DISSERTATION

Time-Predictable Java Chip-Multiprocessor

ausgeführt zum Zwecke der Erlangung des
akademischen Grades eines

Doktors der technischen Wissenschaften

unter der Leitung von

O.Univ.Prof. Dipl.-Ing. Dr. Herbert Grünbacher
und
Univ.Ass. Dipl.-Ing. Dr. Martin Schöberl
Institut für Technische Informatik,
Real-Time Systems Group

eingereicht an der Technischen Universität Wien,
Fakultät für Informatik

von

Christof Pitter
Matr.-Nr. 0327072
Markhofgasse 4/17, A–1030 Wien

Wien, im März 2009 _____

Time-Predictable Java Chip-Multiprocessor

Embedded systems are frequently used in real-time applications. Such applications must undergo timing analysis to ensure the timing constraints are met and the mission succeeds. Static worst-case execution time (WCET) analysis yields safe and precise upper bounds of tasks for a given hardware platform. It is preferred to measurement-based analysis methods because it guarantees to consider all possible execution times.

The purpose of this thesis is to design a novel chip-multiprocessor (CMP) solution for the development of Java real-time applications. This chip-multiprocessor system consists of a global physical memory accessible to all processors. A memory arbiter resolves concurrent access of multiple CPUs to the main memory. This architecture enables simple communication by accessing shared data objects. This thesis investigates if a shared memory multiprocessor can serve as a hardware platform for real-time applications. The great challenge is that tasks running on different CPUs of a CMP influence each others' execution times when accessing memory. Therefore, the system's arbiter must limit these interdependencies to be able to analyze WCETs of individual tasks. An adaptation of a static WCET tool for use with the multiprocessor architecture shall permit straightforward WCET analysis results.

In this study, the proposed CMP design is implemented using field-programmable gate array technology. Three different arbitration policies are developed: a fixed priority, a fair-based, and a time-sliced arbiter. Timing analysis approaches are carried out for the specified memory arbiters. Various CMP configurations with varying number of CPUs are evaluated, analyzed, and compared with respect to their real-time and average-case performance. Different benchmarks are used for executing programs on real hardware.

Results of this study have revealed that only the time-sliced memory arbitration scheme allows a calculation of viable WCET bounds of Java applications. A comparison of different CMP configurations shows that dynamic arbitration mechanisms are less predictable in the temporal domain but show better average-case program performance. The principal conclusion of this research demonstrates that timing analysis is possible for homogeneous multiprocessor systems with a shared memory.

Echtzeit Java Chip-Multiprozessor

Eingebettete Systeme werden häufig für sicherheitskritische Anwendungen verwendet. Solche Anwendungen erfordern eine Analyse des zeitlichen Verhaltens, um die zeitlichen Anforderungen garantieren und ein Fehlverhalten mit dramatischen Konsequenzen verhindern zu können. Die statische Worst-Case Execution Time (WCET) Analyse bestimmt sichere und präzise Grenzen für Programmausführungszeiten auf einer vorgegebenen Hardware-Plattform. Sie wird einer messbasierten Analyse vorgezogen, weil alle möglicherweise auftretenden Ausführungszeiten berücksichtigt werden.

Zweck dieser Dissertation ist die Entwicklung einer innovativen Chip-Multiprozessor (CMP) Lösung, für die Entwicklung von Echtzeitanwendungen mittels Java. Dieses symmetrische Multiprozessorsystem besteht aus mehreren CPUs und einem globalen Hauptspeicher. Ein Arbiter löst Speicherzugriffskonflikte zwischen mehreren Prozessoren auf. Die Kommunikation zwischen unterschiedlichen Prozessoren wird unter Verwendung von gemeinsam genutzter Variablen sichergestellt. Die Dissertation untersucht, ob ein symmetrischer Multiprozessor (SMP) als Plattform für Echtzeitanwendungen eingesetzt werden kann. Die große Herausforderung ist, dass sich die Ausführungszeiten der Tasks gegenseitig durch den gemeinsamen Speicherzugriff beeinflussen. Deshalb muss ein Arbiter diese gegenseitigen Beeinträchtigungen beschränken, um maximale Ausführungszeiten von individuellen Tasks analysieren zu können. Die Adaptierung eines bestehenden Analysetools für die Verwendung mit SMPs soll eine einfache Laufzeitbestimmung ermöglichen.

Ein Teil dieser Forschungsarbeit war die Implementierung des CMPs in der FPGA Technologie. Drei verschiedene Arbitertypen wurden entwickelt: ein Arbiter mit fixer Priorität, ein fairer Arbiter und ein Arbiter der die gemeinsame Speicherbandbreite in Zeitschlitze unterteilt. Das Zeitanalysekonzept wird individuell für jeden Arbiter ausführlich präsentiert. Verschiedene CMP-Konfigurationen mit unterschiedlicher Prozessoranzahl werden evaluiert, analysiert und in Bezug auf ihr Echtzeitverhalten bzw. auf ihre Rechenleistung verglichen. Dazu werden Benchmarks am Prototyp ausgeführt.

Die Dissertationsergebnisse bestätigen, dass nur der Arbiter mit dem Zeitschlitzverfahren eine Berechnung von brauchbaren, maximalen Ausführungszeiten zulässt. Der Vergleich von verschiedenen CMP-Konfigurationen zeigt, dass dynamische Arbitrieralgorithmen im Zeitbereich weniger vorhersagbar sind, jedoch größeres Leistungspotenzial besitzen. Diese Dissertation beweist, dass eine statische Zeitanalyse für einen SMP möglich ist.

Acknowledgements

The research for this thesis has been conducted during my employment as a research assistant at the Institute of Computer Engineering, Real-Time Systems Group within the Vienna University of Technology.

First, I would like to thank my adviser Professor Dr. Herbert Grünbacher for his skillful advice, valuable support, and helpful suggestions. I would like to extend my gratitude towards my secondary adviser Professor Dr. Reinhold Weiss for beneficial comments and suggestions on the thesis.

Special thanks go to my thesis co-adviser and colleague Martin Schöberl. We often had interesting and valuable discussions! I really appreciated your inspiration and motivation before upcoming paper submission deadlines. Your commitment and enthusiasm improved my work considerably during the last four years. Furthermore, I would like to thank Peter Puschner, Wolfgang Puffitsch, and Bernhard Gressl for constructive comments and helpful suggestions on the thesis.

I would like to acknowledge the financial support of the Austrian Federal Ministry of Transport, Innovation, and Technology (BMVIT). This work would not have been possible without their funding.

Special thanks go to my friends from all over the world, for keeping real life more interesting than the virtual one. They are an endless source of joy and inspiration and I would not be the same without them.

I am deeply grateful to Valerie and my family for their constant support. This thesis is dedicated to them.

Contents

List of Figures

List of Tables

1 Introduction

This thesis introduces a Time-Predictable Java Chip-Multiprocessor. It is a homogeneous chip-multiprocessor based on a Java processor core, an implementation of a Java Virtual Machine in hardware. It features a high-performance embedded system for Java real-time applications. This chapter describes the motivation for this project. Furthermore, an overview of the project's major contributions to the field and an outline of the thesis are given.

1.1 Motivation

Today embedded systems are omnipresent in modern society and play an important role in our lives. According to [2], it has been estimated that 99% of all processors aim at the embedded systems market. Turley [16] states that only about 2% of microprocessors are used for personal computers. Embedded devices include household appliances like dish washers or coffee machines, consumer electronic devices like MP3-players or digital cameras. Telecommunication applications like personal digital assistants, cell phones, or networking appliances like switches or routers are further examples. The automotive industry is another key driver of the embedded system market. Today, every car contains from about 40 up to 70 electrical control units [3, 10]. All these applications represent only a random selection of the highest growing segment in the computing market.

1.1.1 Real-Time Embedded Systems

Real-time embedded systems are embedded systems with timing requirements. An accurate definition of a real-time computer system can be found in [7] p. 2:

> A real-time computer system is a computer system in which the correctness of the system behavior depends not only on the logical results of the computations, but also on the physical instant at which these results are produced.

A real-time computer system has to produce logically correct results within a specified period of time. If a correct computation result is late, it will be considered useless. In case of a hard real-time system, a late computation may cause a critical system failure leading to disastrous consequences possibly endangering human life. A soft real-time system can tolerate results that occasionally miss their deadlines. The system still produces correct results [2], but accompanied by an inferior service quality (e.g. flickering of a user interface display).

Real-time embedded systems often have to handle concurrent tasks, such as communication with various peripheral systems, computing values for a control loop, or responding to external events. A natural way to handle these concurrent jobs is to split them up into individual tasks. A task is classified a hard real-time task, if a missed deadline may result in a catastrophe [2]. Every hard real-time system contains at least one so-called hard real-time task. Typical examples are embedded devices used in industries like automotive and rail traffic, medicine, avionics, industrial automation, or nuclear power plants.

Many embedded systems are used for applications that prioritize real-time behavior over processing power. Such real-time systems must undergo a timing analysis. Therefore, the worst-case execution time (WCET) of each application task in the system has to be a known factor. Only if these upper bounds are calculable can the task set be analyzed for schedulability on a given processor.

1.1.2 Worst-Case Execution Time

The WCET is the amount of time a task eventually needs to execute under worst-case conditions on a given processor. In [17], Wilhelm et al. define the goal of WCET analysis concerning the upper bounds of execution time thus:

1. they have to be safe, and

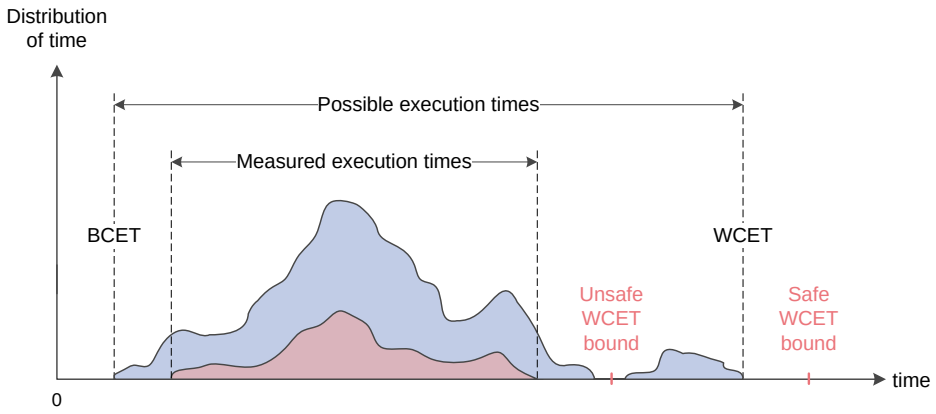

Figure 1.1: Example of safe upper execution time bounds.

2. should be as tight as possible.

The calculated upper time bounds have to be safe in order to ensure hard real-time behavior; otherwise, unpredictable system reactions could put the mission at risk, leading to serious consequences. Moreover, the upper bounds should be as tight as possible to keep the overestimation low in order to conserve resources.

Figure 1.1 shows the variable execution times of a sample program. *Possible execution times* include the best-case execution time (BCET) and the WCET. Additionally, *measured execution times* are shown. The goal of WCET analysis is to find the WCET itself, or an upper bound that is safe and as tight as possible. Unsafe WCET bounds are smaller estimates than the WCET. All upper bounds larger than the WCET are safe. The larger they are, the higher the overestimation of the WCET.

There are three different methods to estimate the WCET of a given task: by measurement, static analysis, or a hybrid approach combining both methods. A WCET analysis by measurement gauges the execution time of a program code using various input data. The estimates are easy to obtain because the analysis is performed on the actual hardware. Therefore, it is especially useful if average-case performance is of interest. A large drawback of the measurement-based method is that the measured WCET result does not reliably confirm that the worst-case program path has been triggered [4], as shown in Figure 1.1.

The objective of a static WCET analysis is to find the maximum execution path and the WCET of a program. It provides a safe upper bound by analyzing the program before runtime, independent of any input values. Even though this method requires an elaborate creation of a precise processor

model, it is the only possibility to obtain a validated upper bound of the application code. Therefore, this analysis method is especially suitable for safety-critical systems.

A hybrid WCET analysis approach starts with a static analysis of the program. The code is split into partitions. Execution times from these code fragments are derived by measurement on real hardware. Finally, these execution times are added to the static analysis model, which calculates the WCET result. No processor model is needed like it is in the static analysis, but safe WCET bounds cannot be guaranteed.

In summary, measurement and hybrid-based analysis can be sufficient for soft real-time systems, but the author believes that static analysis should become the conventional approach to modern hard real-time systems.

1.1.3 Chip-Multiprocessors

Modern applications demand ever-increasing processing power. They act as main drivers for the semiconductor industry. For over 35 years, transistors have been getting faster and clock frequency has adapted accordingly. Additionally, the number of transistors on an integrated circuit at a given cost doubles every 24 months, as described by Moore's Law [9]. The availability of more transistors facilitated an instruction-level parallelism (ILP) approach, which was the primary processor design objective between the mid-1980s and the start of the 21st century. According to [6], processor designers are now reaching the limits of exploiting ILP efficiently. Unfortunately, semiconductor technology has also reached its apex in recent years because of theoretical physical limits. As a result, the frequency, which used to increase exponentially, has leveled off [8].

According to [6], chip-multiprocessors (CMP) are the future in performance enhancement. The CMP technology integrates two or more processing units and a sophisticated communication network into a single integrated circuit. A major advantage of this approach is that any progress in processing power would not be accompanied by an increase in hardware complexity of single processors. Consequently, several processing units coexist on an integrated circuit, utilizing billions of transistors efficiently. According to [18], CMPs combine the significant demands of embedded systems: increased performance, lower power consumption, and cost efficiency.

1.1.4 Java Technology

Traditionally, real-time applications have been designed using assembly languages. Due to hardware advancements and increasing program complexity in embedded system design, a higher level of abstraction was needed.

Subsequently, the C programming language was introduced. More recently, embedded systems have been designed using the C++ programming language, because of its object-oriented feature and several enhancements of C. The increasing size and complexity of today's applications make ever higher demands on the design. The cost of debugging and maintaining the code increases continuously. In summary, the productivity suffers from low-level programming languages.

Java [5] has proven its success in the field of web, desktop and mobile applications. This high-level, object-oriented language has its advantages in the ease of program reuse, robustness, security, and portability. Java does not suffer the pitfalls of the C-based programming languages, e.g. the manual allocation and deallocation of memory or the use of pointers. Originally, the Java platform was not intended for use in real-time systems. The Real-Time Specification for Java (RTSJ) [1], submitted in 1998 and approved in 2002 by the Java Community Process, defines how real-time behavior can be achieved within the Java programming language. An update proposal (JSR-282) improves the RTSJ and includes new enhancements. A further specification for safety-critical Java (JSR-302) is expected soon. It will guarantee that Java-based applications can be certified under the safety-critical development standard in aviation DO-178B level A [11] and other safety critical standards for software.

A project called Java environment for parallel real-time development (JEOPARD) [14] exemplifies the large interest in real-time Java for multicore systems. The consortium of this European Commission funded project, consisting of ten academic and industrial partners, aims at the development of an independent software interface for Java real-time multiprocessor systems. This project uses the proposed processor architecture for the CMP platform evaluation.

The advantages of Java compared to low-level programming languages can be easily outlined from two points of view. From the manager's point of view, Java use shortens the system's time to market, because of the programmer's productivity increase. Additionally, Java developers are a dime a dozen compared to experts developing systems consisting of microprocessors running real-time operating systems. That combination requires great expertise in low-level embedded system programming. From the developer's point of view, Java is considered a simple and easy-to-use programming language compared to C or C++. A large advantage is the automatic memory allocation and garbage collection, which often introduces failures in C-type programs. Large applications are much easier to develop, maintain, and debug. Furthermore, its object-oriented programming model favors reusability.

The Java optimized processor (JOP) [12, 13] is an implementation of the Java Virtual Machine (JVM) in hardware. JOP translates Java bytecodes

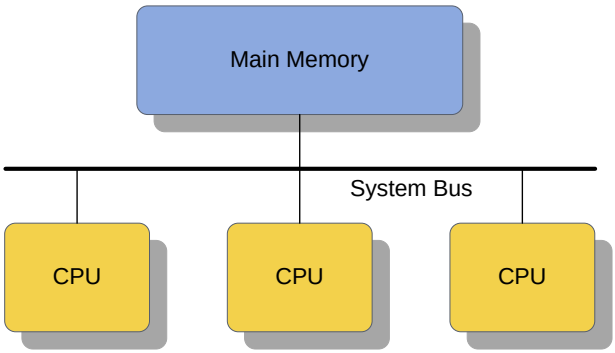

Figure 1.2: Shared memory multiprocessor.

into its own instruction set called microcode. These microcode instructions, implemented in hardware, are executed by the stack architecture. This processor has been designed from scratch to provide a time-predictable execution environment for embedded real-time systems.

1.2 Problem Definition and Objectives

According to [6], two tightly linked major multiprocessor system models exist, the *shared memory model* and the *distributed shared memory model*. The core of the *shared memory model* is a global physical memory accessible to all processors. Multiple CPUs are connected to the memory via a system bus. This architecture enables simple data communication by accessing shared data objects of the common memory. A multiprocessor based on this model is called a symmetric (shared-memory) multiprocessor (SMP), because all processors have symmetric access to the shared memory (see Figure 1.2). It offers uniform memory access times for all CPUs.

In contrast, the *distributed shared memory model* implements a physically distributed memory system. It consists of multiple independent processing nodes with local memory modules, which are connected by an interconnection network. Each CPU can access its own local memory very quickly, but the time it takes to access a memory word located at a distant processing node is much longer. Therefore, this model is called non-uniform memory access (NUMA) architecture and is less appropriate for use in a time-predictable CMP. This thesis investigates SMP architectures.

A fundamental problem in parallel SMP computing is the access of multiple CPUs to the shared memory. A memory arbiter is needed to resolve concurrent access. For use in a time-predictable system, the realization of this memory arbitration mechanism presents two closely related challenges:

- Synchronization of memory access
- Timing analysis of memory access

The arbiter controls the memory access of multiple CPUs to the shared memory. Naturally, if one CPU is accessing the memory, no other CPU can access it simultaneously. They are forced to wait until the currently accessing CPU has completed its memory transfer. In this case, a memory arbiter resolves access conflicts by serializing the read and write operations of the CPUs. The access order is determined by the implemented arbitration algorithm.

In uniprocessor systems, only one processor accesses the memory and the WCET of a memory access can be predicted if memory access latency is predetermined by the memory technology used. However, tasks running on a CMP on different CPUs influence each others' execution times when accessing a shared resource [15], e.g. a shared memory. Therefore, the interdependencies between task execution times have to be removed. A well designed arbitration algorithm is needed to limit the WCET of a task running on a CPU, even though tasks executing on other CPUs may also access the main memory. Consequently, an analysis of WCET bounds is possible.

The objectives of this thesis can be summarized as follows:

- Design and implementation of a homogeneous chip-multiprocessor with global shared memory
  - based on Java technology
  - with time-predictable execution times, if memory provides predictable latencies
  - verified by a prototype implementation in field-programmable gate array (FPGA) technology
- Adaptation of a WCET analysis tool tailored to the CMP
- Comparison and evaluation of different CMP configurations with respect to
  - real-time performance (WCET)
  - average-case performance (average-case execution time - ACET)

1.3 Contributions

This thesis presents a novel Java chip-multiprocessor with a new WCET analysis method that actually makes possible a timing analysis for homogeneous multiprocessor systems with a shared memory. The above mentioned

challenges and objectives are addressed in this thesis. The major contributions are as follows:

- **Time-Predictable Multiprocessor Design**

  The proposed Java chip-multiprocessor is designed for maximum time predictability, where simple and accurate WCET analysis is more important than good average-case performance. Additionally, tight WCET bounds have been a design goal. The CMP architecture is based on multiple time-predictable Java processors and a shared memory. The global physical memory, accessible to all processors, stores all instructions and data. A memory arbiter controls the memory access of multiple CPUs. Synchronization guarantees coordinated access to shared objects among several processors. All components are interconnected with a System-on-Chip (SoC) bus. This CMP operates without caching shared data objects, therefore an identical data perspective is ensured for all CPUs throughout the execution of an application. Consequently, hardware demanding cache coherence mechanisms can be avoided. The proposed multiprocessor is tailored to utilize the multi-threaded nature of real-time applications.

- **Memory Arbitration**

  A memory arbiter is responsible for controlling the memory access of multiple CPUs to the shared memory. Naturally, if one CPU is accessing the memory, other CPUs must not do so at the same time. They are forced to wait until the currently accessing CPU has completed its memory transfer. In this case, a memory arbiter resolves access conflicts by serializing the read and write operations of different CPUs.

  In multiprocessor systems, tasks are running on different CPUs and influence each others' execution times when accessing a shared memory. Therefore, an arbitration algorithm is necessary, which is able to limit the WCET of a task running on a CPU, even though tasks executing on other CPUs may also access the main memory. A time-sliced arbiter that divides the memory bandwidth among the CPUs avoids interferences between task execution times. Consequently, an analysis of tight WCETs is rendered possible.

- **Prototype Implementation**

  A prototype implementation enables CMP architecture validation. Two different hardware platforms have been used for evaluation purposes. They use two different FPGA technologies and have different memory bandwidth capacities. An integration of up to 8 cores could be verified using several different benchmarks.

One of this paper's major contributions is the implementation of three different arbitration policies: a fixed priority, a fair-based, and a time-sliced arbiter. They provide a basis for the comparison of arbitration policies with respect to WCET and average-case performance. Furthermore, several system components had to be designed and implemented, e.g. a synchronization process, and a CMP boot-up sequence.

- **Static WCET Analysis**

  Static WCET analysis finds the worst-case execution time of a given program code for a specific processor model. Overestimation should be kept to a minimum. In a CMP system, tasks running on different CPUs shall not influence each others' execution times when accessing the shared memory. The WCET analysis is primarily dependent on the memory arbiter. In this paper, a static timing analysis approach of each arbitration policy is presented. Some of them turn out not to be viable for hard real-time systems because of their unacceptable WCET results. Only timing analysis using a time-sliced arbiter leads to realistic and viable WCET bounds.

  One contribution of this thesis is the enhancement of JOP's WCET analysis tool for use with multiprocessor systems. The tool can be configured for analysis of different hardware platforms and system configurations.

- **Performance Comparison**

  An implementation of a soft multiprocessor core in an FPGA holds several advantages: ease of rapid prototyping, high configuration and simple verification potential of particular components. Different CMP configurations are evaluated by varying the number of processors, their instruction cache sizes, the memory bandwidth, and arbitration policies. In our experiments, their average-case performance is compared by running different benchmarks on real hardware. Furthermore, CMP versions are compared to a complex Java processor.

1.4 Thesis Outline

The next four chapters of the thesis present the published results of peer-reviewed papers presented at international conferences. These papers appear as individual chapters. Each paper consists of an abstract, an introduction, a related work section, and the main research findings. Each article was written as an individual publication, therefore, a few statements may appear repetitive.

Time Predictable CPU and DMA Shared Memory Access

Christof Pitter and Martin Schoeberl. Proceedings of the International Conference on Field Programmable Logic and Applications (FPL 2007). Amsterdam, Netherlands, August 2007, pages 317-322.

Chapter 2 describes the starting point of my research. This paper evaluates two different timing analysis approaches of a system consisting of one CPU and a direct memory access (DMA) controller. The first implementation of a fixed priority arbiter is responsible for controlling the memory access of both processing units. The DMA controller can be modeled to appear as a hardware-based real-time task which accesses the memory with a regular pattern. Therefore, its WCET is known and the task can be easily integrated into the schedulability analysis. A novel WCET approach describes how each memory access (an occurring read or write access) of the DMA task can be included into the WCET of the application task. Consequently, the application task's WCET increases, but the DMA task can be omitted from the schedulability analysis. Experiments showed that this new approach leads to tighter response times and saves more processing resources for the CPU. In summary, this paper shows that it is possible to analyze the timing behavior of a system consisting of a CPU and a hardware task with a known access pattern both accessing a shared memory.

Towards a Java Multiprocessor

Christof Pitter and Martin Schoeberl. Proceedings of the 5th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2007). Vienna, Austria, 2007, pages 144-151.

Chapter 3 introduces a novel Java multiprocessor architecture for embedded systems. This paper explains why a shared memory model is preferred to a distributed shared memory model for time-predictable multiprocessors. The proposed CMP design consists of a number of Java optimized processor (JOP) cores. Based on the work described in Chapter 2, several improvements on the fixed priority arbiter ensure a solution for simultaneous access of multiple CPUs to the shared main memory. Furthermore, the synchronization of shared data objects is examined. Another interesting aspect of a CMP system, the startup or boot-up, is described in detail. The first FPGA prototype using multiple JOP cores verifies the correct concurrent execution of application tasks. Finally, CMP versions made up of two/three JOPs enable a performance comparison between a single JOP and the CMP versions by running real applications on hardware. The resulting speed-ups encouraged further investigation into the proposed CMP architecture.

Performance Evaluation of a Java Chip-Multiprocessor

Christof Pitter and Martin Schoeberl. Proceedings of the International Symposium on Industrial Embedded Systems (SIES 2008). Montpellier, France, June 2008, pages 34-42.

Chapter 4 evaluates the Java CMP system with respect to average-case performance. Different hardware configurations with varying instruction cache sizes, number of processors, and memory bandwidth are compared. An implementation of a fair-based arbiter guarantees fair memory access among the CPUs. Experiments measure the performance by running three benchmarks on two different FPGA platforms: an embedded industry application, a computationally intensive matrix multiplication, and a synthetic benchmark that continuously accesses a shared data structure. Compared to Chapter 3, one of the prototype boards allows for an integration of up to eight CPUs. Performance results show that a multiprocessor version of a simpler and smaller architecture is more efficient (performance/die area) for parallel workloads than a complex Java processor.

Time-Predictable Memory Arbitration for a Java Chip-Multiprocessor

Christof Pitter. Proceedings of the 6th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2008). Santa Clara, California, 2008, pages 115-122.

Based on the findings of Chapter 2 and 3, this paper proposes a real-time CMP system that allows WCET analysis of tasks running on a homogeneous Java CMP. The core of this CMP is a time-sliced arbiter that divides the memory access bandwidth into equal time slots, one for each CPU. Consequently, WCETs of Java bytecodes can be analyzed depending on the size of the time slot, number of CPUs in the system and memory access time. An adapted WCET analysis tool for use with a CMP system can utilize these results and generates temporal upper bounds for application tasks. A real-world application task is used to compare analyzed results with measured execution times.

Further Analysis and Evaluation

Chapter 6 is based on a submitted paper called *A Real-Time Java Chip-Multiprocessor*. This chapter is dedicated to compare and evaluate CMPs using implemented arbitration policies (fixed priority, fair-based, and a time-sliced policy) with respect to their real-time and average-case performance.

Various CMP configurations are evaluated using a larger application base than described in previously presented chapters.

Conclusion and Outlook

Chapter 7 concludes this thesis by giving a systematic summary of the research process. It recapitulates the main findings of the conducted experiments. Additionally, it provides some further ideas for future research.

Bibliography

[1] Greg Bollella, James Gosling, Benjamin Brosgol, Peter Dibble, Steve Furr, and Mark Turnbull. *The Real-Time Specification for Java*. Java Series. Addison-Wesley, June 2000.

[2] Alan Burns and Andrew J. Wellings. *Real-time systems and programming languages: Ada 95, real-time Java, and real-time POSIX*. International computer science series. Addison-Wesley, third edition, 2001. Revised edition of *Real-time systems and their programming languages*, 1990.

[3] Axel Deicke. The electrical/electronic diagnostic concept of the new 7 series. In *Proceedings of Convergence International Congress & Exposition On Transportation Electronics*, Detroit, USA, October 2002.

[4] Andreas Ermedahl and Jakob Engblom. Execution time analysis for embedded real-time systems. In Sang H. Son Insup Lee, Joseph Y-T. Leung, editor, *Handbook of Real-Time Embedded Systems*, pages 35.1 – 35.17. Chapman & Hall/CRC - Taylor and Francis Group, August 2007.

[5] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, 1997.

[6] John Hennessy and David Patterson. *Computer Architecture: A Quantitative Approach, 4th ed.* Morgan Kaufmann Publishers, 2006.

[7] H. Kopetz. *Real-time systems: design principles for distributed embedded applications*. Kluwer Academic Publishers, 1997.

[8] James Laudon and Lawrence Spracklen. The coming wave of multithreaded chip multiprocessors. *International Journal of Parallel Programming*, 35(3):299–330, June 2007.

[9] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, 1965.

[10] Roman Obermaisser, Philipp Peti, and Fulvio Tagliabo. An integrated architecture for future car generations. *Real-Time Systems*, 36:101–133(33), July 2007.

[11] RTCA. Software considerations in airborne systems and equipment certification. Guideline DO-178A, Radio Technical Commission for Aeronautics, One McPherson Square, 1425 K Street N.W., Suite 500, Washington DC 20005, USA, March 1985.

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

[13] Martin Schoeberl. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture*, 54/1–2:265–286, 2008.

[14] Fridtjof Siebert. Jeopard: Java environment for parallel real-time development. In *JTRES '08: Proceedings of the 6th international workshop on Java technologies for real-time and embedded systems*, pages 87–93, New York, NY, USA, 2008. ACM.

