Evaluation of a Java Processor

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

***Abstract*— In this paper, we will present the evaluation results for a Java processor, with respect to size and performance. The Java Optimized Processor (JOP) is an implementation of the Java virtual machine (JVM) in a low-cost FPGA. JOP is the smallest hardware realization of the JVM available to date. Due to the efficient implementation of the stack architecture, JOP is also smaller than a *comparable* RISC processor in an FPGA.**

Although JOP is intended as a processor for embedded real-time systems, whereas accurate worst case execution time analysis is more important than average case performance, its general performance is still important. We will see that a real-time processor architecture does not need to be slow.

I. Introduction

In this paper, we will present the evaluation results for a Java processor [1], with respect to size and performance. This Java processor is called JOP – which stands for Java Optimized Processor –, based on the assumption that a full native implementation of all Java Virtual Machine (JVM) [2] bytecode instructions is not a useful approach. JOP is a Java processor for embedded real-time systems, in particular a small processor for resource-constrained devices with time-predictable execution of Java programs.

Table I lists the relevant Java processors available to date. Sun introduced the first version of picoJava [3] in 1997. Sun's picoJava is the Java processor most often cited in research papers. It is used as a reference for new Java processors and as the basis for research into improving various aspects of a Java processor. Ironically, this processor was never released as a product by Sun. A redesign followed in 1999, known as picoJava-II that is now freely available with a rich set of documentation [4], [5]. The architecture of picoJava is a stack-based CISC processor implementing 341 different instructions and is the most complex Java processor available. The processor can be implemented [6] in about 440K gates.

aJile's JEMCore is a direct-execution Java processor that is available as both an IP core and a stand alone processor [7], [8]. It is based on the 32-bit JEM2 Java chip developed by Rockwell-Collins. The processor contains 48KB zero wait state RAM and peripheral components. 16KB of the RAM is used for the writable control store. The remaining 32KB is used for storage of the processor stack.

Vulcan ASIC's Moon processor is an implementation of the JVM to run in an FPGA. The execution model is the often-used mix of direct, microcode and trapped execution. As described in [9], a simple stack folding is implemented in order to reduce five memory cycles to three for instruction sequences like *push-push-add*. The Moon2 processor [10] is available as an encrypted HDL source for Altera FPGAs or as VHDL or Verilog source code.

TABLE I
JOP and various Java processors

	Target technology	Size Logic	RAM	Speed [MHz]
JOP	Altera, Xilinx FPGA	1830 LCs	3KB	100
picoJava	No realization	128K gates	38KB	
aJile	ASIC 0.25μ	25K gates	48KB	100
Moon	Altera FPGA	3660 LCs	4KB	
Lightfoot	Xilinx FPGA	3400 LCs	4KB	40
Komodo	Xilinx FPGA	2600 LCs		33
FemtoJava	Xilinx FPGA	2710 LCs	0.5KB	56

The Lightfoot 32-bit core [11] is a hybrid 8/32-bit processor based on the Harvard architecture. Program memory is 8 bits wide and data memory is 32 bits wide. The core contains a 3-stage pipeline with an integer ALU, a barrel shifter and a 2-bit multiply step unit. According to DCT, the performance is typically 8 times better than RISC interpreters running at the same clock speed.

Komodo [12] is a multithreaded Java processor with a four-stage pipeline. It is intended as a basis for research on real-time scheduling on a multithreaded microcontroller. The unique feature of Komodo is the instruction fetch unit with four independent program counters and status flags for four threads. A priority manager is responsible for hardware real-time scheduling and can select a new thread after each bytecode instruction.

FemtoJava [13] is a research project to build an application specific Java processor. The bytecode usage of the embedded application is analyzed and a customized version of FemtoJava is generated in order to minimize the resource usage. FemtoJava is not included in Section IV, as the processor could not run even the simplest benchmark.

Besides the *real* Java processors a few FORTH chips (Cjip [14], PSC1000 [15]) are marketed as Java processors. Java coprocessors (Jazelle [16], JSTAR [17]) provide Java execution speedup for general-purpose processors.

