A Real-Time Java Chip-Multiprocessor

CHRISTOF PITTER
Vienna University of Technology, Austria
and
MARTIN SCHOEBERL
Vienna University of Technology, Austria

Chip-multiprocessors are an emerging trend for embedded systems. In this paper, we introduce a real-time Java multiprocessor 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 shared memory. All components are interconnected via a system on chip bus. The arbiter synchronizes the access of multiple CPUs to the shared main memory. In this paper, three different arbitration policies are presented, evaluated, and compared with respect to their real-time and average-case performance: a fixed priority, a fair-based, and a time-sliced arbiter.

Tasks running on different CPUs of a chip-multiprocessor (CMP) influence each others' execution times when accessing a shared memory. Therefore, the system needs an arbiter that is able to limit the worst-case execution time of a task running on a CPU, even though tasks executing simultaneously on other CPUs access the main memory. Our research shows that timing analysis is in fact possible for homogeneous multiprocessor systems with a shared memory. The timing analysis of tasks, executing on the CMP using time-sliced memory arbitration, leads to viable worst-case execution time bounds.

The time-sliced arbiter divides the memory access time into equal time slots, one time slot for each CPU. This memory arbitration scheme allows for a calculation of upper bounds of Java application worst-case execution times, depending on the number of CPUs, the time slot size, and the memory access time. Examples of worst-case execution time calculation are presented, and the analyzed results of a real-world application task are compared to measured execution time results. Finally, we evaluate the trade-offs when using a time-predictable solution compared to using average-case optimized chip-multiprocessors, applying three different benchmarks. These experiments are carried out by executing the programs on the CMP prototype.

Categories and Subject Descriptors: C.3 [**Special-Purpose and Application-Based Systems**]: Realtime and embedded systems; D.3.4 [**Programming Languages**]: Processors—*Run-time environments, Java*; B.7.1 [**Integrated Circuits**]: Types and Design Styles—*Microprocessors and microcomputers*

Additional Key Words and Phrases: Real-time system, Multiprocessor, Java processor, Shared memory, Worst-case execution time

1. INTRODUCTION

Modern applications demand ever-increasing processing power. They act as the 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

Author's address: Christof Pitter, Martin Schoeberl, Institute of Computer Engineering, Vienna University of Technology, Treitlstr. 3, A-1040 Vienna, Austria. email: pitter.christof@gmail.com, mschoebe@mail.tuwien.ac.at

on an integrated circuit at a given cost doubles every 24 months, as described by Moore's Law [Moore 1965]. 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 Hennessy and Patterson [2006], we 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 [Laudon and Spracklen 2007].

According to Hennessy and Patterson [2006], chip-multiprocessors (CMP) are the future path of performance enhancements. 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 progress in processing power would not be accompanied by an increase in the hardware complexity of the single processors. According to Wolf [2006], CMPs combine the significant advantages of embedded systems: increased performance, lower power consumption, and cost efficiency.

1.1 Java Chip-Multiprocessor

In this paper, we are offering a CMP architecture consisting of a number of Java Optimized Processor (JOP) [Schoeberl 2005b; 2008] cores and a shared memory. The shared memory is uniformly accessible by the homogeneous processing cores. A novel memory arbiter controls the various JOPs' access to the shared memory. It resolves the potential conflict of a simultaneous access to the shared memory. Three different arbitration mechanisms will be evaluated, and compared:

(1) Fixed priority arbitration
(2) Fair arbitration
(3) Time-sliced arbitration

We will show that for real-time systems, only a time-sliced arbitration of the main memory access is a feasible and analyzable solution. Furthermore, we describe the implementation of CMP booting, the CMP thread scheduling and its I/O device interconnection. Additionally, we will present JopCMP, a prototype of the CMP composed of up to 8 JOP cores, integrated in a low-cost FPGA, and connected to an external memory. This prototype is used to measure and evaluate the three arbitration algorithms. The ultimate goal of our research work is a multiprocessor for safety-critical applications.

1.2 WCET Analysis for the Java CMP

Many embedded systems are used for applications that prioritize real-time behavior over processing power. Such hard real-time systems must undergo timing analysis. Therefore, the worst-case execution time (WCET) of each task in the system has to be a known factor. The WCET is the amount of time a task eventually needs to execute under worst-case conditions on a given processor. Wilhelm et al. [2008] define the goal of WCET analysis concerning the upper bounds of execution time thus:

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

The calculated upper time bounds have to be safe in order to ensure hard real-time system 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.

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 obtained by executing the program 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 a measured WCET result does not reliably confirm that the worst-case program path has been triggered [Ermedahl and Engblom 2007].

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 an 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 authors believe that static analysis should become the conventional approach to modern hard real-time systems.

JOP comes with a static WCET analysis tool, which is described by Schoeberl and Pedersen in [2006]. The tool is enhanced for the WCET analysis of a CMP system. The key component for real-time analysis of the CMP is a time-sliced arbiter that splits the memory access bandwidth into time slots, one for each CPU. Therefore, 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 WCET task analysis. Our approach is described using a simple example. Additionally, we will provide measured data of the sample execution time. The results were obtained by running the application on hardware. Subsequently, we were able to compare the analyzed results to measured execution times. Furthermore, the measured and analyzed execution time results of real-world applications show the reliability of the proposed method.

1.3 Contributions and Paper Organization

This paper is based on our previous work [Pitter and Schoeberl 2007b; 2008; Pitter 2008; 2009] on Java based CMP systems. In this paper, we will provide a coherent view of three different arbitration policies with respect to WCET and average-case performance. The time division multiple access (TDMA) based arbiter is the foundation of a time-predictable CMP system. One contribution made by this paper is the enhancement of a WCET analysis tool for the multiprocessor system. Furthermore, the various configurations are evaluated using a larger application base. The proposed architecture is used by the EC funded project Jeopard on real-time Java for multiprocessors [Siebert 2008]. Best to our knowledge, the presented system is the first time-predictable CMP system that includes a WCET analysis tool.

The remainder of the paper is structured as follows: Section 2 outlines work related to this subject. In Section 3, a brief overview of the proposed CMP architecture is given. Three different arbiters are described in detail in Section 4. Section 5 gives a short introduction of the static WCET analysis of JOP. Additionally, it describes the WCET analysis approaches of the different memory arbiters. Section 6, evaluates the performance of the CMPs using three benchmarks. Section 7 discusses our findings. Finally, Section 8 concludes the paper and provides guidelines for future work.

2. RELATED WORK

Three quite different CMP architectures are state-of-the-art in mainstream desktop and server processors: multi-core versions of super-scalar architectures by Intel and AMD [Keltcher et al. 2003], multi-core chips with simple RISC processors like Sun Niagara [Kongetira et al. 2005], and the Cell architecture [Hofstee 2005; Kahle et al. 2005; Kistler et al. 2006]. The Cell is a heterogeneous multiprocessor consisting of a PowerPC microprocessor and eight co-processors. These multiprocessors are not considered viable for time-predictable systems, because their architectures are optimized for average-case performance and not for WCET. Complex hardware complicates the timing analysis.

The following sections describe the progress made in CMP for embedded systems. Furthermore, related work on timing analysis of processor architectures is summarized.