[15] Lothar Thiele and Reinhard Wilhelm. Design for timing predictability. *Real-Time Systems*, 28(2-3):157–177, 2004.

[16] James L. Turley. *The Essential Guide to Semiconductors*. Prentice Hall PTR, 2003.

[17] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David B. Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst*, 7(3):1–53, 2008.

[18] Wayne Wolf. *High-Performance Embedded Computing: Architectures, Applications, and Methodologies*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

2

Time Predictable CPU and DMA Shared Memory Access

Christof Pitter and Martin Schoeberl

Institute of Computer Engineering,
Vienna University of Technology,
Austria

Proceedings of the International Conference on Field Programmable Logic and Applications (FPL 2007). Amsterdam, Netherlands, August 2007, pages 317-322.

Abstract

In this paper, we propose a first step towards a time predictable computer architecture for single-chip multiprocessing (CMP). CMP is the actual trend in server and desktop systems. CMP is even considered for embedded real-time systems, where worst-case execution time (WCET) estimates are of primary importance. We attack the problem of WCET analysis for several processing units accessing a shared resource (the main memory) by support from the hardware. In this paper, we combine a time predictable Java processor and a direct memory access (DMA) unit with a regular access pattern (VGA controller). We analyze and evaluate different arbitration schemes with respect to schedulability analysis and WCET analysis. We also implement the various combinations in an FPGA. An FPGA is the ideal platform to verify

the different concepts and evaluate the results by running applications with industrial background in real hardware.

2.1 Introduction

This paper presents a hard real-time system consisting of a hard real-time application running on a time predictable Java Optimized Processor (JOP) [10] and an additionally direct memory access (DMA) unit with a regular access pattern. This unit is represented by a video graphics array (VGA) controller. Both the CPU and the DMA unit share the main memory of the system. Meeting the deadlines of the tasks of the real-time application is of utmost importance. Therefore, the task set of the system requires a timing validation by schedulability analysis.

A real-time computer system has to produce logical correct results within a specified period of time. If a correct result of a computation is late, the result is considered useless. Such real-time systems (RTS) or safety-critical systems have to handle concurrent tasks, such as communication, calculating values for a control loop, user interface and supervision in embedded systems. A natural way to handle these concurrent jobs is to split them up into individual tasks. Every hard real-time system contains at least one so-called hard real-time task. A task is classified a hard real-time task when a missed deadline may cause a critical failure of the system. Non safety-critical tasks in such a system are soft real-time tasks. If a deadline is missed, the system will still produce correct results [4] but with degraded service.

Safety-critical systems must be predictable in the time domain. It is of utmost importance to be able to analyze the maximal time or WCET of the task. If and only if these upper bounds can be calculated, the schedulability analysis can be performed. They are necessary to test whether a task set can be scheduled on the target system or not [8].

There exist two possibilities for modeling and implementing the RTS consisting of the application running on the CPU and the DMA controller sharing the main memory:

- DMA as soft real-time task
- DMA as hard real-time task

In the first approach, the task of the DMA controller is represented as a soft real-time task. As a consequence, some interference (e.g. flickering on the VGA display) may occur when the deadline of the DMA is violated. Hence, the DMA task performs in a best-effort manner and can be excluded

from schedulability tests. Increasing the buffer size in the DMA controller can help to support a smooth communication between the shared memory and the DMA controller. Nevertheless, this kind of a system is of minor importance because hard RTSs are the target of this paper.

In the second attempt the DMA task as well as the hard real-time application running on the CPU has to meet its deadline. As a consequence, the system contains the application hard real-time tasks and the memory-streaming hard real-time task that are executed simultaneously. Although a VGA display is usually not accounted as a hard real-time task, it serves as a good example demanding a constant amount of data within a known period of time. In the future, the VGA will be replaced by another CPU.

The rest of the paper considers the second approach (hard RTS) and is structured as follows. Section 2.2 presents the related work. In Section 2.3, we explain the two basic options of analyzing the behavior of the RTS and describe how schedulability tests are carried out. Section 2.4 describes the JopVga system and the worst-case analysis results of several experiments. At the end, it discusses the acquired outcome of the paper. Finally, Section 2.5 concludes the paper and gives guidelines for future work.

2.2 Related Work

In [2] Atanassov and Puschner describe the impact of dynamic RAM refresh on the execution time of real-time tasks. The use of DRAM memory in RTSs involves a major drawback because these memory cells have to be periodically refreshed. During the refresh, no memory request can be processed. As a consequence, the request is delayed and the execution time of the task increases.

Many researchers in the real-time community have turned their attention to timing-analysis tools described by Puschner and Burns [8]. The problem is that the available results for uniprocessors are not applicable to modern processor architectures [8, 12]. Multiprocessor systems consisting of shared memories and busses are hard to predict. In addition, the WCET of each individual task depends on the global system schedule.

Even though so much research has been done on multiprocessors, the timing analysis of the systems has been neglected. An example represents the scalable, homogeneous multiprocessor system by Gaisler Research AB. It consists of a centralized shared memory and up to four LEON processor cores that are based on the SPARC V8 architecture [5]. This embedded system is made available as a synthesizable VHDL model and therefore is well suited for SoC designs. LEON is introduced for European space projects as well as for military and demanding consumer applications. Nevertheless, no

literature concerning WCET analysis regarding the multiprocessor has been found.

Another example depicts the ARM11 MPCore [1]. It introduces a pre-integrated symmetric multiprocessor consisting of up to four ARM11 microarchitecture processors. The 8-stage pipeline architecture, independent data and instruction caches and a memory management unit for the shared memory make a timing analysis difficult.

We believe that the impact of WCET of real-time tasks sharing a main memory has not received enough coverage in literature yet. This paper is a first step towards a time-predictable multiprocessor. Providing WCET guarantees and reliable schedules for a multiprocessor system becomes a great challenge.

2.3 CPU/DMA shared memory access

In Section 2.1, we presented two possibilities of integrating the DMA task into the system. Either the DMA controller represents a soft real-time task or it is considered a hard real-time task. This paper addresses the second solution.

2.3.1 Implementation of the DMA hardware controller

The DMA controller accesses the shared memory to read or write data autonomously of the CPU. The volume of the data transfer depends on the I/O-device. Therefore, the application of the device defines the quantity of memory requests within a fixed period of time. Two different options are evaluated to implement the memory access scheme of the DMA controller (see Figure 2.1):

1. The DMA controller accesses the shared memory in a blocked scheme. This approach is used for a fast copy of large blocks of the main memory to another device. Assume this task has the smallest period of all tasks within the system. Hence using fixed-priority scheduling [7] it has the highest priority. After the DMA task is completed, the other tasks of the CPU get permission to access the shared memory depending on their priority. This approach is a good representation of a multimedia task, such as streaming data, performed via DMA.

2. The DMA controller accesses the memory in a timely spread scheme denotes the other extreme. It generates a smaller period because all memory requests are timely spread on the original period. The highest

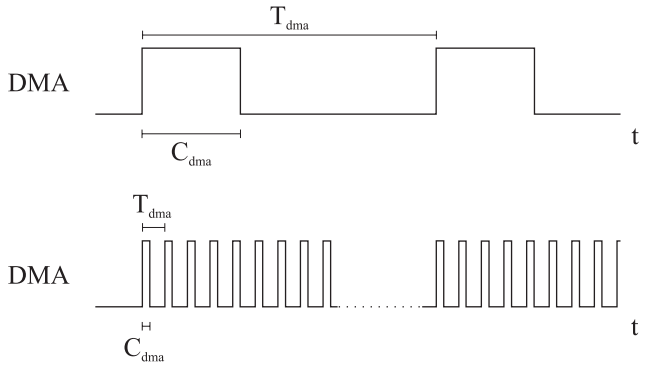

Figure 2.1: The blocked and the spread memory access scheme of the DMA task.

priority of this task is required. If this task does not hold the highest priority, the system will not function correctly because the DMA task will starve when the CPU's software tasks make extensive memory requests.

2.3.2 Task vs. WCET based Analysis

There exist two possibilities to analyze the timing behavior of the RTS:

- DMA access represents an additional real-time task
- DMA access is included in the WCET analysis of each individual application task

The first method considers the DMA task in the schedulability analysis. Hence, all the tasks of the application running on the CPU and the DMA task have to be considered. This simple task set consists of independent periodic tasks with fixed priority. The DMA controller must have the highest priority. The resulting task set can be used for schedulability tests to analyze the timing behavior of the application running on the CPU.

The second approach models the RTS in a different way. The DMA controller and the CPU are accessing shared memory. We are interested in the WCET of the application in spite of the memory communication of the DMA. Therefore, the blocking delay, caused by each possible read or write access of the DMA controller, has to be added to the WCET estimations of the real-time tasks running on the CPU. The results serve bounded WCET estimates of each individual task. As a consequence, the WCET values for the tasks increase, but the DMA task can be omitted from the schedulability analysis.

2.3.3 Schedulability Analysis

The major goal of this paper is the analysis of the timing behavior of the RTS depending on the different options of the memory access of the DMA controller.

Assume that the CPU of the system runs several real-time tasks that are accessing the shared memory. Additionally the DMA controller requests data of the main memory with a regular access pattern. Therefore, the system consists of the DMA task and the tasks of the real-time application running on the CPU. Using fixed-priority scheduling [7], the priorities of the tasks are ordered rate monotonic. The smaller the period the higher is the priority of the task. In order to ensure that all tasks can be completed within their deadlines schedulability tests are carried out.

Utilization-based schedulability test

In [7] it has been shown that a simple schedulability test can be carried out by taking the utilization of the several tasks into account. The utilization is the result of dividing the computation time by the period of the task. If Equation 2.1 holds then all tasks will meet their deadlines. Otherwise, the task set may or may not fail at run-time. C_i denotes the computation time of the task τ_i, T_i is the period of task τ_i and N stands for the number of the tasks to schedule.

$$\sum_{i=1}^{N}(C_i/T_i) \leq N(2^{1/N} - 1) \tag{2.1}$$

If the task set fails, the utilization-based schedulability test cannot guarantee that all tasks will meet their deadlines. Nevertheless, the task set may not fail at run-time.

Response time analysis

A more exact schedulability test by Joseph and Pandya is presented in [6]. The result of this response time analysis for a set of independent tasks provides a necessary and sufficient condition. If the result is positive the task set will be schedulable at run-time. The task set will not be schedulable if the test fails. The worst-case response time R_i of each individual task is calculated and then compared with the task's deadline or period respectively. The equation for the response time is:

$$R_i = C_i + \sum_{j \in hp(i)} \lceil R_i/T_j \rceil \cdot C_j \tag{2.2}$$

The expression $hp(i)$ denotes all tasks with a higher priority than the task τ_i. The smallest R_i that solves Equation 2.2 is the worst-case response time of τ_i. A recurrence relationship can be formed that allows the calculation of the response time [3]:

$$w_i^{n+1} = C_i + \sum_{j \in hp(i)} \lceil w_i^n / T_j \rceil \cdot C_j \tag{2.3}$$

The solution is found when $w_i^{n+1} = w_i^n$. Then w_i^n represents R_i. If one task i has a larger response time than its deadline (or period T_i) the task set cannot be scheduled.

2.4 Evaluation

This section provides an overview of the JopVga system and the corresponding sample application. Furthermore the timing behavior of the RTS is analyzed. The results of the experiments and calculations are compared and classified.

2.4.1 JopVga System

The JopVga system is the hardware used for our experiments. It consists of a time-predictable processor called JOP [10], a VGA controller, an arbiter, a memory interface and an SRAM memory. JOP, the VGA controller, the memory arbiter and the memory interface are implemented on an Altera Cyclone FPGA. As illustrated in Figure 2.2 the external memory is connected to the memory interface. This 1 MByte 32 Bit external SRAM device represents the shared memory of the JopVga system. A SoC bus, called SimpCon [11], connects JOP and the VGA controller with the arbiter. The arbiter is connected via SimpCon to the memory interface.

The arbiter is responsible for setting up the communication between the shared memory and the VGA controller and JOP respectively. It schedules the memory communication of both masters. The shared main memory of the system is divided into two segments: 640 KByte are dedicated to JOP and the remaining 384 KByte are used as frame buffer.

Both JOP and the VGA controller run at a clock frequency of 80 MHz, resulting in a period of 12.5 ns. The SRAM-based shared memory has an access time of 15 ns per 32 bit word. Hence every memory access needs at least 2 cycles. At the moment the arbiter as well as the VGA controller is not capable of using the pipelining approach of the memory access that is introduced by the SimpCon interface [11]. Each memory request of JOP

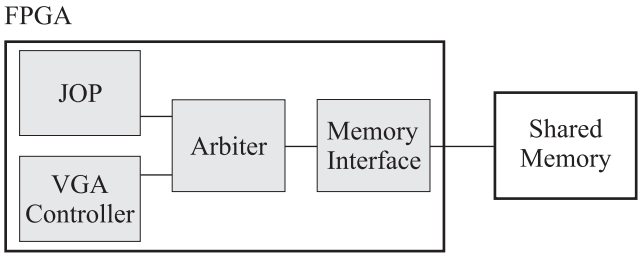

Figure 2.2: JopVga system.

takes 4 cycles and every VGA request takes 3 cycles. The bandwidth of the memory (BW_{mem}) calculates to:

$$BW_{mem} = 4Byte/25ns = 160MByte/s \tag{2.4}$$

The VGA controller uses a VGA resolution of $1024 \cdot 768$ pixels with each pixel consisting of 4 bits. As a consequence the memory used for the VGA display results in 384 KByte ($1024 \cdot 768 \cdot 0.5$ Byte). The horizontal frequency is 60 kHz, which results in a horizontal period of about 17 μs per line and 17 ns per pixel. The vertical frequency is 75 Hz. Therefore the bandwidth used by the VGA calculates to:

$$BW_{vga} = (128 \cdot 4Byte)/17ns = 30.12MByte/s \tag{2.5}$$

Dividing BW_{vga} by BW_{mem} results in 18.83% of the memory bandwidth for the VGA controller. JOP has a bandwidth of 80 MByte/s which results in 50% of the memory bandwidth available.

Application

In order to estimate the worst-case execution time of a RTS all real-time tasks have to be taken into account. In the following the task set illustrated in Table 2.1 represents the periods and the computation times of the system under test. It consists of the VGA task τ_{vga} and two tasks running on JOP τ_{lift} and τ_{kfl}. The priority of 3 depicts the highest prior task. This simple task set illustrates an RTS that is further analyzed using schedulability tests. The VGA task can be modeled as a periodic task with the highest priority and a fixed runtime. The computation time C_{vga} is calculated by multiplying 128 memory accesses times 3 cycles. Using a clock frequency of 80 MHz results in 4.8 μs. T_{vga} is predetermined by the horizontal period of 17 μs.

Two real-world examples with industrial background represent the two tasks running on JOP. `Lift` is a lift controller used in an automation factory. `Kfl` is one node of a distributed RTS to tilt the line over a train for easier loading

Table 2.1: Task set.

τ	T (μs)	C (μs)	Priority
τ_{vga}	17	4.8	3
τ_{lift}	500	162.7	2
τ_{kfl}	3000	782.2	1

and unloading of goods wagon. Both applications consist of a main loop that is executed periodically. In our experiments we use both applications to represent two independent real-time tasks. The WCET of these two tasks are inferred from the WCET analysis tool [12]. In the second column of Table 2.2 the WCET estimates of the two tasks are given in clock cycles. Multiplying those estimates with the clock period result in $\tau_{lift} = 162.7$ μs and $\tau_{kfl} = 782.2$ μs.

2.4.2 Analysis using the Task Approach

In this section the VGA controller represents another real-time task of the system that is taken into account in the analysis of the timing behavior. Both the blocked memory access scheme and the spread memory access scheme, as described in Section 2.3.1, are evaluated.

Blocked

Using the values of Table 2.1 the utilization of each individual task is calculated dividing the computation time C_i by the corresponding period T_i resulting in $U_{vga} = 28.24\%$, $U_{lift} = 32.54\%$ and $U_{kfl} = 26.07\%$. Applying the utilizations to Equation 2.1 results in an overall utilization of 86.85%. The overall utilization may not be more than $3(2^{1/3} - 1) = 78.00\%$ because three tasks are involved. The condition does not hold and consequently this task set fails the utilization-based schedulability test. It cannot be guaranteed that all tasks meet their deadlines. Therefore, a response time analysis is carried out next.

The response time R_{vga} is the same as the computation time because this task has the highest priority.

$$R_{vga} = C_{vga} = 4.8\mu s \tag{2.6}$$

The response time of the next lower prior task τ_{lift}, denoted as R_{lift}, is the addition of the computation time C_{lift} and the time of interference of all higher prior tasks (in that case the interference of τ_{vga}).

$$w_{lift}^{n+1} = C_{lift} + \lceil w_{lift}^{n} / T_{vga} \rceil \cdot C_{vga} \tag{2.7}$$

The response time is calculated using the values from Table 2.1. The response time of R_{lift} is found when w_{lift}^{n+1} equals to w_{lift}^{n}. It is 229.9 μs which is less than T_{lift}. Finally, R_{kfl} has to be calculated. R_{kfl} is the addition of the computation time C_{kfl} and the time of interference of the two higher priority tasks τ_{vga} and τ_{lift}. The result of R_{kfl} is 1999.4 μs. All the response times are smaller than their appropriate periods and hence the response time analysis has a positive outcome. This response time calculation ensures that the tasks will meet their deadlines because the successful analysis is sufficient and necessary [4] even though the utilization-based schedulability test could not be passed.

Spread

The VGA task accesses the memory in a timely spread scheme. The new values for the period and the computation time of τ_{vga} are calculated by dividing the original period $T_{vga} = 17\mu$s of Table 2.1 by the cycle time of 12.5ns. It results in 1360 cycles. We need 128 memory requests for each line on the VGA. Hence $T_{vga} = 10$ cycles and $C_{vga} = 3$ cycles. The remaining 80 cycles are not used. Hence the values for the VGA task change to $T_{vga} =$ 125 ns and $C_{vga} = 37.5$ ns.

Using these values for JOP's tasks and the values for τ_{lift} and τ_{kfl} the utilization-based schedulability test logically results in a similar overall utilization of 88.61% as described in the previous section. A small divergence between the results can be explained by the remaining 80 cycles that are not used in this memory access scheme. Again, the utilization test is negative.

Therefore, the response time analysis is used. R_{vga} is equal to $C_{vga} = 37.5$ ns. R_{lift} calculates to 232.5 μs and $R_{kfl} = 2279.6$ μs. The positive result of the response time analysis shows that the task set can be scheduled. As in the utilization-based test, the remaining 80 cycles affect the results of R_{lift} and R_{kfl}. Both response times are larger than in the blocked memory access scheme. As a consequence, the spread memory access scheme is worse than the blocked scheme.

2.4.3 Analysis using the WCET method

The second approach to include the DMA unit in the schedulability analysis is to model the DMA access in the WCET values for memory access of the software tasks. Each instruction that accesses memory has to include the maximum delay due to a possible memory access by the DMA unit.

Table 2.2: WCET estimates given in clock cycles.

App	JOP only	JOP with VGA	Increase
Kfl	62573	83131	33%
Lift	13016	16118	24%

Table 2.3: Task set for the WCET method.

τ	T (μs)	C (μs)	Priority
τ_{lift}	500	201.5	2
τ_{kfl}	3000	1039.1	1

Blocked

Using the blocked memory access scheme and the WCET method for analysis is not a reasonable approach. One would have to account the delay of the whole block of memory requests of the VGA to each memory access of JOP. That results in a very conservative WCET for each memory access of JOP. Hence, it is not further investigated.

Spread

The WCET analysis tool [12] can be parameterized with respect to the memory access time (the wait states for memory read, memory write, and the cache load). The memory access from the VGA unit takes 2 cycles plus 1 cycle in the arbiter to switch between the two masters. Therefore, we add 3 cycles to the wait states.

Table 2.2 shows the WCET values in clock cycles for different applications for the stand-alone processor and when adding the DMA device. We see an increase of 24% to 33% of the WCET for the tasks.

These values are conservative as each memory access is modeled with the maximum blocking time. For a single memory access (such as bytecode `getfield`) this is the best we can do without further analysis of the instruction pattern. However, for cache loading we could include the access pattern (in our example one access per 10 clock cycles) into the analysis of the cache load time. Previously experiments showed that the load time for the method cache produces most of the memory requests. Consequently, with inclusion of the access pattern into the WCET analyzer tool, we can provide tighter WCET values. This is a new approach to WCET analysis, as in all current approaches the WCET analysis is independent from the schedulability analysis. Schedulability analysis is usually the next step and assumes known WCET values.

Table 2.4: Comparison of the response times of the task approach with blocked DMA (C_1 and R_1) and the WCET method with spread DMA access (C_2 and R_2).

τ	T (μs)	C_1 (μs)	R_1 (μs)	C_2 (μs)	R_2 (μs)
τ_{vga}	17	4.8	4.8	–	–
τ_{lift}	500	162.7	229.9	201.5	201.5
τ_{kfl}	3000	782.2	1999.4	1039.1	1845.1

The computation time values of Table 2.3 are calculated by multiplying the WCET estimates of τ_{lift} and τ_{kfl} from Table 2.2 with the clock period of 12.5 ns. The values for those two tasks are the basis for the schedulability test. The utilization-based test results in 74.94%. Only two tasks are taken into account and hence this result is less than $2(2^{1/2} - 1) = 82.84\%$. Even though this test is positive, we also investigate the response time analysis.

The response time analysis for the WCET estimates of the system with the VGA results in tighter response times than in the analysis using the task approach of Section 2.4.2. R_{lift} is the same as $C_{lift} = 201.5$ μs and R_{kfl} calculates to 1845.1 μs.

2.4.4 Discussion

Table 2.4 shows the results of the evaluation for both analyses: column 3 and 4 (C_1 and R_1) for the DMA task approach (showing the better solution with blocked DMA mode) and column 5 and 6 (C_2 and R_2) for the DMA-WCET approach. We can see that both τ_{lift} and τ_{kfl} have a higher WCET (C_2) in the DMA-WCET approach. As we do not have to include the VGA task in the response time analysis, the response time R_2 is less for both tasks despite the fact that C_2 is higher.

The result shows that the inclusion of the DMA access into the WCET analysis provides tighter worst-case response times than considering the DMA as an additional task. The difference can be explained as follows: most instructions on JOP do not access the main memory; they use the internal stack cache for data. The pipeline is filled from the instruction cache most of the time. In the task approach, all instructions are *blocked* by the VGA task. The WCET analysis is more exact as it delays only those instructions, which do actually access the main memory and the cache load events.

To validate our calculations and measurements we run all mentioned task sets on real hardware and the JopVga system respectively. No deadline violation of any task could ever be observed when performing these experiments.

Table 2.5: Comparison of the system performances in iterations/s.

App	JOP only	blocked VGA	spread VGA
Kfl	12163	11562	11628
Lift	9643	9194	9356

2.4.5 Benchmarks

Although the solution is aimed at RTSs, i.e. a time predictable system, the average case performance is still interesting. The system under test is the JopVga system. The FPGA platform enables us to compare the performance of the system with an enabled VGA controller versus one with a disabled VGA controller. The results are achieved by running real applications in real hardware. For our measurements, we use the embedded Java benchmark suite JavaBenchEmbedded as described in [9]. The result is iterations per second, which means a higher value illustrates a better performance. In Table 2.5, the benchmark results are shown.

The `Kfl` application is slowed down just by 4.4% due to the memory contention with the spread VGA memory access scheme. The WCET estimates of the same task resulted in an increase of up to 33% (see Table 2.2). Another benchmark called `Lift` experiences an even smaller slowdown of 3.0% due to the contention with the VGA task. The WCET estimates of the same task resulted in an increase of 24%.

To recapitulate, although we use about one third of the memory bandwidth for the DMA unit both applications suffer less than that one third in their execution time. This result is a promising indication that the memory system is not the bottleneck of the single CPU with the VGA. There is enough headroom for further devices. The result is promising for our further plans on a CMP version with several JOPs sharing a single memory.

2.5 Conclusion and Future Work

In this paper, we have analyzed a system with a processor and a DMA unit with respect to WCET and schedulability. We have found two ways to model the influence of the DMA unit to the application tasks: 1.) the DMA access as an additional real-time task and 2.) include the DMA memory access in the WCET analysis of the individual application tasks. We found that the second approach results in a tighter estimation and enables more processing resources for the application.

We will investigate the possibility to include a known memory access pattern

into the WCET analysis of the cache loading to find tighter WCET estimates. On a cache load, we can guarantee that only a maximum number of memory loads can conflict with the DMA unit. The next step is the application of our findings to a system with several CPUs – the CMP JOP system. In that case, the memory access pattern is less predictable. Therefore, the shared resource can only be modeled by the WCET approach.

Bibliography

[1] ARM. ARM11 MPCore Processor, technical reference manual. Available at: http://www.arm.com, August 2006.

[2] Pavel Atanassov and Peter Puschner. Impact of dram refresh on the execution time of real-time tasks. In *Proc. IEEE International Workshop on Application of Reliable Computing and Communication*, pages 29–34, Dec. 2001.

[3] Neil C. Audsley, Alan Burns, Robert I. Davis, Ken Tindell, and Andy J. Wellings. Fixed priority pre-emptive scheduling: An historical perspective. *Real-Time Systems*, 8(2-3):173–198, 1995.

[4] Alan Burns and Andrew J. Wellings. *Real-time systems and programming languages: Ada 95, real-time Java, and real-time POSIX*. International computer science series. Addison-Wesley, third edition, 2001. Revised edition of *Real-time systems and their programming languages*, 1990.

[5] SPARC International Inc. *The SPARC Architecture Manual: Version 8*. Prentice Hall, Englewood Cliffs, New Jersey 07632, 1992.

[6] Mathai Joseph and Paritosh K. Pandya. Finding response times in a real-time system. *Comput. J*, 29(5):390–395, 1986.

[7] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973.

[8] Peter Puschner and Alan Burns. A review of worst-case execution-time analysis. *Journal of Real-Time Systems*, 18(2/3):115–128, May 2000.

[9] Martin Schoeberl. Evaluation of a Java processor. In *Tagungsband Austrochip 2005*, pages 127–134, Vienna, Austria, October 2005.

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

[11] Martin Schoeberl. SimpCon - a simple and efficient SoC interconnect. Available at: http://www.opencores.org/, 2007.

[12] Martin Schoeberl and Rasmus Pedersen. WCET analysis for a Java processor. In *JTRES '06: Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems*, pages 202–211, New York, NY, USA, 2006. ACM Press.

3

Towards a Java Multiprocessor

Christof Pitter and Martin Schoeberl

Institute of Computer Engineering,
Vienna University of Technology,
Austria

Proceedings of the 5th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2007). Vienna, Austria, 2007, pages 144-151.

Abstract

This paper describes the first steps towards a Java multiprocessor system on a single chip for embedded systems. The chip multiprocessing (CMP) system consists of a homogeneous set of processing elements and a shared memory. Each processor core is based on the Java Optimized Processor (JOP). A major challenge in CMP is the shared memory access of multiple CPUs. The proposed memory arbiter resolves possible emerging conflicts of parallel accesses to the shared memory using a fixed priority scheme. Furthermore, the paper describes the boot-up of the CMP. We verify the proposed CMP architecture by the implementation of the prototype called JopCMP. JopCMP consists of multiple JOPs and a shared memory. Finally yet importantly, the first implementation of the CMP composed of two/three JOPs in an FPGA enables us to present a comparison of the performance between a single-core JOP and the CMP version by running real applications.

Keywords: Multiprocessor, Java, Shared Memory

3.1 Introduction

Modern applications demand ever-increasing computation power. They act as the main drivers for the semiconductor industry. For over 35 years, the speed of transistors has become faster and the frequency of the clock rate accordingly. Additionally the number of transistors on an integrated circuit for a given cost doubles every 24 months, as described by Moore's Law [20]. The availability of more transistors has been used by introducing the instruction-level parallelism (ILP) approach, which was the primary processor design objective between the mid-1980s and the start of the 21st century. According to [9], we are reaching the limits of exploiting ILP efficiently. Unfortunately, semiconductor technology has also run across limitations in recent years because of the theoretical limits informed by the theory of physics, e.g., electrical signals cannot travel faster than the speed of light. As a result, the frequency that used to increase exponentially has leveled off [17].

To sustain the rapid growth of computation power new system architecture advancements have to be made. According to [9] the future direction of computer systems is chip multiprocessing (CMP). Such a system combines two or more processing elements and a sophisticated communication network on a single chip. A major advantage of this approach is that progress in computation power does not come along with an increase of the hardware complexity of the single processors. However, the shared components like memory and I/O of a multiprocessor system produce new challenges that have to be addressed.

Real-time programs are naturally multi-threaded and a good candidate for on-chip multiprocessors with shared memory. The Java virtual machine (JVM) thread model supports threads that share the main memory. Therefore, a multiprocessor JVM in hardware is a viable option. The basis for the CMP architecture has been set with the Java Optimized Processor (JOP) [23, 24, 25]. JOP is the implementation of a JVM in hardware. The used application model is described in [22, 28] and is based on the Ravenscar Ada profile [7]. In order to generate a small and predictable processor, several advanced and resource-consuming features (such as instruction folding or branch prediction) were omitted from the design of JOP. The resulting low resource usage of JOP makes it possible to integrate more than one processor in a low-cost field programmable gate array (FPGA).

