3.1.

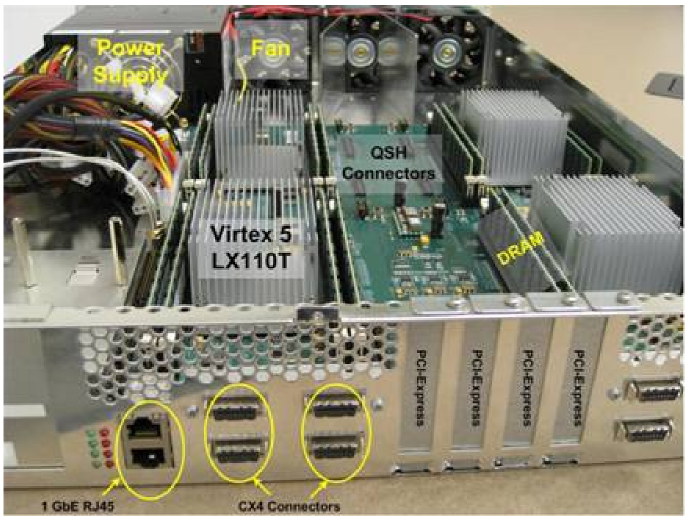


Figure 3.1: BEE3 platform

TMbox schematics

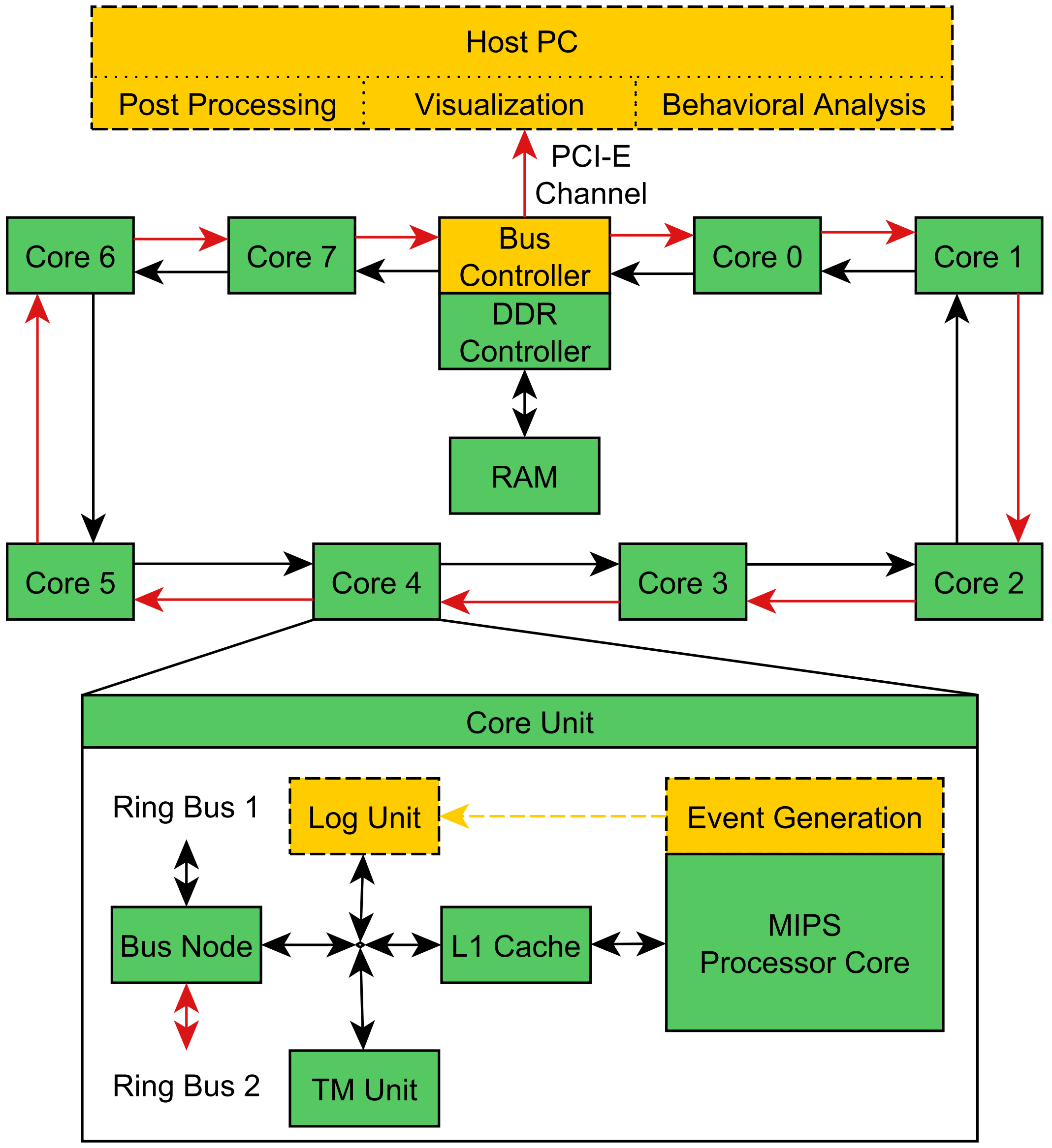


Figure 3.2: 8 Core TMbox system block diagram

Several components (log unit, event generation, bus controller) of the TMbox system were modified or added during the implementation phase of this study thesis. Figure 3.2 shows the modified hard- and software components of the TMbox system in yellow color and the added components additionally with dashed lines. The black ring bus transfers memory read/write requests and responses while the red ring bus transfers invalidation and event messages. Additional information about the TMbox system can be found in *"TMbox: A Flexible and Reconfigurable 16-Core Hybrid Transactional Memory System"* [10].

The following paragraphs describe the units which were re-used from the TMbox system. The new units are described in section 3.4.

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 TM unit is necessary for HTM application support. It contains the read- and write-set of the currently running transaction. Some TM 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 busses. 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 Profiling workflow

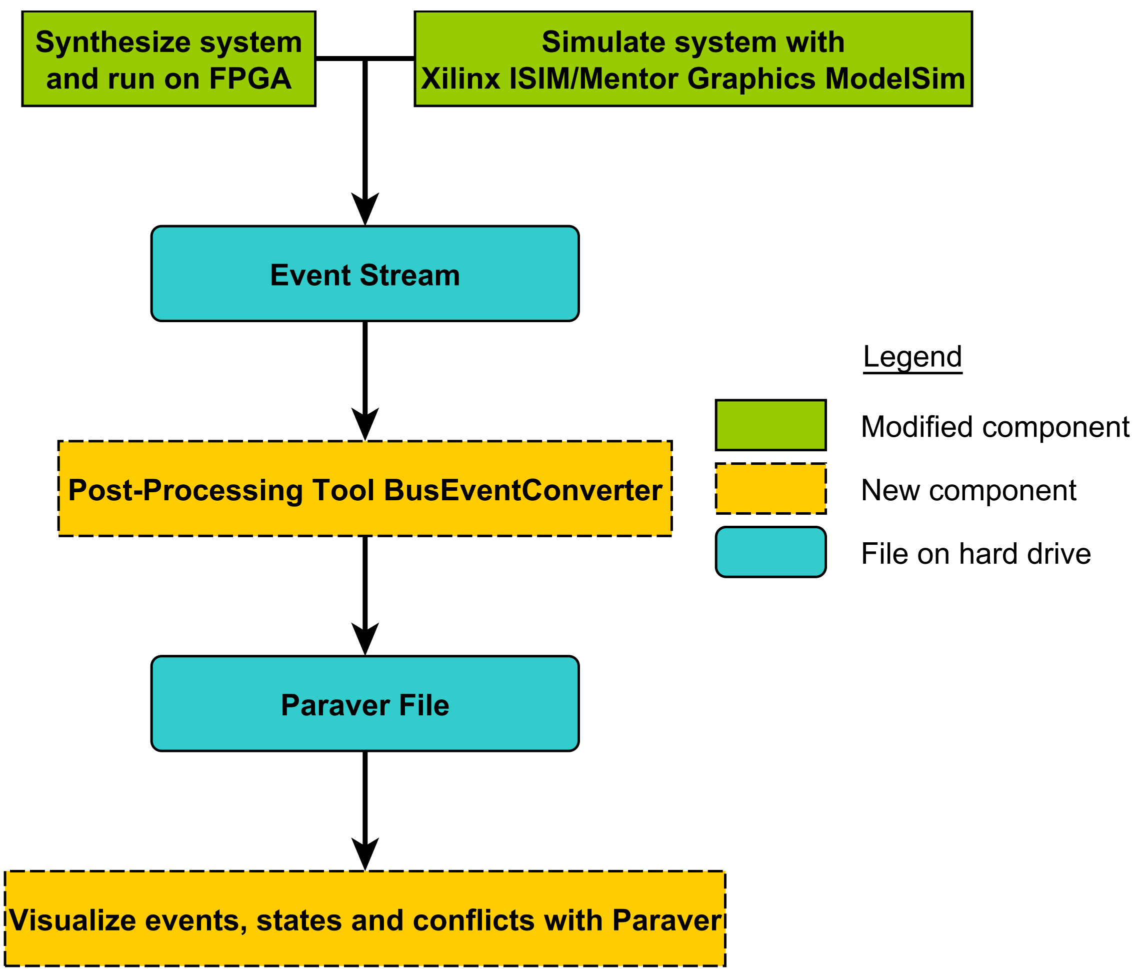

Synthesize system and run on FPGA
Simulate system with Xilinx ISIM/Mentor Graphics ModelSim
Event Stream
Post-Processing Tool BusEventConverter
Paraver File
Visualize events, states and conflicts with Paraver
Legend
Modified component
New component
File on hard drive

Figure 3.3: Profiling workflow

A simplified representation of the profiling workflow is given in Figure 3.3. First, an event stream is generated by either simulating the system with Xilinx ISIM / Mentor Graphics ModelSim or by synthesizing and running it on the BEE3 platform. The event stream is saved in a file and then fed into the post-processing tool BusEventConverter for further processing. This tool was created during this study thesis. It creates a file, which is finally used to visualize and analyze events, states and conflicts with the Paraver tool. The components of the profiling workflow are described in the next sections, for further information about BusEventConverter see section 3.5.1, about Paraver see section 3.5.2.