2.1 Embedded Multiprocessors

In the embedded system domain, there are two different types of CMP architecture:

(1) heterogeneous multiprocessors

(2) homogeneous multiprocessors

Multiprocessors with a heterogeneous architecture combine a core CPU for controlling and communication tasks, and additional special processing elements, which are often tailored to specific applications. Some examples of heterogeneous multiprocessors include the ST Nomadik [Artieri et al. 2004], designed for mobile multimedia applications, the Philips Nexperia PNX-8500 [Dutta et al. 2001], aimed at digital video entertainment systems, or the TI OMAP family [Martin and Chang 2003], designed to support 2.5G and 3G wireless applications.

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

2.1.1 *ARM.* The ARM11 MPCore [ARM 2006] 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.

2.1.2 *LEON.* Gaisler Research AB designed and implemented a homogeneous multiprocessor system called LEON3-FT-MP [Gaisler and Catovic 2006]. It consists of one centralized shared memory and four LEON3-FT processor cores that are based on the SPARC V8 instruction set architecture [SPARC International Inc. 1992]. All the CPUs, additional I/O controllers and memory controllers are connected using two AMBA-specified advanced high-performance buses (AHB) [ARM 1999]. One AHB runs at the CPUs' fre-

quency and connects the processors to the shared memory controller. The low-speed bus connects all other peripheral devices.

According to the AMBA specification, a CPU takes on the role of a master because it initiates 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. Even though the AHB arbitration protocol specification is well defined, no priority strategies or arbitration algorithms are specified. LEON's AHB arbiter implementation uses fixed priority. As will be shown later, a fixed priority arbiter is a problematic option for real-time systems.

2.1.3 *MicroBlaze.* MicroBlaze-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 [Xilinx 2007]. 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 individual application needs (i.e. peripheral controllers or cache sizes).

Memory and peripheral devices are connected via the on-chip peripheral bus (OPB) [IBM 2001]. Xilinx provides an OPB bus arbiter [Xilinx 2005] that can integrate up to 16 masters into the system. The available arbitration schemes include fixed priority (FP) or least recently used (LRU) algorithms.

2.1.4 *NIOS II.* Altera's Nios II [Altera 2007b] and the System-on-a-Programmable-Chip (SOPC) Builder [Altera 2007c] 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. Nios II can be customized to meet the application requirements: three different models, from non-pipelined up to a 6-stage pipeline. Avalon [Altera 2007a] 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 hides all connection details from the user. While the Avalon specification can be used freely, the System Interconnect Fabric is the property of Altera.

For multiprocessor systems, the System Interconnect Fabric integrates an arbitration module [Altera 2007a]. 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.

2.1.5 *PRET.* The core objective of the research collaboration between the universities of Berkley and Columbia is to implement a processor architecture for real-time embedded systems that is as predictable with regard to time as it is in the range of computed values. In [2008], Lickly et al. are proposing a precision-timed architecture (PRET), which combines a SPARC-based processor architecture with time-predictable features. A 6-stage thread-interleaved pipeline executes 6 threads in parallel, one thread at each stage. Hence, data forwarding can be avoided. Furthermore, scratchpad memories are used in place of common data and instruction caches. Access to the main memory is controlled by a so-called memory wheel. It allocates a pre-determined time slot for each thread to access the memory. The research group has presented a model of the PRET architecture in SystemC and demonstrated applications running in simulation.

2.1.6 *Discussion.* The described multiprocessors are still using backplane style buses that are not appropriate for an SoC interconnection. Furthermore, there is no use for a com-

plex 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 [Schoeberl 2007], which is further described in Section 3.3. Moreover, we are using a fixed priority, a fairness-based, and a time-sliced arbitration algorithm.

JOP, the processor used for the proposed CMP system, is open source and freely available under the GNU GPL. 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 soft-core processor avoids a lock-in to a single FPGA vendor, as is the case for MicroBlaze and Nios.

2.2 WCET Analysis of Multiprocessors

WCET analysis is crucial to the timing analysis of hard real-time systems. The task set of a real-time system requires a timing validation by schedulability analysis [Joseph and Pandya 1986; Liu and Layland 1973]. Hence, the WCET of each task has to be calculated. Only if these upper execution time bounds are known, the schedulability analysis can be performed. Consequently, the analysis result shows whether the task deadlines will be met, subsequently guaranteeing that all tasks can be executed by the system.

WCET analysis has been an active and well-established research area in the uniprocessor domain for years. Both Puschner & Burns [2000], and Wilhelm et al. [2008] give a broad overview of the WCET research. Nevertheless, not all of these achievements can be applied to multiprocessor systems. They are based on the assumption that tasks are independent and cannot influence one another. Using modern multiprocessors with shared resources (i.e. a shared memory), tasks influence each others' execution times and cannot be analyzed independently.

One research group (from the University of Linkoeping) has studied the WCET analysis of multiprocessors [Andrei et al. 2008; Rosen et al. 2007]. 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. Their 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, so all input data must be placed into the private memory before the task can start executing. Consequently, in most cases the execution time of a task can never be influenced by other tasks (see: simple-task model [Kopetz 1997]). However, the communication bus serves as a communication interface between CPUs and private memories, and 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, a TDMA-based bus sharing policy is used, as several CPUs may request a cache line from their private memories simultaneously.

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

3. JOPCMP ARCHITECTURE

According to [Wolf 2006], a multiprocessor system consists of three major subsystems: processing elements, memory and an interconnection network. JopCMP implements the

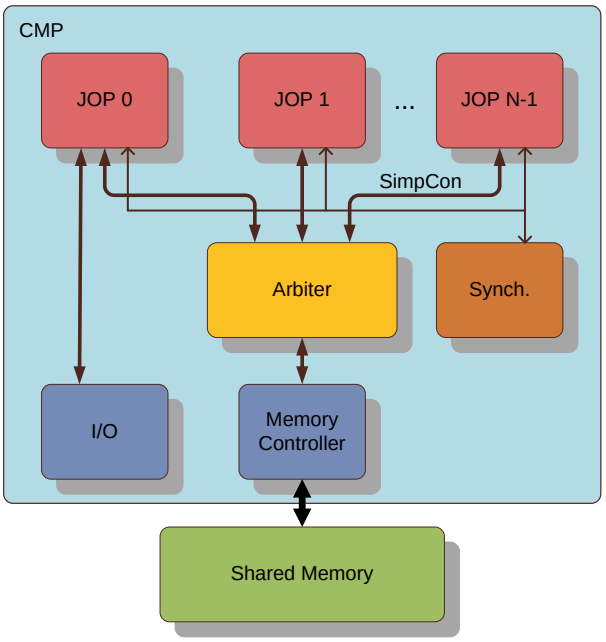

Fig. 1. JopCMP Architecture.

symmetric (shared-memory) multiprocessor (SMP) model. Several Java processors provide the basis of a homogeneous CMP. The interconnection network is responsible for connecting multiple processors to the memory. An arbiter is part of this network and controls the memory access to the shared memory. An SoC bus, called SimpCon [Schoeberl 2007], is used to connect the processing cores to the arbiter, and the arbiter to the memory controller. We consider the synchronization of shared data as a further major subsystem of an SMP. It is responsible for coordinating the access to shared objects. Figure 1 illustrates the typical architecture of an FPGA technology implementation. The following sections describe the different elements in more detail.

3.1 The Java Optimized Processor (JOP)

The Java optimized processor (JOP) [Schoeberl 2005b; 2008] is an implementation of the Java Virtual Machine (JVM) in hardware. This processor has been designed from scratch to provide a time-predictable execution environment for embedded real-time systems. 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 consumes lower logic resources compared to other Java processors.