In this paper, we propose a CMP architecture consisting of a number of JOPs and a shared memory. The shared memory is uniformly accessible

by the homogeneous processing cores. An arbitration unit takes care of conflicts due to parallel memory requests. Furthermore, we describe the implementation of the caching and synchronization mechanisms. Additionally we present JopCMP, the first prototype of the CMP with JOP cores. JopCMP is composed of multiple JOP cores, integrated in an FPGA, and an external memory. A novel memory arbiter controls the memory access of the various JOPs to the shared memory. It resolves possible emerging conflicts of parallel accesses to the shared memory. In comparison to existent memory arbiters of system-on-chip (SoC) busses (e.g. AMBA [3]), the proposed arbitration process is performed in the same cycle as the request happens. This increases the bandwidth and eases the time predictability of each memory access. Therefore, the implementation of the JopCMP represents a first step towards a time predictable multiprocessor. The ultimate goal of our research work is a multiprocessor for safety-critical applications. Moreover, an acceptable performance compared with mainstream non real-time Java systems is an objective.

The rest of the paper is structured as follows. Section 3.2 presents related work. In Section 3.3, we describe the proposed CMP architecture and go into details of the memory model and caching. Additionally the issue of synchronization is examined. Section 3.4 describes the first implementation of the JopCMP system, including the boot-up sequence and the shared memory access management of the CMP. Section 3.5 presents experiments with the JopCMP prototype. We compare the performance of the CMP against a single processor. Finally, Section 3.6 concludes the paper and provides guidelines for future work.

3.2 Related Work

In this paper, we argue that the replication of a simple pipeline on a chip is a more effective use of transistors than the implementation of super-scalar architectures. The following two subsections are about the progress made in CMP.

3.2.1 Mainstream Multiprocessors

Due to the power wall [9], the trend towards CMP can be seen in mainstream processors. Currently, three quite different architectures are state-of-the-art:

1. Multi-core versions of super-scalar architectures (Intel/AMD)
2. Multi-core chip with simple RISC processors (Sun Niagara)

3. The CELL architecture

Mainstream desktop processors from Intel and AMD include two or four super-scalar, simultaneous multithreading processors, which are just replications of the original, complex cores. Sun took a completely different approach with its Niagara T1 [15]. The T1 contains eight processor cores. Each core consists of a simple six-stage, single-issue pipeline similar to the original five-stage RISC pipeline. The additional pipeline stage adds fine-grained multithreading. The first version of the chip contains just a single floating-point unit that is shared by all eight processors. The design is targeted to server workloads.

The Cell multiprocessor [10, 13, 14] is an example of a heterogeneous multiprocessor system. The Cell contains, besides a PowerPC microprocessor, eight synergistic processors (SP). The bus is clocked at half of the processor speed (1.6 GHz). It is organized in four rings each 128 bit wide, two in each direction. A maximum of three non-overlapping transfers on each ring are possible. The SPs contain on-chip memory instead of a cache. All memory management, e.g. transfer between SPs or between on-chip memory and main memory, is under program control, which makes programming a difficult task.

All of the above CMP architectures are optimized for average case performance and not for worst-case execution time (WCET). The complex hardware complicates the timing analysis. Our overall goal is a homogeneous CMP design that is analyzable with respect to WCET. The paper presents the first step towards this.

3.2.2 Embedded Multiprocessors

In the embedded system domain, two different CMP architectures are distinguished:

1. heterogeneous multiprocessors
2. homogeneous multiprocessors

Multiprocessors with a heterogeneous architecture combine a core CPU for controlling and communication tasks and additional digital signaling processing elements, interface processors or mobile multimedia processing units. These units are connected together using multi-level buses or switches. Some functional units may have their own individual memories along with shared memory structures. They are often tailored for specific applications. Some examples of heterogeneous multiprocessors include the Nomadik [1] from

ST designed for mobile multimedia applications, the Nexperia PNX-8500 [8] from Philips aimed at digital video entertainment systems, or the OMAP family [19] from TI designed to support 2.5G and 3G wireless applications.

Gaisler Research AB designed and implemented a homogeneous multiprocessor system. It consists of a centralized shared memory and up to four LEON processor cores that are based on the SPARC V8 architecture [11]. This embedded system is available as a synthesizable VHDL model. Therefore, it is well suited for SoC designs. We could not find any literature concerning WCET analysis regarding the multiprocessor.

Another example is the ARM11 MPCore [4]. It introduces a pre-integrated symmetric multiprocessor consisting of up to four ARM11 microarchitecture processors. The 8-stage pipeline architecture, independent data and instruction caches, and a memory management unit for the shared memory make a timing analysis difficult.

The two leaders of the FPGA market Altera and Xilinx both provide software tools and intellectual property (IP) processors to design CMP systems [2, 5]. The Nios II CPUs depict the processing units of the multiprocessor architecture from Altera. It is easy to create a CMP architecture using the GUI interface of the System-on-a-programmable-chip (SOPC) builder, a tool of the Altera Quartus II design suite. Nevertheless, the dependence on specific IP cores is unavoidable when designing such a system.

In this paper, we concentrate on homogeneous multiprocessors consisting of two or more similar CPUs sharing a main memory. Even though much research has been done on multiprocessors, the timing analysis of the systems has so far been disregarded.

3.3 CMP Architecture

According to [12, 30], two different possibilities of a tightly coupled multiprocessor system exist (see Figure 3.1). The core of the *Shared Memory Model* is a global physical memory equally accessible to all processors. These systems enable simple data sharing through a uniform mechanism of reading and writing shared structures in the common memory. This multiprocessor model is called symmetric (shared-memory) multiprocessor (SMP) because all processors have symmetric access to the shared memory. This architecture is known as UMA (uniform memory access).

In contrast, the *Distributed Shared Memory Model* implements a physically distributed-memory system (often called a multicomputer). It consists of multiple independent processing nodes with local memory modules, connected by a general interconnection network like switches or meshes. Com-

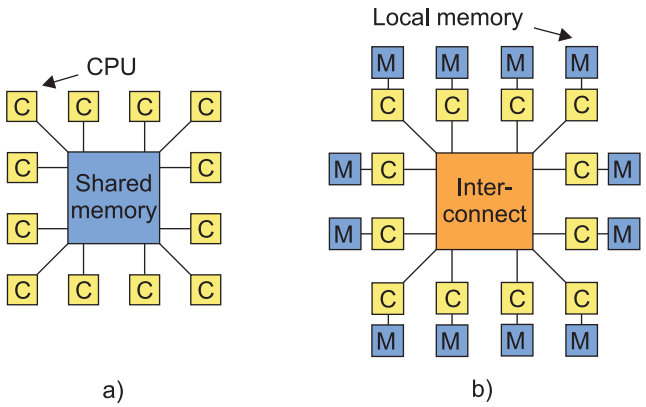

Figure 3.1: CMP Memory Models: a) Shared memory model, b) Distributed shared memory model.

munication between processes residing on different nodes involves a message-passing model that requires extensive additional data exchange. The messages have to take care of data distribution across the system and manage the communication. This architecture is called non-uniform memory access (NUMA) because the time for a memory access depends on the location of the memory word.

For time-predictability, the NUMA architecture is less appropriate. Each CPU can access its own local memory very fast but the time to access a data word of another memory in the distributed system takes much longer. Consequently, the memory access times can vary extensively. In the SMP architecture, each memory request takes the same time independent of the CPU. Additionally, no message-passing communication system that could limit the bandwidth of the interconnection is needed. Therefore, the SMP architecture is the best choice for analyzing tight bounds of a memory access.

3.3.1 JVM Memory Model

The JVM defines various runtime data areas that are used during the execution of a program [18]. Some of these data areas are shared between threads, while others exist separately for each thread.

Stack: Each thread has a private stack area that is created at the same time as the thread containing a frame with return information for a method, a local variable area, and the operand stack. Local variables and the operand stack are accessed as frequently as registers in a standard processor. According to [23], a Java processor should provide some caching mechanism of this data area.

Heap: The heap is the data area where all objects and arrays are allocated. The heap is shared among all threads. A garbage collector (GC) reclaims storage for objects. The GC of the proposed CMP runs on a designated processor.

Method area: The method area is shared among all threads. It contains static class information such as field and method data, the code for the methods and the constant pool. The constant pool is a per-class table, containing various kinds of constants such as numeric values or method and field references. The constant pool is similar to a symbol table. Part of this area, the code for the methods, is very frequently accessed (during instruction fetch) and therefore is a good candidate for caching.

The specification of the JVM mandates that the heap and the method area be shared among the threads. This memory model favors the shared global memory model as the adequate solution for the multiprocessor using JOP cores. One single shared address space is accessible from all processors.

3.3.2 CMP Cache Memory

Many SMP architectures support caching of private and shared data [9]. Since many memory requests can be served by the caches, the number of accesses to the main memory decreases. Nevertheless, caching of shared data may result in cache coherence problems in a multiprocessor environment. Assume shared data are cached and each processor has a copy of a data word in its own cache. If one processor changes the data word, the other processor will not notice that the data word has become invalid. Hence, two different CPUs could see different values in their caches of exactly the same memory location. There exist cache coherence techniques, e.g. snooping protocols or directory based mechanisms, to secure that no cache coherence problems can arise. Nevertheless, these cache coherence mechanisms require processing overhead and latencies.

In the JVM, each thread has its own JVM stack. The thread very often accesses this memory area. Therefore, in a CMP system it is cached in a so-called stack cache of the corresponding CPU. No cache conflicts can occur because this data is private for each thread.

The method area of the JVM is shared among all the threads. Nevertheless, it is cached in the method cache [23] of each CPU. This area is a read-only area. Consequently, no cache coherence conflicts can occur.

The heap of the JVM is the memory region that is not cached, as data cache WCET analysis is problematic. The heap contains all objects that are

created by a running Java application. Therefore, the heap represents the memory area used for communication between the multiple CPUs of a CMP.

To summarize, the CMP architecture operates without any cache coherence mechanisms, as cache coherence conflicts are avoided by our CMP architecture.

3.3.3 Synchronization

Synchronization is an essential part of a multiprocessor system with shared memory. The CMP synchronization support has two important responsibilities:

- Protect access to shared objects
- Avoid priority inversion

The first responsibility of synchronization is to protect access to shared objects. As already mentioned in Section 3.3.2, the heap inside the JVM contains the objects that are shared between threads. If multiple threads need to access the same objects or class variables concurrently, their access to the data must be properly managed. Otherwise, the program will have unpredictable behavior. Therefore, the JVM associates a lock with each object and class. Only one thread can hold the lock at any time. When the thread no longer needs the lock, it returns it to the JVM. If another thread has requested the same lock, the JVM passes the lock to that thread. The traditional approach implementing such objects centers around the use of critical sections: only one process operates on the object at a given time.

JOP, the implementation of the JVM in hardware, solves this problem by a straightforward approach. Suppose several threads are executed depending on the priority of each thread. If one thread accesses a shared object and enters a so-called critical section, it will have to hold exclusive ownership. Therefore, JOP provides two software constructs called `monitorenter` and `monitorexit`. In hardware, the synchronization mechanism is implemented by disabling and enabling interrupts at the entrance and exit of the monitor. This simple form of synchronization disallows any context switches until the thread leaves the critical section. Even though this is a viable option for single processor systems, the price is high for this approach. Consequently, the processor cannot interleave programs in different critical sections. This may lead to a degradation of the execution performance. Therefore, a couple of constructs to implement critical sections in hardware [29] exist, e.g. the atomic Read-Modify-Write operation based on a test and set instruction.

Especially in the CMP, the synchronization solution with deactivation and activation of interrupts does not suffice. Mutual exclusion cannot be guaranteed because we cannot prevent other CPUs from running in parallel. Threads of different processors may simultaneously access the shared object. Therefore, a synchronization unit is essential for the CMP. If one processor wants to access a shared object, it will have to request a lock. Either the CPU receives the lock or the request is rejected because another CPU is using the object. With the grant of the lock, the processor resides in the critical section and cannot be interrupted. After the processor does not access the shared object anymore, it will release the lock immediately. Another CPU, which is waiting for the lock, will get the permission to access the memory object. The first implementation will make only one global lock available for the heap. Later we will investigate to use multiple locks. The access of each object of the heap will be controlled by its corresponding lock. Though the introduction of multiple locks will increase concurrency, it may induce the risk of deadlock see [32]. A deadlock is a condition in which two or more threads cannot advance because they request a lock that is held by another thread.

The second responsibility of the CMP synchronization support is the avoidance of priority inversion. If a low priority thread holds a shared resource, a high priority thread cannot access the same resource. Consequently, the high priority thread cannot interrupt the low priority thread until the low priority thread releases the lock. Assume one or more medium priority threads preempt the low-priority task. Consequently, the high priority thread can be delayed indefinitely, because the medium priority jobs will take precedence over the low priority task and the high priority task as well. An unbounded priority inversion and in fact no time predictability would be the consequence. Solutions for priority inversion for single processor systems include the priority ceiling protocol or the priority inheritance protocol [16]. The work of Wang et al. [31] presents two algorithms of priority inheritance locks for multiprocessor real-time systems. If a low priority processor locks a high priority processor, the low priority processor will inherit the highest priority of the waiting processors. Hence, no medium priority processors get the chance of interrupting the low priority processor and consequently lock the high priority processor for an indefinite time. The time, the high priority CPU has to wait is bounded.

Even though we know that hardware assisted transactional memory models could be a promising approach for our multiprocessor, we concentrate on memory locks for synchronization in this paper. Future work will investigate the use of transactional memory for the CMP.

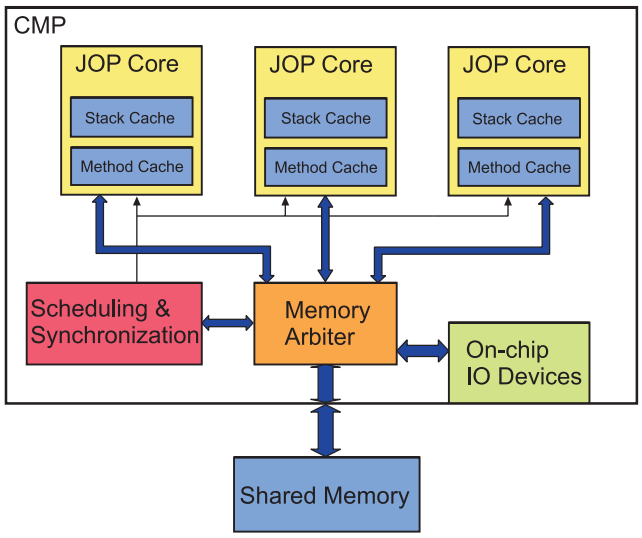

Figure 3.2: Time predictable CMP architecture.

3.3.4 CMP using JOPs

The proposed chip multiprocessing system (see Figure 3.2) uses the SMP architecture. It consists of a shared memory that is uniformly accessible by a number of homogeneous processors. The JOP cores are connected to the shared memory via a memory arbiter that is further explained in Section 3.4.2. This arbiter has to control the memory access of the various JOPs to the shared memory. It resolves possible emerging conflicts of parallel accesses to the shared memory dependent on the priority of the CPU that requested access. Each CPU is assigned a unique priority in the system.

Each core contains a local method and stack cache. Furthermore, the depicted CMP architecture shows a scheduling and synchronization unit. The preemptive scheduler is assigned to distribute the real-time tasks among the processors. Synchronization has the responsibility to coordinate access to the shared objects by a mutual exclusion mechanism. Due to different priorities of the multiple processors, a low priority processor shall not be able to block a high priority processor indefinitely. Therefore, this priority inversion problem is solved by using priority inheritance locks for shared objects.

On-chip IO devices, such as a controller for real-time Ethernet or a real-time field bus, may be mapped to shared memory addresses and are connected via the memory arbiter.

3.4 Implementation

In the following section, we describe our implementation of a CMP system based on JOP. Multiple cores are connected via a low-latency arbiter to a shared main memory. We subsequently refer to the prototype as JopCMP.

3.4.1 JopCMP Boot-up Sequence

One interesting issue for a CMP system is the question how the startup or boot-up is performed. Before we explain the CMP solution, we need an understanding of the boot-up sequence of JOP in an FPGA. On power-up, the FPGA starts the configuration state machine to read the FPGA configuration data either from a Flash or via a download cable (for development). When the configuration has finished an internal reset is generated. After that reset, microcode instructions are executed starting from address 0. At this stage, we have not yet loaded any application program (Java bytecode). The first sequence in microcode performs this task. The Java application can be loaded from an external Flash or via a serial line (or USB port) from a PC. The microcode assembly configured the mode. Consequently, the Java application is loaded into the main memory. To simplify the startup code we perform the rest of the startup in Java itself, even when some parts of the JVM are not yet setup.

In the next step, a minimal stack frame is generated and the special method `Startup.boot()` is invoked. From now on JOP runs in Java mode. The method `boot()` performs the following steps:

- Send a greeting message to *stdout*
- Detect the size of the main memory
- Initialize the data structures for the garbage collector
- Initialize `java.lang.System`
- Print out JOP's version number, detected clock speed, and memory size
- Invoke the static class initializers in a predefined order
- Invoke the `main` method of the application class

The boot-up process is the same for all processors until the generation of the internal reset and the execution of the first microcode instruction. From that point on, we have to take care that *only one* processor performs the initialization steps.

All processors in the CMP are functionally identical. Only one processor is designated to boot-up and initialize the whole system. Therefore, it is necessary to distinguish between the different CPUs. We assign a unique CPU identity number (CPU ID) to each processor. Only processor CPU0 is designated to do all the boot-up and initialization work. The other CPUs have to wait until CPU0 completes the boot-up and initialization sequence. At the beginning of the booting sequence, CPU0 loads the Java application. Meanwhile, all other processors are waiting for an *initialization finished* signal of CPU0. This busy wait is performed in microcode. When the other CPUs are enabled, they will run the same sequence as CPU0. Therefore, the initialization steps are guarded by a condition on the CPU ID.

In our current prototype, we let all additional CPUs also invoke the main method of the application. This is a shortcut for a simple evaluation of the system[1]. In a future version, the additional CPUs will invoke a system method to be integrated into the normal scheduling system.

3.4.2 Memory Arbiter

The general structure of a system-on-a-chip (SoC) architecture combines SoC modules with a data exchange interconnection. A CMP is formed of several processors connected to a shared memory module. Some sort of SoC interconnection has to enable data exchange between the SoC modules.

One major design decision regards the type of interconnection that is used. Our system requires a fast, point-to-point connection between each CPU and the memory. We still have to keep in mind that a parallel access to the memory is not possible and would lead to a conflict. Therefore, some kind of synchronization mechanism has to take care of this problem. Additionally, each memory access should be as fast as possible and time predictable.

Communication on SoC is an active research area with focus on network-on-chip (NoC) [6]. NoC is not the appropriate architecture for JopCMP. First, the interconnection of JopCMP does not have to be a network because our system consists of a couple of masters (JOP) and only one slave (shared memory). The communication between the masters takes place using shared memory and not packet oriented messages. Consequently, there is no use of a network connecting all modules with each other. A NoC usually introduces long latencies that cannot be tolerated for our memory system (we avoid data caches to achieve better temporal predictability). Furthermore, the network contentions within the routers may cause varying latencies [6] that make WCET analysis more complex and conservative.

[1] In the main method we execute different applications based on the CPU ID.

For the JopCMP the simple SoC interconnect (SimpCon) [26] is used to connect the SoC modules. This synchronous on-chip interconnection is intended for read and write transfers via point-to-point connections. The master starts the transaction. The read or write request, as well as the address and data of the slave, is valid for one cycle. If the slave needs the address or data longer than a cycle, it has to store it in a register. Consequently, the master can continue to execute its program until the result for a read is needed. The slave informs the master by a signal called *rdy_cnt* when the requested data will be available. In addition, this signal also serves as an early notification of the completion of the data access. This mechanism allows the master to send a new request before the former has been completed. This form of pipelining permits fast data transfer.

SimpCon is well suited for on-chip point-to-point connections. Nevertheless, the specification does not support synchronization of connecting multiple masters to a shared slave. Therefore, we introduce a central arbiter. The SimpCon interface can be used as interconnect between the masters and the arbiter and the arbiter and the slave. In this case, the arbiter acts as slave for each JOP and as master for the shared memory. The arbitration and signal routing is completely transparent for the masters and the slaves. No bus request (as e.g., in AMBA [3]) phase has to precede the actual bus transfer. Without contention, the arbiter introduces zero cycle latency for a transaction.

The arbiter is designed for the SimpCon interface. In the JopCMP architecture, it plays the important role of controlling the memory access of the various CPUs to the shared memory. It resolves possible emerging conflicts of parallel accesses to the shared memory. The implemented arbitration scheme uses fixed priority. As already mentioned in Section 3.4.1, each CPU is assigned a unique ID. This CPU ID establishes the priority for each CPU. The CPU with the lowest CPU ID has top priority. The memory arbiter dissolves any simultaneous memory accesses by determining an order of precedence.

Zero cycle latency is the design objective of the memory arbiter. Assume that two processors want to access the shared memory at the same clock cycle. Consequently, the arbiter has to decide which CPU is granted the request. This arbitration process is performed in the same cycle as the request happens. Consequently, the time to access the memory is reduced and the bandwidth increases extensively. Whether this form of zero cycle arbitration would scale to a large number of processors is an open question and the topic of future work.

In [21] two different approaches of analyzing the timing behavior are compared with respect to schedulability analysis and WCET analysis. The system consists of a direct memory access (DMA) and JOP, both accessing a shared memory using the proposed arbiter. Pitter and Schoeberl treat the

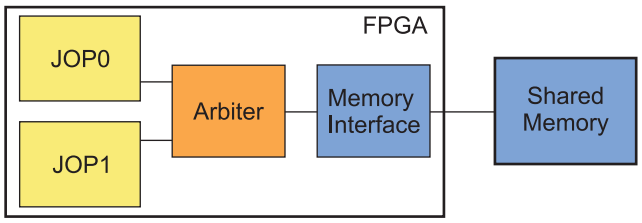

Figure 3.3: Dual-core JopCMP system.

DMA task as the top priority real-time task with a regular memory access pattern. The outcome of this paper shows that this arbiter allows WCET analysis by modeling the DMA memory access into the WCET values of the tasks running on JOP. Each memory access is modeled with the maximum blocking time: three cycles of the DMA unit. In the JopCMP system, the DMA unit is replaced by another JOP. Therefore, the blocking times can be much higher; for example, because of a cache load. We will investigate different approaches to solve this problem in our future work in order to provide a time predictable shared memory access scheme.

3.4.3 JopCMP Prototype

The first prototype of the proposed CMP consists of multiple JOPs [23], the proposed memory arbiter, a memory interface and an SRAM memory. Both CPUs, the memory arbiter and the memory interface are implemented on an Altera Cyclone FPGA. SimpCon [26], the SoC bus, connects the JOPs with the arbiter. The arbiter is connected via SimpCon to the memory interface. As illustrated in Figure 3.3 the memory interface connects the external memory to the FPGA. This 1 MByte, 32-bit external SRAM device represents the shared memory of the JopCMP.

The dual-core JopCMP runs at a clock frequency of 80 MHz, resulting in a period of 12.5 ns per cycle. The SRAM-based shared memory has an access time of 15 ns per 32-bit word. Hence, every memory access needs a minimum of 2 cycles.

The JopCMP system features a couple of different I/O interfaces, such as a serial interface, a USB interface and several I/O pins of the FPGA board. Usually, the application pretends either one CPU owns the exclusive access of the I/O ports, or more processors share I/O interfaces. The benchmarks used for prototyping our JopCMP do not require I/O access of all CPUs. Consequently, the top priority JOP holds the sole access to the I/O world.

To summarize, the CPUs share the main memory but only one JOP is able to communicate with external devices. Despite this, the possibility of sharing the I/O can be implemented using an arbitration unit for the I/O access.

Table 3.1: Benchmark results in iterations/s for a single-core JOP at different clock frequencies.

App	75 MHz	80 MHz	100 MHz
Kfl	13768	14667	18347
Lift	12318	13138	16425

3.5 Experiments

In this section, we provide performance measurements obtained on real hardware. The FPGA platform enables us to compare the performance between a single-processor system and the JopCMP system composed of multiple JOP cores. The measured results are achieved by running real applications in real hardware. We use the embedded Java benchmark suite called JavaBenchEmbedded, as described in [23], for our experiments.

We make use of two real-world examples with industrial background herein after referred to as `Lift` and `Kfl` to represent two independent tasks. `Lift` is a lift controller used in an automation factory. `Kfl` is one node of a distributed real-time system to tilt the line over a train for easier loading and unloading of goods wagons. Both applications consist of a main loop that is executed periodically. Only one task executes on a single CPU at a given time in our experiments. We measure the execution in iterations per second, which means that a higher value implies a better performance.

The baseline for the comparison is the performance of a single-core. Table 3.1 shows the performance numbers at different clock frequencies for the single-core.

3.5.1 Comparison at CMP Frequency

In the first experiments, we compare JopCMP versions with two and three cores against a single-core at the same clock frequency. Those measurements provide insights how limiting the memory bandwidth is.

We start with a comparison of the performance measurements of a dual-core JopCMP against the performance of a traditional single-core JOP. Both systems run at the same clock frequency of 80 MHz. Running only one task on JOP results in 14667 iterations/s for `Kfl`. Task `Lift` achieves 13138 iterations/s (as shown in Table 3.1).

Table 3.2 shows the benchmark results running two tasks simultaneously on a dual-core JopCMP system at a frequency of 80 MHz. First, we measured the execution by running two `Lift` tasks, one on each CPU. The speedup of the overall system is calculated by dividing the sum of the performance of

Table 3.2: Benchmark results in iterations/s of a dual JopCMP system at a clock frequency of 80 MHz.

Processor	JOP0	JOP1	JOP0	JOP1	JOP0	JOP1
Appl.	Lift	Lift	Kfl	Lift	Lift	Kfl
Result	12951	12951	14435	12374	12574	14296

both tasks by the performance of the task running on the single processor. This calculates to

$$Speedup_{dualcore} = \frac{12951 + 12951}{13138} \approx 1.97. \tag{3.1}$$

Each task running on a different CPU executes 12951 iterations/s. The result indicates that the two tasks do not access the memory very often in parallel. Otherwise, the memory contention would reflect a difference between the results. Additionally, each result does not diverge greatly from the result of the single processor. The outcome of the experiment is that the JopCMP is 1.97 times faster than the single JOP for `Lift` both running at 80 MHz.

In the next experiment, the high priority JOP0 executes the `Kfl` task and JOP1 runs the task `Lift`. The results show that the task running on CPU0 is slowed down just by 1.6% comparing to the execution of the single task on the single JOP. Furthermore, the low priority CPU1 executes the main loop of `Lift` 12374 times per second. This task is slowed down by 5.8% due to the memory contention with the second JOP.

We exchange the tasks between the CPUs for a further experiment. Task `Lift` experiences a decrease of 4.3% and task `Kfl` decreases by 2.5%. The results of Table 3.2 indicate that task `Lift` experiences a larger slowdown than task `Kfl`, irrespective whether it is executed on CPU0 or on CPU1.

In conclusion, the processing performance greatly increases due to the use of two JOPs in the JopCMP system. The comparison of the measurements between the dual-core and the single JOP shows, that each single task in the JopCMP system experiences only a small slowdown in performance as measured by iterations per second. The cause is the access contention to the shared memory.

The maximum frequency of the tri-core JopCMP is 75 MHz. Therefore, we measure the speedup of the CMP system compared to JOP running at a clock frequency of 75 MHz. The `Lift` achieves 12318 iterations/s on JOP. Table 3.3 depicts the results of the CMP. Equation 3.2 presents the calculation of the speedup. Only 7% slowdown relative to the theoretical maximum speedup

Table 3.3: Benchmark results in iterations/s of a tri-core JOP system at a clock frequency of 75 MHz.

Processor	JOP0	JOP1	JOP2
Appl.	Lift	Lift	Lift
Result	11736	11538	11260

of 3 indicates either a large headroom in the memory interface or that the benchmark is actually small enough to fit into the method cache.

$$Speedup_{tricore} = \frac{11736 + 11538 + 11260}{12318} \approx 2.80. \tag{3.2}$$

3.5.2 Comparison Against a Single Core

As we see in Table 3.4 the maximum clock frequency depends on the number of cores. For a fair comparison of the full system speedup, we have to take the reduced clock frequency for JopCMP into account. Therefore, we compare the dual- and tri-core speedup against a single-core where all designs are clocked at their maximum frequency.

Equation 3.3 shows that the real speedup of a dual-core version of JOP, measured with the benchmark `Lift`, against a 100 MHz single-core is about 58%.

$$Speedup_{dualcore} = \frac{12951 + 12951}{16425} \approx 1.58. \tag{3.3}$$

In the last experiment, we compare the performance of JOP with a tri-core JopCMP system both running at their maximum clock frequencies. JOP executes the `Lift` 16425 iterations/s at a clock speed of 100 MHz (see Table 3.1). Table 3.3 shows the benchmark results running the `Lift` task simultaneously on a tri-core JopCMP at the maximum frequency of 75 MHz. The speedup of the tri-core system calculates to

$$Speedup_{tricore} = \frac{11736 + 11538 + 11260}{16425} \approx 2.10. \tag{3.4}$$

Even though the CMP runs at a reduced clock frequency of 75 MHz, the tri-core JopCMP provides a 2.1 times better overall performance compared to the traditional single-core JOP at 100 MHz.