From the Table I we can see that JOP is the smallest realization of a hardware JVM in an FPGA and also has the highest clock frequency.

In the following section, a brief overview of the architecture of JOP is given, followed by a more detailed description of the microcode. Section III compares JOP's resource usage with

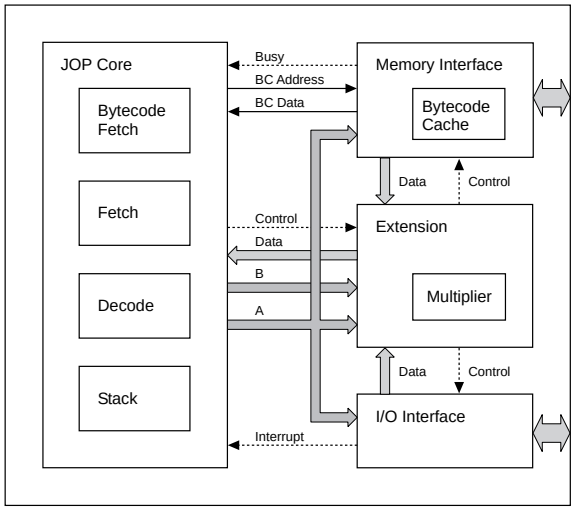

Fig. 1. Block diagram of JOP

other soft-core processors. In the Section IV, a number of different solutions for embedded Java are compared at the bytecode level and at the application level.

II. JOP Architecture

JOP is a stack computer with its own instruction set, called microcode in this paper. Java bytecodes are translated into microcode instructions or sequences of microcode. The difference between the JVM and JOP is best described as the following:

> The JVM is a CISC stack architecture, whereas JOP is a RISC stack architecture.

Figure 1 shows JOP's major function units. A typical configuration of JOP contains the processor core, a memory interface and a number of IO devices.

The processor core contains the three microcode pipeline stages *microcode fetch*, *decode* and *execute* and an additional translation stage *bytecode fetch*. The module called extension provides the link between the processor core, and the memory and IO modules. The ports to the other modules are the address and data bus for the bytecode instructions, the two top elements of the stack (A and B), input to the top-of-stack (Data) and a number of control signals. There is no direct connection between the processor core and the external world.

The memory interface provides a connection between the main memory and the processor core. It also contains the bytecode cache. The extension module controls data read and write. The *busy* signal is used by the microcode instruction wait[1] to synchronize the processor core with the memory unit. The core reads bytecode instructions through dedicated buses (BC address and BC data) from the memory subsystem.

The extension module performs three functions: (a) it contains hardware accelerators (such as the multiplier unit in this example), (b) the control for the memory and the I/O module, and (c) the multiplexer for the read data that is loaded into the top-of-stack register. The write data from the top-of-stack (A) is connected directly to all modules.

[1]The busy signal can also be used to stall the whole processor pipeline. This was the change made to JOP by Flavius Gruian [18]. However, in this synchronization mode, the concurrency between the memory access module and the main pipeline is lost.

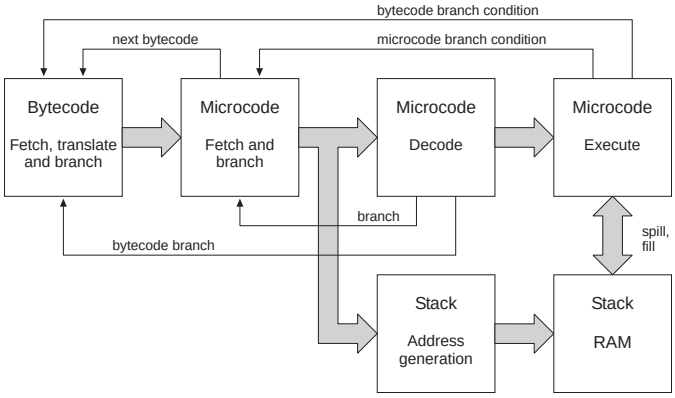

Fig. 2. Datapath of JOP

A. The Processor Pipeline

JOP is a fully pipelined architecture with single cycle execution of microcode instructions and a novel approach to mapping Java bytecode to these instructions. Figure 2 shows the datapath for JOP.