Java processors usually do not execute Java bytecodes directly, because some instructions are too complex to be implemented in hardware. Therefore, JOP translates the bytecodes into its own instruction set called microcode. These microcode instructions, implemented in hardware, are executed by the stack architecture. Most bytecodes can be translated to a single microcode or a sequence of microcode instructions. Hence, the com-

plex instruction set of the JVM is transferred into a reduced instruction set. A few, more complex bytecodes, e.g., `new`, are implemented in Java methods.

JOP's core consists of a 4-stage pipeline. The first pipeline stage, *bytecode fetch*, fetches a bytecode from the instruction cache and calculates the microcode address. The subsequent 3 stages of the pipeline called *microcode fetch*, *microcode decode*, and *microcode execute* operate on native 8-bit microcode instructions. The top two stack machine elements are stored in registers. All instructions operate on these two registers in the *microcode execute* unit.

3.2 Memory Hierarchy

A shared memory is a global physical memory where all instructions and data are stored, accessible to all processors. A memory controller connects the CPUs integrated on the FPGA to the shared off-chip memory. Additionally, each JOP has access to two fast local memories, referred to as cache memories.

Each application thread has a reserved stack area in the memory. This thread private data is very frequently accessed, similar to registers in a typical register machine. Therefore, JOP caches this data in a so-called stack cache [Schoeberl 2005a] in an on-chip RAM. The spilling and filling of the stack cache is controlled by microcode instructions.

Additionally, an instruction cache – called method cache [Schoeberl 2004] – limits memory access frequency by caching bytecode instructions of complete methods. A cache miss and consequently a new method load from the read-only method area can only occur at invoke or return bytecodes.

According to the JVM specification [Lindholm and Yellin 1999], the heap stores JVM shared data. Our CMP architecture operates without caching the heap's shared data objects, therefore a coherent view of all CPUs' accessed data is ensured throughout. Our design facilitates the avoidance of hardware demanding cache coherence mechanisms.

3.3 Interconnection Network

The selection of an interconnection network topology is a major decision in multiprocessor architecture design. We use the simple SoC interconnect (SimpCon) [Schoeberl 2007] to connect the SoC modules. This synchronous on-chip bus is intended for read and write transfers via point-to-point connections. Only a master can initiate a transaction via a write or read request. Compared to other commonly used SoC buses like Avalon [Altera 2007a], or AMBA [ARM 1999], this specification does not work like a backplane bus. No bus request phase has to precede the actual bus transfer. Furthermore, the master's driven control, address, and data lines are only valid for a single clock cycle. A slave has to register all signals (e.g. the address) needed for several clock cycles. Consequently, the master can continue to execute its program until it needs a read result. The slave informs the master of the time the requested data will be available, through a signal called *rdy_cnt*. Additionally, the signal serves as an early notification of data access completion. This mechanism allows the master to send a new request before the former has been fulfilled. This form of pipelining permits fast data transfers.

SimpCon is well suited for on-chip point-to-point connections. Nevertheless, the specification does not support the synchronization of multiple masters to one slave. Therefore, we have introduced a central arbiter that controls memory access of multiple CPUs to the shared memory. The arbiter acts as slave for each JOP and as master for the memory controller. Section 4 is dedicated to memory arbitration.

3.4 Synchronization

Shared memory SMP systems need a synchronization mechanism. 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 have introduced a synchronization unit to the 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 1. JOP will be granted access if no other processor is holding the lock. Otherwise, it must wait until the other processor has finished 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. One side effect of a single lock is the avoidance of deadlock through design. Further information on synchronization of JopCMP can be found in [Pitter and Schoeberl 2007b].

3.5 CMP Boot-up Sequence

One interesting aspect of a CMP system is how the startup or boot-up is performed. On power-up, the FPGA starts the configuration state machine to read the FPGA configuration data either from a Flash memory or via a download cable from the PC during the development process. When the configuration has finished, an internal reset is generated. After this 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 memory, via a PC serial line, or an USB-port. The next step is the generation of a minimal stack frame. From then on, JOP runs in Java mode and invokes the special method `Startup.boot()`, even though some parts of the JVM are not yet setup. The method `boot()` performs the following steps:

—Sends a greeting message to *stdout*

—Detects the size of the main memory

—Initializes the data structures for the garbage collector

—Initializes `java.lang.System`

—Prints out JOP's version number, detected clock speed, and memory size

—Invokes the static class initializers in a predefined order

—Invokes the application class `main` method

The boot-up process is the same for all processors up to the execution of the first microcode instructions. At that moment, *only one* processor is allowed to perform 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 different CPUs. A unique CPU identity number (CPU_{ID}) is assigned to each processor. Only processor CPU_0 is designated to perform all the boot-up and initialization work. The other CPUs have to wait until CPU_0 has completed the boot-up and initialization sequence.

3.6 CMP Scheduling

The scheduler on each core is a preemptive, priority based real-time scheduler. As each thread gets a unique priority, no FIFO queues within priorities are needed. The best analyzable real-time CMP scheduler does not allow threads to migrate between cores. Each thread is pinned to a single core at creation. Therefore, standard scheduling analysis can be performed on a per core base.

Similar to the uniprocessor version of JOP, the application is divided into an initialization phase and a mission phase. During the initialization phase, a predetermined core executes only one thread that has to create all data structures and the threads for the mission phase. During transition to the mission phase all created threads are started.

The uniprocessor real-time scheduler for JOP has been enhanced to facilitate the scheduling of threads in the CMP configuration. Each core executes its own instance of the scheduler. The scheduler is implemented as `Runnable`, which is registered as an interrupt handler for the core local timer interrupt. The scheduling is not tick-based. Instead, the timer interrupt is reprogrammed after each scheduling decision. During the mission start, the other cores and timer interrupts are enabled.

Another interesting option to use a CMP system is to execute exactly one thread per core. In this configuration scheduling overheads can be avoided and each core can reach an utilization of 100% without missing a deadline. To explore the CMP system without a scheduler, a mechanism is provided to register objects, which implement the `Runnable` interface, for each core. When the other cores are enabled, they execute the `run` method of the `Runnable` as their *main* method.

3.7 I/O Devices

Each core contains a set of local I/O devices, needed for the runtime system (e.g., timer interrupt, lock support). The serial interface for program download and a *stdio* device is connected to the first core.

For additional I/O devices two options exist: either they are connected to one core, or shared by all/some cores. The first option is useful when the bandwidth requirement of the I/O device is high. As I/O devices are memory mapped they can be connected to the main memory arbiter in the same way as the memory controller. In that case the I/O devices are shared between the cores and standard synchronization for the access is needed. For high bandwidth demands a dedicated arbiter for I/O devices or even for a single device can be used.

An interrupt line of an I/O device can be connected to a single core or to several cores. As interrupts can be individually disabled in software, a connection of all interrupt lines to all cores provides the most flexible solution.

3.8 Hardware Platform

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. All designs are clocked at 90 MHz and the main memory access time is 4 cycles for a 32-bit read operation, and 6 cycles for a 32-bit write operation.

All configurations consume the same amount of on-chip memory per core: 1 KB stack cache and 2 KB of method cache. This configuration makes it possible to synthesize an 8-way version of the CMP in the low-cost FPGA.