Table 3.4: Comparison of resource consumption between JOP and the JopCMP versions.

Processor	Resources (LC)	Memory (KB)	fmax (MHz)
JOP	2815	7.63	100
Dual-core	5540	15.62	80
Tri-core	8219	23.42	75

To recapitulate, the performance measurements provide a promising indication that the memory system is not the bottleneck of the JopCMP. It seems that there is enough headroom for further devices. As future research, we will evaluate if the single cycle arbitration is worth the reduced clock frequency or if a pipeline stage that introduces an additional cycle latency is a better solution. Furthermore, we will evaluate the influence of different arbitration schemes on the WCET of the individual tasks.

3.5.3 Resource Consumption

Finally, Table 3.4 shows the resource consumptions and the maximum frequencies of a typical version of JOP and the JopCMP versions implemented in an Altera EP1C12 FPGA. We can see the differences in the resource consumptions by looking at the two basic structures of an FPGA, Logic Cells (LC) and embedded memory blocks. The dual-core consumes roughly twice as much of LCs and memory blocks as JOP. Nevertheless, only 46% of LCs and 52% of memory blocks of the low-cost Cyclone FPGA are used. It runs at a clock frequency of 80 MHz. The tri-core JopCMP runs at a maximum clock frequency of 75 MHz. It requires 68% of the total LCs and 78% of the total memory blocks available on the FPGA. Therefore, this low-cost FPGA does not provide enough space to integrate additional JOPs. In summary, the experimental results and the resource consumptions of the JopCMP prototype are encouraging. We can observe a slight degradation of the maximum clock frequency when using more and more JOP cores due to the increasing combinational logic of the arbiter. Nevertheless, we will further evaluate CMP implementations with additional JOPs sharing a single memory in future work.

3.6 Conclusion

In this paper, we introduced a CMP architecture consisting of a number of JOP cores and a shared memory. We demonstrated the effectiveness of the

architecture by the first prototype of a Java multiprocessor called JopCMP. Correct executions of real application tasks verified the implementation with multiple JOP cores. Experiments showed the correct functioning of the implemented boot-up and the memory arbiter. Measurements made a comparison between a single JOP and the JopCMP possible, and endorsed further pursuit of the CMP approach. Future research will show how the CMP system will behave when integrating more than three processing cores in an FPGA.

We have to admit, that the lack of the implementation of a synchronization mechanism prevents more advanced experiments; not any with objects (e.g. producer-consumer problem), shared by multiple processors, could be carried out. This implementation of the combined synchronization mechanism and priority inversion avoidance defines our future work. Additionally, we will investigate real-time multiprocessor scheduling for the proposed CMP.

Furthermore, we will integrate the maximum latency due to collisions on the memory into the WCET tool [27] as we have done for the simpler case of DMA devices [21]. We will investigate the influence on the WCET with different arbitration schemes.

Acknowledgement

The TPCM-project received support from the Austrian FIT-IT SoC initiative, funded by the Austrian Ministry for Traffic, Innovation and Technology (BMVIT) and managed by the Austrian Research Promotion Agency (FFG) under grant 813039.

Bibliography

[1] Richard Chesson Mark Hopkins Alain Artieri, Viviana D'Alto and Wade D. Peterson Marco C. Rossi. Nomadik - open multimedia platform for next generation mobile devices, TA305 technical article. http://www.st.com, September 2004.

[2] Altera. Creating Multiprocessor Nios II Systems Tutorial, V 7.1. http://www.altera.com, May 2007.

[3] ARM. AMBA specification (rev 2.0), May 1999.

[4] ARM. ARM11 MPCore Processor, technical reference manual. Available at: http://www.arm.com, August 2006.

[5] Vasanth Asokan. White Paper: Xilinx Platform Studio (XPS), Designing Multiprocessor Systems in Platform Studio. http://www.xilinx.com, April 2007.

[6] Tobias Bjerregaard and Shankar Mahadevan. A survey of research and practices of network-on-chip. *ACM Comput. Surv*, 38(1), 2006.

[7] Brian Dobbing and Alan Burns. The ravenscar tasking profile for high integrity real-time programs. In *Proceedings of the 1998 annual ACM SIGAda international conference on Ada*, pages 1–6. ACM Press, 1998.

[8] Santanu Dutta, Rune Jensen, and Alf Rieckmann. Viper: A multiprocessor SOC for advanced set-top box and digital TV systems. *IEEE Design & Test of Computers*, 18(5):21–31, 2001.

[9] John Hennessy and David Patterson. *Computer Architecture: A Quantitative Approach, 4th ed.* Morgan Kaufmann Publishers, 2006.

[10] H. Peter Hofstee. Power efficient processor architecture and the cell processor. In *HPCA*, pages 258–262, 2005.

[11] SPARC International Inc. *The SPARC Architecture Manual: Version 8*. Prentice Hall, Englewood Cliffs, New Jersey 07632, 1992.

[12] V. Milutinovic J. Protic, M. Tomasevic. Distributed shared memory: concepts and systems. *Parallel & Distributed Technology: Systems & Applications, IEEE*, 4:63 – 71, 1996.

[13] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *j-IBM-JRD*, 49(4/5):589–604, 2005.

[14] Michael Kistler, Michael Perrone, and Fabrizio Petrini. Cell multiprocessor communication network: Built for speed. *Micro, IEEE*, 26:10–25, 2006.

[15] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*, 25(2):21–29, 2005.

[16] H. Kopetz. *Real-time systems: design principles for distributed embedded applications*. Kluwer Academic Publishers, 1997.

[17] James Laudon and Lawrence Spracklen. The coming wave of multithreaded chip multiprocessors. *International Journal of Parallel Programming*, 35(3):299–330, June 2007.

[18] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, USA, second edition, 1999.

[19] Grant Martin and Henry Chang. *Winning the SOC Revolution, Chapter 5*. Kluwer Academic Publishers, 2003.

[20] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, 1965.

[21] Christof Pitter and Martin Schoeberl. Time predictable CPU and DMA shared memory access. In *International Conference on Field-Programmable Logic and its Applications (FPL 2007)*, Amsterdam, Netherlands, August 2007.

[22] P. Puschner and A. J. Wellings. A profile for high integrity real-time Java programs. In *4th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC)*, 2001.

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

[24] Martin Schoeberl. A time predictable Java processor. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE 2006)*, pages 800–805, Munich, Germany, March 2006.

[25] Martin Schoeberl. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture*, doi:10.1016/j.sysarc.2007.06.001, 2007.

[26] Martin Schoeberl. SimpCon - a simple and efficient SoC interconnect. Available at: http://www.opencores.org/, 2007.

[27] Martin Schoeberl and Rasmus Pedersen. WCET analysis for a Java processor. In *JTRES '06: Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems*, pages 202–211, New York, NY, USA, 2006. ACM Press.

[28] Martin Schoeberl, Hans Sondergaard, Bent Thomsen, and Anders P. Ravn. A profile for safety critical java. In *10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'07)*, pages 94–101, Santorini Island, Greece, May 2007. IEEE Computer Society.

[29] William Stallings. *Operating Systems: Internals and Design Principles, 5th ed.* Prentice Hall, 2005.

[30] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[31] Cai-Dong Wang, Hiroaki Takada, and Ken Sakamura. Priority inheritance spin locks for multiprocessor real-time systems. In *ISPAN*, pages 70–76. IEEE Computer Society, 1996.

[32] Amy Williams, William Thies, and Michael D. Ernst. Static deadlock detection for java libraries. In Andrew P. Black, editor, *ECOOP*, volume 3586 of *Lecture Notes in Computer Science*, pages 602–629. Springer, 2005.

4

Performance Evaluation of a Java Chip-Multiprocessor

Christof Pitter and Martin Schoeberl

Institute of Computer Engineering,
Vienna University of Technology,
Austria

Proceedings of the International Symposium on Industrial Embedded Systems (SIES 2008). Montpellier, France, June 2008, pages 34-42.

Abstract

Chip multiprocessing design is an emerging trend for embedded systems. In this paper, we introduce a Java multiprocessor system-on-chip called JopCMP. It is a symmetric shared-memory multiprocessor and consists of up to 8 Java Optimized Processor (JOP) cores, an arbitration control device, and a global shared memory. All components are interconnected with a system-on-chip bus.

This paper focuses on the performance evaluation of different hardware configurations of the multicore system. Therefore, we vary the instruction cache sizes, the number of processors and the memory bandwidth. Within our experiments, we measure the performance by running three benchmarks on real hardware: an embedded application from industry, a computationally intensive matrix multiplication and a synthetic benchmark that continuously

accesses a shared data structure. Two different field-programmable gate arrays are used for the presented experiments.

Our results illustrate the promises and limits of the proposed multiprocessor architecture concerning synchronization, memory bandwidth and caching. Furthermore, we compare the performance and size of JopCMP with a complex Java processor.

4.1 Introduction

It is expected that chip-level multiprocessing (CMP) will be the future path of performance enhancements [10]. CMP technology integrates two or more processing units and a sophisticated communication network into a single integrated circuit (IC). In actual desktop and server architectures, two trends can be seen: (1) integration of two to four out-of-order super-scalar CPUs (Intel/AMD) on a single die or (2) integration of 8 very simple in-order RISC pipelines (Sun's Niagara [16] and IBM's CELL [15]). According to [27], multiprocessing is also common in embedded systems as it combines the goals of increasing performance, lower power consumption and cost effectiveness.

In this paper, we propose a CMP architecture composed of multiple Java Optimized Processor (JOP) cores and a shared memory. The shared memory is uniformly accessible by the homogeneous processing cores. A system-on-chip (SoC) bus connects the devices of the system. A fairness-based arbitration unit takes care of memory contention due to parallel memory requests; it employs a fair scheduling strategy, which divides the memory access bandwidth into equal shares. In contrast to existent memory arbiters of SoC buses (e.g. AMBA [6] or Avalon [3]), the proposed arbitration process is performed in the same cycle as the request happens. This feature increases the memory bandwidth.

JopCMP synchronizes the access of multiple CPUs to shared data structures by a global lock. Due to the implementation of JopCMP in field-programmable gate array (FPGA) technology, we are able to modify the number of processing cores easily. Additionally, the size of the instruction cache of each processor can be configured. Hence, we are able to analyze the behavior of JopCMP in detail. We use three benchmarks with different workload characteristics. A real-world application from industry – a lift controller in an automation factory – represents a medium computationally intensive, fully parallelized application without any accesses to shared data structures. A matrix multiplication benchmark represents a computationally intensive algorithm. This benchmark offers good potential for parallelism with low synchronization overheads. The third benchmark, parallel access to a hash

table, represents a workload with a high conflict on a shared data structure. It will stress our current solution for the multiprocessor synchronization. The results of the experimental measurements illustrate the promises and limits of the proposed multiprocessor solution.

The rest of the paper is structured as follows. Section 4.2 presents related work and delivers an insight into JOP. In Section 4.3, we describe the proposed CMP architecture and the interconnection network. Furthermore, the memory controller and the synchronization unit of JopCMP are summarized concisely. Section 4.4 presents the evaluation method (benchmarks and hardware platforms) and the benchmark results. Finally, Section 4.5 concludes the paper and provides guidelines for future work.

4.2 Related Work

The embedded system domain distinguishes between two multiprocessor architectures: heterogeneous multiprocessors and homogeneous multiprocessors. Heterogeneous multiprocessors are often tailored to specific features of the application. The architecture of the system usually combines a core CPU for controlling and communicating tasks and additional processing devices for specific functions, i.e. digital signal processing elements, interface processors or mobile multimedia processing units.

This paper concentrates on homogenous multiprocessors. These systems combine a number of identical CPU cores. In the following sections, we describe three embedded multiprocessor solutions and their interconnection systems that are available in FPGA technology. We also provide an overview of JOP, the processor used in our multiprocessor solution.

4.2.1 LEON

Gaisler Research AB designed and implemented a homogeneous multiprocessor system called LEON3-FT-MP [9]. It consists of a centralized shared memory and four LEON3-FT processor cores that are based on the SPARC V8 instruction set architecture [14]. Each CPU consists of a 7-stage pipeline with separate 16 KB data and instruction caches, a memory management unit, a floating-point unit and hardware support for multiplication and division. All the CPUs, additional I/O controllers and the memory controllers are connected using two advanced high-performance buses (AHB) of the AMBA specification [7]. One AHB runs at the CPUs' frequency of 266 MHz and connects the processors with the memory controller of the shared memory. Additionally the high-speed AHB communicates with the low-speed AHB (running at 133 MHz) using an AHB/AHB bridge. The low-speed

AHB connects all other peripheral devices with lower speed requirements to the system. The bus frequencies reported in [9] are estimations of a 0.13μ ASIC implementation. A prototype in a Xilinx Virtex-4 can run at 40 MHz.

According to the AMBA specification, a CPU defines the role of a master because it initiates the transactions with other components (slaves). The pipelined AHB bus can integrate up to 16 masters into an SoC. An arbiter controls the shared system bus. AHB specifies all interface signals between the masters and the arbiter and the arbiter and the slaves. Even though the specification of the arbitration protocol of the AHB is well defined, no priority strategies or arbitration algorithms are specified. The Leon implementation of the AHB arbiter uses fixed priorities. Our proposed JopCMP system includes several different arbiters. For this evaluation, we use a fair arbitration algorithm. Without a fairness-based arbiter, a system consisting of more than 4 CPUs will not exploit its performance. The Leon multicore system supports two operating systems: eCos and RTEMS. Software is developed in C/C++.

4.2.2 MicroBlaze

MicroBlaze [29] based CMPs can be designed with the Xilinx Embedded Development Kit (EDK). MicroBlaze is a 32-bit reduced instruction set computer (RISC) optimized for FPGA implementation. The pipeline length of the CPU can be configured to either 3 or 5 stages. It implements the Harvard architecture with separate instruction and data buses. The CPU can be tailored to the application needs (i.e. peripheral controllers or cache sizes).

Memory and peripheral devices are connected via the on-chip peripheral bus (OPB) [12]. OPB is part of the bus hierarchy called CoreConnect [11], an open standard for SoC communication proposed by IBM. Xilinx provides an OPB bus arbiter [13, 28] that can integrate up to 16 masters into the system. The available arbitration algorithms include fixed priority (FP) or least recently used (LRU). A full-featured GNU tool chain is available for software development in C/C++.

4.2.3 NIOS II

Altera's Nios II [4] and the System-on-a-Programmable Chip (SOPC) Builder [5] support the design and implementation of CMPs in Altera's FPGA technology. The Nios RISC architecture implements a 32-bit instruction set similar to the MIPS instruction set architecture. The sizes of its instruction and data caches are configurable. Nios II can be customized to meet the application requirements: three different models from non-pipelined up to a 6-stage pipeline. Examples of customizable features are a floating-point

unit, memory controllers and different communication controllers. Avalon [1] is the SoC bus used by the SOPC Builder. It connects the master and slave components to the System Interconnect Fabric. This System Interconnect Fabric encapsulates all connection details from the user. While the Avalon specification can be used freely, the System Interconnect Fabric is Altera's property.

For multiprocessor systems, the System Interconnect Fabric integrates an arbitration module [2]. In contrast to traditional shared bus architectures, the interconnection allows multiple masters to access different slaves simultaneously. This eliminates the bottleneck of one shared bus if one master may access a slave and another master wants to access a different slave in parallel. For a multiprocessor system where two or more masters frequently access one slave (the shared memory), the System Interconnect Fabric provides no advantage. If several masters request data from the same slave, an arbiter will determine which master will gain access. All other masters are forced to wait. The arbitration logic can be configured in the SOPC Builder. The arbitration schemes include fairness-based shares, round-robin scheduling, burst transfers, and minimum share value. The Nios II system supports the uClinux operating system and the C/C++ GNU tool chain is available.

4.2.4 Java Optimized Processor (JOP)

The Java optimized processor (JOP) [22, 23, 24] is an implementation of the Java Virtual Machine (JVM) in hardware. JOP translates the Java intermediate bytecodes to its own instruction set called microcode. These microcode instructions, implemented in hardware, are executed by the stack architecture. The CPU has a 4-stage pipeline.

Each thread has a local stack area. This thread private data is accessed very often. Therefore, JOP caches this data in a so-called stack cache [21]. Additionally, a kind of instruction cache (called method cache [20]) limits the memory access frequency and increases the processing power. Complete methods, shared among all the threads, are cached there. According to the JVM specification [17], the heap stores the shared data of the VM. All objects that are created by a Java application are stored on the heap. Caching of these objects is not implemented. A typical JOP configuration contains the CPU core, the method and stack cache, a memory interface and several I/O controllers.

JOP is designed for embedded real-time systems where the analysis of the worst-case execution time (WCET) of all threads is possible. Hence, a couple of typical architectural advancements, used to increase the average processing power, have been omitted. Examples include branch prediction or out-of-

order execution. Nevertheless, JOP shows good average performance and lower logic resources consumption in comparison to other Java processors. Therefore, our Java multiprocessor system is based on JOP.

4.2.5 Discussion

According to [8], the inter-core communication in CMPs offers more bandwidth than traditional backplane buses used for building traditional SMPs. Additionally, the latency of a transfer is much lower on an SoC bus. The described multiprocessors are still using backplane style buses that are not appropriate for a SoC interconnection. Furthermore, there is no use for a complex bus hierarchy in our design. Our system consists of a couple of CPUs connected to a single shared memory. Therefore, our choice of the interconnection network is the simple SoC bus called SimpCon [25], which is further described in Section 4.3.2. Moreover, we use a fairness-based arbitration algorithm. To our knowledge, Leon's IP library does not include a fair arbiter. A disadvantage of Nios II based multiprocessor systems is that data cache coherency is not supported. The data caches are disabled for a multiprocessor system. In our proposed solution, we limit data caching to the thread private JVM stack.

JOP is open source and freely available for academic research. Every single part of the processor core can be customized and configured. JOP is technology independent (like LEON) and has been ported to FPGAs from Altera, Xilinx, and Actel. This property avoids a lock-in to a single FPGA vendor, as it is the case for MicroBlaze and Nios.

4.3 Overview of JopCMP

According to [27], a multiprocessor system consists of 3 major subsystems: processing elements, memory and an interconnection network. JopCMP implements the symmetric (shared-memory) multiprocessor (SMP) model [10]. JOPs provide the basis of the homogeneous CMP as depicted in Figure 4.1. These processing elements perform computations in parallel. Instructions and data are stored in a single shared memory. The interconnection network is responsible to connect multiple processors with the memory. An arbiter is part of this network and controls the memory access to the shared memory. An SoC bus is used to connect the processing cores to the arbiter, and the arbiter to the shared memory. The arbiter acts as slave for each JOP and as master for the memory controller. We are convinced that synchronization of shared data is a further major subsystem of an SMP. It is responsible to coordinate access to shared objects. Following sections describe the elements in more detail.

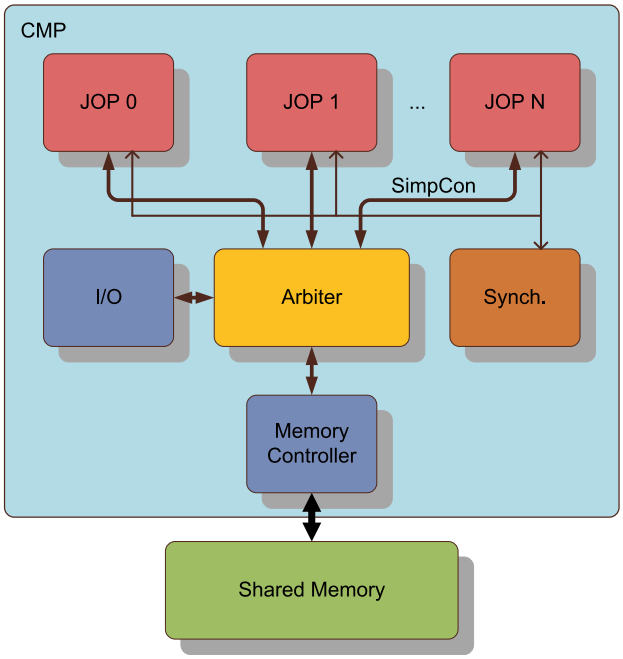

Figure 4.1: Overview of JopCMP.

4.3.1 Memory Hierarchy

JOP's memory hierarchy with its caches (see Section 4.2.4) and the shared memory architecture fit very well to each other. JOP does not support caching of Java objects. Hence, cache coherency and consistency issues cannot arise. The instruction cache is read-only and therefore not an issue. The stack cache contains only thread local data and no cache coherency protocol (e.g. snooping) is needed. Avoiding such a protocol in an FPGA saves resources and does not impair the maximum clock frequency of the CMP. Our multicore solution avoids cache coherency conflicts by design.

4.3.2 Interconnection Network

The selection of the interconnection network topology is a major design decision of a multiprocessor architecture. We use a SoC bus with a central arbitration unit.

Traditionally, a bus is a set of wires that connects multiple masters and multiple slaves of a system on a printed circuit board. A master initiates each communication. All communication channels between the interfaces of the exchanging devices represent the so-called backplane bus. The bus arbiter implements a certain priority algorithm that controls this shared

bus. If multiple masters request access to the bus, the arbiter will allocate the shared bus resource to one master. All other masters are forced to wait. Consequently, multiple masters cannot concurrently drive the bus.

The simple SoC interconnect (SimpCon) [25] is used to connect SoC components on a single IC. This synchronous on-chip interconnection is intended for read and write transfers via point-to-point connections. Only a master can initiate a transaction via a write or a read request. In comparison to other commonly used SoC buses like Avalon [2], this specification does not work like a backplane bus. The master's driven control, address, and data lines are only valid for a single clock cycle. A slave has to register any signals (e.g., the address) that are needed for several clock cycles. Consequently, the master can continue to execute its task until the data of an access is needed. Furthermore, the slave can early inform the master (up to two cycles ahead) when a bus transaction will finish. Therefore, pipelined memory accesses and consequently fast data transfers are possible.

4.3.3 Fairness-based Arbiter

SimpCon is well suited for on-chip point-to-point connections. We introduce a central arbiter to connect multiple masters (JOPs) to one slave (memory controller). This arbitration device controls the memory access of multiple CPUs to the shared memory. An adequate priority policy has to be implemented to resolve competing memory requests of the CPU cores. If memory request contention happens, only one master is granted access and all others are forced to wait.

Usually, the arbitration policy of the arbiter depends on the application needs. An example of a dynamic arbitration scheme depending on the CPU priorities is described in [18]. Each CPU of the system is assigned a unique priority. If memory access contention occurs, the CPU with the highest priority will be granted access. This arbitration policy can be used for real-time systems where one CPU executes hard real-time[1] tasks and the other ones execute tasks with minor requirements regarding deadlines. Consequently, the hard real-time CPU gets the highest priority of the system.

In this paper, we analyze the performance of a balanced CMP. Therefore, an arbitration policy is implemented that guarantees fairness among the CPUs accessing the shared memory. Furthermore, starvation of any CPUs is prohibited. Each CPU in the system is assigned a unique CPU identity (CPU_{ID}), starting from 0 up to the number of CPUs-1. Our fair arbitration policy uses a wrapping counter. It changes the permission, which CPU is allowed to access the memory. The value of the counter has the same range

[1] A hard real-time task has to deliver its results on time. A single miss of a deadline may result in a disastrous accident.

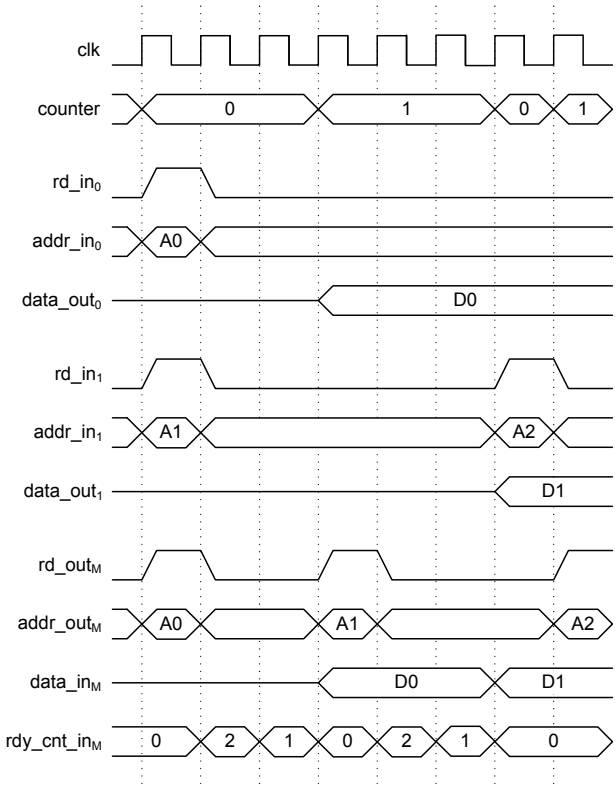

Figure 4.2: Memory access arbitration of the fairness-based arbiter.

as the CPU identities. At the time of completion of the prior memory access, the counter is advanced. If the new counter value equals a requesting CPU_{ID} and the memory is ready to execute a memory access, the memory access will be processed and the current value of the counter will be halted until completion of the data transmission. In the case that the CPU with CPU_{ID} that equals the value of the counter does not want to access the memory, the counter is immediately advanced.

Figure 4.2 shows an arbitration scenario of a 2-way CMP system with 2 cycles memory access time. The signals *clk* and *counter* are internal signals of the arbiter. All other signals are either input or output signals of the arbiter illustrated by the signal's name. Furthermore, the subscripts indicate whether the signals belong to a specific CPU (denoted by the CPU_{ID}) or to the memory controller. Some SimpCon signals, i.e. the signals for write access, are disregarded in Figure 4.2.

At the first clock cycle, both CPU_0 and CPU_1 want to perform a read access to the shared memory simultaneously. CPU_0 is immediately allowed to perform the read access because the counter's value equals to 0 and the

memory is idle ($rdy\_cnt\_in_M$ equals to 0). Consequently, the read enable signal of the memory (rd_out_M) is driven high and the memory address ($addr\_out_M$) is asserted. The read request of CPU_1 is registered in the arbiter. It has to wait until completion of the memory access of CPU_0, indicated by the value 0 of signal $rdy\_cnt\_in_M$ and by the received data on $data\_in_M$ and $data\_out_0$ accordingly. At completion of the memory access, the counter is already incremented by one and the registered memory access of CPU_1 is processed. When the data is available, the counter already equals to 0. Other than CPU_1, CPU_0 does not request a memory access and the counter is advanced in the following cycle. CPU_1's access is not granted because the counter value equals to 0 in the current clock cycle. In the next cycle, the registered memory access of CPU_1 is processed.

This arbitration algorithm implements a fair partitioning of the memory bandwidth. The more CPUs are part of the system the higher is the probability that the counter matches the CPU_{ID} with a pending memory request after a successful access. Therefore, a high workload will result in a saturation of the memory bandwidth. In case of low contention between several CPUs, this scheme wastes memory bandwidth (and performance) because delays of memory access grants are introduced.

The arbiter performs the arbitration decision in the same cycle as the request arrives. Therefore, we have a zero-cycle arbitration protocol. No additional cycle is lost for arbitration and the memory access latency is not affected. Zero-cycle arbitration latency is an advantage in comparison to existing arbiters like AMBA [7] and Avalon [2], which always use an extra cycle for a so-called bus request phase. Consequently, each access takes more time. This fairness-based arbiter is implemented at Register Transfer Level (RTL) in VHDL. The arbitration process is primarily implemented in combinational logic without considerably decreasing the clock frequency of the whole system.

The arbiter is fully scalable with respect to the number of connected masters. Compared to existing arbiters like AMBA [7] or CoreConnect [13] the maximum number of connected masters is not limited to 16. Hence, the application determines the quantity of connected masters.

4.3.4 Memory Controller

The memory controller is the interconnection between the arbiter and the shared memory. Controllers for different types of memory are available for the SimpCon bus. The controller supports pipelined read and write commands. Pipelining of SimpCon, the arbiter, and the memory controller allows back-to-back reads from the memory. Furthermore, due to the definition of the SimpCon interconnect, it is possible to use the registers in the IO cells

of the FPGA for all latency-sensitive SRAM signals (address bus, data bus, and control lines).

4.3.5 Synchronization

Shared memory SMP systems need a synchronization mechanism. The CPUs exchange data by reading and writing shared data objects. In order to ensure that a CPU has exclusive access to such an object, synchronization is necessary.

Therefore, we introduced a synchronization unit in hardware that controls one global lock. If one core wants to access a shared object, it will request the lock using the synchronization interconnection depicted in Figure 4.1. JOP will be granted access if no other processor of the system is holding the lock. Otherwise, it must wait until the other processor completes accessing the shared object.