Three stages form the JOP core pipeline, executing microcode instructions. An additional stage in the front of the core pipeline fetches Java bytecodes – the instructions of the JVM – and translates these bytecodes into addresses in microcode. Bytecode branches are also decoded and executed in this stage. The second pipeline stage fetches JOP instructions from the internal microcode memory and executes microcode branches. Besides the usual decode function, the third pipeline stage also generates addresses for the stack RAM. As every stack machine instruction has either *pop* or *push* characteristics, it is possible to generate fill or spill addresses for the *following* instruction at this stage. The last pipeline stage performs ALU operations, load, store and stack spill or fill. At the execution stage, operations are performed with the two topmost elements of the stack.

A stack machine with two explicit registers for the two topmost stack elements and automatic fill/spill needs neither an extra write-back stage nor any data forwarding. Details of this two-level stack architecture are described in [19]. The short pipeline results in short branch delays. Therefore, a hard to analyze, with respect to Worst Case Execution Time (WCET), branch prediction logic can be avoided.

B. Interrupt Logic

Interrupts are considered hard to handle in a pipelined processor, meaning implementation tends to be complex (and therefore resource consuming). In JOP, the bytecode-microcode translation is used cleverly to avoid having to handle interrupts in the core pipeline.

Interrupts are implemented as special bytecodes. These bytecodes are inserted by the hardware in the Java instruction

stream. When an interrupt is pending and the next fetched byte from the bytecode cache is an instruction, the associated special bytecode is used instead of the instruction from the bytecode cache. The result is that interrupts are accepted at bytecode boundaries. The worst-case preemption delay is the execution time of the *slowest* bytecode that is implemented in microcode. Bytecodes that are implemented in Java (see Section II-D) can be interrupted.

The implementation of interrupts at the bytecode-microcode mapping stage keeps interrupts transparent in the core pipeline and avoids complex logic. Interrupt handlers can be implemented in the same way as standard bytecodes are implemented i.e. in microcode or Java. This special bytecode can result in a call of a JVM internal method in the context of the interrupted thread. This mechanism implicitly stores almost the complete context of the current active thread on the stack.

C. Cache

A pipelined processor architecture calls for higher memory bandwidth. A standard technique to avoid processing bottlenecks due to the higher memory bandwidth is caching. However, standard cache organizations improve the average execution time but are difficult to predict for WCET analysis. Two time-predictable caches are proposed for JOP: a *stack cache* as a substitution for the data cache and a *method cache* to cache the instructions.

As the stack is a heavily accessed memory region, the stack – or part of it – is placed in on-chip memory. This part of the stack is referred to as the *stack cache* and described in [19]. Fill and spill of the stack cache is subjected to microcode control and therefore time-predictable.

In [20], a novel way to organize an instruction cache, as *method cache*, is given. The cache stores complete methods, and cache misses only occur on method invocation and return. Cache block replacement depends on the call tree, instead of instruction addresses. This *method cache* is easy to analyze with respect to worst-case behavior and still provides substantial performance gain when compared against a solution without an instruction cache.

D. Microcode

The following discussion concerns two different instruction sets: *bytecode* and *microcode*. Bytecodes are the instructions that make up a compiled Java program. These instructions are executed by a Java virtual machine. The JVM does not assume any particular implementation technology. Microcode is the native instruction set for JOP. Bytecodes are translated, during their execution, into JOP microcode. Both instruction sets are designed for an extended[2] stack machine.

1) Translation of Bytecodes to Microcode: To date, no hardware implementation of the JVM exists that is capable of executing *all* bytecodes in hardware alone. This is due to the following: some bytecodes, such as new, which creates and initializes a new object, are too complex to implement in hardware. These bytecodes have to be emulated by software.

[2]An extended stack machine contains instructions that make it possible to access elements deeper down in the stack.

