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

This paper focuses on the performance evaluation of different hardware configurations of the multicore system. Therefore, we vary the instruction cache sizes, the number of processors and the memory bandwidth. Within our experiments, we measure the performance by running three benchmarks on real hardware: an embedded application from industry, a computationally intensive matrix multiplication and a synthetic benchmark that continuously accesses a shared data structure. Two different field-programmable gate arrays are used for the presented experiments.

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

I. Introduction

It is expected that chip-level multiprocessing (CMP) will be the future path of performance enhancements [1]. CMP technology integrates two or more processing units and a sophisticated communication network into a single integrated circuit (IC). In actual desktop and server architectures, two trends can be seen: (1) integration of two to four out-of-order super-scalar CPUs (Intel/AMD) on a single die or (2) integration of 8 very simple in-order RISC pipelines (Sun's Niagara [2] and IBM's CELL [3]). According to [4], multiprocessing is also common in embedded systems as it combines the goals of increasing performance, lower power consumption and cost effectiveness.

In this paper, we propose a CMP architecture composed of multiple Java Optimized Processor (JOP) cores and a shared memory. The shared memory is uniformly accessible by the homogeneous processing cores. A system-on-chip (SoC) bus connects the devices of the system. A fairness-based arbitration unit takes care of memory contention due to parallel memory requests; it employs a fair scheduling strategy, which divides the memory access bandwidth into equal shares. In contrast to existent memory arbiters of SoC buses (e.g. AMBA [5] or Avalon [6]), the proposed arbitration process is performed in the same cycle as the request happens. This feature increases the memory bandwidth.

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

The rest of the paper is structured as follows. Section 2 presents related work and delivers an insight into JOP. In Section 3, we describe the proposed CMP architecture and the interconnection network. Furthermore, the memory controller and the synchronization unit of JopCMP are summarized concisely. Section 4 presents the evaluation method (benchmarks and hardware platforms) and the benchmark results. Finally, Section 5 concludes the paper and provides guidelines for future work.

II. Related Work

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

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

A. LEON

Gaisler Research AB designed and implemented a homogeneous multiprocessor system called LEON3-FT-MP [7]. It consists of a centralized shared memory and four LEON3-FT processor cores that are based on the SPARC V8 instruction set architecture [8]. Each CPU consists of a 7-stage pipeline with separate 16 KB data and instruction caches, a memory management unit, a floating-point unit and hardware support for multiplication and division. All the CPUs, additional I/O controllers and the memory controllers are connected using two advanced high-performance buses (AHB) of the AMBA specification [9]. One AHB runs at the CPUs' frequency of 266 MHz and connects the processors with the memory controller of the shared memory. Additionally the high-speed AHB communicates with the low-speed AHB (running at 133 MHz) using an AHB/AHB bridge. The low-speed AHB connects all other peripheral devices with lower speed requirements to the system. The bus frequencies reported in [7] are estimations of a 0.13μ ASIC implementation. A prototype in a Xilinx Virtex-4 can run at 40 MHz.

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

B. MicroBlaze

MicroBlaze [10] based CMPs can be designed with the Xilinx Embedded Development Kit (EDK). MicroBlaze is a 32-bit reduced instruction set computer (RISC) optimized for FPGA implementation. The pipeline length of the CPU can be configured to either 3 or 5 stages. It implements the Harvard architecture with separate instruction and data buses. The CPU can be tailored to the application needs (i.e. peripheral controllers or cache sizes).

Memory and peripheral devices are connected via the on-chip peripheral bus (OPB) [11]. OPB is part of the bus hierarchy called CoreConnect [12], an open standard for SoC communication proposed by IBM. Xilinx provides an OPB bus arbiter [13], [14] that can integrate up to 16 masters into the system. The available arbitration algorithms include fixed priority (FP) or least recently used (LRU). A full-featured GNU tool chain is available for software development in C/C++.