The hardware lock allows fast implementation of the bytecodes `monitorenter` and `monitorexit` that are used by the JVM for synchronization. For short critical sections, this feature compensates for the less reactive behavior of a single global lock. A side effect of a single lock is the avoidance of deadlock by design. Further information on synchronization of JopCMP can be found in [18].

4.4 Performance Evaluation

In this section, we evaluate the performance of our CMP architecture. We present and compare the performance of our multicore depending on the number of processors and the size of their instruction caches. The benchmarks highlight that several processors working in parallel outperform a uniprocessor that executes the same workload sequentially. Two different hardware platforms set up the basis of the presented experiments. Furthermore, we include FPGA synthesis results and compare performance and size of JopCMP with the complex Java processor picoJava [26].

4.4.1 Benchmarks

Using a multi-core system, application development is more complex because the application code has to be spread out among different processors. We evaluate the CMP with three different benchmarks:

- a real-world embedded application from industry (`Lift`),

- a matrix multiplication (`MMul`) and
- an application that is operating on a hash table (`HTable`).

Our benchmark methodology is as follows: `Lift` and `HTable` are executed several times (16384 and 256 respectively). This workload is distributed evenly on the multiprocessor versions. The benchmark `MMul` performs automatic distribution of the workload.

Lift Application

`Lift` is a real-world example with industrial background. This embedded application is a lift controller used in an automation factory. `Lift` is part of the embedded Java benchmark suite called JavaBenchEmbedded, as described in [22]. In fact, the benchmark is written to measure uniprocessor performance. Nevertheless, we use it for executing several `Lift` tasks on multiple CPUs concurrently. Consequently, this benchmark presents a medium computational, fully parallelized application without any accesses to shared data structures and synchronization needs.

Matrix Multiplication

The benchmark `MMul` is designed to give some idea about the performance of a computationally intensive algorithm with good potential for parallelism. The benchmark multiplies two matrices of dimension 100x100. This calculation results in 1 million multiplication operations. Each row of the resulting matrix is calculated by a single CPU. A synchronization variable secures that the next idle CPU takes the next unsolved row until the result is achieved. The benchmark measures the elapsed time for the calculation. `MMul` is classified as a parallel workload – computational intensive with low synchronization overhead.

Hash Table Access

Hash tables are often used data structures to manage and lookup different data objects. Each value of a hash table is associated with a key. The key permits efficient access to the value. `HTable` presents an interesting application for the JopCMP; multiple CPUs access the shared data structure in a tight loop leading to severe synchronization conflicts. `HTable` measures the elapsed time until a fixed number of read, insert and delete operations are performed.

4.4.2 Hardware Platforms

We use two different hardware platforms for our evaluation. They differ in FPGA technology and memory bandwidth.

Altera DE2 Board

The system has been prototyped on Altera's Development and Education Board (DE2 Board) with a low-cost Cyclone II (EP2C35) FPGA. It has a capacity of 33,000 logic elements (LEs) and 483,000 bits of on-chip memory. This FPGA can be populated with up to 8 JOP cores. The DE2 Board contains 512 KB SRAM connected via a 16-bit data bus. A single read operation for a 32-bit data item takes 4 clock cycles. On the DE2 board, we run all systems with the same clock frequency (90 MHz). This frequency is the maximum value that all different configurations can run and that can be configured with the PLL.

Cycore Board

The Cycore FPGA board contains the older Cyclone I FPGA (EP1C12) from Altera. It has a capacity of 12,000 LEs and 239,000 bits of on-chip memory. A 1 MB, 15 ns SRAM is connected via a 32-bit data bus. With this SRAM, it is possible to perform a 32-bit memory read in two cycles for system frequencies up to 100 MHz. Therefore, the memory bandwidth is two times higher than the bandwidth of the Altera DE2 board.

4.4.3 Measurements

To evaluate JopCMP, we compare the performance of different multicore configurations with the single JOP version under varying workloads and FPGA platforms.

As it is expected that the memory bandwidth will restrict the number of useful cores we also measure the consumed bandwidth. We have integrated a memory access counter into the memory controller to measure the number of cycles the memory bus is busy. Equation 4.1 gives the memory load relative to the available memory bandwidth. The resulting memory bandwidth utilization depends on the size of the instruction cache. It decreases with larger cache sizes because memory access frequency drops as well.

$$Utilization_{Mem.Bandwidth} = \frac{MemoryAccessTime}{ExecutionTime} \tag{4.1}$$

Table 4.1: Execution time and memory bandwidth utilization of Lift, Altde2 @ 90 MHz

	1 KB Cache		2 KB Cache		4 KB Cache	
Number of JOP Cores	Time (ms)	Util. (%)	Time (ms)	Util. (%)	Time (ms)	Util. (%)
1	1255	40	1158	32	1158	32
2	769	66	662	56	662	56
4	613	83	484	77	484	77
8	595	85	459	81	–	–

Table 4.1 shows the measured execution time and memory bandwidth utilization of `Lift` running at a frequency of 90 MHz on the Altera DE2 board. The first column gives the number of JOP cores of the system. Additionally, the size of the instruction cache is varied between 1, 2 and 4 KB for each CPU. The execution time and the memory bandwidth utilization are measured for each combination of number of CPUs and cache size. A 4 KB cache version of 8 cores is missing, as it does not fit into the available FPGA. One JOP with an instruction cache of 2 KB executes `Lift` in 1158 ms and the measured memory bandwidth utilization is 32%. A dual-core performs about 1.8 times faster than a single JOP. Actually, a 4-processor system with 1 KB of cache nearly doubles the performance of a single-core. The same system with more cache experiences a speedup of 2.4, no matter if either a 2 KB or a 4 KB instruction cache is used. Using more processors does not provide significantly better performance.

Furthermore, Table 4.1 gives information on the memory bandwidth utilization (denoted Util.) of the system. The utilization decreases with larger caches. Nevertheless, no real difference between 2 KB or 4 KB can be seen. We conclude from this measurement that the kernel of the `Lift` benchmark is small enough to fit into the 2 KB cache. Although the consumed memory bandwidth does not reach the theoretical possible 100%, we see only minor performance differences between a 4-core and 8-core system.

Table 4.2 depicts the results of the measurement of `MMul` on the DE2 platform. The computationally intensive algorithm demonstrates its good potential for parallelism. The speedup of the CMPs consisting of 2 and 4 cores comes up to our expectations with speedups of 1.5 and 1.7 accordingly. 8 cores provide no additional significant speedup. We assume that a combination of high memory conflicts and increased synchronization cost becomes

Table 4.2: Execution time and memory bandwidth utilization of MMul, Altde2 @ 90 MHz

	1 KB Cache		2 KB Cache		4 KB Cache	
Number of JOP Cores	Time (ms)	Util. (%)	Time (ms)	Util. (%)	Time (ms)	Util. (%)
1	2957	52	2957	52	2957	52
2	1932	79	1932	79	1932	79
4	1773	86	1773	86	1773	86
8	1771	86	1771	86	–	–

Table 4.3: Execution time and memory bandwidth utilization of HTable, Altde2 @ 90 MHz

	1 KB Cache		2 KB Cache		4 KB Cache	
Number of JOP Cores	Time (ms)	Util. (%)	Time (ms)	Util. (%)	Time (ms)	Util. (%)
1	413	27	410	26	410	26
2	423	29	408	26	408	26
4	341	31	346	29	344	29
8	263	32	262	30	–	–

noticeable. The memory bandwidth utilization remains constant at 86% independent of a 4- or 8-way CMP. Increasing the cache size does not result in any performance improvements or any change of the consumed memory bandwidth.

Table 4.3 shows the measurements of `HTable` running at a frequency of 90 MHz on the DE2 board. Unlike `Lift` and `MMul`, this benchmark results in a small performance slowdown comparing a single JOP with a 2-core system with 1 KB of cache. Only slight speedups with the 2 KB and 4 KB versions can be seen. This originates from the application's characteristics. It combines low computational demands with high synchronization overhead. Nevertheless, CMPs with more CPUs cover this overhead by introducing more processing power. The 8-core JopCMP with 2 KB of cache is about

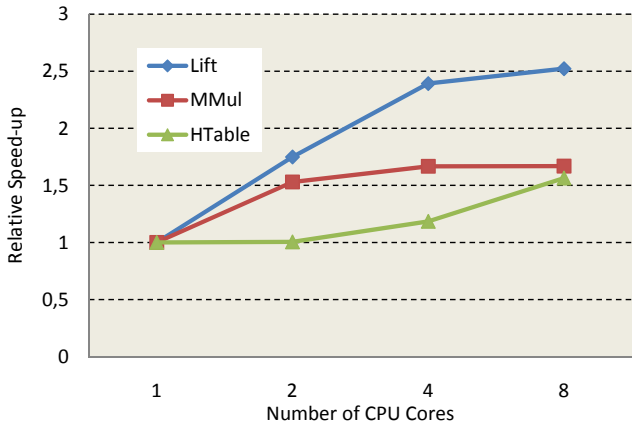

Figure 4.3: Performance comparison of JopCMP, running three different benchmarks.

1.6 times faster than the comparable single-JOP. `HTable`'s high synchronization demands are easily noticeable by the figures of the memory bandwidth utilization. These results show a very small increase when adding more CPUs to the CMP.

Figure 4.3 summarizes the measured execution times of the configurations with 2 KB cache size. The horizontal axis describes the number of CPUs and the vertical axis illustrates the relative speed-up. The relative speedup is the relation between the execution time on a single core and a multi core version. We can see different saturation points for `Lift` and `MMul`. Interesting is the result of the `HTable` benchmark: it only shows a significant speedup with many cores and we do not see the saturation at 8 cores. For this benchmark, it would be interesting to run a 16-way multicore version.

Table 4.4 shows `Lift` benchmark results based on the Cycore FPGA board. Running at a clock frequency of 60 MHz, the 2-way CMP with 2 KB cache is 1.9 times faster than JOP. The system consisting of three processors is 2.5 times faster. In comparison to the DE2 board, the memory bandwidth utilization of 48% of the 3-core CMP with 2 KB cache is lower than every 2-core configuration of the DE2 platform. The measurements confirm that the memory of the Cycore board is about 2 times faster than the memory on the DE2 board. Synthesis results show that JopCMP can achieve a maximal clock frequency of 60 MHz on the Cyclone I FPGA. This is reasonable because the Cyclone I series is an older technology compared to Cyclone II. Furthermore, a 3-way JopCMP with 2 KB cache size is the maximum that can be integrated into this FPGA.

Table 4.4: Execution time and memory bandwidth utilization of Lift, Cycore @ 60 MHz

	1 KB Cache		2 KB Cache		4 KB Cache	
Number of JOP Cores	Time (ms)	Util. (%)	Time (ms)	Util. (%)	Time (ms)	Util. (%)
1	1506	24	1455	18	1455	18
2	844	45	779	35	779	35
3	661	57	578	48	–	–

4.4.4 Comparison with a Complex Java Processor

picoJava II [26], Sun's hardware implementation of the JVM, consumes more resources than JOP. An interesting comparison is whether a single, complex processor or a multiprocessor based on simple processors performs better for multi-threaded workloads. In the server domain Sun has chosen to implement 8 simple RISC cores (with a 6-stage in-order pipeline) in the CMP Niagara [16] while Intel and AMD still use their complex super-scalar out-of-order architectures.

Puffitsch has implemented the picoJava II in an FPGA [19] on the same board (DE2) that we use for our evaluation. picoJava II runs at 40 MHz in the Cyclone II. The resource consumption is 27,562 LEs and 381 KBit memory when configured with 16 KB instruction and 16 KB data cache.

Puffitsch could run the `Lift` and the `HTable` benchmark in the picoJava FPGA implementation. Table 4.5 shows the comparison between picoJava and various versions of JopCMP with 2 KB instruction cache. The performance of the `Lift` benchmark and the resource consumption are given relative to picoJava. picoJava executes the `Lift` benchmark in 632 ms. A single JOP needs 1158 ms for the same workload. That means that picoJava is about two times faster at 40 MHz than a single JOP version at 90 MHz. Note, that the resource consumption (LEs and memory for the caches) of picoJava is much higher than for JOP. The two-processor configuration is 5% slower than picoJava and consumes 1/5 of the FPGA resources. A 4-way processor configuration of JOP is about 30% faster than picoJava, but consumes less than half of the resources.

The benchmark `HTable` executes in 165 ms on picoJava. On this synthetic, lock intensive benchmark picoJava is almost three times faster than JOP. picoJava uses two lock registers to cache the last two obtained locks. This hard-

Table 4.5: Performance and size of JopCMP relative to picoJava in the same FPGA board

Number of JOP Cores	Performance	Size	Memory
1	0.55	0.11	0.12
2	0.95	0.22	0.24
4	1.30	0.43	0.47
8	1.38	0.84	0.94

ware feature allows execution of bytecodes `monitorenter` and `monitorexit` in 3 and 2 cycles when the lock is found in one of the lock registers. JOP needs for those operations with the global lock 18 and 20 cycles.

4.4.5 Synthesis Results

Table 4.6 shows the utilization of different multicore systems within the FPGA device EP2C35 on the DE2 board. The size of the instruction cache of all JOPs is configured to 2 KB. With such a small cache, the utilization of logic elements and on-chip memory is balanced. However, 2 KB instruction cache and 0.5 KB data cache (the stack cache) are very small, even for embedded processors. For a CMP system based on FPGA technology, we would prefer devices with a different LE to on-chip memory ratio.

Surprisingly, the frequency does not change significantly with the number of JOP cores in the system. The timing analysis results are obtained with Altera Quartus II. The maximum frequency varies between 110 MHz (for a single JOP) and 96 MHz (for an 8-way CMP). Using the phase-locked loop (PLL) of the FPGA, the clock frequency of all configurations is configured to 90 MHz.

Our arbiter scales quite well with respect to the maximum clock frequency. The arbiter performs the arbitration decision with zero cycle latency without hurting the maximum clock frequency with more bus masters.

4.5 Conclusion and Future Work

We have shown that even in a medium sized low-cost FPGA it is possible to run 8 cores of a Java processor in parallel. The performance enhancements of factors 1.7 and 2.5 with 2- and 4-way cores for a real-world application are promising. We did not expect a linear improvement in speed. A speedup

Table 4.6: Synthesis Results on the Cyclone II FPGA (EP2C35)

Number of JOP Cores	Resources (LE)	(%)	On-chip Memory (KBit)	(%)	Frequency (MHz)
1	3,033	9	45	10	110
2	6,196	19	90	19	104
4	11,946	36	180	38	101
8	23,252	70	360	76	96

logarithmic to the number of cores would satisfy future processing demands as the number of transistors that can be integrated into a single chip is still increasing exponentially.

However, 2 out of 3 benchmarks saturated at 4 cores. For the benchmarks with almost no synchronization like `Lift` and `MMul`, the bottleneck is the memory bandwidth. The access to the hash table needs a lot of synchronization. For this benchmark, the memory bandwidth utilization is almost independent of the numbers of processors or the instruction cache size.

The bandwidth of the memory is limited on the DE2 board due to the narrow 16-bit interface to the SRAM. It even limits the performance of a single processor. A comparison between the performance of the Cycore board and the DE2 board shows that the Cycore board running at 2/3 of the clock frequency is not that much slower. We conclude that the configuration of the DE2 memory interface limits the usable number of cores to four.

We estimate that fast memory and caching can increase the number of useful cores to about 8. However, additional cache memories are not an option with the logic to memory relation of current FPGAs. The on-chip memory is the limiting factor for the configuration with 8 cores. For 8 cores, we had to limit the instruction cache to 2 KB. Reducing the instruction cache further to 1 KB does not impair the performance of the small benchmarks. However, we see a performance reduction in the application benchmark `Lift`: a 4-processor version with 2 KB instruction cache performs better than an 8-processor version with 1 KB instruction cache. Even the two-processor version with 2 KB cache is almost as fast as the 8-processor version with 1 KB cache.

Comparing our JopCMP against a complex Java processor, such as picoJava II, we conclude that a multiprocessor version of a simpler and smaller architecture is more efficient (performance/die area) for parallel workloads. With independent instances of the application benchmark `Lift` a 4-core version of JopCMP is 1.3 times faster than picoJava with a die area of about

45% of picoJava.

We saturate the memory access up to 86%. Theoretically, 100% bandwidth utilization is possible. We have not saturated the memory bandwidth, as the arbiter does not yet fully support the pipelining of the SimpCon specification. The pipeline is flushed on a switch between cores. We assume that an enhancement of the arbiter will result in 100% utilization of the memory bandwidth with 8 cores.

For applications with lot of inter-thread communication, the single global lock is clearly the bottleneck (as seen by the hash table test). We consider two different paths of enhancements: (1) adding hardware for several, independent locks and (2) implement a hardware transactional memory. Hardware transactional memory is the more complex solution. However, it results in an automatically finer grained locking that will improve the performance of the concurrent hash table access.

In this paper, we have evaluated the Java CMP system with respect to average case throughput. However, JOP is designed as a real-time processor to simplify the WCET analysis. The ultimate goal of our research is a Java multiprocessor architecture for embedded real-time systems that can be analyzed with respect to the WCET of individual tasks.

Acknowledgement

We thank Wolfgang Puffitsch for executing the benchmarks on the picoJava processor and providing the result for our comparison.

The research leading to these results has received funding from the Austrian Research Programme FIT-IT under contract number 813039 (TPCM) and the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement number 216682 (JEOPARD).

Bibliography

[1] Altera. Avalon interface specification, April 2005.

[2] Altera. Avalon Memory-Mapped Interface Specification (v3.3), May 2007.

[3] Altera. Creating Multiprocessor Nios II Systems Tutorial, V 7.1. http://www.altera.com, May 2007.

[4] Altera. Nios II Processor Reference Handbook (ver 7.2), October 2007.

[5] Altera. Quartus II Handbook Volume 4: SOPC Builder (ver 7.2), October 2007.

[6] ARM. AMBA specification (rev 2.0), May 1999.

[7] Arm. AMBA Specification (rev 2.0), May 1999.

[8] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, December 2006.

[9] J. Gaisler and E. Catovic. Multi-Core Processor Based on LEON3-FT IP Core (LEON3-FT-MP). In *DASIA 2006 - Data Systems in Aerospace*, volume 630 of *ESA Special Publication*, July 2006.

[10] John Hennessy and David Patterson. *Computer Architecture: A Quantitative Approach, 4th ed.* Morgan Kaufmann Publishers, 2006.

[11] IBM. The CoreConnect Bus Architecture. http://www.ibm.com, 1999.

[12] IBM. On-chip peripheral bus architecture specifications v2.1, April 2001.

[13] IBM. 32-Bit OPB Arbiter Core Databook revision 1, March 2007.

[14] SPARC International Inc. *The SPARC Architecture Manual: Version 8.* Prentice Hall, Englewood Cliffs, New Jersey 07632, 1992.

[15] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *j-IBM-JRD*, 49(4/5):589–604, 2005.

[16] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*, 25(2):21–29, 2005.

[17] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, Reading, MA, USA, second edition, 1999.

[18] Christof Pitter and Martin Schoeberl. Towards a Java multiprocessor. In *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems (JTRES 2007)*, Vienna, Austria, September 2007. ACM Press.

[19] Wolfgang Puffitsch. picoJava-II in an FPGA. Master's thesis, Vienna University of Technology, 2007.

[20] Martin Schoeberl. A time predictable instruction cache for a Java processor. In *On the Move to Meaningful Internet Systems 2004: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2004)*, volume 3292 of *LNCS*, pages 371–382, Agia Napa, Cyprus, October 2004. Springer.

[21] Martin Schoeberl. Design and implementation of an efficient stack machine. In *Proceedings of the 12th IEEE Reconfigurable Architecture Workshop (RAW2005)*, Denver, Colorado, USA, April 2005. IEEE.

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

[23] Martin Schoeberl. A time predictable Java processor. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE 2006)*, pages 800–805, Munich, Germany, March 2006.

[24] Martin Schoeberl. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture*, doi:10.1016/j.sysarc.2007.06.001, 2007.

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

[26] Sun. *picoJava-II Microarchitecture Guide*. Sun Microsystems, March 1999.

[27] Wayne Wolf. *High-Performance Embedded Computing: Architectures, Applications, and Methodologies*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[28] Xilinx. OPB Arbiter product specification (v1.10c), December 2005.

[29] Xilinx. MicroBlaze Processor Reference Guide, Embedded Development Kit EDK 9.2i. http://www.xilinx.com, June 2007.

5

Time-Predictable Memory Arbitration for a Java Chip-Multiprocessor

Christof Pitter

Institute of Computer Engineering,
Vienna University of Technology,
Austria

Proceedings of the 6th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2008). Santa Clara, California, 2008, pages 115-122.

Abstract

In this paper, we propose an approach to calculate worst-case execution times (WCET) of tasks running on a homogeneous Java multiprocessor. These processors access a shared main memory. Hence, the tasks running on different CPUs may influence the execution times of each other. Therefore, we implemented a time division multiple access arbiter that divides the memory access time into equal time slots, one time slot for each CPU. This memory arbitration allows calculating upper bounds for the execution time of Java bytecodes depending on the number of CPUs, the size of the time slot, and the memory access time. A WCET analysis tool can utilize these results and generate temporal, upper bounds for application tasks. We further explore how the size of the time slot and the number of CPUs in the system influ-

ence the WCET results. Furthermore, a real-world application task is used to compare the analyzed results with measured execution times. This paper describes the timing analysis of a time-predictable Java multiprocessor with shared memory.

Keywords: Chip-Multiprocessor, Java, Shared Memory, Worst-case Execution Time

5.1 Introduction

Multiprocessors and particularly chip-multiprocessors (CMP) are gaining importance in the embedded market. The CMP technology integrates two or more processing units and a sophisticated communication network into a single integrated circuit. A heterogeneous CMP combines different processing units connected with their memories that are customized to one single part of the application. The homogeneous CMP approach consists of identical CPUs to increase the computation power for general-purpose applications. To save resources and keep the price of the semiconductors low, a single shared main memory is used.

Today, many embedded systems are used for applications where real-time behavior is more important than computation power. Such hard real-time systems must undergo a timing analysis. Therefore, the worst-case execution time (WCET) of each task in the system has to be known. The WCET is the maximum execution time of a piece of code. It is the time a process could eventually need to execute under worst conditions on a given processor. In [20], Wilhelm et al. define the goal of WCET analysis concerning the upper bounds of the execution time:

1. they have to be safe and
2. should be as tight as possible.

Most importantly, the calculated upper timing bounds have to be safe in order to ensure hard real-time behavior of the system. The results must not underestimate the WCET values; otherwise, unpredictable behavior of the system could put the mission at risk with potentially serious consequences. Moreover, the upper bounds should be as tight as possible to keep the overestimation low in order to conserve resources.

There are three different methods to estimate the WCET of a task: static analysis; by measurement; or a hybrid approach combining both methods. The large drawback of the measurement-based method is that these estimates cannot reliably guarantee that the worst-case situation – provoked by

a special rare occurring input – is part of the measurement [2]. Furthermore, this kind of analysis is rather complex and time-consuming. Measurement-based analysis can be sufficient for soft real-time systems, but the authors believe that static analysis should be the state-of-the-art of modern hard real-time systems. Using static WCET analysis, the WCET of tasks is analyzed before runtime, independent of any input values. The objective of this analysis is to find the path with the maximum execution time of the program code.

This paper proposes a static WCET analysis of a homogeneous CMP with a shared memory. The CMP system is composed of multiple Java Optimized Processor (JOP) [15, 17] cores and a shared memory. The shared memory is uniformly accessible by the homogeneous processing cores. A system-on-chip (SoC) bus connects the devices of the system.

JOP comes with a static WCET analysis tool, which is described by Schoeberl and Pedersen in [18]. A contribution of this paper is the extension of the WCET tool for timing analysis of the multiprocessor system. The key component for real-time analysis of the CMP is the arbiter that divides the memory access bandwidth into time slots, one for each CPU. Hence, we can analyze the WCET of Java bytecodes depending on the size of the time slot, the number of CPUs in the system, and the memory access time. These execution times are the basis for the WCET analysis of tasks. Our approach is described using a simple example. Additionally, we provide measured data of the execution time of the example. The results are obtained by running the application in hardware. Consequently, we are able to compare the analyzed results with measured execution times. Furthermore, the measured and analyzed execution time results of a real-world application show the reliability of the proposed method.

The rest of the paper is structured as follows. Section 5.2 outlines related work on WCET analysis. Section 5.3 presents the CMP system. Additionally, we introduce the approach for time-predictable arbitration of the shared memory access. Section 5.4 summarizes the static WCET analysis based on JOP. Furthermore, it outlines the WCET analysis of the JopCMP system and gives an example. Section 5.5 compares the obtained WCET results with measured execution times. Finally, Section 5.6 concludes the paper and provides guidelines for future work.

5.2 Related Work

WCET analysis is crucial to timing analysis of hard real-time systems. The task set of a real-time system requires timing validation by schedulability analysis [4, 8]. Hence, the WCET of each task has to be calculated. If and

only if these upper bounds of the execution times are known, the schedulability analysis can be performed. Consequently, the result of the analysis shows whether the deadlines of the tasks will be met (guaranteeing that all tasks can be executed by the system) or not.

WCET analysis has been an active and well-established research area for years in the uniprocessor domain. Both Puschner and Burns [12], and Wilhelm et al. [20] give a broad overview of the WCET research. Nevertheless, not all these achievements can be applied to multiprocessor systems. They are based on the assumption that tasks are independent and cannot influence each other. Using modern multiprocessors with shared resources (i.e. a shared memory), tasks influence each other's execution times and cannot be analyzed in isolation.

To the best of our knowledge, only one research group (from university of Linköping) has studied the WCET analysis of multiprocessors [1, 13]. These publications are based on a multiprocessor system-on-chip with a shared communication bus, connecting several CPUs with two different types of memory. Each CPU has a private memory and all the processing units share a common memory for communication. A CPU is equipped with instruction and data caches, which are used to fetch data and instructions from the private memory. During execution, a task can only access private memory and no shared data objects. Hence, all input data must be placed into the private memory before the task can start executing. Consequently, in most cases the execution time of the task can never be influenced by other tasks (compare the simple-task model [5]). However, the communication bus serves as a communication interface between the CPUs and the private memories, and the CPUs and the shared memory. If a cache miss occurs during task execution, data has to be fetched from private memory using the communication bus. Therefore, some sort of bus arbitration is necessary because several CPUs may request a cache line from their private memories simultaneously.

In this paper, we introduce our approach to WCET analysis of a multiprocessor using shared resources. Even though the application tasks running on different CPUs may influence the execution times of each other, we are able to bound the WCET of the real-time tasks.

5.3 JopCMP Architecture

According to [21], a multiprocessor system consists of three major subsystems: processing elements, memory and an interconnection network. JopCMP implements the symmetric (shared-memory) multiprocessor (SMP) model [3]. Several JOPs provide the basis of the homogeneous CMP as depicted in

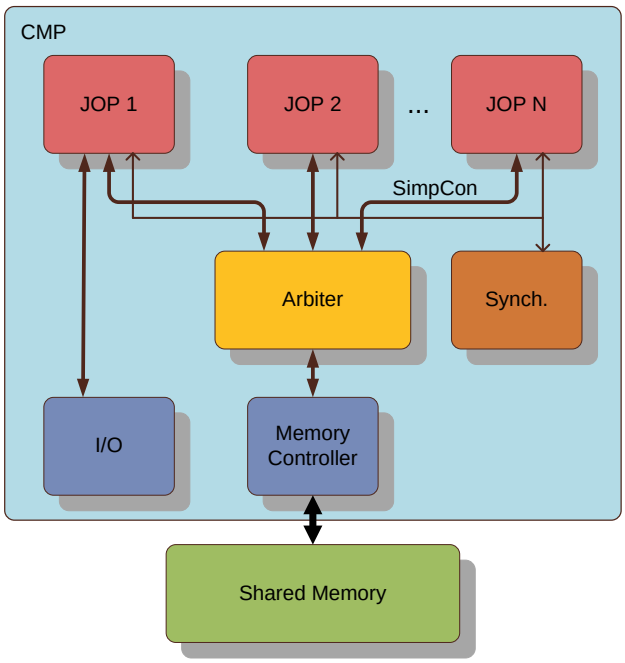

Figure 5.1: JopCMP Architecture.

Figure 5.1. JOP is an implementation of the Java Virtual Machine (JVM) in hardware [15, 17]. It features a stack cache for the private data of each thread. Additionally, a kind of instruction cache (called method cache) limits the memory access frequency and increases the processing power. These real-time processing elements perform computations in parallel. Instructions and data are stored in a single shared memory. The interconnection network is responsible for connecting multiple processors with the memory. An arbiter is part of this network and controls the memory access to the shared memory. An SoC bus, called SimpCon [16], is used to connect the processing cores to the arbiter, and the arbiter to the shared memory. We consider the synchronization of shared data as a further major subsystem of an SMP. It is responsible for coordinating access to shared objects. A detailed description of the JopCMP architecture can be found in [10].

5.3.1 Arbitration Challenge

The arbitration of a time-predictable CMP with a shared memory can be divided into two closely coupled challenges:

- Control of the shared memory access
- Timing analysis of the memory access

The arbiter is responsible for partitioning the memory access bandwidth between the CPUs in the system. It controls the memory access of multiple CPUs to the shared memory. Naturally, if one CPU is accessing the shared memory, no other CPU can simultaneously access the memory. They are forced to wait until the CPU has completed its memory transfer. The arbitration unit takes the supervisory and control role for the shared access. Two different arbitration policies exist: dynamic and static arbitration approaches.