3.3 A rationale for event-based monitoring

HTM and application behavior can be split into a stream of small events containing information about state changes. The events are later recomposed during post-processing using the BusEventConverter tool. 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 ring bus only during bus idle phases and therefore does not influence the application behavior and its runtime characteristics. An alternative would be to collect and send the complete HTM state each time it changes, causing a large amount of data to be transferred, consuming more bandwidth than the chosen approach.

Event format

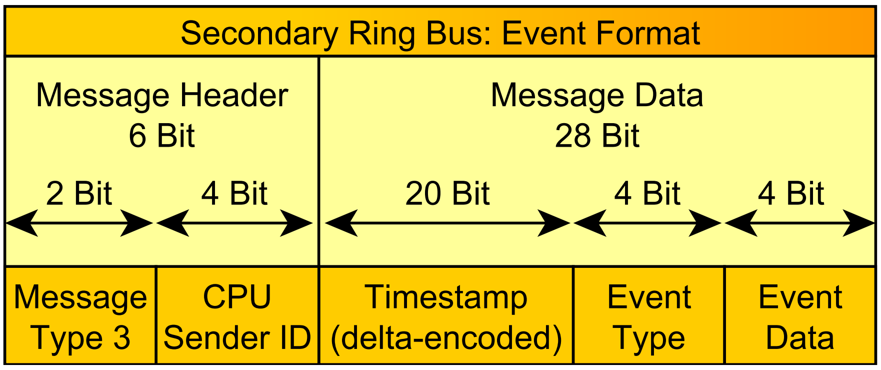


Figure 3.4: Event diagram

Figure 3.4 shows the format of an event. The division into two parts, message header and message data, is given by the fixed format of the secondary (invalidation) ring bus. The special message type 3 distinguishes event messages from already defined invalidation messages of the TMbox system.

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 an accuracy of 1 cycle and a temporal space of about 1 Mio. cycles between two events occurring on one processor. The TMbox system has a FPGA clock frequency of 50 MHz. To prevent a timestamp overflow an event must therefore be sent every 20 milliseconds. A timestamp overflow would otherwise cause the “real” time (during application runtime) and the reconstructed time after post-processing to diverge. A timestamp overflow is however an unusual case: Events are created during normal system operation with reasonable HTM activity with a much higher frequency than required by this technical limitation. A solution to rule out timestamp overflows would be to add the generation of a special no-operation event, whenever the 20 milliseconds without event generation time limit would be reached.

The 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 reaching a hardware constraint (capacity abort) or in most cases by getting a matching invalidation message from another processor.

Event types

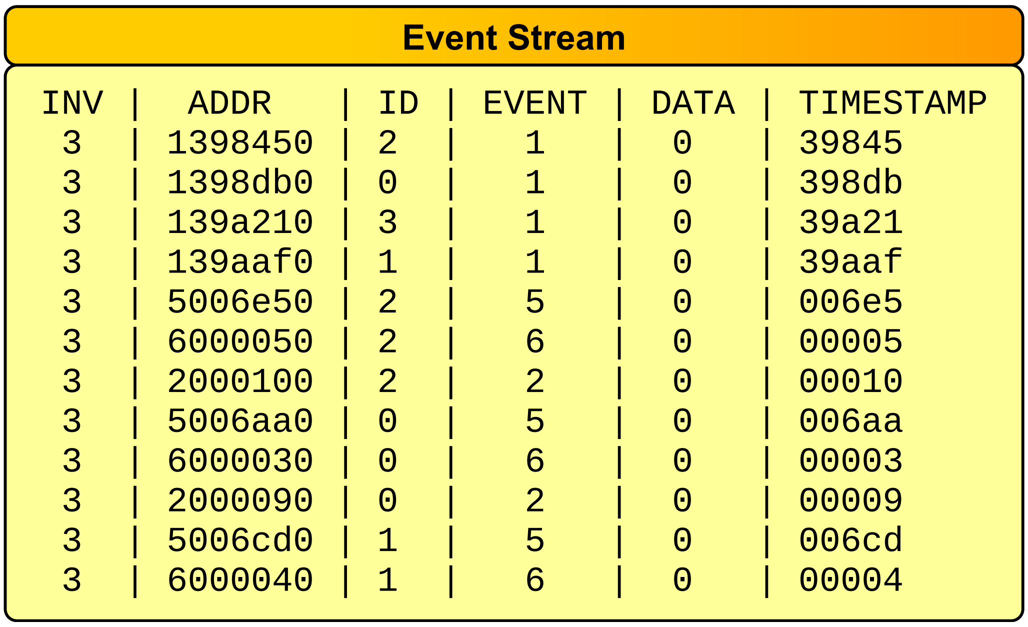
The event types defined for transactions are:

- Start
- Commit
- Abort
- Invalidation
- Try locking ring bus for commit
- Succeeded locking ring bus

The generation and capturing of these events allows to rebuild the HTM state 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). Currently up to 16 different event types can be defined, allowing an easy addition of new event types in future projects.

Generated event stream

The generated event stream can be easily transferred and saved. Figure 3.5 shows a short example of an 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 3.5: Monitoring infrastructure 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).

3.4 Monitoring units

The following paragraphs describe the design of the hardware units created or modified during this study thesis. The units are written in VHDL and Verilog.

3.4.1 Event Generation Unit

The event generation unit monitors the HTM unit state, generating events whenever the state changes. The generated events cover all state changes possible during runtime. They are augmented with additional data that is useful later on for behavior analysis. This additional data includes for instance the CPU ID, which caused an TM abort.

The processor core contains a finite state machine (FSM) handling internal processor state transitions. The event generation unit is embedded into this FSM. A pseudocode sample of the modified FSM is later shown in Figure 4.2.

3.4.2 Log Unit

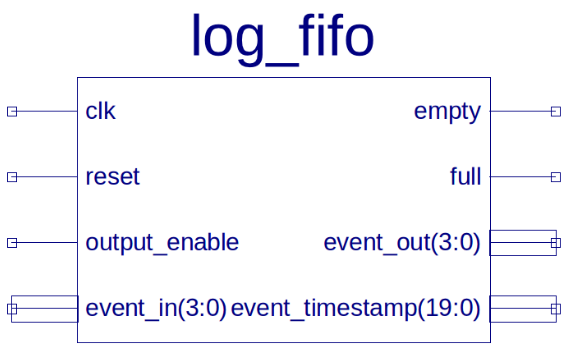


Figure 3.6: The log_fifo event capturing and saving unit

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 secondary ring bus (invalidation bus) to the bus controller. The transfer is only done whenever the ring bus is idle, to prevent a disturbance of application timing behavior. Therefore a buffer is used to buffer events. The buffer size can be set during synthesis using VHDL generics.

Experiments did show that a buffer size of 32 entries 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, has been used for this purpose. The chart 3.7 shows the maximum number of used buffer entries over time for the assembler program running on an 8 core system. The highest peak is at four used entries and the average use is between one and two entries. This shows that a buffer size of 32 entries gives enough headroom to prevent a loss of monitoring events even during application phases with a high frequency of generated events.

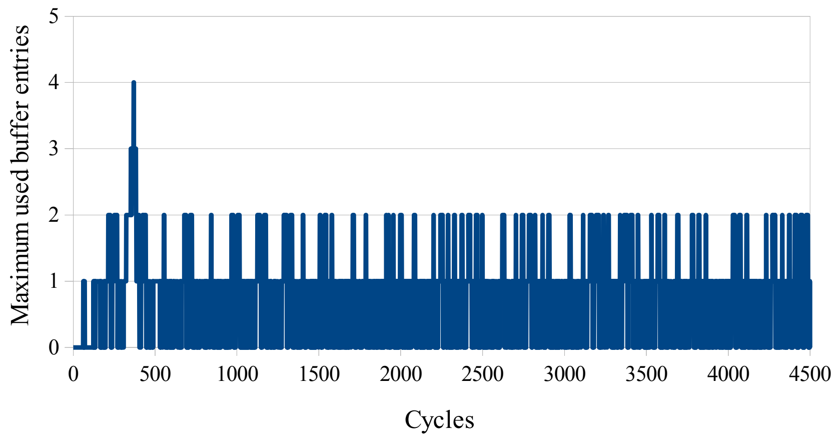


Figure 3.7: Peak used buffer entries

Location of Log Unit

The location of the log unit influences the scope of the available logging data and the complexity of the necessary design changes. Three possible locations for the unit have been identified:

1. Tightly integrated into processor core

   The internal processor state can be easily monitored by embedding the log unit 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 busses and signals to the log unit.

2. Located in the DDR RAM controller

   This approach allows to log memory access patterns. Access to other TM related data is, on the other hand, difficult without breaking up the modularization of the TMbox system. It is also challenging to distinguish between operations done on different processors.

3. Meet in the middle: Between bus node and cache unit

   This method allows a great extensibility of the log unit with a broad amount of available loggable data. It is also relatively easy to non-intrusively connect the log unit to the rest of the system.

The third location has been chosen for implementation. It delivers a broad amount of logging data, while keeping the complexity of necessary changes to the TMbox system at a reasonable level.

3.4.3 Bus Controller Unit

The bus controller unit gathers all events and transfers them over a PCI Express channel to a host PC. The PCI Express channel has a fixed bandwidth, set during synthesis. In the case of events flowing in at a higher rate than they can be transferred to the PC a FIFO buffer is used to prevent a loss of events.

3.5 Post-Processing

After the supervised application has finished running, the post processing tool BusEventConverter reads and checks the event stream, rebuilds HTM and application states, generates statistics and outputs data suitable for later processing with an visualization and analysis tool, explained in the next section.