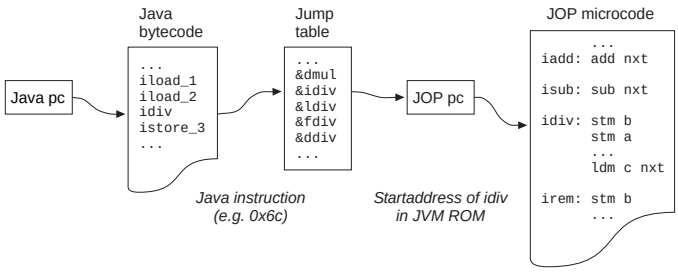

Fig. 3. Data flow from the Java program counter to JOP microcode

To build a self-contained JVM without an underlying operating system, direct access to the memory and I/O devices is necessary. There are no bytecodes defined for low-level access. These low-level services are usually implemented in *native* functions, which mean that another language (C) is native to the processor. However, for a Java processor, bytecode is the *native* language.

One way to solve this problem is to implement simple bytecodes in hardware and to emulate the more complex and *native* functions in software with a different instruction set (sometimes called microcode). However, a processor with two different instruction sets results in a complex design.

Another common solution, used in Sun's picoJava [4], is to execute a subset of the bytecode native and to use a software trap to execute the remainder. This solution entails an overhead (a minimum of 16 cycles in picoJava) for the software trap.

In JOP, this problem is solved in a much simpler way. JOP has a single *native* instruction set, the so-called microcode. During execution, every Java bytecode is translated to either one, or a sequence of microcode instructions. This translation merely adds one pipeline stage to the core processor and results in no execution overheads. With this solution, we are free to define the JOP instruction set to map smoothly to the stack architecture of the JVM, and to find an instruction coding that can be implemented with minimal hardware.

Figure 3 gives an example of this data flow from the Java program counter to JOP microcode. The fetched bytecode acts as an index for the jump table. The jump table contains the start addresses for the JVM implementation in microcode. This address is loaded into the JOP program counter for every bytecode executed.

Every bytecode is translated to an address in the microcode that implements the JVM. If there exists an equivalent microinstruction for the bytecode, it is executed in one cycle and the next bytecode is translated. For a more complex bytecode, JOP just continues to execute microcode in the subsequent cycles. The end of this sequence is coded in the microcode instruction (as the *nxt* bit).

2) Compact Microcode: For the JVM to be implemented efficiently, the microcode has to *fit* to the Java bytecode. Since the JVM is a stack machine, the microcode is also stack-oriented. However, the JVM is not a pure stack machine.

Method parameters and local variables are defined as *locals*. These locals can reside in a stack frame of the method and are accessed with an offset relative to the start of this *locals* area. Additional local variables (16) are available at the microcode level. These variables serve as scratch variables, like registers in a conventional CPU. However, arithmetic and logic operations are performed on the stack.

Some bytecodes, such as ALU operations and the short form access to *locals*, are directly implemented by an equivalent microcode instruction (with a different encoding). Additional instructions are available to access internal registers, main memory and I/O devices. A relative conditional branch (zero/non zero of TOS) performs control flow decisions at the microcode level. For optimum use of the available memory resources, all instructions are 8 bits long. There are no variable-length instructions and every instruction, with the exception of wait, is executed in a single cycle. To keep the instruction set this dense, two concepts are applied:

Two types of operands, immediate values and branch distances, normally force an instruction set to be longer than 8 bits. The instruction set is either expanded to 16 or 32 bits, as in typical RISC processors, or allowed to be of variable length at byte boundaries. A first implementation of the JVM with a 16-bit instruction set showed that only a small number of different constants are necessary for immediate values and relative branch distances.

In the current realization of JOP, the different immediate values are collected while the microcode is being assembled and are put into the initialization file for the local RAM. These constants are accessed indirectly in the same way as the local variables. They are similar to initialized variables, apart from the fact that there are no operations to change their value during runtime, which would serve no purpose and would waste instruction codes.