A dynamic arbitration policy resolves simultaneous access at runtime. The fixed priority policy is an example. Each CPU of the system is assigned a unique priority. If memory access contention occurs, the CPU with the highest priority will be granted access to the memory. All other CPUs will have to wait. This arbitration algorithm implements a dynamic decision scheme depending on the CPU priorities.

The static arbitration policy strictly defines the access pattern before runtime. Consequently, no arbitration is necessary during execution time. This policy is typical for real-time systems because it provides information for the timing analysis. An example of this policy is the time division multiple access (TDMA) scheme.

What is the problem with time predictability of the CMP? In uniprocessor systems, only one processor accesses the memory and we can predict the WCET of a memory access. However, tasks running on different CPUs influence each others' execution times when they access a shared resource [19], i.e. a shared memory. Therefore, we want to remove the interdependencies between the execution times of tasks. If we already know the pattern of how another task of a CPU accesses the memory, this will make the arbitration and the analysis a lot easier (compare the work of Pitter and Schoeberl in [9]). Usually, we cannot exactly predict the memory access pattern of multiple CPUs. Therefore, we need an arbitration algorithm that is able to bound the WCET of a task running on a CPU, even though tasks executing on other CPUs may also access the main memory. Consequently, the analysis of WCETs is possible.

According to [1, 11, 13], a TDMA based policy guarantees a constant bandwidth to each processor. We agree that this arbitration policy is well suited for time predictability in multiprocessor systems with shared resources. Each processor is assigned a predefined part of the bandwidth. The easiest solution to implement this idea is the division of time. Consequently, each CPU gets an allocated time slot for accessing the shared memory.

We implemented a TDMA arbiter in the hardware description language VHDL. It is fully configurable concerning the number of CPUs and the size of the time slot. Hence, we are capable of running JopCMP with a TDMA arbiter in an FPGA development board. It contains a Cyclone-I FPGA

(EP1C12) from Altera and a 1 MB, 15 ns SRAM that provides the main memory of the system. This prototyping hardware is used for experimental measurements of execution times of tasks, as described in Section 5.5.

5.4 WCET Analysis

This section starts with a short introduction of the static WCET analysis of JOP. The remaining sections describe the WCET analysis approach of JopCMP.

5.4.1 Static WCET Analysis based on JOP

Real-time processors like JOP have simpler and less powerful architectures than modern CPUs. Several advanced features that increase the average-case performance (i.e. data caches, out-of-order execution, and branch prediction) are disregarded [17]. Although these methods speedup the execution of programs, they impede the predictability of the timing behavior because the WCET depends on the execution history. Hard real-time processors like JOP benefit from a hardware model that assigns an accurate execution time to each machine instruction.

Using JOP's WCET analysis tool [18], the WCET result of a task can be obtained. A Java program is compiled into class files that include the JVM language instructions called bytecodes. For static WCET analysis, the bytecode sequence is transformed into a directed graph of basic blocks called control flow graph (CFG). Each basic block consists of bytecode instructions. JOP translates each bytecode to a microcode or a sequence of microcode instructions that are executed by the processor. Every microcode has a fixed execution time. Hence, each basic block can be assigned an exact execution time.

Furthermore, flow facts have to be added to the Java program code in advance. In general, this is the only way to bound the loops and calculate the frequency of execution of the basic blocks. The CFG including the flow facts and the mapping to the hardware make the WCET analysis possible using the implicit path enumeration technique (IPET) [6].

5.4.2 Multiprocessor WCET Approach

With the use of the TDMA arbitration scheme, the WCET of an arbitrary memory access of a CPU can be calculated using Equation 5.1.

$$WCET_{access} = (n-1) \cdot t_{time\ slot} + t_{access} \tag{5.1}$$

The unit of $WCET_{access}$ is clock cycles. n specifies the number of CPUs in the system. The more CPUs integrated into the system, the longer the WCET of a single memory access. t_{access} describes the memory access time in clock cycles. The width of a time slot ($t_{time\ slot}$) is given in clock cycles, which is configurable. It should be configured as small as possible to reduce the WCET_{access}. The larger the time slot, the higher the WCET_{access} for a single memory access. The minimum size of the time slot is fixed with t_{access}. Otherwise, a processing unit could never successfully access the memory in one time slot.

To find out the overall WCET, the size of the time slot of the arbiter is of major importance. The size of the time slot is dependent on the memory access time. Usually, processors run with a higher clock frequency than memories. The gap between the processor and the memory frequency is further widening [3]. Therefore, the usage of caches is a common approach to reduce the memory access frequency. Nevertheless, the problem with the memory access cannot be circumvented. If the memory access time cannot keep up with the CPU frequency, the processor stalls until the data is available. This delay can be referred to as wait states, which has a large impact on the WCET. A large number for each memory access will degrade the performance of the system.

In our approach, we analyze the bytecodes' WCETs that are dependent on:

- Number of JOPs integrated in the CMP
- Size of the time slot
- Memory access time

First, the memory access pattern of each bytecode has to be investigated. The number of JOPs and the size of the time slot have to be defined. This configuration of the system introduces a fixed TDMA memory access scheme where each CPU is assigned a time slot of the TDMA period. Subsequently, all preconditions are accomplished to determine the WCET of each bytecode using the algorithm described in Section 5.4.4. JOP's WCET analysis tool uses the generated bytecode estimates to calculate the WCET of the Java source code.

5.4.3 Bytecode Memory Access Pattern

JOP translates most of the bytecodes to its native microcode instructions. Each bytecode is composed of a microcode instruction or a series of microcode instructions. Some bytecodes are actually implemented in hardware. A couple of bytecodes are implemented in Java. The timing analysis

Type	Bytecode	Memory Area
const	ldc, ldc_w, ldc2_w	Method area
get	getfield, getstatic	Heap
put	putfield, putstatic	Heap
array	aaload, aastore, baload, bastore, caload, castore, daload, dastore, faload, fastore, iaload, iastore, laload, lastore, saload, sastore, arraylength	Heap
call	invokeinterface, invokespecial, invokestatic, invokevirtual	Method area
return	areturn, dreturn, freturn, ireturn, lreturn, return	Method area
new	anewarray, multianewarray, new, newarray	Heap
switch	lookupswitch, tableswitch	Method Area
cast	checkcast, instanceof	Heap

Table 5.1: Bytecodes accessing the shared memory.

of these bytecodes is not part of this work because they have to be analyzed like general Java source code.

According to the JVM specification [7], the heap and the method area are shared data areas, whereas the stack is a private data area for each thread. In JOP, the heap and the method area are located in the main memory. Consequently, all bytecodes that work on these areas have to be carefully examined. Some bytecodes access the memory several times, some only once. Hence, it makes sense to have a closer look at the different instructions. Table 5.1 summarizes the bytecodes that access the main memory. As stated before, some bytecodes are implemented in Java, i.e. bytecodes of type NEW, SWITCH and CAST. Therefore, they are disregarded in the proposed analysis.

Most memory access patterns of the bytecodes can be statically analyzed; i.e. bytecodes that access the heap and those of type CONST. The pattern is only dependent on the memory access time. If the memory access time is known, the memory access pattern of the bytecodes can be analyzed regardless of the source code of the program. An example of such a bytecode is *ldc*, which pushes a single word constant onto the stack. Therefore, only one memory access to the method area is needed. JOP translates this bytecode into a series of microcodes. If the memory access time is known, the memory access pattern can be specified using JOP's bytecode implementation. Another example is *iaload*, which is implemented in hardware. For the analysis of the memory access pattern, we determine the VHDL implementation in

combination with ModelSim simulations.

The memory access patterns of the bytecodes of type CALL and RETURN need a dynamic analysis and are more difficult to attain. Each JOP is equipped with an instruction cache that caches complete Java methods [14]. Consequently, the memory access patterns of these bytecodes vary, depending on the history of the execution. If the method is already in the cache, no additional memory accesses are needed to load the method into the cache. If a cache miss occurs, JOP will have to load the whole method into the cache. Depending on a cache hit or a cache miss and the length of the method that has to be loaded, the access pattern has to be created for each individual occurrence of the bytecode in the source code. Therefore, we integrated the generation of the patterns into the WCET analysis tool where the cache information is available. Again, JOP's microcodes and ModelSim simulations make it possible to analyze the timing of the memory accesses.

5.4.4 WCET Analysis of Bytecodes

Listing 5.1 shows a simplified version of the algorithm to find the WCET of the bytecodes. The inner loop of this algorithm calculates the execution time of the *bytecode* starting at *position*. The *bytecode* describes the memory access pattern of the instruction. It has a predefined length and each element contains either a READ/WRITE request or a NOP (no memory access). If the indexed element is a NOP, the execution time illustrated by *execTime* will be advanced. Additionally, the variable *position* is increased by 1, which defines the position of the *tdmaPattern* array. If the element of the bytecode equals either a READ or a WRITE access, the *tdmaPattern* will decide whether this CPU will be allowed to access the memory. In case another CPU is on turn to access the memory (the element equals to 0), *execTime* and *position* are advanced until the CPU is allowed access again. The WRITE case of the switch-case statement is similar to the READ case and is therefore omitted from the listing.

The outer loop changes the starting position of the calculation in each iteration. The constant TDMA_PERIOD is defined by the multiplication of the number of CPUs by the size of the time slot. Each iteration of this loop calculates an execution time value of the bytecode. If the new *execTime* is greater than the current worst-case execution time, it will be assigned to *wcet*. Hence, the resulting WCET of the bytecode depending on the number of CPUs and the size of the time slot is available after the last iteration.

Listing 5.1: Algorithm to find the WCET of the bytecodes.

```
int wcet=0;

for(i=0;i<TDMA_PERIOD;i++){
    execTime=0;
    position=i;

    for(j=0;j<bytecode.length;j++){
        switch(bytecode[j]){
            case NOP:
                execTime++;
                position++;
                break;

            case READ:
                while(tdmaPattern[position]!=1){
                    execTime++;
                    position++;
                }

                execTime++;
                position++;
                break;

            case WRITE:

                ...

                break;
        }
    }
    if(wcet<execTime){
        wcet=execTime;
    }
}
```

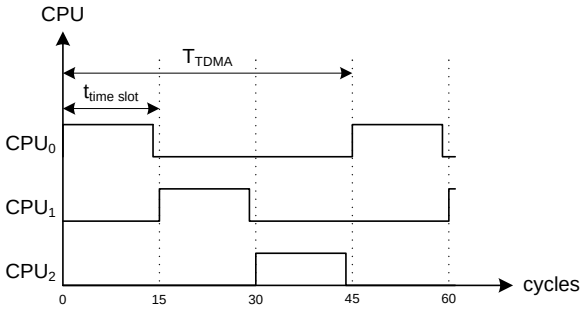

Figure 5.2: Time slots of the CPUs.

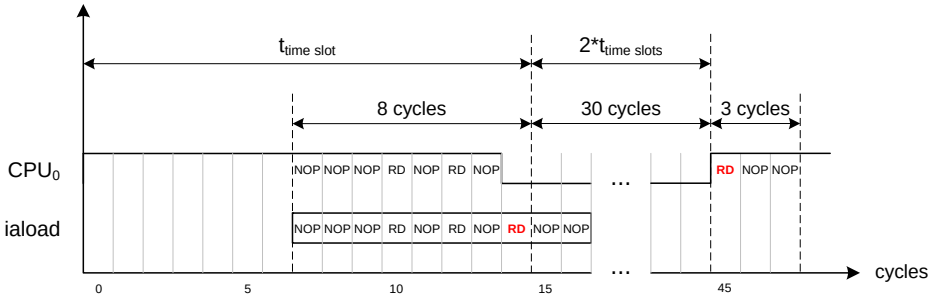

Figure 5.3: WCET calculation of iaload.

WCET Calculation of Bytecode iaload

This simple example exemplifies the calculation of the bytecode *iaload*. JopCMP contains 3 CPUs and the time slot is configured to 15 clock cycles. A read access to the main memory takes 2 cycles. Figure 5.2 shows the TDMA memory access pattern for each CPU. Time slots to access the shared memory are allocated for each processor. It should be noted that a time slot of 15 cycles permits each CPU to access the memory until the 14^{th} cycle. In the 15^{th} cycle, a read access cannot be permitted. Otherwise, we cannot guarantee that the next CPU is able to access the memory in the first cycle of its time slot. (This originates from the pipelining transactions of the SimpCon specification [16]).

We want to evaluate the WCET of the bytecode *iaload*. The following access pattern is given for *iaload* = {NOP, NOP, NOP, RD, NOP, RD, NOP, RD, NOP, NOP}. We can see that *iaload* performs three read accesses to the memory. The WCET is calculated when the index of the outer loop equals to 7. This scenario is shown in Figure 5.3. *iaload* starts with an NOP operation in the 8^{th} cycle of the CPU's time slot. We can see that the 3^{rd} RD access of *iaload* cannot immediately be executed. Therefore, it is delayed two time slots until it is allowed to access the memory. The WCET results

Table 5.2: Java bytecodes and basic blocks of the loop.

Block	Addr.	Bytecode	Cycles	BB Cycles
B1	0:	iconst_0	1	
	1:	istore_3	1	2
B2	2:	iload_3	1	
	3:	iload_0	1	
	4:	if_icmpge	4	6
B3	7:	aload_1	1	
	8:	iload_3	1	
	9:	aload_1	1	
	10:	iload_3	1	
	11:	iaload	41	
	12:	iload_2	1	
	13:	iadd	1	
	14:	iastore	46	
	15:	iinc	8	
	18:	goto 2	4	105

in 41 cycles.

5.4.5 Loop Example

In the following, we systematically analyze the WCET of a simple loop to show how the WCET is calculated. Listing 5.2 shows the source code where a scalar s is added to a vector. This loop is parallelizable because each iteration of the statement in the loop body is independent. Therefore, the loop body could be easily executed on different CPUs in parallel.

Listing 5.2: Simple Loop.

```
for (i=0; i<10; i++) { // @WCA loop=10
    a[i]=a[i]+s;
}
```

As explained before, the WCET estimate for each bytecode accessing the main memory has to be calculated depending on the configuration of the CMP. For this example, JopCMP consists of 3 CPUs and the time slot for each CPU is specified as 15 clock cycles. Consequently, one TDMA period is 45 cycles. The bytecodes and basic blocks of the example, as generated by the WCET analysis tool, are shown in Table 5.2. The fourth column

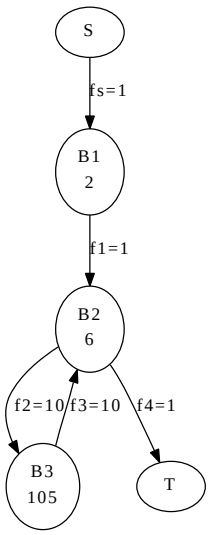

Figure 5.4: Control flow graph of the simple loop.

presents the execution time in clock cycles for each bytecode, and the fifth column gives the execution time for each basic block. If we compare the bytecodes with Table 5.1, only *iaload* and *iastore* access the main memory. Therefore, their WCETs are dependent on the configuration of the system.

The CFG, illustrated in Figure 5.4, is constructed from the basic blocks. The vertices represent the basic blocks labeled with their name and execution time. All edges are labeled with an execution frequency. Hence, the WCET can be calculated and the result is 1118 cycles.

We can also measure the execution time of this simple example by running the JopCMP on the FPGA development board described in Section 5.3. The measured execution time of the loop example results in 987 cycles. This result and the analytical WCET estimate diverge slightly. This overestimation of the analytical result is not surprising because the analysis always takes the WCET for the bytecodes *iaload* and *iastore* into account. In the measurement, some array accesses are executed in fewer clock cycles than the worst case.

5.5 Results

We use the simple loop as an example to find the WCET estimates for varying system configurations. In Table 5.3, the analyzed WCETs of different system configurations are shown. The number of CPUs is varied between 1 and 32. Additionally, the size of the time slot changes between 3 and 48 clock cycles. The first row of the results shows that the WCET of the single JOP is 488

Table 5.3: Analyzed WCET of the loop example depending on the system configuration.

Configuration # of CPUs	Time Slot (cycles)	*Analyzed* WCET (cycles)
1	–	488
2	3	708
2	4	668
2	5	728
2	6	708
2	7	728
2	8	768
2	9	698
2	10	718
2	11	738
2	12	758
2	24	998
2	48	1478
4	3	1068
4	6	1068
4	12	1238
4	24	1958
4	48	3398
8	3	1788
8	6	1788
8	12	2198
8	24	3878
8	48	7238
16	3	3228
16	6	3228
16	12	4118
16	24	7718
16	48	14918
32	3	6108
32	6	6108
32	12	7958
32	24	15398
32	48	30278

Table 5.4: Analyzed WCET and measured execution time of the loop example.

Configuration # of CPUs	Time Slot (cycles)	Analyzed WCET (cycles)	Measured Exec. (cycles)
1	–	488	488
2	3	708	670
2	6	708	666
2	9	698	554
2	12	758	716
2	24	998	502
2	48	1478	931
3	3	888	838
3	6	888	747
3	9	878	815
3	12	998	735
3	24	1478	765
3	48	2438	1410

cycles. No number is given for the time slot of this configuration because a single JOP does not have to share the main memory bandwidth.

The second part of the table shows the results of the dual-core JopCMP with varying time slot sizes. In general the WCET increases continuously with larger time slot sizes. The systems with more CPUs show a similar behavior concerning the size of the time slot. When comparing the analyzed WCET results of the 2-way JopCMP, both systems with the time slot equal to 3 and 6 cycles have the same WCET. Only the two bytecodes *iaload* and *iastore* of the loop access the main memory. The execution times of all other bytecodes of the loop are not affected by the size of the time slot. The configuration with the time slot equal to 3 results in a WCET of 22 cycles for *iaload* and 24 cycles for *iastore*. If the time slot is 6 cycles, the WCETs of *iaload*=17 and *iastore*=29 cycles. We can see that the sums of the two WCETs of the bytecodes are the same, independent of whether the size of the time slot equals 3 or 6 cycles. Consequently, the execution times of both loop bodies are identical. Note that the configuration with a 4-cycle time slot results in the lowest execution time. In this configuration, the sum of *iaload* and *iastore* is minimized.

The more CPUs integrated into the system, the longer the WCETs. The configuration with 4 CPUs and 6-cycle time slot results in a WCET of 1068

cycles. *iaload* executes in 29 cycles and *iastore* in 53 cycles. Even though, the number of CPUs in the system is doubled, the WCET only increases by 51% compared to the 2-way CMP with the same time slot value.

Table 5.4 compares the measured execution time and the analyzed WCET of the simple loop example. One can see that the measured execution time and the WCET estimate are equal for a single JOP system. This originates from the characteristics of the example. The WCET result and the measured execution time are the same, because only one execution path exists in the code. Furthermore, one can see that the WCET estimates are tight for CMP versions with minor time slots. The configurations with a slot size of 9 cycles result in the lowest WCETs. The measured execution times vary greatly depending on the time slot. It results in 716 cycles for the 2-way CMP with a 12-cycle time slot. The same system with a slot size equal to 24 cycles executes in only 502 cycles. This result shows that *iaload* and *iastore* are frequently executed in only one time slot in the configuration with 24 cycles. In the worst case, they need two time slots to execute, which explains the large difference between the measured and the analyzed execution time.

Furthermore, we used a benchmark called Lift as another example to calculate some WCETs. Lift is a real-world example with an industrial background. This embedded application is a lift controller used in an automation factory. It is part of the embedded Java benchmark suite called JavaBenchEmbedded, as described in [15]. Table 5.5 shows that the WCET of a single JOP results in 8689 cycles. Each CPU of a dual-core JopCMP with a time slot size of three cycles executes the Lift benchmark in 12391 cycles in the worst case. Therefore, the WCET increases only by 43%. The tri-core CMP version experiences an increase of 83% in the execution time compared to the single JOP. Whereas one JOP executes Lift only once, the CMP configuration executes the benchmark three times in parallel.

The last column of Table 5.5 illustrates the pessimism of the WCET analysis. Even though we cannot pretend to measure the WCET, the pessimism ratio gives us an idea of the quality of our analyzed results. One can see that there is not much difference between the measurement and the analysis of the single JOP and the 2-way CMP version with a reasonable size of the time slot. Nevertheless, the conservatism of the analysis does not increase greatly with three JOPs. Unfortunately, we are not able to integrate more than 3 JOP cores into the available Cyclone-I FPGA. Therefore, no values of the measured execution times and corresponding pessimism ratios are available for the 4-way CMP. We can see that the pessimism is in an acceptable range for a multiprocessor WCET analysis.

Table 5.5: Analyzed WCET and measured execution time of the Lift benchmark.

Configuration # of CPUs	Time Slot (cycles)	Analyzed WCET (cycles)	Measured Exec. (cycles)	Pessimism (Ratio)
1	–	8689	5830	1.49
2	3	12391	7587	1.63
2	6	12580	7305	1.72
2	9	13621	7623	1.79
2	12	14867	8131	1.83
2	24	19391	8409	2.31
2	48	28618	9499	3.01
3	3	15872	9234	1.72
3	6	16186	8721	1.86
3	9	18072	9720	1.86
3	12	20456	11097	1.84
3	24	29735	12489	2.38
3	48	48106	14211	3.36
4	3	18827	–	–
4	6	19876	–	–
4	9	22621	–	–
4	12	26171	–	–
4	24	40079	–	–
4	48	67594	–	–

5.6 Conclusion and Future Work

In this paper, we have described the extension of JOP's WCET tool for a homogeneous multiprocessor with a shared memory. The key component of real-time analysis of JopCMP is a TDMA arbiter that divides the memory access bandwidth into equal shares. Hence, we can analyze the WCET of Java bytecodes depending on the size of the time slot, number of CPUs in the system and the memory access time. They are used in the WCET analysis tool to calculate WCET estimates. A simple loop example is presented and WCET estimates are compared to real measurement results.

In the future, we will conduct extensive experiments using more benchmarks to investigate the frequencies of the bytecodes that access the main memory. The applications determine the size of the time slot to achieve tight WCET results. It is defined to be short in size if single memory accesses are dominant in the application code. A medium slot size is the solution for frequent field and array accesses and a large time slot size for predominant method cache load accesses. Consequently, we will be able to better justify, whether to define the size of the time slot larger or smaller. Furthermore, we want to extend the solution and present an idea how tighter WCET estimates can be obtained, in order to increase the accuracy of the analysis.

We will investigate the use of a percentage-based arbitration of the available memory access bandwidth. Hence, the memory bandwidth per CPU will be adjusted, dependent on the workload of the multiple CPUs. For example, if CPU_0 needs 60% of the available memory bandwidth for example, it will receive 60% of the bandwidth share or the time slots accordingly.

Acknowledgement

The research leading to these results has received funding from the Austrian Research Programme FIT-IT under contract number 813039 (TPCM).

Bibliography

[1] Alexandru Andrei, Petru Eles, Zebo Peng, and Jakob Rosen. Predictable implementation of real-time applications on multiprocessor systems-on-chip. In *VLSI Design*, pages 103–110. IEEE Computer Society, 2008.

[2] Andreas Ermedahl and Jakob Engblom. Execution time analysis for embedded real-time systems. pages 35.1 – 35.17. Chapman & Hall/CRC - Taylor and Francis Group, August 2007.

[3] John Hennessy and David Patterson. *Computer Architecture: A Quantitative Approach, 4th ed.* Morgan Kaufmann Publishers, 2006.

[4] Mathai Joseph and Paritosh K. Pandya. Finding response times in a real-time system. *Comput. J*, 29(5):390–395, 1986.

[5] H. Kopetz. *Real-time systems: design principles for distributed embedded applications*. Kluwer Academic Publishers, 1997.

[6] Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In *Workshop on Languages, Compilers, & Tools for Real-Time Systems*, pages 88–98, 1995.

[7] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, USA, second edition, 1999.

[8] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973.

[9] Christof Pitter and Martin Schoeberl. Time predictable CPU and DMA shared memory access. In *International Conference on Field-Programmable Logic and its Applications (FPL 2007)*, Amsterdam, Netherlands, August 2007.

[10] Christof Pitter and Martin Schoeberl. Performance Evaluation of a Java Chip-Multiprocessor. In *Proceedings of the IEEE Third Symposium on Industrial Embedded Systems (SIES 2008)*, Montpellier, France, June 2008.

[11] F. Poletti, D. Bertozzi, L. Benini, and A. Bogliolo. Performance Analysis of Arbitration Policies for SoC Communication Architectures. *Design Automation for Embedded Systems*, 8:189–210(22), 200306/09.

[12] Peter Puschner and Alan Burns. A review of worst-case execution-time analysis. *Journal of Real-Time Systems*, 18(2/3):115–128, May 2000.

[13] Jakob Rosen, Alexandru Andrei, Petru Eles, and Zebo Peng. Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip. In *RTSS*, pages 49–60. IEEE Computer Society, 2007.

[14] Martin Schoeberl. A time predictable instruction cache for a Java processor. In *On the Move to Meaningful Internet Systems 2004: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2004)*, volume 3292 of *LNCS*, pages 371–382, Agia Napa, Cyprus, October 2004. Springer.

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

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

[17] Martin Schoeberl. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture*, 54/1–2:265–286, 2008.

[18] Martin Schoeberl and Rasmus Pedersen. WCET analysis for a Java processor. In *Proceedings of the Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2006)*, pages 202–211, Paris, France, October 2006. ACM.

[19] Lothar Thiele and Reinhard Wilhelm. Design for timing predictability. *Real-Time Systems*, 28(2-3):157–177, 2004.

[20] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David B. Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst*, 7(3), 2008.

[21] Wayne Wolf. *High-Performance Embedded Computing: Architectures, Applications, and Methodologies.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

6

Further Analysis and Evaluation

This chapter is based on a submitted paper called *A Real-Time Java Chip-Multiprocessor*. It provides a coherent view of three different arbitration policies described in previous chapters for use in a CMP with shared memory. This chapter analyzes and evaluates different arbitration configurations with respect to WCET and average-case performance. In comparison to Chapter 5, configurations with more processors are evaluated.

6.1 Memory Arbitration Revisited

The memory arbitration of a real-time CMP with a shared memory presents a number of closely related challenges:

- Synchronization of memory access
- Timing analysis of memory access
- Zero-cycle arbitration
- Scalability with the number of CPUs
- Support of burst mode memory access

The arbiter controls the memory access of multiple CPUs to the shared memory. Naturally, if one CPU is accessing the memory, no other CPU

can access it concurrently. It is forced to wait until the CPU on turn has completed its memory transfer. In this case, a memory arbiter resolves this possible parallel access by serializing the CPUs' read and write operations.

Two different arbitration policies exist: the dynamic and the static arbitration approach. A dynamic arbitration policy resolves concurrent access of two or more CPUs at runtime. An example is a priority-based arbitration scheme where each CPU in the system is assigned a unique priority. If memory access contention occurs, the CPU with the highest priority will be granted access to the memory and all other CPUs will have to wait.

The static arbitration policy defines the access pattern before runtime. Consequently, no arbitration is necessary during execution time. Implementation of this policy is typical for real-time systems to prevent memory access contention. Each CPU has an a priori allocated time to perform its operations on the memory.

In uniprocessor systems, only one processor accesses the memory and the WCET of a memory access can be predicted. However, tasks running on a CMP on different CPUs influence each others' execution times when accessing a shared resource [10], e.g. a shared memory. Therefore, a removal of the interdependencies between task execution times is the stated objective. As described in Chapter 2, a fixed memory access pattern of an application task, e.g. a DMA unit, simplifies the arbitration and timing analysis. Usually, memory access patterns of multiple CPUs cannot be accurately predicted. Therefore, an arbitration algorithm is needed that enables WECT analysis of a task running on a CPU, even though tasks executing on other CPUs may also access the main memory. Consequently, an analysis of bounded WCETs is possible.

All described arbiter designs perform an arbitration decision in the same cycle the request arrives. No additional cycle is lost for arbitration and memory access latency is not affected. Subsequently, memory access time is reduced and the bandwidth increases considerably. Furthermore, arbiters can be configured for variable numbers of CPUs. The CMP system is thus customized to the application needs.

Typically, CPUs access a memory either using single read/write accesses or a burst access mode. Single read or write accesses are used to exchange data and instructions. The burst access mode ensures fast data transmission of consecutive chunks of memory. This feature has become state-of-the-art to load cache memories. Therefore, memory arbiters have to support the burst mode to access a memory.