C. NIOS II

Altera's Nios II [15] and the System-on-a-Programmable Chip (SOPC) Builder [16] support the design and implementation of CMPs in Altera's FPGA technology. The Nios RISC architecture implements a 32-bit instruction set similar to the MIPS instruction set architecture. The sizes of its instruction and data caches are configurable. Nios II can be customized to meet the application requirements: three different models from non-pipelined up to a 6-stage pipeline. Examples of customizable features are a floating-point unit, memory controllers and different communication controllers. Avalon [17] is the SoC bus used by the SOPC Builder. It connects the master and slave components to the System Interconnect Fabric. This System Interconnect Fabric encapsulates all connection details from the user. While the Avalon specification can be used freely, the System Interconnect Fabric is Altera's property.

For multiprocessor systems, the System Interconnect Fabric integrates an arbitration module [18]. In contrast to traditional shared bus architectures, the interconnection allows multiple masters to access different slaves simultaneously. This eliminates the bottleneck of one shared bus if one master may access a slave and another master wants to access a different slave in parallel. For a multiprocessor system where two or more masters frequently access one slave (the shared memory), the System Interconnect Fabric provides no advantage. If several masters request data from the same slave, an arbiter will determine which master will gain access. All other masters are forced to wait. The arbitration logic can be configured in the SOPC Builder. The arbitration schemes include fairness-based shares, round-robin scheduling, burst transfers, and minimum share value. The Nios II system supports the uClinux operating system and the C/C++ GNU tool chain is available.

D. Java Optimized Processor (JOP)

The Java optimized processor (JOP) [19]–[21] is an implementation of the Java Virtual Machine (JVM) in hardware. JOP translates the Java intermediate bytecodes to its own instruction set called microcode. These microcode instructions, implemented in hardware, are executed by the stack architecture. The CPU has a 4-stage pipeline.

Each thread has a local stack area. This thread private data is accessed very often. Therefore, JOP caches this data in a so-called stack cache [22]. Additionally, a kind of instruction cache (called method cache [23]) limits the memory access frequency and increases the processing power. Complete methods, shared among all the threads, are cached there. According to the JVM specification [24], the heap stores the shared data of the VM. All objects that are created by a Java application are stored on the heap. Caching of these objects is not

implemented. A typical JOP configuration contains the CPU core, the method and stack cache, a memory interface and several I/O controllers.

JOP is designed for embedded real-time systems where the analysis of the worst-case execution time (WCET) of all threads is possible. Hence, a couple of typical architectural advancements, used to increase the average processing power, have been omitted. Examples include branch prediction or out-of-order execution. Nevertheless, JOP shows good average performance and lower logic resources consumption in comparison to other Java processors. Therefore, our Java multiprocessor system is based on JOP.

E. Discussion

According to [25], the inter-core communication in CMPs offers more bandwidth than traditional backplane buses used for building traditional SMPs. Additionally, the latency of a transfer is much lower on an SoC bus. The described multiprocessors are still using backplane style buses that are not appropriate for a SoC interconnection. Furthermore, there is no use for a complex bus hierarchy in our design. Our system consists of a couple of CPUs connected to a single shared memory. Therefore, our choice of the interconnection network is the simple SoC bus called SimpCon [26], which is further described in Section III-B. Moreover, we use a fairness-based arbitration algorithm. To our knowledge, Leon's IP library does not include a fair arbiter. A disadvantage of Nios II based multiprocessor systems is that data cache coherency is not supported. The data caches are disabled for a multiprocessor system. In our proposed solution, we limit data caching to the thread private JVM stack.

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

III. Overview of JopCMP

According to [4], a multiprocessor system consists of 3 major subsystems: processing elements, memory and an interconnection network. JopCMP implements the symmetric (shared-memory) multiprocessor (SMP) model [1]. JOPs provide the basis of the homogeneous CMP as depicted in Figure 1. These processing elements perform computations in parallel. Instructions and data are stored in a single shared memory. The interconnection network is responsible to connect multiple processors with the memory. An arbiter is part of this network and controls the memory access to the shared memory. An SoC bus is used to connect the processing cores to the arbiter, and the arbiter to the shared memory. The arbiter acts as slave for each JOP and as master for the memory controller. We are convinced that synchronization of shared data is a further major subsystem of an SMP. It is responsible to coordinate access to shared objects. Following sections describe the elements in more detail.