3.5.1 The BusEventConverter tool

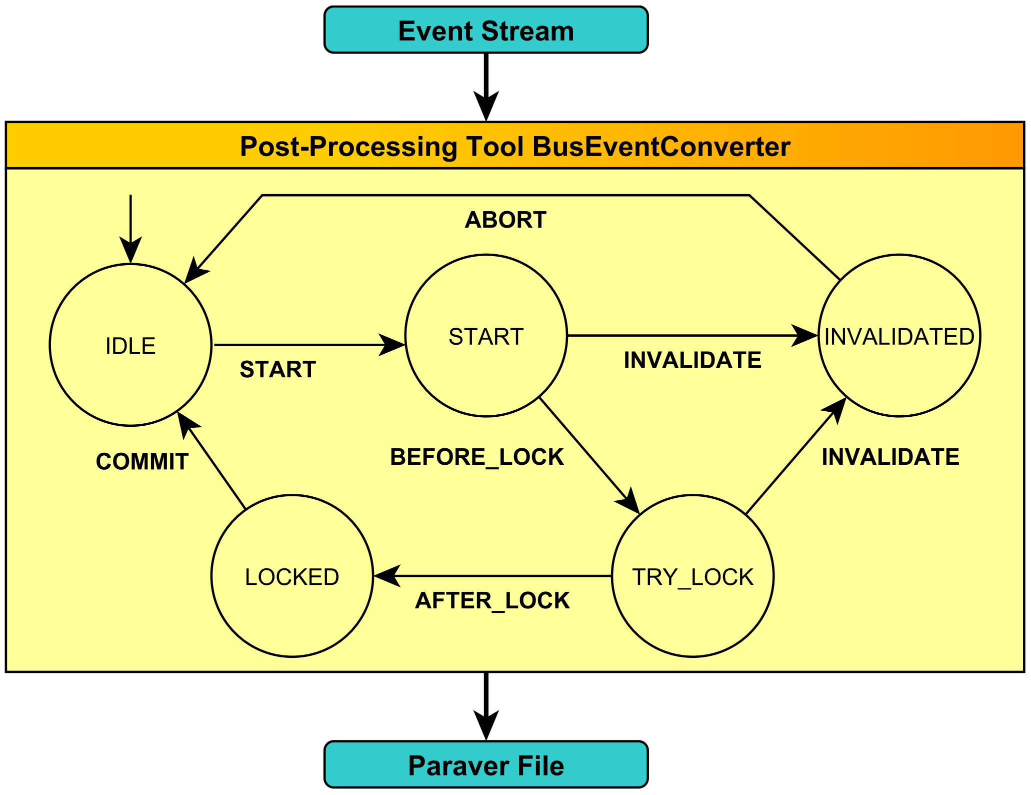


Figure 3.8: Monitoring infrastructure post processing

The event stream, which is generated by the monitoring units of the monitoring infrastructure, 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. This design principle is also called a workflow pipeline. The BusEventConverter workflow pipeline is shown in Figure 3.9. The passes developed during this study thesis, which are shown in the workflow pipeline picture, are explained in the following sections.

The workflow pipeline approach allows a flexible development of the BusEventConverter program functionality: New passes can be easily fitted between existing passes. Already existing passes can be replaced with passes with extended and/or different functionality. Future projects can, for example, work on adding passes for automatic phase detection.

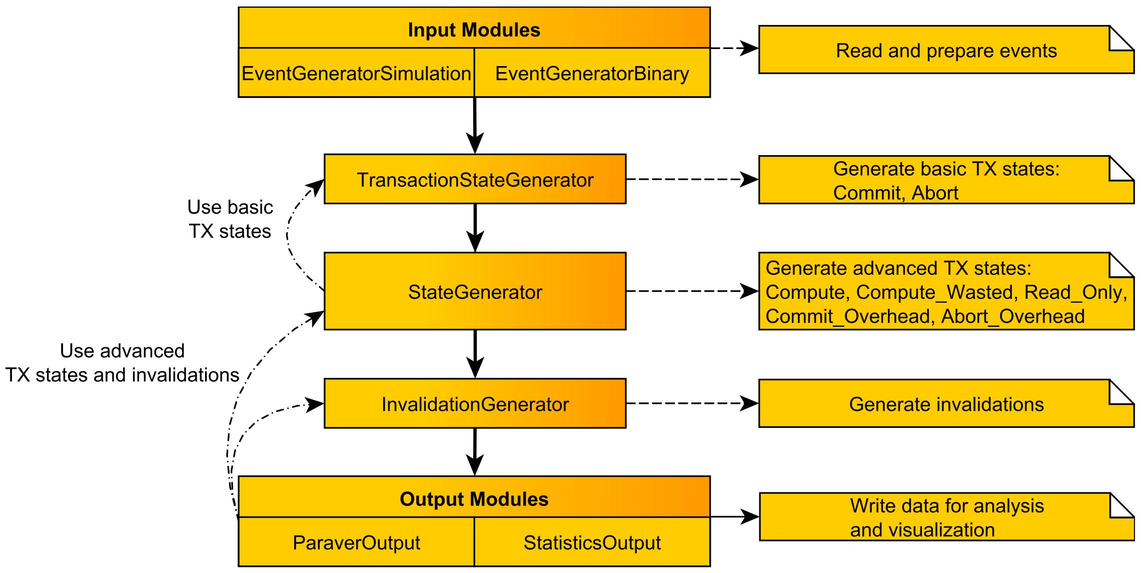

Input Modules
EventGeneratorSimulation
EventGeneratorBinary
Read and prepare events
TransactionStateGenerator
Generate basic TX states: Commit, Abort
Use basic TX states
StateGenerator
Generate advanced TX states: Compute, Compute_Wasted, Read_Only, Commit_Overhead, Abort_Overhead
Use advanced TX states and invalidations
InvalidationGenerator
Generate invalidations
Output Modules
ParaverOutput
StatisticsOutput
Write data for analysis and visualization

Figure 3.9: BusEventConverter workflow pipeline

The BusEventConverter tool is written in Java 1.6 and runs on Windows, Linux and MacOS. Figure 3.10 shows the classes of the BusEventConverter tool. Classes associated with error and file handling have been omitted for the sake of brevity.

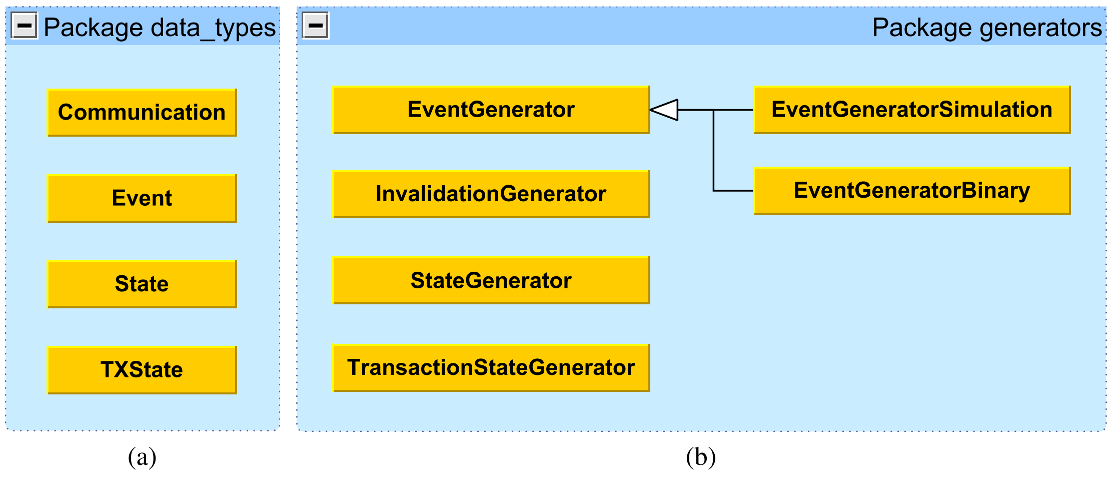


Figure 3.10: Classes of post processing tool BusEventConverter

Input modules

The input module of BusEventConverter can be exchanged to accomodate for multiple input formats. Currently two input modules are available: The first module, which reads data output by simulation and the second module, which reads data output by running the TMbox system on the BEE3 platform. The resulting data of a simulation with Xilinx ISIM / Mentor Graphics ModelSim is saved as textual data, whereas the FPGA module generates binary data.

Generators

Each generator uses a specific finite state machine (FSM) to process the data created by the input module or a previous generator.

The TransactionStateGenerator reads the events from an input module and generates basic transaction states. Each transaction is classified as either committed or aborted.

The StateGenerator augments these basic transaction states with several sub-states. For instance, a committed transaction with an non-empty write-set is divided into three sub-states:

1. **Computation Phase:** The program does transactional reads and writes and processes data to get results
2. **Try Lock Phase:** The transaction is validated and prepared for commit by trying to get a commit lock
3. **Commit Lock Phase:** The commit lock is acquired and transactional data is moved to memory

The successful completion of the last phase concludes a committed transaction.

The InvalidationGenerator generates invalidations between processor cores. An invalidation is the cause for an abort of a transaction. Data includes the processor core ID of the processor that sent the invalidation and the processor core ID of the processor, on which the aborted transaction was running.

Output module

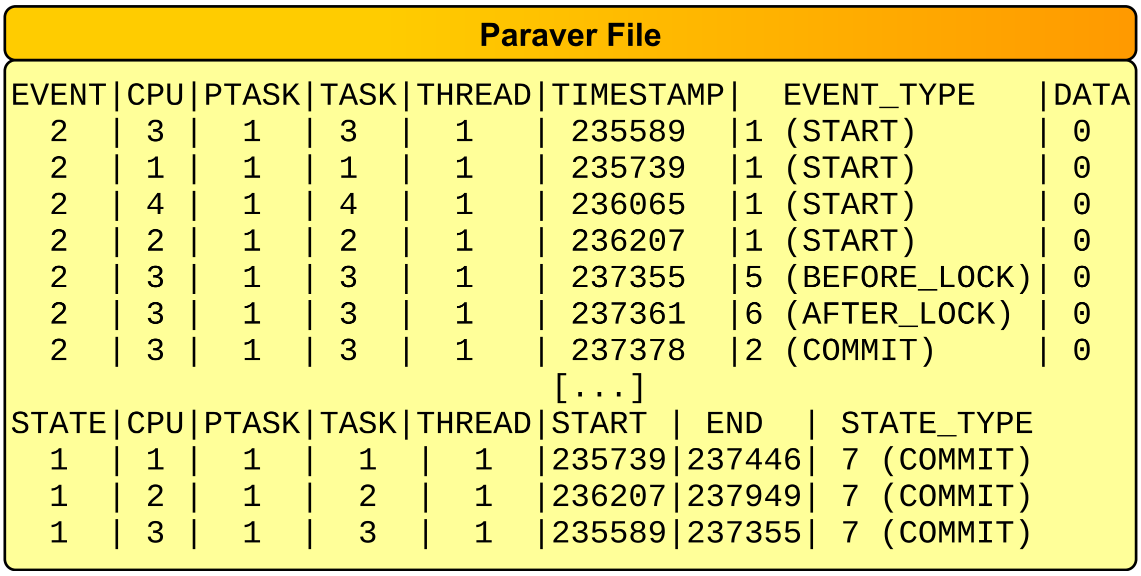
Two output modules currently exist: The first module checks consistency and outputs statistics like event count, number of event types, number of states, etc. This module is used mainly for debugging purposes. The second module outputs data suitable for visualization and analysis with the Paraver tool, which is explained in the next section. A sample output of a BusEventConverter run is given in Listing 3.1.

```
Using Simulation Reader
Generating Events: ......................... Done
Generating TX States: 1 2 3 4
Writing output:
- Header
- Events
- States
- Invalidations
Processed 2550 events, 736 TXStates, 1812 states
Filtered 58 events
```

Listing 3.1: Sample output of a BusEventConverter run

Paraver file

The generated file is now ready for visualization and analysis. A schematic example of such a Paraver file is given in Figure 3.11.