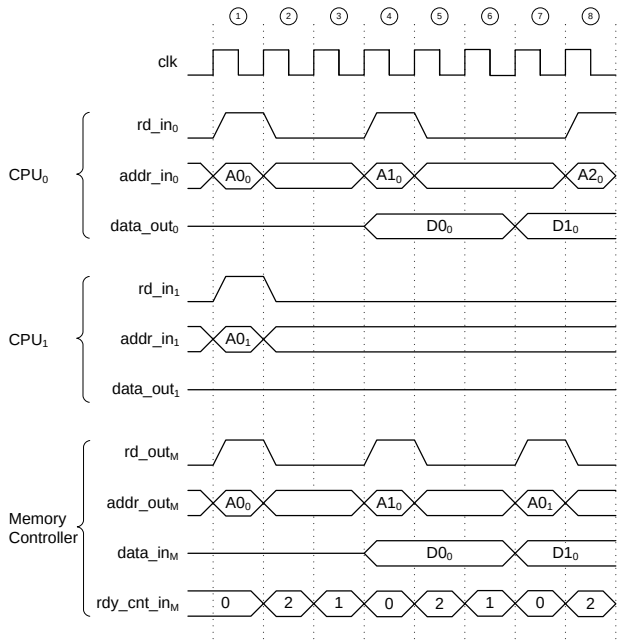

Figure 6.1: Memory access arbitration of the fixed priority arbiter.

6.1.1 Fixed Priority Arbiter

The fixed priority arbitration policy is a typical example of a dynamic arbitration scheme. Each CPU in the system is assigned a unique CPU identity, hereinafter referred to as CPU_{ID}. This CPU_{ID} establishes priority for each CPU. The CPU with the lowest CPU_{ID} has top priority to access the shared memory. The memory arbiter solves simultaneous memory accesses by determining an access priority order.

In Figure 6.1, an arbitration scenario of a 2-way CMP with a memory access time of 2 cycles is shown. The cycle numbers are specified at the top of the figure. All depicted signals are either input or output signals of the arbiter, illustrated by the signals' names. Furthermore, the subscripts indicate whether the signals belong to a specific CPU (denoted by the CPU_{ID}) or to the memory controller. Some SimpCon signals are disregarded in Figure 6.1, e.g. the signals for write access.

At the first clock cycle, both CPU_0 and CPU_1 want to perform a read access to the shared memory. CPU_0 is immediately granted access given that the memory is idle ($rdy\_cnt\_in_M$ equals to 0) because it has a higher priority than CPU_1. Consequently, the read enable signal of the memory (rd_out_M) is driven high and the memory address ($addr\_out_M$) is asserted. The read request of CPU_1 is registered in the arbiter. It has to wait until CPU_0 has finished accessing the memory, indicated by the value 0 of signal

$rdy\_cnt\_in_M$ and no further request of CPU_0 is pending. At the fourth cycle, CPU_1's registered request has to wait again until completion of another memory access of CPU_0. At the seventh cycle, CPU_0's data is available and the registered memory access of CPU_1 is processed. At the eighth cycle, CPU_0 wants to access the memory again. This read access is registered in the arbiter and will be performed after CPU_1 has completed its memory access.

The fixed priority arbiter has been used for a WCET analyzable configuration of a single CPU and a DMA device [6]. The DMA device, e.g. a graphics controller, performs a regular memory access within a short period of time and is assigned top priority.

6.1.2 Fair Arbiter

The fair arbiter implements an arbitration policy that guarantees fairness among the CPUs accessing the shared memory. Memory access can be either a pipelined or a single access. Furthermore, starvation of any CPU is prohibited. Each CPU in the system is assigned a unique CPU identity (CPU_{ID}), starting from 0 up to the number of CPUs-1. Our fair arbitration policy uses a wrapping counter. It changes the permission, which CPU is allowed to access the memory at a given time. The value of the counter has the same range as the CPU identity. As soon as the preceding memory access is complete, the counter is advanced by one. If the new counter value is the same as a requesting CPU_{ID} and the memory is ready to execute a memory access, memory access will be processed and the current counter value remains the same until the data transmission has finished. If the counter shows a CPU_{ID} that does not want to access the memory, the counter is immediately advanced.

Figure 6.2 shows an arbitration scenario of a 2-way CMP system with a memory access time of 2 cycles. The cycle numbers are specified at the top of the figure. The signals *clk* and *counter* are internal signals of the arbiter. All other signals are either input or output signals of the arbiter, as indicated by their names. Furthermore, the subscripts indicate whether signals belong to a specific CPU (denoted by the CPU_{ID}) or to the memory controller.

At the first clock cycle, both CPU_0 and CPU_1 want to simultaneously perform a read access to the shared memory. CPU_0 is immediately allowed to perform the read access because the counter's value is 0 and the memory is idle ($rdy\_cnt\_in_M$ equals to 0). Consequently, the read enable signal of the memory (rd_out_M) is driven high and the memory address ($addr\_out_M$) is asserted. The read request of CPU_1 is registered in the arbiter. It has to wait until CPU_0 has finished accessing the memory, as indicated by the value 0 of signal $rdy\_cnt\_in_M$ and, accordingly, by the received data on

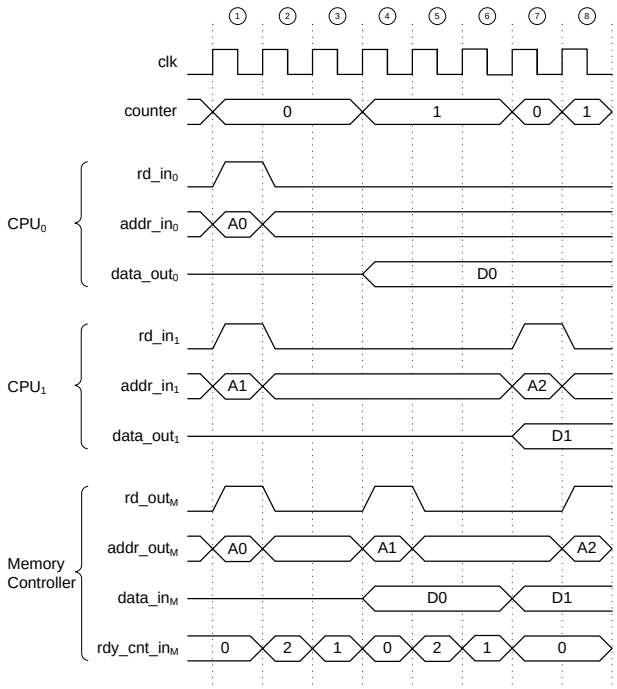

Figure 6.2: Memory access arbitration of the fair arbiter.

$data\_in_M$ and $data\_out_0$. At the fourth cycle the memory access has been completed, the counter increments by one and the registered memory access of CPU_1 is processed. At the seventh cycle the data is available, the counter shows a 0 value. Consequently, CPU_1 is not granted access. As opposed to CPU_1, CPU_0 does not request a memory access. The counter is therefore advanced at the eighth cycle and the registered memory access of CPU_1 is processed.

The more CPUs are part of the system the higher is the probability that the counter matches a CPU_{ID} with a pending memory request after a successful access. Therefore, a high workload will result in a saturation of the memory bandwidth. In case of low competition among several CPUs, this scheme wastes memory bandwidth (and performance) because delays without any memory access can occur.

6.1.3 Time-sliced Arbiter

According to [2, 7, 8], a time-sliced or time division multiple access (TDMA) arbitration policy guarantees constant bandwidth for each processor. Each processor is assigned a predefined part of the bandwidth, which is mapped to an appropriate time slot, so each CPU has an a priori allocated time to per-

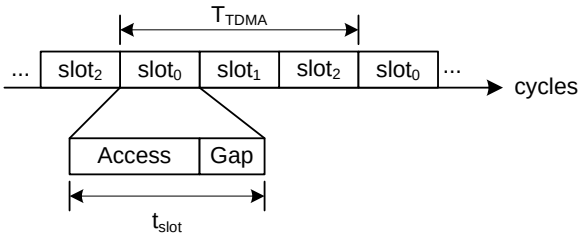

Figure 6.3: TDMA period consisting of three time slots.

form its operations on the memory. The author agrees that this arbitration policy is suitable for timing analysis of multiprocessor systems with shared resources. The arbitration configuration provides the needed information for timing analysis.

Figure 6.3 shows the TDMA memory access pattern for a CMP system with 3 CPUs. Each CPU is allocated a time slot to access the shared memory in every TDMA period. This time slot, configured to a predefined number of clock cycles, is divided into an access time and an access gap. Memory operations of the corresponding CPU can only be started during access time. During the gap segment, the CPU is not allowed to start a memory access because a switch to the next CPU is necessary. This gap permits the next CPU on turn to access the shared memory in the first cycle of its time slot, because every memory access requested in the previous time slot is reliably completed. The size of the gap is depends on the memory access time. The larger the memory access time, the larger the gap.

6.2 Timing Analysis

This section describes WCET analysis of a CMP system using the specified memory arbiters. Furthermore, WCET results are compared to measured execution times and a WCET performance prospect is given.

6.2.1 Fixed Priority Arbitration Approach

Common for all arbitration approaches is the fact that the WCET of a single memory access is the sum of two parts. One part represents the maximum waiting time before the memory access can be executed. The other part represents the CPU's memory access time without any memory contention. WCET results are given in clock cycles.

The fixed priority arbitration policy assigns a unique priority to each CPU. If memory access contention occurs, the CPU with the highest priority will

be granted access to the memory. Using this arbitration policy, the WCET of a memory request of the highest priority CPU, indicated by the subscript 0, can be calculated thus:

$$WCET_0 = \max_{\forall i \neq 0} \{t_{WCET_i} - 1\} + t_0 \tag{6.1}$$

whereby t_0 denotes the memory access time of CPU_0. The other part of the equation represents the maximum waiting time. Let i be a variable that can take any number between 1 up to the number of CPUs-1 and t_{WCET_i} be the maximum duration of all possible memory access types of CPU_i. On the one hand, this variable can represent a single memory access but on the other hand, it can account for a full method load into the method cache. In the worst possible scenario, one or more CPUs in the system request a memory access one cycle earlier than CPU_0. Therefore, CPU_0 has to wait $\max\{t_{WCET_i} - 1\}$ cycles until it can read from or write to the memory. Consequently, the WCET of a single memory access of the highest priority CPU is the load time of the longest method of all lower priority CPUs added to the memory access time of CPU_0.

Calculating the WCET of a lower priority CPU memory access is either rendered impossible or the result represents a very conservative estimate, depending on the number of CPUs. In case of a 3-way CMP, for example, the WCET of the lowest priority CPU cannot be estimated because the higher priority CPUs in the system may prevent that CPU from accessing the memory indefinitely.

A fixed priority arbiter can be used for systems that execute hard real-time tasks on the top priority CPU, and tasks with non-critical timing requirements on all other CPUs.

6.2.2 Fair Arbitration Approach

The fair arbiter implements a fair access to the shared memory for all CPUs of the CMP. This policy avoids starvation of a CPU. The WCET of a memory access by an individual CPU can be calculated using Equation 6.2.

$$WCET_j = \sum_{\forall i \neq j} t_{WCET_i} + t_j \tag{6.2}$$

As in the case of the fixed priority CPU, t_{WCET_i} is the WCET of all memory access types of CPU_i. Again, this variable can be either a single memory access or a full method load. In the case of a CPU method load, the internal counter of the arbiter is stopped until the full method load has been completed. After that, the counter is advanced and the next CPU is allowed to

access the memory. The worst-case scenario for a single CPU memory access can be estimated to be the load time of the longest method of each CPU until the CPU can access the shared memory.

6.2.3 Time-sliced Arbitration Approach

The TDMA arbitration policy defines a static memory access pattern. Each CPU is assigned an allocated time slot. Using the TDMA arbitration scheme, the WCET of a single memory access from an individual CPU can be calculated with Equation 6.3:

$$WCET_j = (t_{gap} - 1) + (n - 1) \cdot t_{slot} + t_j \tag{6.3}$$

whereby n specifies the number of CPUs in the system, and t_{slot} defines the size of the time slot in clock cycles. t_j describes the memory access time of CPU_j. In the worst-case scenario, CPU_j wants to access a memory in the first cycle of the gap segment (t_{gap}) of its own time slot.

The part describing this scenario has not been described in detail by the Equation 5.1 in Chapter 5. Nevertheless, it has been considered in the WCET analysis, as demonstrated by the code in Listing 5.1 and its description of the algorithm.

The WCET of a single memory access increases with the number of CPUs in the system. Moreover, the size of the time slot of the arbiter is of major importance. The minimum time slot size is predetermined and depends on the memory access time. Otherwise, a processing unit could never successfully access the memory within one time slot.

Applying Equation 6.3 to individual instances of memory access results in conservative bytecode WCET bounds. The novel method presented in Section 5.4 calculates the WCET for complete bytecode instructions instead of analyzing the WCET of a single memory access. The bytecode WCETs are dependent on:

- the number of JOPs integrated in the CMP
- time slot size
- memory access time

Assuming all configurations are set and the memory access time is known, the WCET of each bytecode can be determined as described in Section 5.4. JOP's WCET analysis tool uses the generated bytecode estimates to calculate the WCET of a Java application.

Table 6.1: Analyzed WCET and measured execution time of the Lift benchmark.

Configuration		Analyzed	Measured		Pessimism
# CPUs	Time Slot (cycles)	WCET (cycles)	Best Exec. (cycles)	Worst Exec. (cycles)	(Ratio)
1	–	10567	6309	6818	1.55
2	6	18793	8634	10463	1.80
2	12	18417	8472	9663	1.91
2	18	20529	9900	11263	1.82
2	24	22713	9988	11275	2.01
4	6	32001	13604	17579	1.82
4	12	31683	14238	17007	1.86
4	18	37917	17532	20875	1.82
4	24	44505	18604	21415	2.08
8	6	58305	24452	32747	1.78
8	12	58275	27528	33615	1.73
8	18	72693	35100	41755	1.74
8	24	88089	37228	42823	2.06

6.2.4 WCET vs. Measured Execution Time

In Chapter 5, the used hardware only permits an integration of up to 3 JOPs. Therefore, the CMP system using the TDMA arbiter has been prototyped on Altera's Development and Education Board (DE2 Board) with a low-cost Cyclone II (EP2C35) FPGA. This FPGA can be populated with up to 8 JOP cores, each core equipped with a 1 KB stack cache and a 2 KB method cache. The DE2 Board contains 512 KB SRAM connected via a 16-bit data bus. The design is clocked at 90 MHz and the main memory is a 16-bit SRAM with an access time of 4 cycles for a 32-bit read operation, and 6 cycles for a 32-bit write operation.

A benchmark called `Lift` is used to calculate WCETs of a real-world application with an industrial background. This embedded application is a lift controller used in an automation factory. `Lift` is part of an embedded Java benchmark suite called JavaBenchEmbedded, as described in [9].

Table 6.1 compares the measured execution time and the analyzed WCET of the `Lift` benchmark. The first two columns describe the different system configurations. Column 3 shows that the analyzed WCET of `Lift` on a single

JOP results in 10567 cycles. The result of the dual-core JopCMP with a time slot size of 12 cycles is 18417 cycles (worst-case scenario). Even though both CPUs execute the `Lift` benchmark simultaneously, their WCETs increase only by 74%. The 8-way CMP with a 12-cycle time slot experiences an increase of a factor of 5.5 in the worst case compared to the single JOP. Whereas one JOP executes `Lift` only once, the CMP configuration executes the benchmark 8 times concurrently.

Measured execution time results are illustrated in the fourth and fifth columns. Several measurements are carried out for each configuration, so the best case and the worst case are represented in the table. It has to be noted that the measured worst case is probably not the real WCET. As the simulation environment does not cover all data possibilities, there is no guarantee that the path for the real WCET has been triggered. The last column of Table 6.1 illustrates the pessimism of the WCET analysis. It is calculated by dividing the analyzed WCET by the worst measured execution time. The pessimism ratio gives an idea of the quality of our analyzed results. The pessimism of a single CPU is 1.55. The analysis is not a great deal more conservative in configurations with more CPUs and a reasonable time slot size. This slightly WCET increase can be explained as the analysis always takes the WCET of each individual bytecode due to memory arbitration into account. Nevertheless, the author believes that the pessimism in such cases is within an acceptable range for a multiprocessor WCET analysis.

6.2.5 WCET Performance

One interesting issue is how the CMP system scales with respect to the WCET. The goal using the time-predictable CMP system is twofold: (1) to provide a system where safe WCET bounds can be estimated and (2) to enhance the performance by means of multiple cores within the CMP.

The WCET results from Table 6.1 permit an estimation of a WCET performance increase. Executing the `Lift` benchmark simultaneously, increases the WCET on the individual cores, but also the number of iterations that are executed within the whole system. JOP executes `Lift` in 10567 cycles in the worst case. Assuming a 12-cycle time slot, an 8-way CMP needs 58275 cycles for completion of 8 simultaneously executed `Lift` benchmarks. Equation 6.4 calculates the resulting speed-up to be 1.45.

$$Speedup_{CMP} = \frac{WCET_{CPU}}{WCET_{CMP}} \cdot 8 \approx 1.45 \tag{6.4}$$

Figure 6.4 shows the WCET performance through multiprocessing. There is a measurable WCET speed-up for the TDMA arbiter with relatively small

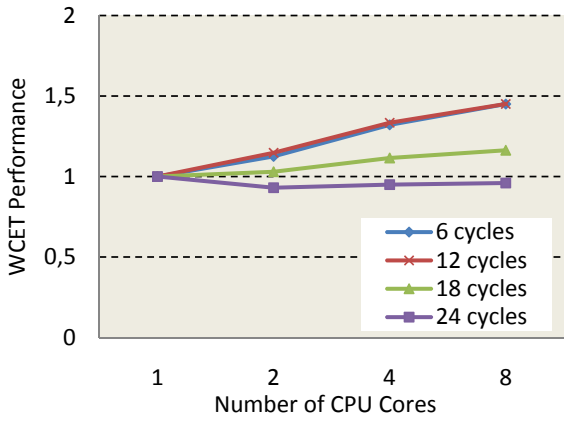

Figure 6.4: WCET performance of the Lift benchmark.

slot sizes. Choosing a larger slot size actually decreases performance. Compared to an average-case performance increase, as shown in the following section, the WCET performance enhancements are moderate.

6.3 Performance Evaluation

Although the CMP is designed for hard real-time systems with time-predictable task execution, the average-case performance is still interesting. Applying three different arbitration schemes, the trade-offs using a time-predictable solution compared to using an average-case optimized CMP system can be explored. The benchmarks highlight that several processors working simultaneously outperform a uniprocessor that executes the same workload in sequence. Again, the FPGA-based platform is used to evaluate different configurations.

6.3.1 Benchmarks

Using a multi-core system, application development is more complex because the application code has to be split up among several processors. Three different benchmarks are used for the CMP evaluation:

- a real-world embedded application in industrial use (`Lift`),
- a matrix multiplication (`MMul`), and
- an embedded TCP/IP stack (`ejip`).

Our benchmark methodology is as follows: `Lift` is executed 10000 times. This workload is distributed evenly among the processors. The benchmarks `MMul` and `ejip` perform an automatic distribution of the workload.

Lift Application

The `Lift` benchmark, introduced in Section 6.2.4, is actually written to measure uniprocessor performance. Nevertheless, it is used for executing several `Lift` tasks on multiple CPUs concurrently. This benchmark thus represents a medium computational, fully parallelized application without any synchronization needs.

Matrix Multiplication

The benchmark `MMul` is designed to give an idea of the performance of a computationally intensive algorithm showing good parallelism potential. The benchmark multiplies two matrices with a dimension measuring 100x100. This calculation results in 1 million multiplication operations. Each row of the resulting matrix is calculated by a single CPU. A synchronization variable secures that the next idle CPU takes the next unsolved row until the desired result is achieved. The benchmark measures the elapsed time for the calculation. `MMul` is classified as a parallel workload – computationally intensive with low synchronization overhead.

Embedded TCP/IP Stack

As an example of an application with several communicating threads, an embedded TCP/IP stack for Java called `ejip` is used. The benchmark explores the possibility of parallelizing the TCP/IP stack. The application that uses the TCP/IP stack is an artificial example of a client thread requesting a service (vector multiplication) from a server thread. That benchmark consists of 5 threads: 3 application threads (client, server, result), and 2 TCP/IP threads executing the link layer as well as the network layer protocol.

6.3.2 Measurements

Table 6.2 shows the measured execution time of `Lift`, running at a frequency of 90 MHz on the Altera DE2 board. The first column gives the number of JOP cores in the system. The results of three different arbiters are shown. Configurations using

- the fixed priority arbiter,

Table 6.2: Performance comparison of different arbiter types using the Lift benchmark.

Number of JOP cores	Fixed Exec. time (ms)	Fair Exec. time (ms)	TDMA Exec. time (ms)
1	702	702	702
2	389	399	469
4	336	292	405
8	340	277	395

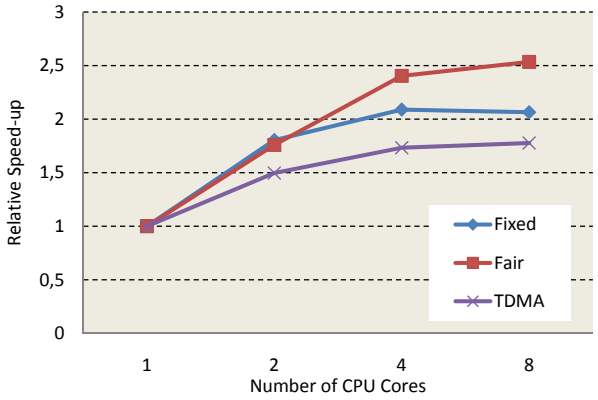

Figure 6.5: Performance comparison of the Lift benchmark using different arbiters.

- the fair arbiter, and
- the time-sliced arbiter with a time slot of 12 cycles.

The execution time is measured for each combination of number of CPUs and arbitration policy. One JOP executes the `Lift` workload in 702 ms and does not have to share memory bandwidth. A dual-core system performs about 1.8 times faster than a single JOP, using a fixed priority or a fair arbiter. The configuration with a TDMA arbiter shows a performance improvement of 50%. A 4-processor system using a fixed arbiter doubles the performance of a single-core. The same system with a fair-based arbiter experiences a speed-up of 2.4. A time-sliced arbiter cannot keep up with these system speed-ups but is still 73% faster than a single JOP. Executing the workload on more than 4 processors does not give a significantly better performance, irrespective of the used arbiter type.

Table 6.3: Performance comparison of different arbiter types using the MMul benchmark.

Number of JOP cores	Fixed Exec. time (ms)	Fair Exec. time (ms)	TDMA Exec. time (ms)
1	839	839	839
2	461	467	556
4	306	315	421
8	305	306	336

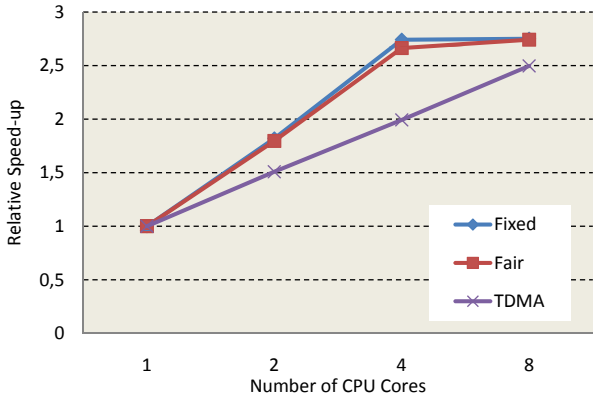

Figure 6.6: Performance comparison of the MMul benchmark using different arbiters.

Figure 6.5 summarizes the performance results. The horizontal axis describes the number of CPUs, the vertical axis illustrates the relative speed-up. The relative speed-up is the relation between the execution time on a single core and a multi-core version. The figure shows that configurations up to 4 cores using different arbiters scale adequately. Using an 8-core CMP with a fixed priority arbiter results in a performance slow-down. The reason is the large competition among the CPUs to access the memory.

Table 6.3 and Figure 6.6 show the measurement results of `MMul`. This computationally intensive algorithm shows a good potential for parallelism. Execution time for the fair arbiter configuration is shorter than reported in Chapter 4, as the benchmark code is optimized to avoid memory access as much as possible. The speed-ups of CMP versions consisting of 2 and 4 cores lived up to our expectations with 1.5 (TDMA) and 1.8 (fixed and fair), respectively. The fixed arbitration policy scales well using up to 4 proces-

Table 6.4: Performance comparison of different arbiter types using the ejip benchmark.

Number of JOP cores	Fixed Exec. time (ms)	Fair Exec. time (ms)	TDMA Exec. time (ms)
1	305	305	305
2	193	196	245
4	125	124	210
8	271	223	334

sor cores. Adding more CPUs to the system does not result in a better performance. Whereas the first four CPUs calculate 97% of the final result altogether, the other CPUs calculate only 3%. Notably, CPU_6 and CPU_7 suffer from starvation because they never get their turn to calculate a single multiplication. The fair arbiter distributes a workload evenly among all the processors. Each CPU contributes to the final result by calculating either 12 or 13%. Nevertheless, 8 cores do not provide a significant speed-up. Also, the TDMA arbiter distributes the workload evenly and is only 10% slower than the fair arbiter using 8 CPUs.

The results for the embedded TCP/IP example `ejip` are shown in Table 6.4. This application, consisting of five communicating threads, scales quite well using up to 4 cores. Applying the fair or the fixed priority arbiter, performance increases by a factor of 2.5. Even the TDMA-based system is about 45% faster than a single core solution. The overheads introduced by 8 cores, with only 5 cores executing threads, leads to a performance decrease compared to the 4-core system shown in Figure 6.7. The performance decrease of configurations using the fair and the TDMA arbitration can be explained as the algorithms allocate memory bandwidth to three CPUs that do not execute any threads.

One bottleneck in the TCP/IP stack is a global buffer pool. All layers communicate via this single pool. The single pool is not an issue for a uniprocessor system, however simultaneously running threads in a CMP system compete more often for pool access. A revised version of the TCP/IP stack would use dedicated queues, preferably non-blocking, single reader/writer queues, for the communication between layers. Furthermore, a finer parallelization within the TCP/IP stack needs to be explored to fully utilize the available CPUs.

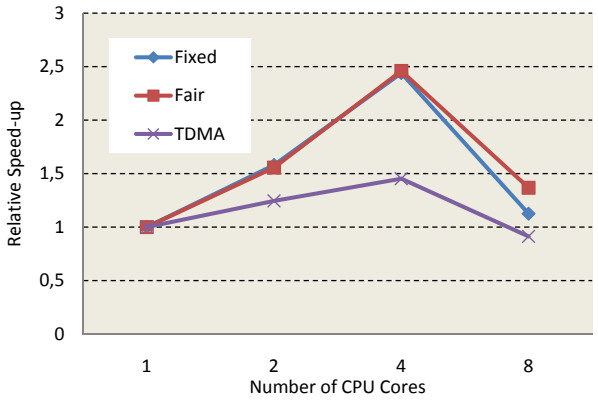

Figure 6.7: Performance comparison of the ejip benchmark using different arbiters.

6.4 Discussion

Three different arbiters were implemented to experiment with different CMP systems, highlight advantages and disadvantages, and find the most practical field of application. Table 6.5 summarizes the differences between various arbitration policies.

6.4.1 Starvation

CMPs using a fair or a TDMA arbiter cannot suffer from starvation because each CPU gets the chance to access the memory. A fixed priority arbiter may cause starvation of CPUs, because higher priority CPUs might access the shared memory first. This situation can occur when more than 2 CPUs are integrated in a CMP. The two highest priority CPUs could alternately access the shared memory. Consequently, the third CPU will never get to access the memory.

6.4.2 Predictability

Even though a timing analysis approach of each arbitration policy is presented in Section 6.2, some of them are not viable for hard real-time systems. The proposed timing analysis approach of a TDMA arbiter enables the WCET calculation of Java bytecodes instead of the WCET for a single memory access. The WCET is dependent on the number of CPUs and the time slot size. It is preferable to keep the time slot short if single memory access is dominant in the application code. A medium time slot size is the

Table 6.5: Comparison of the arbitration policies.

Property	Fixed	Fair	TDMA
Starvation	-	+	+
Predictability	-	+/-	+
Performance	+/-	+	-
Pipelined Split Transaction	-	-	+
CPU-overlapping Pipelining	-	-	+

solution for frequent field and array accesses, a large slot size for predominant cache load accesses. Further experiments are necessary to be able to better define whether to make time slot sizes larger or smaller. Appendix B shows bytecode WCETs of different CMP configurations.

A fair-based arbiter enables timing analysis of all tasks running on different CPUs. Nevertheless, the WCET of a single memory access by a CPU is a very conservative estimate, because a possible method cache load by each CPU has to be taken into account. The resulting WCET values are not feasible. If a method cache load could be split into several transactions, this arbitration could also be a viable solution.

Using the fixed priority arbiter, only the highest priority CPU of a CMP is predictable. WCETs cannot be calculated for programs executing on the other CPUs. Therefore, this arbitration policy can be used for real-time systems where one CPU executes hard real-time and the other ones non-critical tasks.

6.4.3 Performance

Using three benchmarks, the speed-up capability of real-world applications is demonstrated using multiple processors cores. The speed-up increase has lived up to expectations for systems using a fair arbiter. The `Lift` benchmark executes 2.5 times faster on an 8-way CMP than on a single JOP. The performance improvement of CMP systems with a fixed priority arbiter scales well up to 4 cores. When integrating more CPUs, some of them may suffer from starvation using fixed priority arbitration. As expected, the average-case performance of systems using the TDMA arbiter cannot keep up with the performance improvements of other arbiters. Furthermore, experimental results with the `ejip` benchmark show that more cores does not certainly result in a better CMP performance.