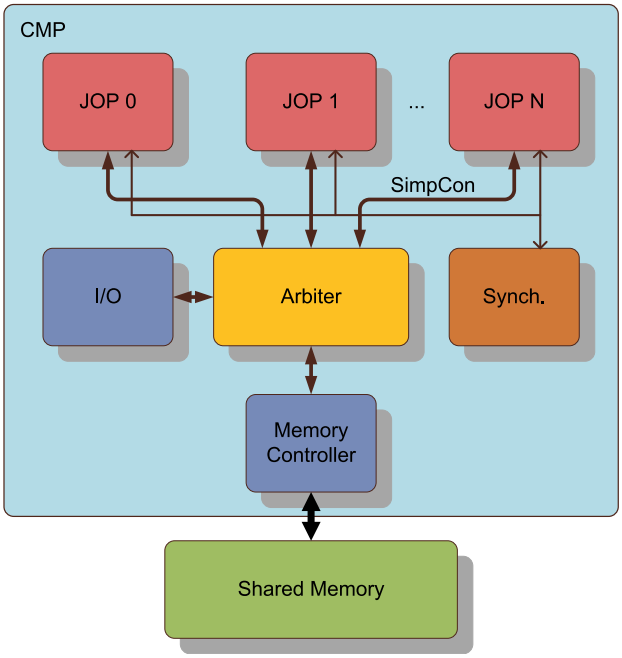

Fig. 1. Overview of JopCMP.

A. Memory Hierarchy

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

B. Interconnection Network

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

Traditionally, a bus is a set of wires that connects multiple masters and multiple slaves of a system on a printed circuit board. A master initiates each communication. All communication channels between the interfaces of the exchanging devices represent the so-called backplane bus. The bus arbiter implements a certain priority algorithm that controls this shared bus. If multiple masters request access to the bus, the arbiter will allocate the shared bus resource to one master. All other masters are forced to wait. Consequently, multiple masters cannot concurrently drive the bus.

The simple SoC interconnect (SimpCon) [26] is used to connect SoC components on a single IC. This synchronous on-chip interconnection is intended for read and write transfers via point-to-point connections. Only a master can initiate a transaction via a write or a read request. In comparison to other commonly used SoC buses like Avalon [18], this specification

does not work like a backplane bus. The master's driven control, address, and data lines are only valid for a single clock cycle. A slave has to register any signals (e.g., the address) that are needed for several clock cycles. Consequently, the master can continue to execute its task until the data of an access is needed. Furthermore, the slave can early inform the master (up to two cycles ahead) when a bus transaction will finish. Therefore, pipelined memory accesses and consequently fast data transfers are possible.

C. Fairness-based Arbiter

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

Usually, the arbitration policy of the arbiter depends on the application needs. An example of a dynamic arbitration scheme depending on the CPU priorities is described in [27]. Each CPU of the system is assigned a unique priority. If memory access contention occurs, the CPU with the highest priority will be granted access. This arbitration policy can be used for real-time systems where one CPU executes hard real-time[1] tasks and the other ones execute tasks with minor requirements regarding deadlines. Consequently, the hard real-time CPU gets the highest priority of the system.

In this paper, we analyze the performance of a balanced CMP. Therefore, an arbitration policy is implemented that guarantees fairness among the CPUs accessing the shared memory. Furthermore, starvation of any CPUs is prohibited. Each CPU in the system is assigned a unique CPU identity (CPU_{ID}), starting from 0 up to the number of CPUs-1. Our fair arbitration policy uses a wrapping counter. It changes the permission, which CPU is allowed to access the memory. The value of the counter has the same range as the CPU identities. At the time of completion of the prior memory access, the counter is advanced. If the new counter value equals a requesting CPU_{ID} and the memory is ready to execute a memory access, the memory access will be processed and the current value of the counter will be halted until completion of the data transmission. In the case that the CPU with CPU_{ID} that equals the value of the counter does not want to access the memory, the counter is immediately advanced.