Paraver File

```
EVENT|CPU|PTASK|TASK|THREAD|TIMESTAMP|   EVENT_TYPE   |DATA
  2  | 3 |  1  | 3  |  1   | 235589  |1 (START)       | 0
  2  | 1 |  1  | 1  |  1   | 235739  |1 (START)       | 0
  2  | 4 |  1  | 4  |  1   | 236065  |1 (START)       | 0
  2  | 2 |  1  | 2  |  1   | 236207  |1 (START)       | 0
  2  | 3 |  1  | 3  |  1   | 237355  |5 (BEFORE_LOCK)| 0
  2  | 3 |  1  | 3  |  1   | 237361  |6 (AFTER_LOCK) | 0
  2  | 3 |  1  | 3  |  1   | 237378  |2 (COMMIT)      | 0
                          [...]
STATE|CPU|PTASK|TASK|THREAD|START | END   | STATE_TYPE
  1  | 1 |  1  |  1  |  1  |235739|237446| 7 (COMMIT)
  1  | 2 |  1  |  2  |  1  |236207|237949| 7 (COMMIT)
  1  | 3 |  1  |  3  |  1  |235589|237355| 7 (COMMIT)
```

Figure 3.11: Paraver file

The file contains several Paraver records: The first segment of the file contains the Paraver event definitions, 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. The different types are explained with more details in the next section.

Paraver file format

The Paraver file format has been figured out by looking through the Paraver source code available at the BSC website [20]. A specification of the Paraver file format is shown for further reference in Listing 3.2. This specification should only be used with a grain of salt, as an official specification could not be found.

Paraver trace files contain a header and a set of records [20]. There are three different record types:

- **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.

```
-- Header --

#Paraver (<DD/MM/YYYY at HH:MM>):<lasttime>_<units>:<
   hard_architecture>:1:<soft_architecture>(x,x,x),<
   communicators>

-- State --

1:<cpu>:<ptask>:<task>:<thread>:<begin>:<end>:<state>

-- Event --

2:<cpu>:<ptask>:<task>:<thread>:<timestamp>:<event_type
   >:<event_value>

-- Communication --

3:<send_cpu>:<send_ptask>:<send_task>:<send_thread>:<
   logical_send>:<physical_send>:<recv_cpu>:<recv_ptask
   >:<recv_task>:<recv_thread>:<logical_recv>:<
   physical_recv>:<size>:<tag>
```

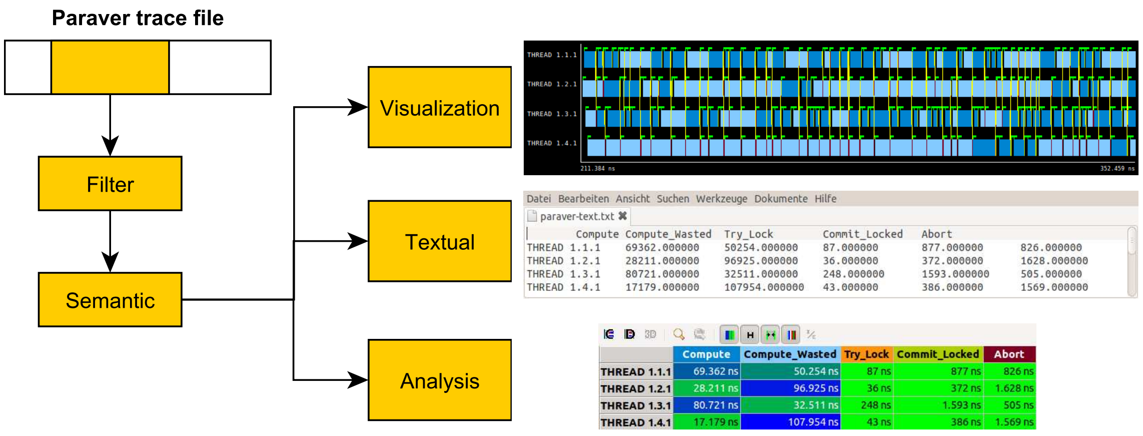
Listing 3.2: Paraver File Format

3.5.2 The Paraver visualization and analysis program

Paraver [20] 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" [1], one of the most powerful supercomputers in Europe, located at the BSC.

Paraver structure and features

The Paraver visualization and analysis workflow is shown in Figure 3.12.



```
             Compute  Compute_Wasted   Try_Lock        Commit_Locked   Abort
THREAD 1.1.1          69362.000000     50254.000000    87.000000       877.000000      826.000000
THREAD 1.2.1          28211.000000     96925.000000    36.000000       372.000000      1628.000000
THREAD 1.3.1          80721.000000     32511.000000    248.000000      1593.000000     505.000000
THREAD 1.4.1          17179.000000     107954.000000   43.000000       386.000000      1569.000000
```

	Compute	Compute_Wasted	Try_Lock	Commit_Locked	Abort
THREAD 1.1.1	69.362 ns	50.254 ns	87 ns	877 ns	826 ns
THREAD 1.2.1	28.211 ns	96.925 ns	36 ns	372 ns	1.628 ns
THREAD 1.3.1	80.721 ns	32.511 ns	248 ns	1.593 ns	505 ns
THREAD 1.4.1	17.179 ns	107.954 ns	43 ns	386 ns	1.569 ns

Figure 3.12: 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.

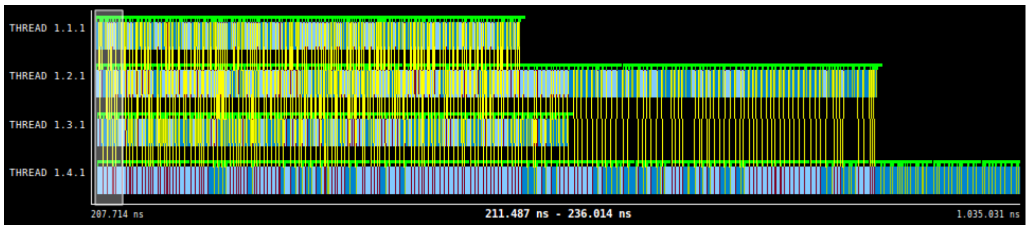
The semantic module afterwards assigns a numeric value to each state and event. This can later be used to compute a system-level overview (see also Figure 5.4c).

The visualization, textual and analysis modules comprise the main parts of Paraver. The use of this parts is shown in the next chapter.

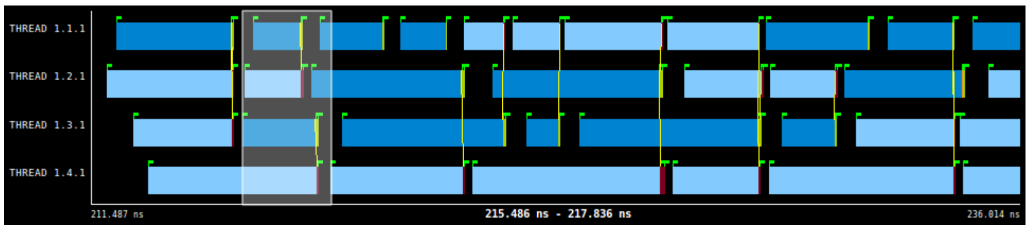
[1]http://www.bsc.es/plantillaA.php?cat_id=200

Handling of large traces

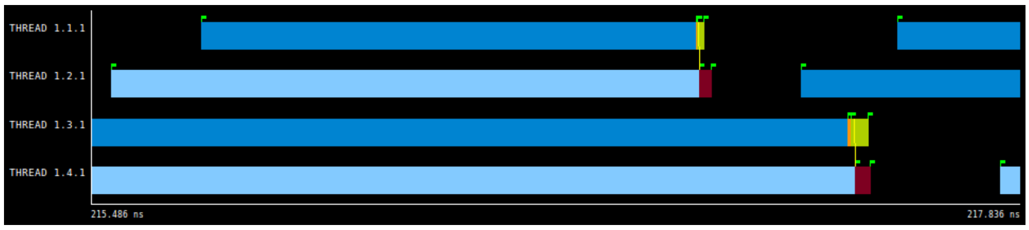
The monitoring of long running applications, which run on many-core systems, creates particularly large traces. Paraver is designed to handle these traces efficiently. The user can freely zoom in and out of traces, displaying only interesting parts of the visualization of a trace. A demonstration of this feature is given in Figure 3.13. The white box marks the zoomed area, e.g. the white box in (a) is fully shown in (b) and the white box in (b) is fully shown in (c). It is possible to show multiple visualization types simultaneously. These visualizations can be synchronized to show the same timespan (as seen in Figures 5.4 and 5.5).



(a)



(b)



(c)

Figure 3.13: Increasing zoom levels of an program trace

4 Implementation

All new designed units have been created from scratch and tested individually using unit tests and test workbenches. Whenever a unit worked right it was integrated into the TMbox system. After an unit was integrated the whole system was tested for proper operation using special test programs. This process was repeated until every changed unit had been tested and integrated. The approach lead to a steady progress during the implementation phase.

Adding the infrastructure to the TMbox system increased FPGA chip usage by a few percent. Please refer to table 4.1 and the corresponding Figure 4.1 for a detailed comparison of a TMbox system with and without the monitoring infrastructure.

