Towards a Java Multiprocessor

Pitter, Christof; Schoeberl, Martin

Published in:
Proceedings of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2007)

Publication date:
2007

Document Version
Early version, also known as pre-print

Towards a Java Multiprocessor

Christof Pitter
Institute of Computer Engineering
Vienna University of Technology, Austria
cpitter@mail.tuwien.ac.at

Martin Schoeberl
Institute of Computer Engineering
Vienna University of Technology, Austria
mschoebe@mail.tuwien.ac.at

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.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Real-time 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*

Keywords

Multiprocessor, Java, Shared Memory

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 ap-

JTRES '07 September 26-28, 2007 Vienna, Austria

plications. 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 2 presents related work. In Section 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 4 describes the first implementation of the JopCMP system, including the boot-up sequence and the shared memory access management of the CMP. Section 5 presents experiments with the JopCMP prototype. We compare the performance of the CMP against a single processor. Finally, Section 6 concludes the paper and provides guidelines for future work.

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.

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.

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. CMP ARCHITECTURE

According to [12, 30], two different possibilities of a tightly coupled multiprocessor system exist (see Figure 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. Communication 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

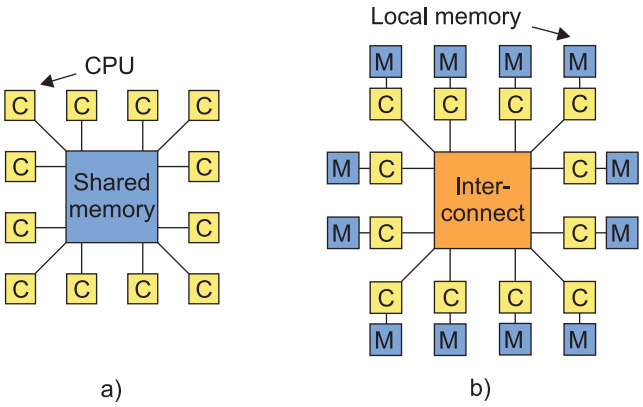

Figure 1: CMP Memory Models: a) Shared memory model, b) Distributed shared memory model.

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.

3.1 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.

3.2 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 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 3.2, 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.

3.4 CMP using JOPs

The proposed chip multiprocessing system (see Figure 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 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,

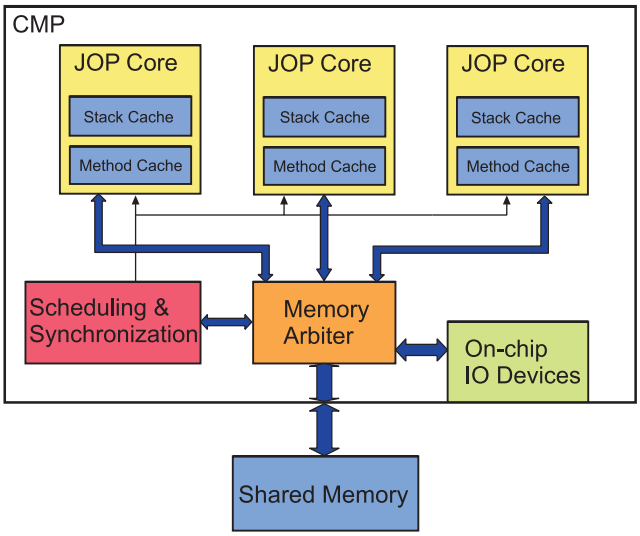

Figure 2: Time predictable CMP architecture.

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.

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.

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.

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.

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 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 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

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

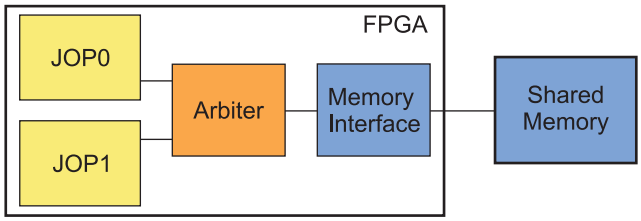

Figure 3: Dual-core JopCMP system.

to solve this problem in our future work in order to provide a time predictable shared memory access scheme.

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 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.

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 1 shows the performance numbers at different clock frequencies for the single-core.

Table 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

Table 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

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 1).

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

$$Speedup_{dualcore} = \frac{12951 + 12951}{13138} \approx 1.97. \quad (1)$$

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 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.

Table 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

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 depicts the results of the CMP. Equation 2 presents the calculation of the speedup. Only 7% slowdown relative to the theoretical maximum speedup 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.

$$Speedup_{tricore} = \frac{11736 + 11538 + 11260}{12318} \approx 2.80. \quad (2)$$

5.2 Comparison Against a Single Core

As we see in Table 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 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%.

$$Speedup_{dualcore} = \frac{12951 + 12951}{16425} \approx 1.58. \quad (3)$$

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 1). Table 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

$$Speedup_{tricore} = \frac{11736 + 11538 + 11260}{16425} \approx 2.10. \quad (4)$$

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.

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.

5.3 Resource Consumption

Finally, Table 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

Table 4: Comparison of resource consumption between JOP and the JopCMP versions.

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

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.

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.

7. REFERENCES

[1] R. C. M. H. Alain Artieri, Viviana D'Alto and W. D. P. 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. http://www.arm.com, August 2006.

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

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

[7] B. Dobbing and A. 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] S. Dutta, R. Jensen, and A. 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] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach, 4th ed.* Morgan Kaufmann Publishers, 2006.

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

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

[12] V. M. 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] M. Kistler, M. Perrone, and F. Petrini. Cell multiprocessor communication network: Built for speed. *Micro, IEEE*, 26:10–25, 2006.

[15] P. Kongetira, K. Aingaran, and K. 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] J. Laudon and L. Spracklen. The coming wave of multithreaded chip multiprocessors. *International Journal of Parallel Programming*, 35(3):299–330, June 2007.

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

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

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

[21] C. Pitter and M. 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] M. Schoeberl. *JOP: A Java Optimized Processor for Embedded Real-Time Systems*. PhD thesis, Vienna University of Technology, 2005.

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

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

[27] M. Schoeberl and R. 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] M. Schoeberl, H. Sondergaard, B. Thomsen, and A. 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] W. Stallings. *Operating Systems: Internals and Design Principles, 5th ed.* Prentice Hall, 2005.

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

[31] C.-D. Wang, H. Takada, and K. Sakamura. Priority inheritance spin locks for multiprocessor real-time systems. In *ISPAN*, pages 70–76. IEEE Computer Society, 1996.

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