4. MEMORY ARBITRATION

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

The arbiter controls the memory access of multiple CPUs to the shared memory. Naturally, if one CPU is accessing the shared memory, no other CPU can access the memory at the same time. It is forced to wait until the CPU on turn has completed its memory transfer. In this case, a memory arbiter resolves these access conflicts 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 simultaneous accesses at runtime. Each CPU in the system is assigned a priority. The fixed and the fairness based arbitration policies are examples of dynamic arbiters.

The static arbitration policy strictly defines the access pattern before runtime. Consequently, no arbitration decision is necessary at runtime. Implementation of this policy is typical for real-time systems where 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 [Thiele and Wilhelm 2004], e.g. a shared memory. We wanted to remove the interdependencies between task execution times. Therefore, an arbitration algorithm is necessary that 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.

Arbiters perform an arbitration decision in the same cycle the request arrives. In comparison to existing arbiters like Avalon [Altera 2007a], or AMBA no additional cycle is lost for arbitration. Subsequently, memory access time is reduced and the bandwidth increases.

Our implemented arbiters can be configured for variable numbers of CPUs. Compared to existing arbiters like AMBA [ARM 1999] or CoreConnect [IBM 2007], the maximum number of connected masters is not limited. As a result, the CMP system can be customized to the application needs.

4.1 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.

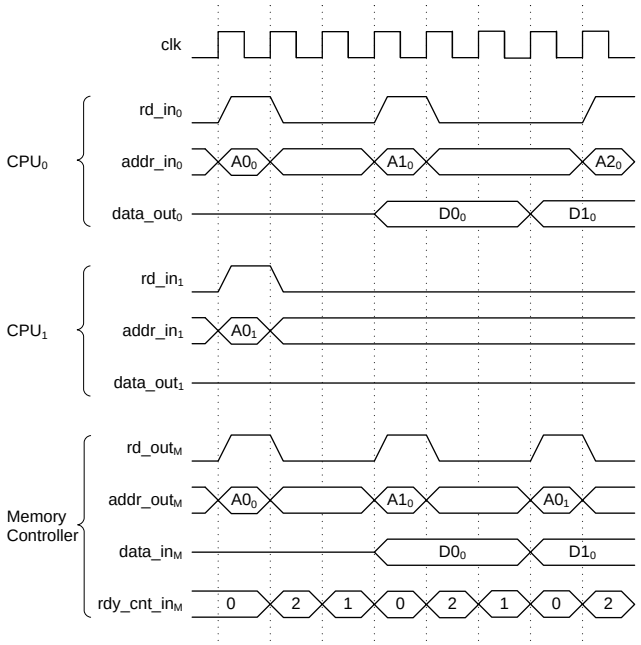

Fig. 2. Memory access arbitration of the fixed priority arbiter.

In Figure 2, an arbitration scenario of a 2-way CMP with a memory access time of 2 cycles is shown. 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 2, 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 because it has a higher priority than CPU_1 given that 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, indicated by the value 0 of signal $rdy\_cnt\_in_M$ and no further request of CPU_0 is pending. As soon as CPU_0's data is available, the registered memory access of CPU_1 is processed. In the last cycle indicated in Figure 2, CPU_0 wants to access the memory again. This read access is registered in the arbiter and is 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 [Pitter and Schoeberl 2007a]. The DMA device, e.g. a display refresh unit, performs a regular memory access within a short period of time and is assigned top priority.

4.2 Fair Arbiter

The fair arbiter implements an arbitration policy that guarantees fairness among the CPUs accessing a shared memory. Each CPU in the system is assigned a unique CPU identity

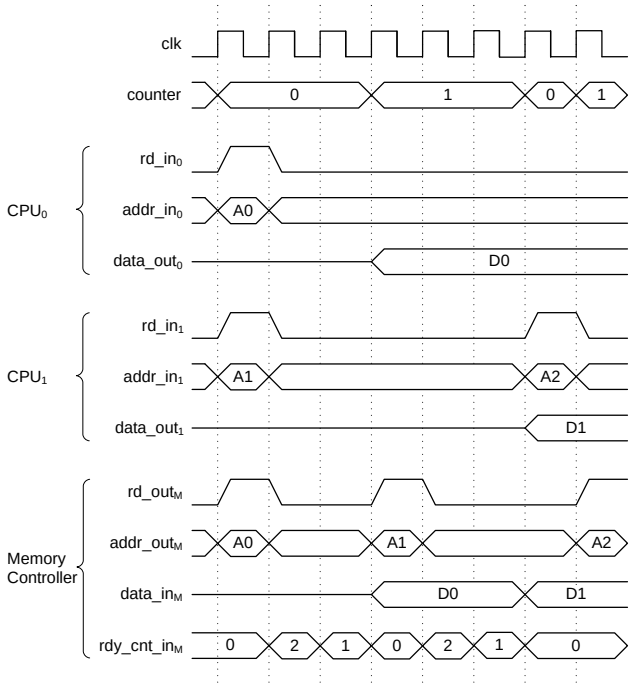

Fig. 3. Memory access arbitration of the fair arbiter.

(CPU_{ID}). Our fair arbitration policy uses a wrapping counter. 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 3 shows an arbitration scenario of a 2-way CMP system with a memory access time of 2 cycles. 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 $data\_in_M$ and $data\_out_0$. When the memory access has been completed, the counter increments by one and the registered memory access of CPU_1 is processed. When the data is available, the counter already shows a 0 value. As opposed to CPU_1, CPU_0 does not request a memory access. The

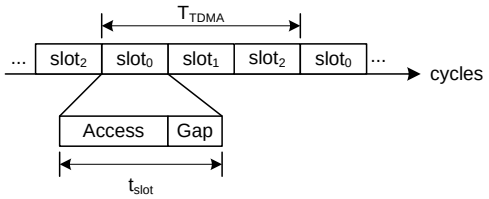

Fig. 4. Time slots of the CPUs.

counter is therefore advanced in the following cycle and CPU_1's registered memory access is processed.

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 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.

4.3 Time-sliced Arbiter

According to [Andrei et al. 2008; Poletti et al. 0609; Rosen et al. 2007], a 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 perform its operations on the memory. We agree that this arbitration policy is well suitable for timing analysis of multiprocessor systems with shared resources. The arbitration information provides significant data for timing analysis.

Figure 4 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 an outstanding memory request can be finished, but the CPU cannot initiate a new request. This gap permits the next CPU on turn to access the shared memory in the first cycle of its time slot. The size of the gap is dependent on memory access time. The larger the memory access time, the larger the gap.

5. TIMING ANALYSIS

This section starts with a short introduction of the static WCET analysis of JOP. The remaining sections describe the WCET analysis of a CMP system using different memory arbiters.

5.1 Static WCET Analysis based on JOP

Real-time processors like JOP have simpler and less powerful architectures than modern CPUs. Several advanced features increasing average-case performance (e.g. data caches, out-of-order execution, and branch prediction) are disregarded [Schoeberl 2008]. Although these methods speed up program execution, they impede the timing behavior predictability 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 [Schoeberl and Pedersen 2006], the WCET of a task can be obtained. Java programs are compiled to form class files that include JVM instructions called bytecodes. For static WCET analysis, the bytecode sequence is transformed into a directed graph of basic blocks called a control flow graph (CFG). Each basic block consists of several bytecode instructions. JOP translates each bytecode into a microcode or a sequence of microcode instructions that are executed by the processor. Every microcode has a fixed execution time, therefore each basic block can also be assigned an exact execution time.