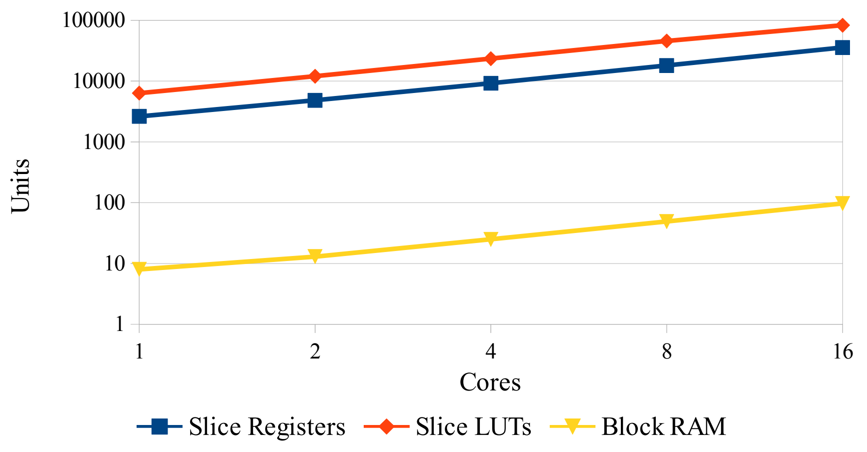
The complete system scales linearly when increasing the number of processor cores. When looking closer it can also be seen that increasing the number of processor cores does actually decrease the amount of used FPGA units per core.

FPGA	Cores				
unit type	1	2	4	8	16
Registers[1]	2,537	4,671	8,953	17,527	34,697
Registers[2]	2,627	4,822	9,227	18,046	35,705
Increase	3.55 %	3.23 %	3.06 %	2.96 %	2.91 %
LUTs[1]	6,215	11,973	23,027	45,887	82,871
LUTs[2]	6,340	12,027	23,411	45,683	82,923
Increase	2.01 %	0.45 %	1.67 %	-0.44 %	0.06 %
BRAMs[1]	7	12	24	45	89
BRAMs[2]	8	13	25	49	97
Increase	14.29 %	8.33 %	4.17 %	8.89 %	8.99 %

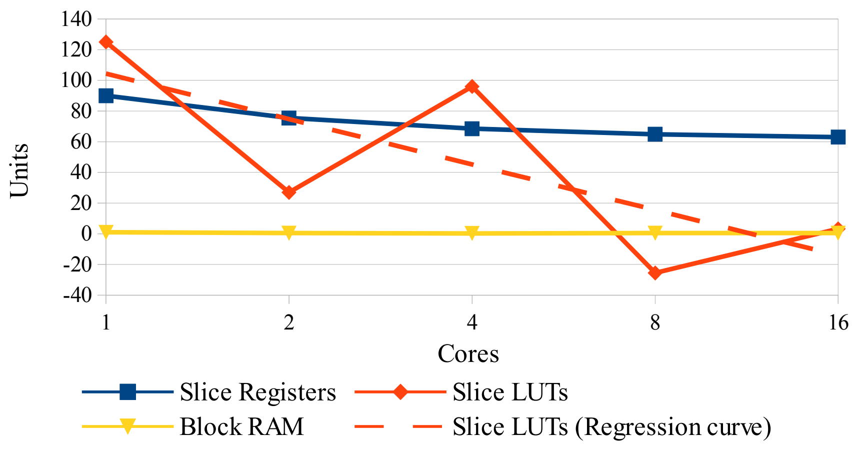
[1] without monitoring infrastructure
[2] with monitoring infrastructure

Table 4.1: FPGA usage of TMbox system

An unexpected result can be seen when comparing the LUTs used for an 8 core system: Adding the monitoring infrastructure to the system actually decreased the amount of used LUTs. Assuming that the FPGA synthesis software does not do a complete and comprehensive optimization, which would be computationally expensive, this can be explained that under this very specific circumstances the optimization pass does find better optimization opportunities during synthesis.



(a) Complete system (processor cores + monitoring infrastructure)



(b) Increase per Core

Figure 4.1: TMbox FPGA Usage (with monitoring infrastructure)

The TMbox system uses a FSM to manage the internal states of a processor. This FSM contains the current state of the processor cache and reacts to memory requests and answers coming from the ring bus. Figure 4.2 shows, in pseudocode, a simplified image of this FSM. The full TMbox FSM contains 11 states and 131 transitions. For simplicity only the four states relevant to TM operations are displayed in the Figure. The newly added event emitting code parts are marked with red color. The event emitting code was implemented as unintrusive as possible, the changes in the FSM were in fact accomplished by adding about 30 lines of VHDL code.

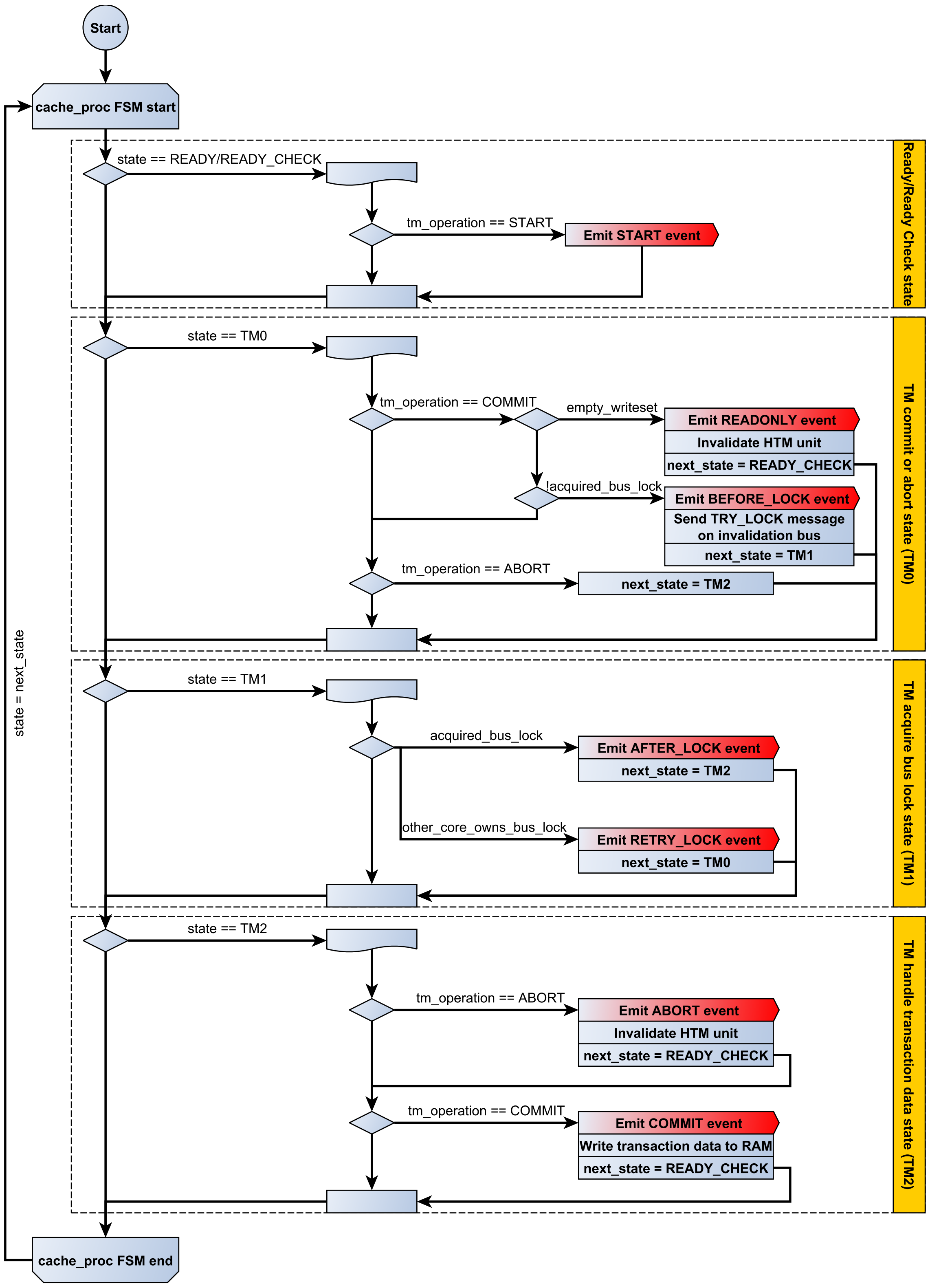


Figure 4.2: TMbox cache state FSM

FPGA space usage and impact on clock frequency

The next Figure depicts a visual comparison of the FPGA usage of the monitoring infrastructure compared to the overall space usage of the TMbox system:

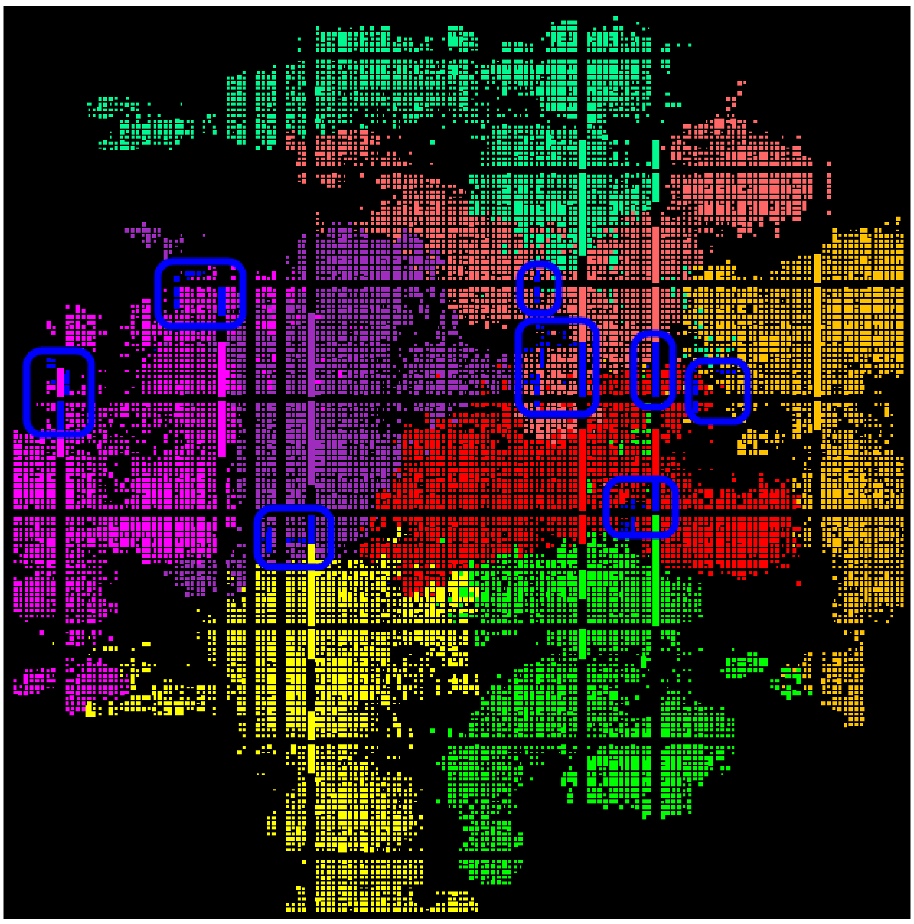


Figure 4.3: Monitoring infrastructure space usage

Figure 4.3 shows a synthesized, translated, mapped and routed visualization of the FPGA chip assignment for an 8 core TMbox system. The Figure was created using Xilinx PlanAhead. Each colored pixel in the Figure is a used hardware unit on the FPGA, for instance a LUT, BRAM, register, clock generator etc. All units with the same color belong to the same processor core. The dark blue parts with a blue border are the newly added units of the monitoring infrastructure.

The implementation of the monitoring infrastructure did not have an impact on the maximum clock frequency of the TMbox system (50 MHz). Therefore there was no need to manually identify and optimize critical data and clock paths in the FPGA.

5 Results

This chapter describes the visualization and analysis aspects of the developed monitoring infrastructure and shows two examples of HTM application behavior analysis.

The primary goal of this study thesis has been achieved: The implementation of a monitoring infrastructure for the TMbox platform with no runtime overhead and therefore no change in application runtime characteristics. This was confirmed experimentally. The main factor responsible for achieving this goal is the low priority transmission of monitoring data together with the buffering of to be transferred data.

Further goals have also been accomplished: An accurate overview of the behavior of a multi-threaded HTM application can be obtained, visualized and analyzed with this monitoring infrastructure. Several analysis metrics are provided, such as contention/abort rate, contention between specific threads, time spent in committed and aborted transactions and overhead caused by the HTM system. These metrics allow to compare different implementations of an application and to optimize its overall runtime and scalability on a given HTM system.

A given application can furthermore be analyzed concerning different hardware parameters (for instance number of processors, HTM unit parameter settings, type of bus interconnection, etc.). These parameters can be varied systematically to reveal the influence of hardware parameters on a given application's runtime behavior and it's performance. Based on these insights an optimization of the underlying HTM hardware can be made.

The runtime behavior of an HTM program can also be visualized. This enables the programmer to identify the specific characteristics of different parts of an application and to detect parts with sub-optimal behavior. This allows to optimize the poorly performing parts of the application.

The images in this section have been created using the post processing tool BusEventConverter and the visualization and analysis tool Paraver. They show the currently available visualization views. These views support the TM application programmer in his optimization efforts. Additional views can be created using Paraver, but this process is unfortunately complicated, as the visualization parts of Paraver are undocumented and had to be partially reverse engineered during this study thesis.

Some of the already known bottlenecks of the TMbox system were confirmed and quantified with these analysis capabilities. One of these bottlenecks is the increasing memory access latency when adding more processor cores to the system, an inherent attribute of a ring bus type core interconnect. Future work on the TMbox system may therefore evaluate different approaches to core interconnection, leading to an improved system.

5.1 Introduction to Paraver visualization

Paraver is used to show a timeline of the flow of different HTM states. Each color corresponds to a certain HTM state. The colors are explained in the next section.

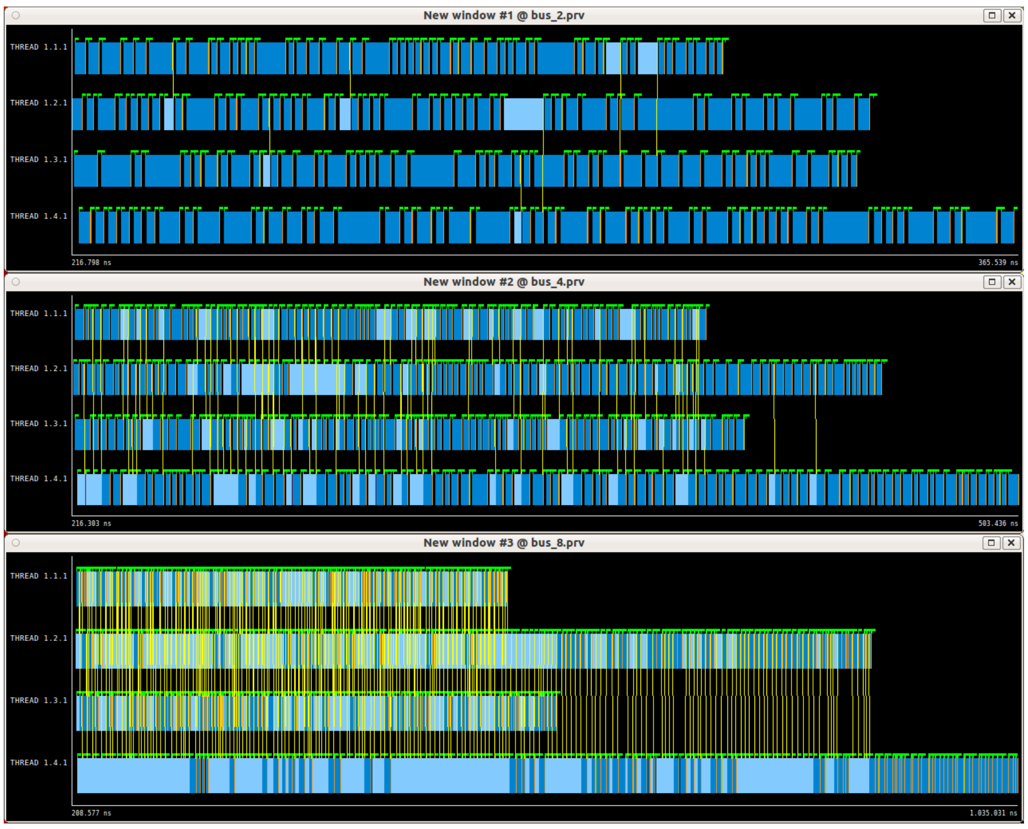


Figure 5.1: Visualization of changing contention levels

Figure 5.1 shows three runs of an application, each with increasing contention levels (amount of aborts). The application simulates financial transactions and runs on 4 cores. Dark blue parts indicate computation done in committed transactions, light blue parts on the contrary indicate computation done in aborted transactions, i.e. wasted work. Yellow lines connect transactions being aborted with the transaction causing the abort. A green flag on top of the timeline of a thread indicates an event (change of HTM state).

5.2 States of a transaction

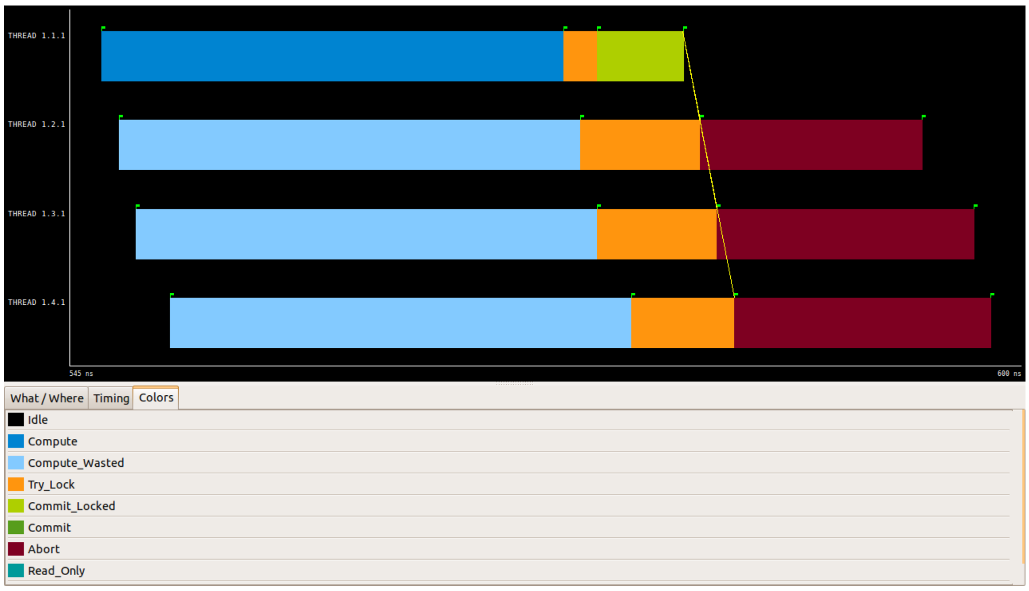


Figure 5.2: Visualization of four transactions

Figure 5.2 shows a close up picture of four transactions executing in parallel. Each color signifies a different transaction state. The following states are recognized and can be analysed:

⬛ "**Idle**": The application is currently not using the HTM unit

🟦 "**Compute**": Time spent doing calculations and transactional reads and writes in successfully commited transactions (i.e. "Useful work")

🟦 "**Compute_Wasted**": Time spent doing calculations and transaction reads and writes in aborted transactions (i.e. "Wasted work")