A similar solution is used for branch distances. The assembler generates a VHDL file with a table for all found branch constants. This table is indexed using instruction bits during runtime. These indirections during runtime make it possible to retain an 8-bit instruction set, and provide 16 different immediate values and 32 different branch constants. For a general purpose instruction set, these indirections would impose too many restrictions. As the microcode only implements the JVM, this solution is a viable option.

To simplify the logic for instruction decoding, the instruction coding is carefully chosen. For example, one bit in the instruction specifies whether the instruction will increment or decrement the stack pointer. The offset to access the *locals* is directly encoded in the instruction. This is not the case for the original encoding of the equivalent bytecodes (e.g. *iload_0* is 0x1a and *iload_1* is 0x1b).

3) Flexible Implementation of Bytecodes: As mentioned above, some Java bytecodes are very complex. One solution already described is to emulate them through a sequence of microcode instructions. However, some of the more complex bytecodes are very seldom used. To further reduce the resource implications for JOP, in this case local memory, bytecodes can even be implemented by *using* Java bytecodes. During the assembly of the JVM, all labels that represent an entry point for the bytecode implementation are used to generate the translation table. For all bytecodes for which no such label is found, i.e. there is no implementation in microcode, a *not-implemented* address is generated. The instruction sequence at this address invokes a static method from a system class (com.jopdesign.sys.JVM). This class contains 256 static methods, one for each possible bytecode, ordered by the bytecode value. The bytecode is used as the index in the method table of this system class. This feature also allows for the easy configuration of resource usage versus performance.

III. Resource Usage

Cost, alongside energy consumption, is an important issue for embedded systems. The cost of a chip is directly related to the die size (the cost per die is roughly proportional to the square of the die area [21]). Chips with fewer gates also consume less energy. Processors for embedded systems are therefore optimized for minimum chip size.

One major design objective in the development of JOP was to create a small system that could be implemented in a low-cost FPGA. Table II shows the resource usage for different configurations of JOP and different soft-core processors implemented in an Altera EP1C6 FPGA [22]. Estimating equivalent gate counts for designs in an FPGA is problematic. It is therefore better to compare the two basic structures, Logic Cells (LC) and embedded memory blocks.

All configurations of JOP contain a memory interface to a 32-bit static RAM and an 8-bit FLASH for the Java program and the FPGA configuration data. The minimum configuration implements multiplication and the shift operations in microcode. In the basic configuration, these operations are implemented as a sequential Booth multiplier and a single-cycle barrel shifter. The typical configuration also contains some useful I/O devices such as an UART and a timer with interrupt logic for multi-threading. The typical configuration of JOP needs about 30% of the LCs in a Cyclone EP1C6, thus leaving enough resources free for application-specific logic.

As a reference, NIOS [23], Altera's popular RISC soft-core, is also included in the list. NIOS has a 16-bit instruction set, a 5-stage pipeline and can be configured with a 16 or 32-bit datapath. Version A is the minimum configuration of NIOS. Version B adds an external memory interface, multiplication support and a timer. Version A is comparable with the minimal configuration of JOP, and Version B with its typical configuration.

SPEAR [24] (Scalable Processor for Embedded Applications in Real-time Environments) is a 16-bit processor with deterministic execution times. SPEAR contains predicated instructions to support single-path programming [25]. SPEAR is included in the list as it is also a processor designed for real-time systems.

To prove that the VHDL code for JOP is as portable as possible, JOP was also implemented in a Xilinx Spartan-3 FPGA [26]. Only the instantiation and initialization code for

TABLE II
FPGA SOFT-CORE PROCESSORS

Processor	Resources [LC]	Memory [KB]	fmax [MHz]
JOP Minimal	1,077	3.25	98
JOP Basic	1,452	3.25	98
JOP Typical	1,831	3.25	101
Lightfoot[3]	3,400	4	40
NIOS A	1,828	6.2	120
NIOS B	2,923	5.5	119
SPEAR[4]	1,700	8	80

TABLE III
GATE COUNT ESTIMATES FOR VARIOUS PROCESSORS

Processor	Core [gate]	Memory [gate]	Sum. [gate]
JOP	11K	40K	51K
picoJava	128K	314K	442K
aJile	25K	590K	615K
Pentium MMX			1125K

the on-chip memories is vendor-specific, whilst the rest of the VHDL code can be shared for the different targets. JOP consumes about the same LC count (1844 LCs) in the Spartan device, but has a slower clock frequency (83MHz).