In addition, flow facts have to be added to the Java program code in advance. In general, this is the only way to limit the loops and calculate the basic block frequency execution. The CFG, including the flow facts and the mapping to the hardware, make WCET analysis possible using the implicit path enumeration technique (IPET) [Li and Malik 1995].

5.2 Fixed Priority Arbitration Approach

The common factor in all arbitration approaches is 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. Throughout this paper, the WCET is measured 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:

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

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 instances of memory access 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 to the method cache. In the worst possible scenario, one or more CPUs in the system request memory access during the previous clock cycle of 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.

5.3 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 2.

$$WCET_j = \sum_{\forall i \neq j} t_{WCET_i} + t_j \tag{2}$$

As in the case of the fixed priority CPU, t_{WCET_i} is the WCET of all instances of memory access 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.

5.4 Time-sliced Arbitration Approach

The TDMA arbitration policy strictly defines the 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 3:

$$WCET_j = (t_{gap} - 1) + (n-1) \cdot t_{slot} + t_j \tag{3}$$

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 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. Later on, we will examine whether a smaller or a larger time slot configuration achieves lower WCET bounds. Nevertheless, the minimum size of the time slot is predetermined and dependent on the memory access time. Otherwise, a processing unit could never successfully access the memory within one time slot.

Applying Equation 3 to individual instances of memory access results in a conservative WCET for bytecodes. To provide tighter WCET bounds, our method calculates the WCET for complete bytecode instructions instead of analyzing the WCET of a single memory access. The WCET is dependent on:

—the number of JOPs integrated in the CMP

—time slot size

—memory access time

First, the memory access pattern of each bytecode has to be investigated. The number of JOPs has to be defined as well as the size of the time slot. This system configuration introduces a fixed TDMA memory access scheme whereby each CPU is assigned a time slot within the TDMA period. Within these set conditions, the WCET of each bytecode can be determined using the algorithm described in Section 5.4.2. JOP's WCET analysis tool uses the generated bytecode estimates to calculate the WCET of a Java application.

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 I. Bytecodes accessing a shared memory.

5.4.1 *Bytecode Memory Access Pattern.* JOP translates most of the bytecodes into its native set of microcode instructions. Each bytecode consists of one or a series of microcode instructions. Some bytecodes are actually implemented in the hardware. A couple of bytecodes are implemented in Java. The timing analysis of these bytecodes was not included in this paper because these bytecodes are analyzed like normal Java code.

The heap and method areas are shared data areas located in the main memory. Consequently, all bytecodes accessing these memory areas have to be examined. Some bytecodes access the memory several times in a row, some only once. Therefore, it makes sense to have a closer look at several different instructions. Table I summarizes the bytecodes that access the main memory. As stated before, some bytecodes are implemented in Java, e.g. bytecodes of type NEW, SWITCH and CAST, so they have been disregarded in the proposed analysis.

Most bytecode memory access patterns can be analyzed statically, e.g. 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 bytecode memory access pattern can be analyzed regardless of the program source code. 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 the hardware. To analyze the memory access pattern, we examine the VHDL implementation in combination with ModelSim simulations.

The memory access patterns of type CALL and RETURN bytecodes require a dynamic analysis. Each JOP is equipped with an instruction cache that caches complete Java methods [Schoeberl 2004]. Consequently, the memory access patterns of these bytecodes vary, depending on execution history. If the method is already in the cache, no additional memory access is 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