Comparing the CMP system to a complex Java processor such as picoJava II, described in Chapter 4, the conclusion is that a multiprocessor version of a simpler and smaller architecture is more efficient (performance/die area) for parallel workloads. With independent instances of the application benchmark `Lift`, an 8-core CMP version is around 1.4 times faster than picoJava with a die area of about 90% of picoJava.

6.4.4 Implementation Features

The arbiters implemented within this research work are fully configurable with respect to the number of connected CPUs. Compared to existing arbiters like AMBA [3] or CoreConnect [4], the maximum number of connected masters is not limited. As a result, the application and the hardware resources determine the quantity of connected processors.

Another advantage over existing arbiters like Avalon [1], or AMBA is the zero-cycle arbitration latency. The synchronization of the shared memory access does not need an extra cycle for its arbitration decision. The synthesis results of Table 4.6 show that clock frequency scales quite well with an increasing number of CPUs. Therefore, an evaluation of an additional pipeline stage has not been conducted.

Additionally, all implemented arbiters feature a burst memory access mode for loading cache memories. Therefore, back-to-back reads of the memory are supported. Compared to other arbiters, the TDMA arbiter supports CPU-overlapping pipelining. This feature allows for a full saturation of the available memory bandwidth.

6.4.5 Real-Time Speed-up of Multicore

The WCET analysis of an application task for a TDMA-based system showed only a moderate speed-up (up to a factor of 1.5 for an 8-core system). A lower increase of real-time performance compared to the increase of average-case performance has been expected. However, the achievable speed-up was less than expected. It has to be noted that the measured workload consisted of independent tasks without communication overheads. This result serves as a foundation for further improvements.

The reduced memory bandwidth calls for more research on time-predictable caching. The WCET analysis tool only considers the leaf nodes of a call tree for the method cache analysis. Tighter bounds on method cache misses will directly pay off for a system with a pressure on memory bandwidth. First experiments with a local cache analysis showed a WCET reduction of 15% using an 8-way configuration [11].

Furthermore, a time-predictable solution for the caching of heap-allocated data will have to be considered. A small, fully associative buffer similar to a victim cache [5] will allow detection of some cache hits in the WCET analysis.

Bibliography

[1] Altera. Avalon Memory-Mapped Interface Specification (v3.3), May 2007.

[2] Alexandru Andrei, Petru Eles, Zebo Peng, and Jakob Rosen. Predictable implementation of real-time applications on multiprocessor systems-on-chip. In *VLSI Design*, pages 103–110. IEEE Computer Society, 2008.

[3] ARM. AMBA Specification (rev 2.0), May 1999.

[4] IBM. 32-Bit OPB Arbiter Core Databook revision 1, March 2007.

[5] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 364–373, Seattle, WA, May 1990.

[6] Christof Pitter and Martin Schoeberl. Time predictable CPU and DMA shared memory access. In *International Conference on Field-Programmable Logic and its Applications (FPL 2007)*, Amsterdam, Netherlands, August 2007.

[7] F. Poletti, D. Bertozzi, L. Benini, and A. Bogliolo. Performance Analysis of Arbitration Policies for SoC Communication Architectures. *Design Automation for Embedded Systems*, 8:189–210(22), 200306/09.

[8] Jakob Rosen, Alexandru Andrei, Petru Eles, and Zebo Peng. Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip. In *RTSS*, pages 49–60. IEEE Computer Society, 2007.

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

[10] Lothar Thiele and Reinhard Wilhelm. Design for timing predictability. *Real-Time Systems*, 28(2-3):157–177, 2004.

[11] Andy Wellings and Martin Schoeberl. Thread-local scope caching for real-time Java. In *Proceedings of the 12th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2009)*, Tokyo, Japan, March 2009. IEEE Computer Society.

7

Conclusion and Outlook

This section starts with a short summary of the major goals described in this thesis. Furthermore, the research course is outlined and the main findings and results are presented. The conclusion will demonstrate the relevance of this thesis to current scientific work and will give some ideas for future research.

7.1 Thesis Goal

The main research goal of this thesis was the design and development of a homogeneous CMP with a shared memory for real-time embedded systems based on Java. Multiple CPUs increase the processing power but the system should remain predictable in the temporal domain. Even though threads execute on different CPUs and access the shared memory, interaction of execution times due to memory access have to be prevented. Consequently, the execution times of different threads can be analyzed separately. Another objective was the development of a static WCET analysis concept with a corresponding tool to provide a fully featured package for real-time Java application development.

7.2 Research Compendium

In the beginning, this thesis explored a new timing analysis approach of a system consisting of one processor and a hardware thread, both accessing a shared memory. Results achieved in this part of research were encouraging, so the design for a chip-multiprocessor system composed of multiple Java processors with shared memory was initiated. The first prototype showed that the proposed architecture operated correctly. Furthermore, first benchmark execution results running on multiple cores supported this research direction. Memory arbitration played the most important role within the multiprocessor system design. It had to control simultaneous access by multiple CPUs to the shared memory. First, a fixed-priority arbiter was implemented. Later on, a fair-based arbiter was introduced to obtain average-case performance results of applications. Finally, a TDMA arbiter turned out to be the best solution for a time-predictable Java chip-multiprocessor. It provides a basis for a successful WCET analysis. An evaluation of the CMP system concerning static WCET analysis and average-case performance completes the research paper.

7.2.1 Main Findings and Results

The following points describe the author's most valuable findings and contributions gained during this research project. They show that the objectives set out by this thesis, defined in Section 1, have been accomplished.

- Timing analysis is in fact possible for a homogeneous multiprocessor system with a shared memory. Tasks running on different CPUs usually influence each others' execution times when accessing shared memory. The TDMA arbitration concept removes the interdependencies between task execution times due to bandwidth partitioning. WCET estimates of single application tasks can be analyzed in isolation.
- The design of a time-predictable CMP architecture is presented in detail. A prototype implementation, integrating up to 8 JOP cores in a low-cost FPGA, shows a successful execution of real-world programs.
- WCET analysis approaches of different arbiters show that a time-sliced arbiter is the only option for obtaining viable WCET estimations. While resolving simultaneous access conflicts to the shared memory, it enables WCET analysis of Java bytecodes by splitting the memory access bandwidth into equal time slots.
- Timing analysis of Java applications generates WCET bounds given in clock cycles before runtime. There is no need for any measurements. A

static WCET analysis tool, enhanced for use with multiprocessors, produces these WCET estimates and completes a real-time development solution.

- A comparison of CMP configurations using different arbiters shows that dynamic arbitration mechanisms are less predictable in the temporal domain but more powerful for average-case program execution.
- The course of this research confirmed the assumption that a real-time processor has to be designed from scratch for time-predictability. Both publications from Thiele & Wilhelm [9] and Lickly et al. [4] share this opinion. Microprocessors used for real-time systems have to be as predictable with regard to time as they are in the range of computed values. Several advanced microprocessor features that increase average-case performance cannot be analyzed precisely. Therefore, the following principle should be pursued for real-time system design: *Build a processor for WCET analysis, don't use a common processor tailored to average-case performance and try to analyze it!*

7.3 Thesis Relevance

7.3.1 Real-Time Java

Interest for an increased abstraction level has gained the embedded system industry's acceptance for Java. Nevertheless, it is still on the verge of breakthrough in the real-time system sector. A step in the right direction is a European Commission funded project called *Java environment for parallel real-time development* (JEOPARD) [8]. The consortium members started the development of an independent software interface for Java real-time multiprocessor systems in January 2008. During this project, the proposed real-time Java CMP is used for further research activities on real-time garbage collection [5, 7] and future time-predictable processor architecture [6, 10]. The author is convinced of Java's success in developing future time-predictable systems.

7.3.2 Simultaneous Multiprocessing

This section raises a provocative question: *Are simultaneous multiprocessors the future for increasing processing performance?* Although multiple profound arguments have been made that support the use of the most popular multiprocessor architecture, some doubts are emerging about future efficient use of SMP computing.

In the 20th century, a computer system's performance was dominated by the low CPU clock rate. Since then, times have changed and the biggest problem has been to provide the CPU with enough data as quickly as possible, because CPU clock rate has outperformed memory speed. This growing speed imbalance, commonly known as "memory bottleneck", is intensified if multiple CPUs are using a shared memory bus. The pipelines of multiple processors have to be kept busy. Therefore, caches have been used to lower memory bandwidth demands. Consequently, processors can continue executing their tasks quickly and can contribute significantly to the multiprocessor performance.

According to Hennessy and Patterson [1], this computer architecture only scales up to a few dozen cores. Typical state-of-the-art desktop and server processors integrate two or four CPUs. In this thesis, benchmark executions scale well in CMP configurations with up to four cores. Integrating more CPUs does not yield superior results. Even though a CMP with four cores should be satisfactory for current real-time applications, further enhancements can only be attained by increasing memory bandwidth or lowering bandwidth demand.

Research and advancements of hardware and software are needed in order to benefit from the SMP concept. New bus architecture innovations are needed to combat slow memory access problems even when the number of CPUs increases. On the software side, a distinct lack of advanced multiprocessor programming techniques is noticeable. Only a few applications have an innate parallel capability to efficiently exploit the available processing performance. Various parallelizable application types include signal and image processing, server applications, and embedded applications. Mapping sequential code to SMP architectures requires new parallelization methodologies in order to exploit the full application performance.

7.4 Outlook

In this project, a complete and detailed timing analysis of a multiprocessor system with a shared memory has been performed. Three different arbitration algorithms have been implemented and analyzed for practical use in real-time systems. Only the TDMA arbitration technique features viable WCET bounds. When carrying out performance comparisons, some issues regarding the speedup of the WCET have been identified in this type of system. Still, many interesting ideas for further system improvements remain unexamined – they would have exceeded the scope of this thesis.

Further research on a time-predictable CMP examines two closely related aspects: improvements to the multiprocessor architecture and the WCET

analysis. Modifications within the architecture might make the analysis more intricate but an increase in processor performance is assured.

7.4.1 Enhanced Memory Arbitration Concepts

An investigation of a percentage-based memory arbitration of the available memory access bandwidth could be an interesting concept to investigate. Based on the time-sliced arbiter, the memory bandwidth per CPU will be adjusted dependent on the workload of multiple CPUs. For example, if one CPU needs 60% of the available memory bandwidth, that is exactly what it will receive. Either this CPU gets more time slots or one larger-sized time slot. A lookup table could be used to dynamically schedule time slots at runtime. For example, it might be necessary to give 100% of the available bandwidth to one CPU due to a high priority event.

Further enhancements of the fair arbiter could result in a promising arbitration solution. In case of a burst memory access (method cache load), this access might be interrupted after a predetermined period of time. This would result in a mix of fair and TDMA-based arbitration. Consequently, using a limited access time, better average-case performance due to a fair arbitration would be combined with the already existing time predictability.

7.4.2 Advanced Synchronization Mechanisms

One major challenge of a multiprocessor with a shared memory is the synchronization of access to shared objects. Multiple processors communicate via these data structures. If multiple threads executing on different CPUs need to access the same objects or class variables simultaneously, their data access must be properly managed. Only then predictable program behavior and data consistency can be ensured.

The first CMP prototype utilizes one global lock for all shared data structures residing on the heap. Even though this synchronization limits the parallel program execution, it does not entail problems with fine grained locking (e.g. livelock or deadlock). More advanced mechanisms like lock-free synchronization algorithms [3] or transactional memory [2] could not be evaluated within the course of this thesis. Especially, transactional memory that executes a number of memory accesses in an atomic transaction is a promising synchronization option. Combining the synchronization of multiprocessors with timing analysis presents a future challenge for predictable multiprocessor architectures.

7.4.3 Dynamic Memory Support

This thesis assumes a main memory with predefined memory access time latencies. Static random access memory (SRAM) is used for prototype implementation because it features short and predetermined memory access times. A large disadvantage of SRAM is that it needs a large number of transistors, namely 6, to store a single bit of information. Therefore, it is very expensive with respect to die area and cost compared to dynamic random access memories (DRAM). These memories only need one transistor and a capacitor for storing a single bit. A capacitor discharges over time and therefore DRAMs must be periodically refreshed to maintain charge. For use in a real-time CMP system, this refresh would not complicate WCET analysis. A designated time slot within the TDMA period could be allocated for refresh execution. Research will have to prove whether longer memory access latencies make DRAM a viable alternative for use in time-predictable CMPs.

7.4.4 WCET Analysis of Bytecode Sequences

Currently, the WCET is analyzed at Java bytecode level. Each bytecode's WCET is calculated separately. For uniprocessor WCET analysis, these bytecode WCETs do not differ from best-case or average-case execution times. However, the multiprocessor analysis searches for the worst-case overlap between the bytecode's memory access pattern and the TDMA memory access scheme. In order to obtain a WCET of an application task, both concepts add up all bytecode WCETs.

A more advanced approach for multiprocessor analysis could analyze basic blocks consisting of numerous successive bytecodes. Consequently, analyzing a sequence of bytecodes would tighten the WCET estimation, because the memory access pattern of a bytecode sequence is taken into account. Furthermore, in order to enhance the quality of WCET bounds, an analysis of multiple basic blocks could be investigated.

7.4.5 WCET Analyzable Data Caching

The gap between memory access time and processor speed is widening. CPUs have to idle until the requested data from the main memory is available. Multiprocessors using a shared memory increase the pressure on the memory bandwidth even further. Typically, caches serve as very fast memories because on a cache hit, data does not have to be fetched from the main memory.

Data caching remains problematic for static timing analysis, because it is difficult to determine whether a data item is in cache or not. Even though each JOP uses a method cache (caching instructions) and a stack cache, caching of heap allocated data has so far been disregarded and requires further exploration.

Time-predictable data caching for multiprocessors including the implementation and analysis of a cache coherence mechanism (e.g. a snooping or a directory-based mechanism) poses a further challenge in analyzing tight execution time bounds.

7.5 Summary

This thesis shows that timing analysis is in fact possible for homogeneous multiprocessor systems with a shared memory. A Java chip-multiprocessor architecture consisting of a number of JOP cores and a shared memory is presented. An arbiter is used to serialize the memory access of multiple processors. Three different arbitration schemes are described, analyzed, and compared for viable use in real-time CMPs.

The key component enabling a WCET analysis of the CMP is a TDMA arbiter that splits the memory access bandwidth into equal shares. Therefore, a WCET analysis of Java bytecodes can be performed instead of analyzing each single memory access. In this research, JOP's WCET tool was enhanced to integrate maximum latencies through memory access collisions of multiple CPUs. Examples of WCET calculations are presented, and WCET estimates are compared to measured results. Several experiments, carried out by executing benchmarks on real hardware, have demonstrated that the performance capability of a time-predictable architecture cannot keep up with an architecture designed for average-case performance. Nevertheless, for real-time applications safe upper WCET bounds are more important than processing performance.

Bibliography

[1] John Hennessy and David Patterson. *Computer Architecture: A Quantitative Approach, 4th ed.* Morgan Kaufmann Publishers, 2006.

[2] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, pages 289–300, New York, NY, USA, 1993. ACM.

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

[4] Ben Lickly, Isaac Liu, Sungjun Kim, Hiren D. Patel, Stephen A. Edwards, and Edward A. Lee. Predictable programming on a precision timed architecture. In *Proceedings of International Conference on Compilers, Architecture, and Synthesis from Embedded Systems*, October 2008.

[5] Wolfgang Puffitsch. Decoupled root scanning in multi-processor systems. In *CASES '08: Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*, pages 91–98, New York, NY, USA, 2008. ACM.

[6] Martin Schoeberl. Time-predictable cache organization. In *Proceedings of the First International Workshop on Software Technologies for Future Dependable Distributed Systems (STFSSD 2009)*, Tokyo, Japan, March 2009. IEEE Computer Society.

[7] Martin Schoeberl and Wolfgang Puffitsch. Non-blocking object copy for real-time garbage collection. In *Proceedings of the 6th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2008)*, September 2008.

[8] Fridtjof Siebert. Jeopard: Java environment for parallel real-time development. In *JTRES '08: Proceedings of the 6th international workshop on Java technologies for real-time and embedded systems*, pages 87–93, New York, NY, USA, 2008. ACM.

[9] Lothar Thiele and Reinhard Wilhelm. Design for timing predictability. *Real-Time Systems*, 28(2-3):157–177, 2004.

[10] Andy Wellings and Martin Schoeberl. Thread-local scope caching for real-time Java. In *Proceedings of the 12th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2009)*, Tokyo, Japan, March 2009. IEEE Computer Society.

A

Measurement-based Verification of Bytecode WCET Analysis

A.1 Verification Goal

The verification goal is to check the WCET analysis of bytecodes. Therefore, bytecode execution times are measured. The bytecode execution time is dependent on the starting point within a TDMA period. Two properties of measured bytecode execution times have to be demonstrated in order to verify multiprocessor WCET analysis:

1. A measured execution time should never be larger than the analyzed WCET. Otherwise, the static WCET analysis tool produces unsafe WCET bounds.
2. Ideally, the maximum measured execution time is the same as the analyzed WCET. Consequently, tight bytecode WCETs are produced due to a precise and accurate hardware model.

A.2 Verification Method

The verification is carried out by measuring bytecode execution times in real hardware. It is assumed that the bytecode starting point within the TDMA period that produces the bytecode WCET is part of the verification process.

Table A.1: Java bytecodes of sample assignment.

Bytecode	Cycles
aload_0	1
iconst_0	1
iaload	25
istore_1	1

A random latency delays the bytecode starting point between consecutive runs. Consequently, it is possible to start the bytecode execution in every possible cycle of a given TDMA period and to cover all different bytecode execution options. To eliminate the risk that a worst-case scenario is not measured, prime numbers are used for the TDMA period and the latency generation.

A.2.1 Hardware Configuration

The hardware board described in Section 3.4.3 is used to obtain measured bytecode execution times. For these experiments, the CMP integrates 3 CPUs and the TDMA period is configured to 21 clock cycles.

A.3 Verification Result of Bytecode iaload

Line 18 of the example code shown in Listing A.1 assigns an array component with index 0 to the local variable `test`. This Java assignment results in the bytecodes described in Table A.1. The WCET analysis of this assignment is only dependent on the bytecode `iaload`, which accesses the memory. All other bytecodes do not access the memory and execute in a single cycle. In summary, the analyzed WCET of this simple Java assignment results in 28 clock cycles.

Listing A.1 shows a method that measures the execution time of the Java assignment `MAX` times. After initializing some local variables, a random number generator is initialized with a prime number `SEED`. In each run of the loop body, a random latency delays the execution of the assignment. `MAX` is initialized to 10000. The `PRIME` constant is chosen to be 183, which generates latencies between 0 and 3470 clock cycles. The variable `result` stores the execution time of each run. Running this measurement in real hardware results in 28 cycles at a maximum. Subtracting the three bytecodes that

Listing A.1: Method for assignment measurement.

```
1  void measure(int [] array) {
2      int test = 0;
3      int random = 0;
4      int start = 0;
5      int stop = 0;
6      int result = 0;
7
8      // Initialization of random generator
9      Random r = new Random(SEED);
10
11     for(int i=0; i<MAX; i++){
12
13         // Random latency generation
14         random = Math.abs(r.nextInt() % PRIME);
15         for(int j=0; j<random; j++);
16
17         start = Native.rdMem(Const.IO_CNT);
18         test = array[0];
19         stop = Native.rdMem(Const.IO_CNT);
20         result = stop-start;
21     }
22 }
```

do not access the memory gives exactly the same result of 25 cycles as the analyzed WCET of `iaload`.

B

Analyzed Bytecode WCETs

When using a TDMA arbiter, bytecode WCETs are dependent on the number of CPUs, time slot sizes and memory access times as described in Chapters 5 and 6. This chapter shows analyzed bytecode WCET results of different CMP configurations.

Assuming memory access times of two cycles for a read and three cycles for a write access, Table B.1 lists bytecodes and their WCETs. Only bytecodes that access the main memory are shown. All other bytecodes are omitted from the table because their WCETs are not affected by a shared memory multiprocessor. The second column presents bytecode WCETs of a single JOP, which does not have to share the memory bandwidth. Therefore, a time slot size is not provided. All other columns show WCET results of different configurations regarding the number of CPUs and the time slot size. For the analysis of return and invoke bytecodes it is assumed that the desired method resides in cache and no cache miss occurs.

	CMP Configuration (# of CPUs/Slot size)									
Opcode	1/-	2/3	2/15	2/30	4/3	4/15	4/30	8/3	8/15	8/30
ldc	8	12	24	39	18	54	99	30	114	219
ldc_w	9	13	25	40	19	55	100	31	115	220
ldc2_w	17	21	33	48	32	63	108	56	123	228
xaload[1]	10	22	26	41	40	56	101	76	116	221
xastore[2]	14	24	31	46	42	61	106	78	121	226
xreturn[3]	23	27	39	54	33	69	114	45	129	234

Table B.1: Bytecode WCETs depending on CMP configuration in clock cycles. The time slot size is varied between 3 and 30 cycles.

	CMP Configuration (# of CPUs/Slot size)									
Opcode	1/-	2/3	2/15	2/30	4/3	4/15	4/30	8/3	8/15	8/30
return	21	25	37	52	31	67	112	43	127	232
getstatic	8	12	24	39	18	54	99	30	114	219
putstatic	10	15	27	42	21	57	102	33	117	222
getfield	13	19	29	44	31	59	104	55	119	224
putfield	16	22	33	48	34	63	108	58	123	228
invokevirtual	100	109	127	154	121	217	274	169	397	514
invokespecial	75	79	97	106	85	157	166	121	277	286
invokestatic	75	79	97	106	85	157	166	121	277	286
arraylength	7	11	23	38	17	53	98	29	113	218

Table B.1: Bytecode WCETs depending on CMP configuration in clock cycles. The time slot size is varied between 3 and 30 cycles.

[1]xaload = iaload, faload, aaload, baload, caload, saload
[2]xastore = iastore, fastore, aastore, castore, sastore
[3]xreturn = ireturn, freturn, areturn

In contrast to the previous table, Table B.2 compares bytecode WCET results using time slot sizes between 6 and 12 cycles.

	CMP Configuration (# of CPUs/Slot size)									
Opcode	1/-	2/6	2/9	2/12	4/6	4/9	4/12	8/6	8/9	8/12
ldc	8	15	18	21	27	36	45	51	72	93
ldc_w	9	16	19	22	28	37	46	52	73	94
ldc2_w	17	29	27	30	53	45	54	101	81	102
xaload	10	17	20	23	29	38	47	53	74	95
xastore	14	29	25	28	53	43	52	101	79	100
xreturn	23	30	33	36	42	51	60	66	87	108
return	21	28	31	34	40	49	58	64	85	106
getstatic	8	15	18	21	27	36	45	51	72	93
putstatic	10	18	21	24	30	39	48	54	75	96
getfield	13	20	23	26	32	41	50	56	77	98
putfield	16	31	27	30	55	45	54	103	81	102
invokevirtual	100	109	133	136	145	187	184	238	331	328
invokespecial	75	82	97	112	118	151	184	190	259	328
invokestatic	75	82	97	112	118	151	184	190	259	328
arraylength	7	14	17	20	26	35	44	50	71	92

Table B.2: Bytecode WCETs depending on CMP configuration in clock cycles. The time slot size is varied between 6 and 12 cycles.

Table B.3 lists bytecode WCET results analyzed for a system with an SRAM memory as specified in Section 6.2. A TDMA arbiter needs a minimum time slot size of 6 clock cycles to assure a successful memory access of multiple CPUs.

	CMP Configuration (# of CPUs/Slot size)									
Opcode	1/-	2/6	2/15	2/30	4/6	4/15	4/30	8/6	8/15	8/30
ldc	10	19	28	43	31	58	103	55	118	223
ldc_w	11	20	29	44	32	59	104	56	119	224
ldc2_w	20	33	38	53	57	68	113	105	128	233
xaload	16	41	34	49	77	64	109	149	124	229
xastore	21	44	59	56	80	119	116	152	239	236
xreturn	23	32	41	56	44	71	116	68	131	236
return	21	30	39	54	42	69	114	66	129	234
getstatic	10	19	28	43	31	58	103	55	118	223
putstatic	13	24	33	48	36	63	108	60	123	228
getfield	17	32	35	50	56	65	110	104	125	230
putfield	21	36	41	56	60	71	116	108	131	236
invokevirtual	105	121	160	169	169	280	289	241	520	529
invokespecial	78	87	130	111	121	220	171	193	400	291
invokestatic	78	87	130	111	121	220	171	193	400	291
arraylength	9	18	27	42	30	57	102	54	117	222

Table B.3: Bytecode WCETs depending on a CMP system with larger memory access times.

C

List of Acronyms

ACET	Average-Case Execution Time
AHB	Advanced High-performance Bus
ASIC	Application-Specific Integrated Circuit
BCET	Best-Case Execution Time
BMVIT	Bundesministerium für Verkehr, Innovation und Technologie
CFG	Control Flow Graph
CMP	Chip-Multiprocessor
CPU	Central Processing Unit
DE2	Development and Education Board 2 from Altera
DMA	Direct Memory Access
DRAM	Dynamic Random Access Memory
EDK	Embedded Development Kit from Xilinx
EP1C12	Cyclone I FPGA from Altera
EP2C35	Cyclone II FPGA from Altera
FP	Fixed Priority
FPGA	Field-Programmable Gate Array
GC	Garbage Collector
GUI	Graphical User Interface
IC	Integrated Circuit
ILP	Instruction-Level Parallelism
IP	Intellectual Property
IPET	Implicit Path Enumeration Technique
JEOPARD	Java Environment for Parallel Real-time Development

JOP	Java Optimized Processor
JSR	Java Specification Request
JVM	Java Virtual Machine
LRU	Least Recently Used
NUMA	Non-Uniform Memory Access
OPB	On-chip Peripheral Bus
PLL	Phase-Locked Loop
RISC	Reduced Instruction Set Computer
RTL	Register Transfer Level
RTS	Real-Time System
RTSJ	Real-Time Specification for Java
SoC	System-on-Chip
SimpCon	Simple SoC Bus
SMP	Symmetric (shared-memory) Multiprocessor
SOPC	System-on-a-programmable-chip builder from Altera
SP	Synergistic Processor
SRAM	Static Random Access Memory
TDMA	Time Division Multiple Access
TPCM	Time-Predictable Chip-Multiprocessor
UMA	Uniform Memory Access
VGA	Video Graphics Array
VHDL	Very High Speed Integrated Circuit (VHSIC) Hardware Description Language
WCET	Worst-Case Execution Time

List of Publications

[1] Christof Pitter. BlueDataControl - Implementation of a Bluetooth Transmission Line for Industrial Use. Master's thesis, Carinthian Tech Institute, Villach, Austria, August 2003.

[2] Christof Pitter. JopCMP - A Java Chip-Multiprocessor for Real-time Systems. In *Proceedings of the 4th International Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC 2008)*, pages 194–196, Barcelona, Spain, July 2008.

[3] Christof Pitter. Time-Predictable Memory Arbitration for a Java Chip-Multiprocessor. In *Proceedings of the 6th international workshop on Java technologies for real-time and embedded systems (JTRES 2008)*, Santa Clara, USA, September 2008. ACM Press.

[4] Christof Pitter and Martin Schoeberl. Time Predictable CPU and DMA Shared Memory Access. In *International Conference on Field-Programmable Logic and its Applications (FPL 2007)*, Amsterdam, Netherlands, August 2007.

[5] Christof Pitter and Martin Schoeberl. Towards a Java Multiprocessor. In *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems (JTRES 2007)*, Vienna, Austria, September 2007. ACM Press.

[6] Christof Pitter and Martin Schoeberl. Performance Evaluation of a Java Chip-Multiprocessor. In *Proceedings of the IEEE Third Symposium on Industrial Embedded Systems (SIES 2008)*, Montpellier, France, June 2008.

Curriculum Vitae

Christof Pitter

Markhofgasse 4/17, A-1030 Wien
Date of Birth: 10/06/1979 in Villach
cpitter@mail.tuwien.ac.at

EDUCATION

10/2003 – ongoing	**Vienna University of Technology** – Institute of Computer Engineering, Real-Time Systems Group PhD-thesis: *"Time-predictable Java Chip-multiprocessor"*
10/1999 – 09/2003	**Carinthia Tech Institute** – University of Applied Sciences, Master thesis: *"BlueDataControl – Implementation of a Bluetooth Transmission Line for Industrial Use"*
09/1996 – 02/1997	**Exchange Semester** in the USA
1990 – 1998	**Bundesrealgymnasium Perau** in Villach

PROFESSIONAL & PRACTICAL EXPERIENCE

10/2006 – ongoing	**Vienna University of Technology** – Institute of Computer Engineering, Real-Time Systems Group *Research Assistant* - Working on the national FIT-IT project TPCM (Time-Predictable Chip-Multiprocessor)
03/2005 – 02/2009	**Vienna University of Technology** – Institute of Computer Engineering, Real-Time Systems Group *Research & Teaching Assistant* - In charge of the laboratory class "Electrical Engineering for Computer Engineering"
09/2004 – 02/2005	**Vienna University of Technology** – Institute of Computer Engineering, Real-Time Systems Group *Study Assistant*
10/2002 – 06/2003	**Kreuzgruber GmbH**, Vienna *Embedded Software Engineer*