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

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

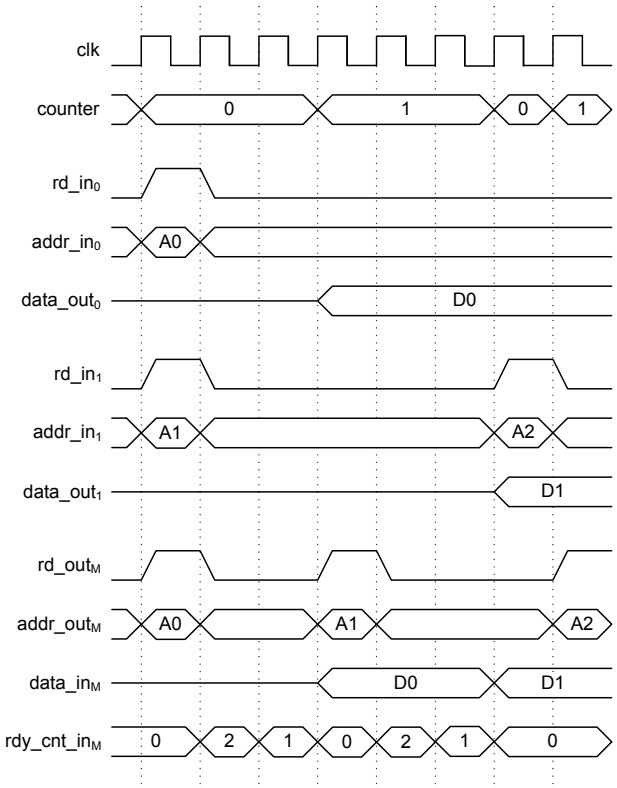

Fig. 2. Memory access arbitration of the fairness-based arbiter.

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

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

introduced.

The arbiter performs the arbitration decision in the same cycle as the request arrives. Therefore, we have a zero-cycle arbitration protocol. No additional cycle is lost for arbitration and the memory access latency is not affected. Zero-cycle arbitration latency is an advantage in comparison to existing arbiters like AMBA [9] and Avalon [18], which always use an extra cycle for a so-called bus request phase. Consequently, each access takes more time. This fairness-based arbiter is implemented at Register Transfer Level (RTL) in VHDL. The arbitration process is primarily implemented in combinational logic without considerably decreasing the clock frequency of the whole system.

The arbiter is fully scalable with respect to the number of connected masters. Compared to existing arbiters like AMBA [9] or CoreConnect [13] the maximum number of connected masters is not limited to 16. Hence, the application determines the quantity of connected masters.

D. Memory Controller

The memory controller is the interconnection between the arbiter and the shared memory. Controllers for different types of memory are available for the SimpCon bus. The controller supports pipelined read and write commands. Pipelining of SimpCon, the arbiter, and the memory controller allows back-to-back reads from the memory. Furthermore, due to the definition of the SimpCon interconnect, it is possible to use the registers in the IO cells of the FPGA for all latency-sensitive SRAM signals (address bus, data bus, and control lines).

E. Synchronization

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

Therefore, we introduced a synchronization unit in hardware that controls one global lock. If one core wants to access a shared object, it will request the lock using the synchronization interconnection depicted in Figure 1. JOP will be granted access if no other processor of the system is holding the lock. Otherwise, it must wait until the other processor completes accessing the shared object.

The hardware lock allows fast implementation of the bytecodes `monitorenter` and `monitorexit` that are used by the JVM for synchronization. For short critical sections, this feature compensates for the less reactive behavior of a single global lock. A side effect of a single lock is the avoidance of deadlock by design. Further information on synchronization of JopCMP can be found in [27].

IV. Performance Evaluation

In this section, we evaluate the performance of our CMP architecture. We present and compare the performance of our multicore depending on the number of processors and the size of their instruction caches. The benchmarks highlight that several processors working in parallel outperform a uniprocessor that executes the same workload sequentially. Two different hardware platforms set up the basis of the presented experiments. Furthermore, we include FPGA synthesis results and compare performance and size of JopCMP with the complex Java processor picoJava [28].

A. Benchmarks

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

- a real-world embedded application from industry (`Lift`),
- a matrix multiplication (`MMul`) and
- an application that is operating on a hash table (`HTable`).

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

1) Lift Application: `Lift` is a real-world example with industrial background. This embedded application is a lift controller used in an automation factory. `Lift` is part of the embedded Java benchmark suite called JavaBenchEmbedded, as described in [19]. In fact, the benchmark is written to measure uniprocessor performance. Nevertheless, we use it for executing several `Lift` tasks on multiple CPUs concurrently. Consequently, this benchmark presents a medium computational, fully parallelized application without any accesses to shared data structures and synchronization needs.