Listing 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;
    }
}
```

the length of the method that has to be loaded, the access pattern has to be created for each instance a bytecode occurs in the source code. We have therefore integrated the pattern generation feature into the WCET analysis tool where the cache information is available.

5.4.2 *WCET Analysis of Bytecodes.* Listing 1 shows a simplified version of the algorithm used to the find the bytecode WCETs. The inner loop of this algorithm calculates the array *bytecode* execution time starting at *position*. The *bytecode* describes the memory access pattern of the instruction. It has a predefined length, which is equal to the length of the microcode instruction sequence. 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 by one. 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 by one until the CPU is allowed

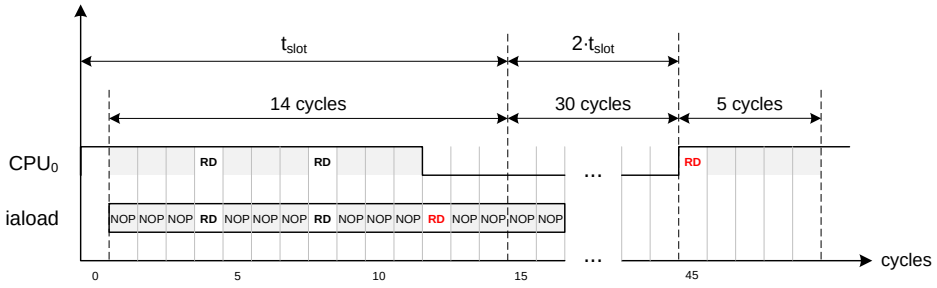

Fig. 5. WCET calculation of iaload.

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 calculation starting point for 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 bytecode execution time value. If the new *execTime* is greater than the current worst-case execution time, it will be assigned to the *wcet* variable. Therefore, the resulting bytecode WCET, depending on the number of CPUs and the size of the time slot, is available after the last iteration.

5.4.3 *WCET Calculation of Bytecode iaload.* Figure 5 shows a calculation of the *iaload* bytecode assuming a CMP configuration of 3 CPUs and a time slot of 15 clock cycles. A read access to the main memory takes 4 cycles. It should be noted that a time slot of 15 cycles permits each CPU to access the memory until the 12^{th} cycle. In the 13^{th}, 14^{th}, and 15^{th} cycle, read access is not granted. Otherwise, memory access cannot be guaranteed for the next CPU in the first cycle of its time slot.

The following access pattern is predetermined for *iaload* = {*NOP, NOP, NOP, RD, NOP, NOP, NOP, RD, NOP, NOP, NOP, RD, NOP, NOP, NOP, NOP*}. We can see that *iaload* performs a read access to the memory three times. The WCET scenario is shown in Figure 5. *iaload* starts with an NOP operation in the 2^{nd} cycle of the CPU's time slot. It can be noted that the 3^{rd} RD access of *iaload* cannot immediately be executed. It is delayed by two time slots until it is allowed to access the memory. The WCET estimate results in 49 cycles.

5.5 A Simple Loop Example

In this section, we will systematically analyze the WCET of a simple loop to show how the WCET is calculated. Listing 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 self-contained. Therefore, the loop body could be easily executed on different CPUs simultaneously.

Listing 2. Simple Loop.

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

Table II. Java bytecodes and basic blocks forming 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	5 :	aload_1	1	
	6 :	iload_3	1	
	7 :	aload_1	1	
	8 :	iload_3	1	
	9 :	iaload	49	
	10 :	iload_2	1	
	11 :	iadd	1	
	12 :	iastore	89	
	13 :	iload_3	1	
	14 :	iconst_1	1	
	15 :	iadd	1	
	16 :	istore_3	1	
	17 :	goto 2	4	152

As explained before, the WCET estimate for each bytecode accessing the main memory has to be calculated according to the CMP configuration. 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 II. The fourth column represents 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 I, only *iaload* and *iastore* access the main memory. Therefore, their WCETs are dependent on the system configuration.

The CFG, illustrated in Figure 6, is constructed from the basic blocks. The vertices represent the basic blocks labeled with their name and execution time. All edges are labeled with their execution frequency. The WCET can now be calculated and the result is 1588 cycles.

We can also measure the execution time of this simple example by running the JopCMP on the FPGA development board described in Section 3. The measured execution time of the loop example results in 1371 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. Within the measuring process, some array accesses are executed using fewer clock cycles than the worst case.

5.5.1 *WCET Dependency on the CMP Configuration.* In Table III, WCET estimates of the loop example are compared for varying system configurations. The number of CPUs varies between 1 and 8. The size of the time slot varies between 6 and 24 clock cycles. The third column shows the analyzed WCET results of the program depending on the CMP

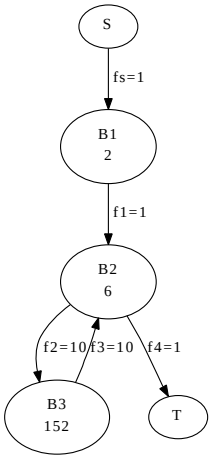

Fig. 6. Control flow graph of the simple loop.

configuration. The WCET estimate of a single JOP is 578 cycles. In this case, no time slot size is given because a single JOP does not have to share the main memory.

Only two bytecodes in the loop, *iaload* and *iastore*, access the main memory. The WCET estimates of all other bytecodes are not affected by the dividing up of memory bandwidth. The WCET varies depending on the time slot size. A time slot size between 12 and 18 cycles results in the smallest WCET estimate, independent of the number of CPUs. With this configurations, *iaload* can be executed within two time slots. If smaller time slot sizes are used, the execution needs more of them. If larger ones are used, the longer TDMA period dominates the execution time. The dual-core configuration with the time slot equaling 12, for example, results in a WCET of 31 cycles for *iaload* and 50 cycles for *iastore*. If the time slot is configured to 18 cycles, the WCET of *iaload* calculates to 37 and *iastore* to 44 cycles. The sums of the two bytecode WCETs are equivalent, regardless of whether the size of the time slot equals 12 or 18 cycles. Consequently, loop body WCETs in both configurations are equivalent as well.

Our experiments show that it is difficult to identify the optimal time slot for best WCET results. It depends on the occurrences of memory accessing bytecode types in the program code. In general, a time slice should be kept to a minimum for a single memory access. Consequently, the length of the TDMA period is minimized as well. Sometimes it can be advantageous to use larger time slots to allow more than one access within a slot (see Table III). The ideal slot size can be determined with the help of the WCET analysis tool.

5.6 WCET vs. Measured Execution Time

In the last column of Table III, measured execution time results are given. The loop programm is executed on one CPU, the other CPUs are idle. It has to be noted that the other CPUs have no influence on the execution time with a TDMA arbiter. The measured execution time and the WCET estimate are the same for a single JOP system, because of the characteristics of the example whereby only one execution path exists in the code.

All measured execution times vary depending on the time slot size. For a 2-way CMP with a 12-cycle time slot the measured execution time results in 965 cycles. The same

Table III. Analyzed WCET and measured execution time of the given loop example.

Configuration		Analyzed	Measured
# of CPUs	Time Slot (cycles)	WCET (cycles)	Exec. (cycles)
1	–	578	578
2	6	1058	965
2	12	1018	965
2	18	1018	747
2	24	1138	987
4	6	1778	1661
4	12	1738	1469
4	18	1738	1470
4	24	2098	1946
8	6	3218	3101
8	12	3178	2905
8	18	3178	2210
8	24	4018	3866

system with a slot size equaling 18 cycles executes using only 747 cycles. This result shows that *iaload* and *iastore* are frequently utilizing fewer time slots in an 18-cycle configuration, which explains the difference between measured and analyzed execution times. Furthermore, we can assert that the WCET estimates lie within an acceptable range.

In addition, we have used a benchmark called `Lift` as another example to calculate WCET estimates. `Lift` is a real-world example 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 [Schoeberl 2005b].

Table IV shows that the WCET of 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). Both CPUs execute the `Lift` benchmark simultaneously, so the WCET increases by 74%. The 8-way CMP version experiences an increase of 551% in execution time 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 IV 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. The authors believe that the pessimism in such cases is within an acceptable range for a multiprocessor WCET analysis.

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

Configuration		Analyzed	Measured		Pessimism
# of 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

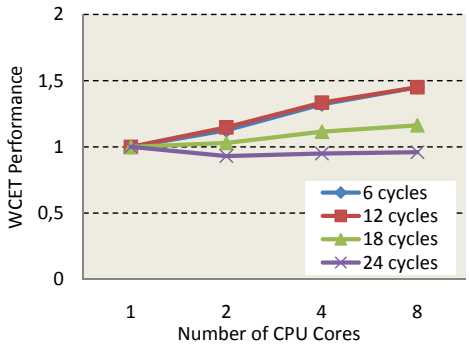

Fig. 7. WCET performance of the Lift benchmark.

5.7 WCET Performance

One interesting question is how the CMP system scales with respect to the WCET. Our 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.