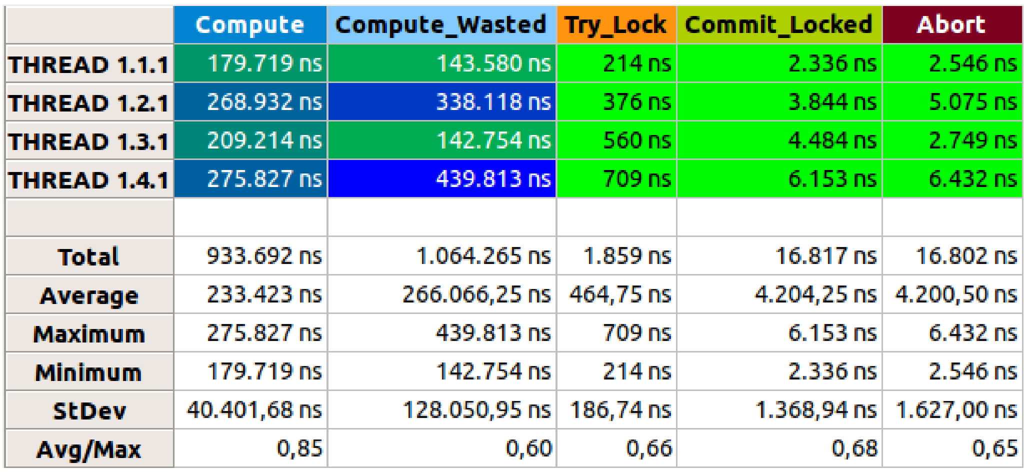
🟧 "**Try_Lock**": A transaction has finished computing, HTM subsystem tries to prepare the commit phase by locking the ring bus

🟩 "**Commit_Locked**": Ring bus lock was acquired, transactional data is stored into RAM

🟥 "**Abort**": Transaction got invalidated, HTM subsystem is reset and transaction is prepared for restart

🟦 "**Read_Only**": A read only transaction (i.e. write set is empty)

5.3 Introduction to Paraver analysis



	Compute	Compute_Wasted	Try_Lock	Commit_Locked	Abort
THREAD 1.1.1	179.719 ns	143.580 ns	214 ns	2.336 ns	2.546 ns
THREAD 1.2.1	268.932 ns	338.118 ns	376 ns	3.844 ns	5.075 ns
THREAD 1.3.1	209.214 ns	142.754 ns	560 ns	4.484 ns	2.749 ns
THREAD 1.4.1	275.827 ns	439.813 ns	709 ns	6.153 ns	6.432 ns
Total	933.692 ns	1.064.265 ns	1.859 ns	16.817 ns	16.802 ns
Average	233.423 ns	266.066,25 ns	464,75 ns	4.204,25 ns	4.200,50 ns
Maximum	275.827 ns	439.813 ns	709 ns	6.153 ns	6.432 ns
Minimum	179.719 ns	142.754 ns	214 ns	2.336 ns	2.546 ns
StDev	40.401,68 ns	128.050,95 ns	186,74 ns	1.368,94 ns	1.627,00 ns
Avg/Max	0,85	0,60	0,66	0,68	0,65

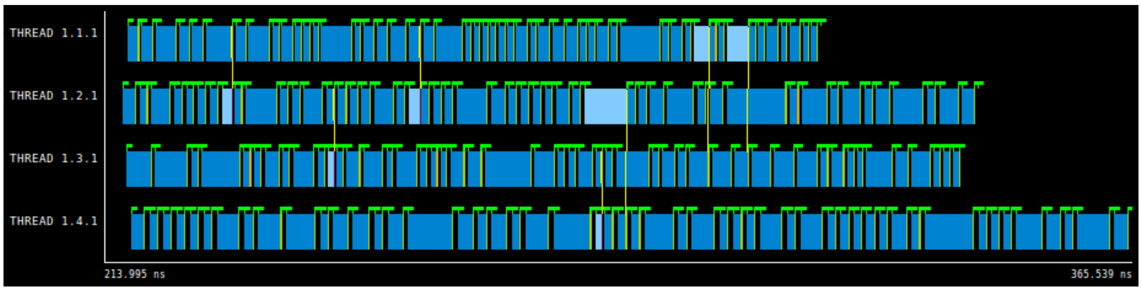
Figure 5.3: Histogram of time spent in HTM stages

Figure 5.3 shows another high-level view of the third run, the run with the highest contention level. This time a histogram of the exact time spent in various HTM stages is displayed. The state names and colors correspond to the states explained in the previous section. The histogram was created using the Paraver analysis module.

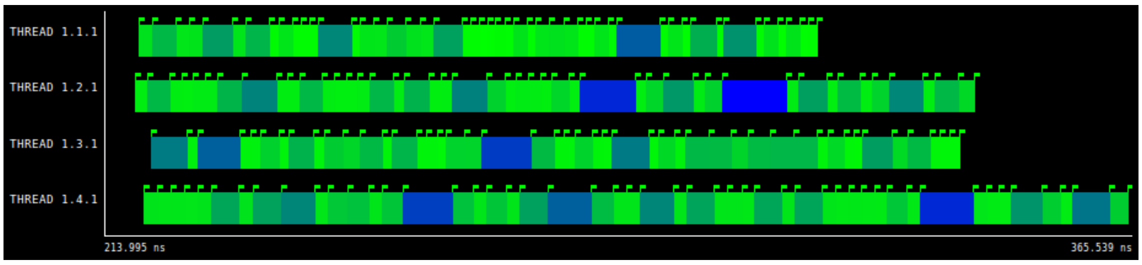
It can be seen that threads 2 and 4 have spent a significantly higher amount of time than the other 2 threads calculating ultimately aborted and thus wasted work. This could be caused by an work imbalance between these threads. This means that work done on threads 2 and 4 creates more conflicts than the work done on threads 1 and 3.

Various metrics useful for analysis can be also calculated from the shown values: The values of rows “Try_Lock”, “Commit_Locked” and “Abort” can be summed up and compared to the overall runtime to get the amount of overhead incurred by the TMbox HTM subsystem. A comparison of rows “Compute” and “Compute_Wasted” gives a rough idea of contention on a system level. The minimum, average, maximum and standard deviation rows show the spread of transaction states, i.e. the difference between the shortest and longest state of a certain type. A more fine-grained analysis down to the contention between two cores can also be done.

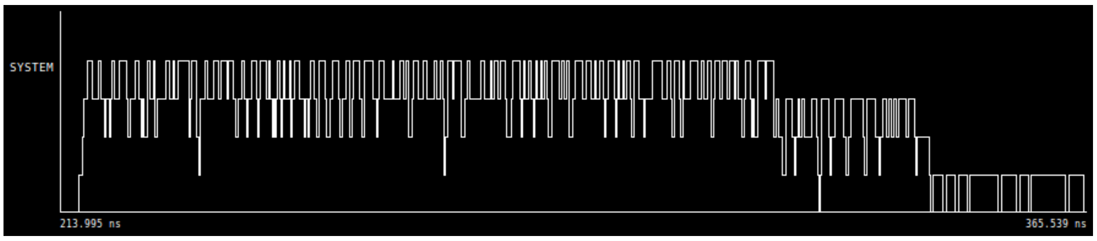
5.4 Visual analysis example I - HTM usage



(a)



(b)



(c)

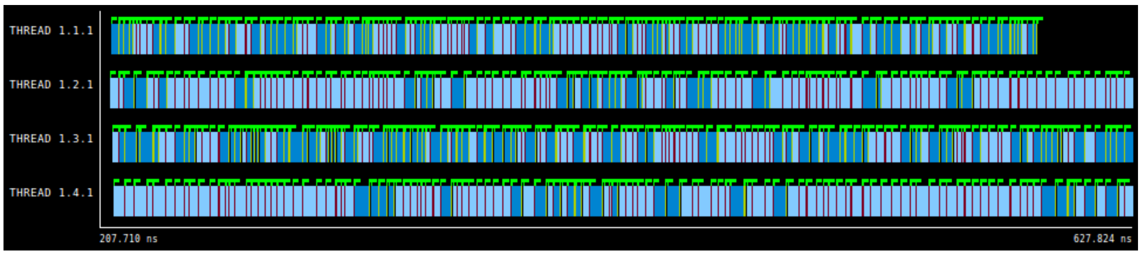
Figure 5.4: Program trace (a) and corresponding rate of commits (b) and number of used HTM units (c)

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

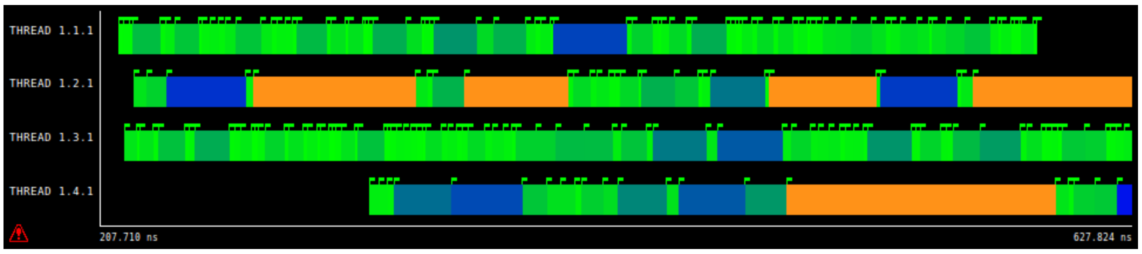
Figure 5.4b shows the rate of commits: Various shades of green correspond to a high rate of commits and a short duration of committing transactions. Blue shades indicate time periods with a low rate of commits and a high duration of committing transactions.

Figure 5.4c shows the number of used HTM units over time (on a system level). During most of the runtime the application uses 2 to 4 HTM units. Later after completion of the first thread the usage changes to between 1 and 3 used HTM 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 HTM unit is used by the last thread. Figure 5.4c is created by the semantic module of Paraver.

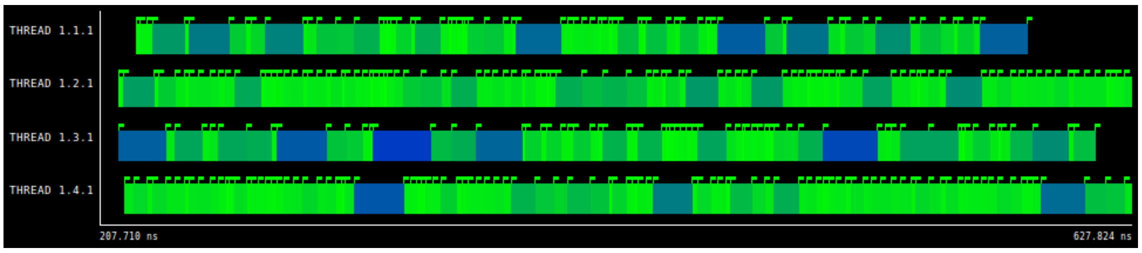
5.5 Visual analysis example II - High contention



(a)



(b)



(c)

Figure 5.5: 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 5.5a correspond to wasted work, i.e. work done in aborted transactions. The Figures 5.5b and 5.5c 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 Summary

The monitoring infrastructure developed in this study thesis allows for an in-depth monitoring, visualization and analysis of HTM applications running on the TMbox platform. The monitored application is not affected in its runtime timing characteristics. Using FPGA technology for the implementation of the monitoring infrastructure allows for fast execution and analysis of long and complex applications. The HTM application behavior is traced and stored for post-processing. Post-processing delivers several analysis metrics that allow to compare different implementations of an application and to optimize their overall runtime and scalability on a given HTM system.