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

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

B. Hardware Platforms

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

TABLE I
EXECUTION TIME AND MEMORY BANDWIDTH UTILIZATION OF LIFT, ALTDE2 @ 90 MHZ

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

TABLE II
EXECUTION TIME AND MEMORY BANDWIDTH UTILIZATION OF MMUL, ALTDE2 @ 90 MHZ

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

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

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

C. Measurements

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

As it is expected that the memory bandwidth will restrict the number of useful cores we also measure the consumed bandwidth. We have integrated a memory access counter into the memory controller to measure the number of cycles the memory bus is busy. Equation 1 gives the memory load relative to the available memory bandwidth. The resulting memory bandwidth utilization depends on the size of the instruction cache. It decreases with larger cache sizes because memory access frequency drops as well.

$$Utilization_{Mem.Bandwidth} = \frac{MemoryAccessTime}{ExecutionTime} \quad (1)$$

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

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

Table II depicts the results of the measurement of `MMul` on the DE2 platform. The computationally intensive algorithm demonstrates its good potential for parallelism. The speedup of the CMPs consisting of 2 and 4 cores comes up to our expectations with speedups of 1.5 and 1.7 accordingly. 8 cores provide no additional significant speedup. We assume that a combination of high memory conflicts and increased synchronization cost becomes noticeable. The memory bandwidth utilization remains constant at 86% independent of a 4- or 8-way CMP. Increasing the cache size does not result in any performance improvements or any change of the consumed memory bandwidth.

Table III shows the measurements of `HTable` running at a frequency of 90 MHz on the DE2 board. Unlike `Lift` and `MMul`, this benchmark results in a small performance slowdown comparing a single JOP with a 2-core system with 1 KB of cache. Only slight speedups with the 2 KB and 4 KB versions can be seen. This originates from the application's characteristics. It combines low computational demands with high synchronization overhead. Nevertheless, CMPs with more CPUs cover this overhead by introducing more processing power. The 8-core JopCMP with 2 KB of cache is about 1.6 times faster than the comparable single-JOP. `HTable`'s high

TABLE III
EXECUTION TIME AND MEMORY BANDWIDTH UTILIZATION OF HTABLE, ALTDE2 @ 90 MHZ

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

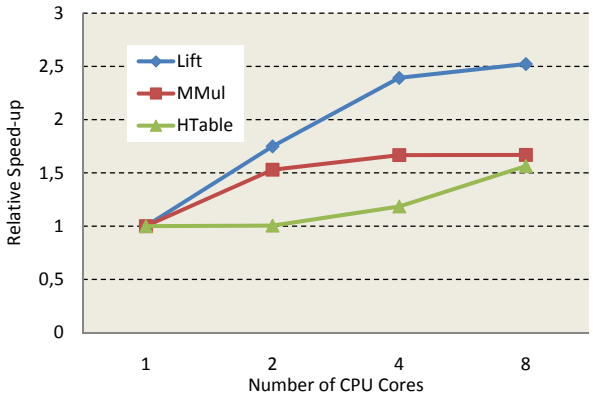

Fig. 3. Performance comparison of JopCMP, running three different benchmarks.

synchronization demands are easily noticeable by the figures of the memory bandwidth utilization. These results show a very small increase when adding more CPUs to the CMP.

Figure 3 summarizes the measured execution times of the configurations with 2 KB cache size. The horizontal axis describes the number of CPUs and the vertical axis illustrates the relative speed-up. The relative speedup is the relation between the execution time on a single core and a multi core version. We can see different saturation points for `Lift` and `MMul`. Interesting is the result of the `HTable` benchmark: it only shows a significant speedup with many cores and we do not see the saturation at 8 cores. For this benchmark, it would be interesting to run a 16-way multicore version.

TABLE IV
EXECUTION TIME AND MEMORY BANDWIDTH UTILIZATION OF LIFT, CYCORE @ 60 MHZ

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

TABLE V
PERFORMANCE AND SIZE OF JOPCMP RELATIVE TO PICOJAVA IN THE SAME FPGA BOARD

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

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

D. Comparison with a Complex Java Processor

picoJava II [28], Sun's hardware implementation of the JVM, consumes more resources than JOP. An interesting comparison is whether a single, complex processor or a multiprocessor based on simple processors performs better for multi-threaded workloads. In the server domain Sun has chosen to implement 8 simple RISC cores (with a 6-stage in-order pipeline) in the CMP Niagara [2] while Intel and AMD still use their complex super-scalar out-of-order architectures.