We can use the WCET numbers from Table IV to estimate a possible increase of WCET performance. 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. Multiplying the increased executed workload, e.g. 8 for an 8-way system, by the performance decrease, e.g. 18%, the resulting speed-up is by a factor of 1.45.

Figure 7 shows the WCET performance through multiprocessing. There is a measurable WCET speed-up for the TDMA arbiter with relatively small 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. PERFORMANCE EVALUATION

Although the CMP is designed for hard real-time systems with a 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, we have used the FPGA-based platform to evaluate different configurations. We have also included FPGA synthesis results.

6.1 Benchmarks

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

—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.

6.1.1 *Lift Application.* The `Lift` benchmark, introduced in Section 5.6, is actually written for an uniprocessor. Nevertheless, we use it for executing several `Lift` tasks on multiple CPUs concurrently. This benchmark thus represents a medium computational, fully parallelized application without any synchronization needs.

6.1.2 *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.

6.1.3 *Embedded TCP/IP Stack.* As an example of an application with several communicating threads, we use an embedded TCP/IP stack for Java, called `ejip`. The benchmark explores the possibility of parallelization of 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.

Table V. 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

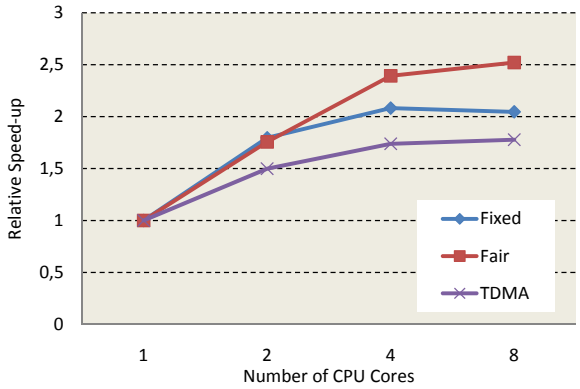

Fig. 8. Performance comparison of the Lift benchmark using different arbiters.

6.2 Measurements

Table V 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,

—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%. Actually, a 4-processor system using a fixed arbiter nearly 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. Executing the workload on more than 4 processors does not give a significantly better performance, irrespective of which arbiter is used.

Figure 8 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

Table VI. 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

Table VII. 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

shows that configurations up to 8 cores using the fair and the TDMA arbiters scale adequately. The saturation point of the fixed arbiter configuration lies at 4 cores. The reason for this slow-down is the large competition among the CPUs to access the memory.

Table VI depicts 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 [Pitter and Schoeberl 2008], as we have optimized the benchmark code. We are assuming that in a CMP system, for this kind of processing task, the code is optimized to avoid memory access as much as possible. The CMP speed-ups, consisting of 2 cores, lived up to our expectations with 1.5 (TDMA) and 1.7 (fixed and fair), respectively. The fixed arbitration policy scales well up to 4 processor cores. Adding more CPUs to the system does not result in a better performance. Whereas the first four CPUs calculate 97 rows altogether, the other CPUs calculate only 3 rows. Notably, CPU_6 and CPU_7 suffer from starvation because they never get their turn to calculate a single row. The fair arbiter distributes a workload evenly among all the processors. Each CPU calculates either 12 or 13 rows contributing to the final result. 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 VII. The application, consisting of five communicating threads, scales quite well up to 4 cores. With the fair 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.

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 real parallel threads compete more often for pool access. As a result, we intend to rewrite

Table VIII. Synthesis results of different CMP configurations on the Cyclone II FPGA (EP2C35).

Number of	Resources (LE)			Memory (KBit)	Frequency (MHz)		
JOP cores	Fixed	Fair	TDMA	all	Fixed	Fair	TDMA
1	3,329	3,329	3,329	62	104	104	104
2	6,534	6,789	6,849	92	102	103	101
4	12,974	13,461	13,568	184	101	100	100
8	25,622	26,877	26,870	368	97	90	90

the TCP/IP stack to use dedicated queues, preferably non-blocking, single reader/writer queues, for the communication between layers. Furthermore, we will explore a finer parallelization within the TCP/IP stack to use the available CPUs.

6.3 Synthesis Results

Table VIII shows the resource consumption of different multicore systems using the three different arbiters, synthesized with Altera Quartus II using the EP2C35 FPGA. Using 2 KB instruction and 1 KB of stack cache for each CPU, the resource consumption of logic elements (LE) and on-chip memory is balanced. CMPs with the same number of CPUs but different arbiter types need fairly the same amount of LEs and identical on-chip memory sizes. An 8-way CMP using the TDMA arbiter requires 81% of the available LEs and 76% of the memory bits on the low-cost Cyclone II. These results show that CMPs based on FPGA technology would prefer a different LE to on-chip memory ratio. Consequently, the anyway small cache sizes could be increased.

The maximum frequency varies between 104 MHz for a single JOP and 90 MHz for an 8-way CMP with the TDMA Arbiter. Using the phase-locked loop (PLL) of the FPGA, the clock frequency of all configurations is set to 90 MHz for the experiments. Even though our arbiters execute an arbitration decision in the same cycle the memory request happens, the different CMP versions scale quite well with respect to maximum clock frequency. Significant clock frequency degradation cannot be observed up to a multicore version with 8 CPUs.

7. DISCUSSION

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

7.1 Starvation

CMPs using a fair or a TDMA arbiter cannot suffer from starvation because each CPU gets the chance to access the memory. Only 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.

Table IX. Comparison of the arbitration policies.

Property	Fixed	Fair	TDMA
Starvation	-	+	+
Predictability	-	+/-	+
Performance	+/-	+	-

7.2 Predictability

Even though a timing analysis approach of each arbitration policy is presented in Section 5, some of them are not viable for hard real-time systems.

With the fixed priority arbiter only the highest priority CPU of a CMP is predictable. WCETs cannot be calculated for all 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.

A fair-based arbiter makes possible the 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. Compared to the TDMA arbiter, the current implementation cannot split a method cache load into several transactions.

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 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.

7.3 Performance

Using three benchmarks, we were able to demonstrate how multiple processors speed up real world applications. 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.

Comparing our JopCMP to a complex Java processor such as picoJava II, our conclusion is that a multiprocessor version of a simpler and smaller architecture is more efficient (performance/die area) for parallel workloads [Pitter and Schoeberl 2008].

7.4 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). We expected a lower increase of a real-time performance compared to the increase of an average-case performance. 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. We assume that the memory access now dominates the WCET. And for a TDMA arbiter the WCET of a single memory access on a 8-way CMP is 8 times longer than for a uniprocessor system. We consider this result a motivation 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 local cache analysis showed a WCET reduction of 15% using an 8-way configuration.

Furthermore, we have to consider a time-predictable solution for the caching of heap-allocated data. A small, fully associative buffer similar to a victim cache [Jouppi 1990] will allow detection of some cache hits in the WCET analysis.

8. CONCLUSION AND FUTURE WORK

Our research shows that timing analysis is in fact possible for homogeneous multiprocessor systems with a shared memory. In this paper, we have presented a Java multiprocessor architecture consisting of a number of JOP cores and a shared memory. An arbiter is used to synchronize memory access by multiple processors. Three different arbitration schemes are described, analyzed, and compared for viable use in real-time CMPs.

The key component enabling WCET analysis of the CMP is a TDMA arbiter. It splits up the memory access bandwidth into equal shares. We analyze the WCET of Java bytecodes instead of the WCET for a single memory access. These WCETs are dependent on time slot sizes, number of CPUs in the system, and memory access time. Therefore, JOP's WCET tool was enhanced to be able to integrate maximum latencies through memory access collisions of multiple CPUs.