The visualization and analysis capabilities of the existing tool Paraver have been leveraged for the purposes of HTM behavior analysis. This novel features enable the programmer to identify the specific characteristics of different parts of an application and to detect parts with sub-optimal behavior.

6.1 Outlook

Some of the design ideas of the monitoring infrastructure can be generalized beyond TM. This will allow the monitoring of other aspects of computer systems:

The event-based design created in this project can be easily extended to enable the analysis of all parts of processor core operations (like cache, ALU and TLB utilization and memory access patterns). This kind of data is useful for behavior and optimization research in other fields of computer science, for instance operating systems and hardware design, and for the construction of adaptive and self-optimizing systems. The visualization and analysis features of Paraver can be used for this purpose by extending the post processing program created in this study thesis with new event types.

Because of the extensibility of the monitoring infrastructure developed during this study thesis STM systems can also be covered (to trace a pure STM or a combined STM/HTM (HybridTM) system). The HTM unit of the TMbox system works on a best effort base (i.e. not all transactions can be executed using HTM). The TM FSM in the processor core, for instance, issues a fallback to STM mode and restarts the transaction when capacity issues arise. More insight about the capabilities, characteristics and optimization possibilities of HybridTM systems can therefore be obtained by extending the monitoring parts of the monitoring infrastructure to gather data about a STM system. Such work is currently in progress.

Additional work on adding phase detection to the post-processing tool BusEventConverter to automate the analysis process is also currently in progress.

Parts of the post-processing can be ported to VHDL/Verilog and synthesized for use on an FPGA. Automatic interpretation of the collected data allows to build an adaptive system, which reconfigures during runtime according to changing application requirements. A related approach has been published in *"An Organic Computing Approach to Sustained Real-time Monitoring"* [21].

Furthermore the general knowledge gained can help to develop new HTM designs, leading to faster and more efficient HTM systems.

6.2 Acknowledgements

From KIT I would like to thank my supervisor Professor Wolfgang Karl for his backing of both this study thesis and the stay abroad in Barcelona and Martin Schindewolf for his advice, constructive criticism and encouragement. 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. I really enjoyed my stay there. A “thank you” also goes to the members of the BSC-MSRC office for the interesting talks during lunch and the helpful information about Barcelona.

This work was supported by a Short Term Scientific Mission (STSM) grant from the European Cooperation in Science and Technology (COST) Action IC1001 (EuroTM) during June and July 2011.

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 TM 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) TM runtime combining HTM and STM support; transactions run in HTM mode and fall back to STM mode when encountering HTM constraints

STM (Software Transactional Memory) TM runtime using standard ISA instructions; no modification of hardware necessary; usually slower than HTM 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

Bibliography

[1] ROSSBACH, Christopher J. ; HOFMANN, Owen S. ; WITCHEL, Emmett: Is transactional programming actually easier? In: *SIGPLAN Not.* 45 (2010), January, S. 47–56. – ISSN 0362–1340

[2] PANKRATIUS, Victor ; ADL-TABATABAI, Ali-Reza ; OTTO, Frank: *Does transactional memory keep its promises? : results from an empirical study*. Karlsruhe : Universität Karlsruhe, Fakultät für Informatik, 2009 (Interner Bericht / Universität Karlsruhe, Fakultät für Informatik ; 2009,12)

[3] BORKAR, Shekhar ; CHIEN, Andrew A.: The Future of Microprocessors. In: *Communications of the ACM* 54 (2011), Mai, S. 67–77. – ISSN 0001–0782

[4] HERLIHY, Maurice ; MOSS, J. Eliot B.: Transactional memory: architectural support for lock-free data structures. In: *Proceedings of the 20th annual international symposium on computer architecture*. New York, NY, USA : ACM, 1993 (ISCA '93). – ISBN 0–8186–3810–9, S. 289–300

[5] CHUNG, Jaewoong ; YEN, Luke ; DIESTELHORST, Stephan ; POHLACK, Martin ; HOHMUTH, Michael ; CHRISTIE, David ; GROSSMAN, Dan: ASF: AMD64 Extension for Lock-Free Data Structures and Transactional Memory. In: *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. Washington, DC, USA : IEEE Computer Society, 2010 (MICRO '43). – ISBN 978–0–7695–4299–7, S. 39–50

[6] CHRISTIE, Dave ; CHUNG, Jae-Woong ; DIESTELHORST, Stephan ; HOHMUTH, Michael ; POHLACK, Martin ; FETZER, Christof ; NOWACK, Martin ; RIEGEL, Torvald ; FELBER, Pascal ; MARLIER, Patrick ; RIVIÈRE, Etienne: Evaluation of AMD's advanced synchronization facility within a complete transactional memory stack. In: *Proceedings of the 5th European conference on Computer systems*. New York, NY, USA : ACM, 2010 (EuroSys '10). – ISBN 978–1–60558–577–2, S. 27–40

[7] CAO MINH, Chi ; CHUNG, JaeWoong ; KOZYRAKIS, Christos ; OLUKOTUN, Kunle: STAMP: Stanford Transactional Applications for Multi-Processing. In: *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, 2008

[8] SAHA, Bratin ; ADL-TABATABAI, Ali-Reza ; JACOBSON, Quinn: Architectural Support for Software Transactional Memory. In: *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. Washington, DC, USA : IEEE Computer Society, 2006 (MICRO 39). – ISBN 0–7695–2732–9, S. 185–196

[9] DICE, Dave ; LEV, Yossi ; MOIR, Mark ; NUSSBAUM, Daniel: 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*. New York, NY, USA : ACM, 2009 (ASPLOS '09). – ISBN 978–1–60558–406–5, S. 157–168

[10] SONMEZ, Nehir ; ARCAS, Oriol ; PFLUCKER, Otto ; UNSAL, Osman S. ; CRISTAL, Adrian ; HUR, Ibrahim ; SINGH, Satnam ; VALERO, Mateo: 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*. Salt Lake City, UT, USA : IEEE Computer Society, 2011 (FCCM '11). – ISBN 978–0–7695–4301–7, S. 146–153

[11] FUNG, Wilson W. L. ; SINGH, Inderpreet ; BROWNSWORD, Andrew ; AAMODT, Tor M.: Hardware Transactional Memory for GPU Architectures. In: *44th IEEE/ACM International Symposium on Microarchitecture (MICRO-44)*. Porto Alegre, Brazil, December 3-7 2011

[12] CEDERMAN, Daniel ; TSIGAS, Philippas ; CHAUDHRY, Muhammad T.: Towards a Software Transactional Memory for Graphics Processors. In: AHRENS, James P. (Hrsg.) ; DEBATTISTA, Kurt (Hrsg.) ; PAJAROLA, Renato (Hrsg.): *EGPGV*, Eurographics Association, 2010. – ISBN 978–3–905674–21–7, S. 121–129

[13] ANSARI, Mohammad ; JARVIS, Kim ; KOTSELIDIS, Christos ; LUJAN, Mikel ; KIRKHAM, Chris ; WATSON, Ian: Profiling Transactional Memory Applications. In: *Proceedings of the 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*. Washington, DC, USA : IEEE Computer Society, 2009. – ISBN 978–0–7695–3544–9, S. 11–20

[14] CHUNG, Jaewoong ; CHAFI, Hassan ; MINH, Chi C. ; MCDONALD, Austen ; CARLSTROM, Brian D. ; KOZYRAKIS, Christos ; OLUKOTUN, Kunle: The Common Case Transactional Behavior of Multithreaded Programs. In: *In Proceedings of the 12th International Conference on High-Performance Computer Architecture*, 2006

[15] ZYULKYAROV, Ferad: Programming, Debugging, Profiling and Optimizing Transactional Memory Programs (PhD Thesis). http://www.feradz.com/ferad-phdthesis-20110525.pdf

[16] HAMMOND, Lance ; CARLSTROM, Brian D. ; WONG, Vicky ; HERTZBERG, Ben ; CHEN, Mike ; KOZYRAKIS, Christos ; OLUKOTUN, Kunle: Programming with

transactional coherence and consistency. In: *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*. New York, NY, USA : ACM, 2004 (ASPLOS-XI). – ISBN 1–58113–804–0, S. 1–13

[17] CHAFI, Hassan ; MINH, Chi C. ; MCDONALD, Austen ; CARLSTROM, Brian D. ; CHUNG, JaeWoong ; HAMMOND, Lance ; KOZYRAKIS, Christos ; OLUKOTUN, Kunle: TAPE: a transactional application profiling environment. In: *Proceedings of the 19th annual international conference on Supercomputing*. New York, NY, USA : ACM, 2005 (ICS '05). – ISBN 1–59593–167–8, S. 199–208

[18] FAURE, Etienne ; BENABDENBI, Mounir ; PECHEUX, Francois: Distributed online software monitoring of manycore architectures. In: *On-Line Testing Symposium, IEEE International* 0 (2010), S. 56–61. ISBN 978–1–4244–7724–1

[19] *BEE3 Research Platform*. http://research.microsoft.com/en-us/projects/bee3/,

[20] *Paraver Website*. http://www.bsc.es/paraver,

[21] BUCHTY, Rainer ; KRAMER, David ; KARL, Wolfgang: An Organic Computing Approach to Sustained Real-time Monitoring. In: HINCHEY, Mike (Hrsg.) ; PAGNONI, Anastasia (Hrsg.) ; RAMMIG, Franz (Hrsg.) ; SCHMECK, Hartmut (Hrsg.): *Biologically-Inspired Collaborative Computing* Bd. 268. Springer Boston, 2008. – ISBN 978–0–387–09654–4, S. 151–162. – 10.1007/978-0-387-09655-1_14