Puffitsch has implemented the picoJava II in an FPGA [29] on the same board (DE2) that we use for our evaluation. picoJava II runs at 40 MHz in the Cyclone II. The resource consumption is 27,562 LEs and 381 KBit memory when configured with 16 KB instruction and 16 KB data cache.

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

The benchmark `HTable` executes in 165 ms on picoJava.

TABLE VI
SYNTHESIS RESULTS ON THE CYCLONE II FPGA (EP2C35)

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

On this synthetic, lock intensive benchmark picoJava is almost three times faster than JOP. picoJava uses two lock registers to cache the last two obtained locks. This hardware feature allows execution of bytecodes `monitorenter` and `monitorexit` in 3 and 2 cycles when the lock is found in one of the lock registers. JOP needs for those operations with the global lock 18 and 20 cycles.

E. Synthesis Results

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

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

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

V. CONCLUSION AND FUTURE WORK

We have shown that even in a medium sized low-cost FPGA it is possible to run 8 cores of a Java processor in parallel. The performance enhancements of factors 1.7 and 2.5 with 2- and 4-way cores for a real-world application are promising. We did not expect a linear improvement in speed. A speedup logarithmic to the number of cores would satisfy future processing demands as the number of transistors that can be integrated into a single chip is still increasing exponentially.

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

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

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

Comparing our JopCMP against a complex Java processor, such as picoJava II, we conclude that a multiprocessor version of a simpler and smaller architecture is more efficient (performance/die area) for parallel workloads. With independent instances of the application benchmark `Lift` a 4-core version of JopCMP is 1.3 times faster than picoJava with a die area of about 45% of picoJava.

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

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

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

ACKNOWLEDGEMENT

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

comparison.

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

REFERENCES

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

[2] P. Kongetira, K. Aingaran, and K. Olukotun, "Niagara: A 32-way multithreaded sparc processor," *IEEE Micro*, vol. 25, no. 2, pp. 21–29, 2005.

[3] 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*, vol. 49, no. 4/5, pp. 589–604, 2005.

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

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

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

[7] J. Gaisler and E. Catovic, "Multi-Core Processor Based on LEON3-FT IP Core (LEON3-FT-MP)," in *DASIA 2006 - Data Systems in Aerospace*, ser. ESA Special Publication, vol. 630, Jul. 2006.

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

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

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

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

[12] ——, "The CoreConnect Bus Architecture," http://www.ibm.com, 1999.

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

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

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

[16] ——, "Quartus II Handbook Volume 4: SOPC Builder (ver 7.2)," October 2007.

[17] ——, "Avalon interface specification," April 2005.

[18] ——, "Avalon Memory-Mapped Interface Specification (v3.3)," May 2007.

[19] M. Schoeberl, "Jop: A java optimized processor for embedded real-time systems," Ph.D. dissertation, Vienna University of Technology, 2005.

[20] ——, "A time predictable Java processor," in *Proceedings of the Design, Automation and Test in Europe Conference (DATE 2006)*, Munich, Germany, March 2006, pp. 800–805.

[21] ——, "A Java processor architecture for embedded real-time systems," *Journal of Systems Architecture*, vol. doi:10.1016/j.sysarc.2007.06.001, 2007.

[22] ——, "Design and implementation of an efficient stack machine," in *Proceedings of the 12th IEEE Reconfigurable Architecture Workshop (RAW2005)*. Denver, Colorado, USA: IEEE, April 2005.

[23] ——, "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)*, ser. LNCS, vol. 3292. Agia Napa, Cyprus: Springer, October 2004, pp. 371–382.

[24] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*, 2nd ed. Reading, MA, USA: Addison-Wesley, 1999.

[25] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The landscape of parallel computing research: A view from berkeley," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2006-183, Dec. 2006.

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

[27] C. Pitter and M. Schoeberl, "Towards a Java multiprocessor," in *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems (JTRES 2007)*. Vienna, Austria: ACM Press, September 2007.

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

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