Furthermore, examples of WCET calculations are presented, and WCET estimates are compared to measurement results. The experiments were carried out by executing three benchmarks on real hardware. Several experiments have demonstrated that we have to accept the exclusivity of applying either a time-predictable architecture or an architecture designed for average-case performance.

The outlook of further research of time-predictable multiprocessor systems has two closely related aspects: the design of multiprocessor architecture and the WCET analysis. Modifications within the architecture might make the analysis more intricate but an increase in processor performance is assured.

We plan to investigate the use of a percentage-based arbitration of the available memory access bandwidth, so memory bandwidth per CPU will be adjusted, dependent on the workload of the multiple CPUs. For example, if one CPU needs 60% of the available memory bandwidth, that is exactly what it will receive.

We also plan to investigate a fair arbiter upgrade. The method cache load might be interrupted after a predetermined period of time. This would result in a mix of fair and TDMA-based arbitration. Consequently, better average-case performance due to fair arbitration would be combined with the time predictability through a maximum access time.

The quality of the WCET analysis results will also be enhanced. We will extend the solution by performing a TDMA analysis on basic blocks in order to tighten the WCET bounds. The increased cache pressure on the memory bandwidth requires further research on time-predictable caching.

Acknowledgement

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).

REFERENCES

ALTERA. 2007a. Avalon Memory-Mapped Interface Specification (v3.3).

ALTERA. 2007b. Nios II Processor Reference Handbook (ver 7.2).

ALTERA. 2007c. Quartus II Handbook Volume 4: SOPC Builder (ver 7.2).

ANDREI, A., ELES, P., PENG, Z., AND ROSEN, J. 2008. Predictable implementation of real-time applications on multiprocessor systems-on-chip. In *VLSI Design*. IEEE Computer Society, 103–110.

ARM. 1999. AMBA Specification (rev 2.0).

ARM. 2006. Arm11 mpcore processor, technical reference manual. http://www.arm.com.

ARTIERI, A., D'ALTO, V., CHESSON, R., HOPKINS, M., ROSSI, M. C., AND PETERSON, W. D. 2004. Nomadik - open multimedia platform for next generation mobile devices, TA305 technical article. http://www.st.com.

DUTTA, S., JENSEN, R., AND RIECKMANN, A. 2001. Viper: A multiprocessor SOC for advanced set-top box and digital TV systems. *IEEE Design & Test of Computers 18,* 5, 21–31.

ERMEDAHL, A. AND ENGBLOM, J. 2007. Execution time analysis for embedded real-time systems. In *Handbook of Real-Time Embedded Systems*, S. H. S. Insup Lee, Jospeh Y-T. Leung, Ed. Chapman & Hall/CRC - Taylor and Francis Group, 35.1 – 35.17.

GAISLER, J. AND CATOVIC, E. 2006. Multi-Core Processor Based on LEON3-FT IP Core (LEON3-FT-MP). In *DASIA 2006 - Data Systems in Aerospace*. ESA Special Publication, vol. 630.

HENNESSY, J. AND PATTERSON, D. 2006. *Computer Architecture: A Quantitative Approach, 4th ed.* Morgan Kaufmann Publishers.

HOFSTEE, H. P. 2005. Power efficient processor architecture and the cell processor. In *HPCA*. 258–262.

IBM. 2001. On-chip peripheral bus architecture specifications v2.1.

IBM. 2007. 32-Bit OPB Arbiter Core Databook revision 1.

JOSEPH, M. AND PANDYA, P. K. 1986. Finding response times in a real-time system. *Comput. J 29,* 5, 390–395.

JOUPPI, N. P. 1990. 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*. Seattle, WA, 364–373.

KAHLE, J. A., DAY, M. N., HOFSTEE, H. P., JOHNS, C. R., MAEURER, T. R., AND SHIPPY, D. 2005. Introduction to the Cell multiprocessor. *j-IBM-JRD 49,* 4/5, 589–604.

KELTCHER, C. N., MCGRATH, K. J., AHMED, A., AND CONWAY, P. 2003. The AMD Opteron processor for multiprocessor servers. *IEEE Micro 23,* 2 (Mar./Apr.), 66–76.

KISTLER, M., PERRONE, M., AND PETRINI, F. 2006. Cell multiprocessor communication network: Built for speed. *Micro, IEEE 26*, 10–25.

KONGETIRA, P., AINGARAN, K., AND OLUKOTUN, K. 2005. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro 25,* 2, 21–29.

KOPETZ, H. 1997. *Real-time systems: design principles for distributed embedded applications*. Kluwer Academic Publishers.

LAUDON, J. AND SPRACKLEN, L. 2007. The coming wave of multithreaded chip multiprocessors. *International Journal of Parallel Programming 35,* 3 (June), 299–330.

LI, Y.-T. S. AND MALIK, S. 1995. Performance analysis of embedded software using implicit path enumeration. In *Workshop on Languages, Compilers, & Tools for Real-Time Systems*. 88–98.

LICKLY, B., LIU, I., KIM, S., PATEL, H. D., EDWARDS, S. A., AND LEE, E. A. 2008. Predictable programming on a precision timed architecture. In *Proceedings of International Conference on Compilers, Architecture, and Synthesis from Embedded Systems*.

LINDHOLM, T. AND YELLIN, F. 1999. *The Java Virtual Machine Specification*, Second ed. Addison-Wesley, Reading, MA, USA.

LIU, C. L. AND LAYLAND, J. W. 1973. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM 20,* 1, 46–61.

MARTIN, G. AND CHANG, H. 2003. *Winning the SOC Revolution, Chapter 5*. Kluwer Academic Publishers.

MOORE, G. E. 1965. Cramming more components onto integrated circuits. *Electronics 38,* 8, 114–117.

PITTER, C. 2008. 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)*. ACM Press, Santa Clara, USA.

PITTER, C. 2009. Time-predictable Java chip-multiprocessor. Ph.D. thesis, Vienna University of Technology, Austria.

PITTER, C. AND SCHOEBERL, M. 2007a. Time predictable CPU and DMA shared memory access. In *International Conference on Field-Programmable Logic and its Applications (FPL 2007)*. Amsterdam, Netherlands.

PITTER, C. AND SCHOEBERL, M. 2007b. Towards a Java multiprocessor. In *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems (JTRES 2007)*. ACM Press, Vienna, Austria.

PITTER, C. AND SCHOEBERL, M. 2008. Performance Evaluation of a Java Chip-Multiprocessor. In *Proceedings of the IEEE Third Symposium on Industrial Embedded Systems (SIES 2008)*. Montpellier, France.

POLETTI, F., BERTOZZI, D., BENINI, L., AND BOGLIOLO, A. 200306/09. Performance Analysis of Arbitration Policies for SoC Communication Architectures. *Design Automation for Embedded Systems 8*, 189–210(22).

PUSCHNER, P. AND BURNS, A. 2000. A review of worst-case execution-time analysis. *Journal of Real-Time Systems 18,* 2/3 (May), 115–128.

ROSEN, J., ANDREI, A., ELES, P., AND PENG, Z. 2007. Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip. In *RTSS*. IEEE Computer Society, 49–60.

SCHOEBERL, M. 2004. 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)*. LNCS, vol. 3292. Springer, Agia Napa, Cyprus, 371–382.

SCHOEBERL, M. 2005a. Design and implementation of an efficient stack machine. In *Proceedings of the 12th IEEE Reconfigurable Architecture Workshop (RAW2005)*. IEEE, Denver, Colorado, USA.

SCHOEBERL, M. 2005b. Jop: A java optimized processor for embedded real-time systems. Ph.D. thesis, Vienna University of Technology.

SCHOEBERL, M. 2007. SimpCon - a simple and efficient SoC interconnect. In *Proceedings of the 15th Austrian Workhop on Microelectronics, Austrochip 2007*. Graz, Austria.

SCHOEBERL, M. 2008. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture 54/1–2*, 265–286.

SCHOEBERL, M. AND PEDERSEN, R. 2006. WCET analysis for a Java processor. In *Proceedings of the 4th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2006)*. ACM Press, New York, NY, USA, 202–211.

SIEBERT, F. 2008. 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*. ACM, New York, NY, USA, 87–93.

SPARC INTERNATIONAL INC. 1992. *The SPARC Architecture Manual: Version 8*. Prentice Hall, Englewood Cliffs, New Jersey 07632.

THIELE, L. AND WILHELM, R. 2004. Design for timing predictability. *Real-Time Systems 28,* 2-3, 157–177.

WILHELM, R., ENGBLOM, J., ERMEDAHL, A., HOLSTI, N., THESING, S., WHALLEY, D. B., BERNAT, G., FERDINAND, C., HECKMANN, R., MITRA, T., MUELLER, F., PUAUT, I., PUSCHNER, P. P., STASCHULAT, J., AND STENSTRÖM, P. 2008. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst 7,* 3, 1–53.

WOLF, W. 2006. *High-Performance Embedded Computing: Architectures, Applications, and Methodologies*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

XILINX. 2005. OPB Arbiter product specification (v1.10c).

XILINX. 2007. MicroBlaze Processor Reference Guide, Embedded Development Kit EDK 9.2i. http://www.xilinx.com.

Received November 2008; revised April 2009; accepted Month Year