From this comparison we can see that we have achieved our objective of designing a small processor. The commercial Java processor, Lightfoot, is 2.3 times larger (and 2.5 times slower) than JOP in the basic configuration. A typical 32-bit RISC processor consumes about 1.6 to 1.8 times the resources of JOP. However, the RISC processor can be clocked 20% faster than JOP in the same technology. The only processor that is similar in size is SPEAR. However, while SPEAR is a 16-bit processor, JOP contains a 32-bit datapath.

Table III provides gate count estimates for JOP, picoJava, the aJile processor, and the Intel Pentium MMX processor that is used in the benchmarks in the next section. Equivalent gate count for an LC[5] varies between 5.5 and 7.4 – we chose a factor of 6 gates per LC and 1.5 gates per memory bit for the estimated gate count for JOP in the table. JOP is listed in the typical configuration that consumes 1831 LCs. The Pentium MMX contains 4.5M transistors [27] that are equivalent to 1125K gates.

We can see from the table that the on-chip memory dominates the overall gate count of JOP, and to an even greater extent, of the aJile processor. The aJile processor is about 12 times larger than JOP.

[3]The data for the Lightfoot processor is taken from the data sheet [11]. The frequency used is that in a Vertex-II device from Xilinx. JOP can be clocked at 100MHz in the Vertex-II device, making this comparison valid.

[4]As SPEAR uses internal memory blocks in asynchronous mode it is not possible to synthesize it without modification for the Cyclone FPGA. The clock frequency of SPEAR in an Altera Cyclone is an estimate based on following facts: SPEAR can be clocked at 40MHz in an APEX device and JOP can be clocked at 50MHz in the same device.

[5]The factors are derived from the data provided for various processors and from the resource estimates in [19].

IV. PERFORMANCE

Running benchmarks is problematic, both generally and especially in the case of embedded systems. The best benchmark would be the application that is intended to run on the system being tested. To get comparable results SPEC provides benchmarks for various systems. However, the one for Java, the SPECjvm98 [28], is usually too large for embedded systems.

Due to the absence of a *standard* Java benchmark for embedded systems, a small benchmark suit that should run on even the smallest device is provided here. It contains several micro-benchmarks for evaluating the number of clock cycles for single bytecodes or short sequences of bytecodes, and two application benchmarks. To provide a realistic workload for embedded systems, a real-time application was adapted to create the first application benchmark (Kfl). The application is taken from one of the nodes of a distributed motor control system [29]. A simulation of both the environment (sensors and actors) and the communication system (commands from the master station) forms part of the benchmark, so as to simulate the real-world workload. The second application benchmark is an adaptation of a tiny TCP/IP stack for embedded Java. This benchmark contains two UDP server/clients, exchanging messages via a loopback device.

As we will see, there is a great variation in processing power across different embedded systems. To cater for this variation, all benchmarks are 'self adjusting'. Each benchmark consists of an aspect that is benchmarked in a loop. The loop count adapts itself until the benchmark runs for more than a second. The number of iterations per second is then calculated, which means that higher values indicate better performance.

All the benchmarks measure how often a function is executed per second. In the Kfl benchmark, this function contains the main loop of the application that is executed in a periodic cycle in the original application. In the benchmark the wait for the next period is omitted, so that the time measured solely represents execution time. The UDP benchmark contains the generation of a request, transmitting it through the UDP/IP stack, generating the answer and transmitting it back as a benchmark function. The iteration count is the number of received answers per second.

The following list gives a brief description of the Java systems that were benchmarked:

JOP is implemented in a Cyclone FPGA, running at 100MHz. The main memory is a 32-bit SRAM (15ns) with an access time of 2 clock cycles. The benchmarked configuration of JOP contains a 4KB method cache [20] organized in 16 blocks.

leJOS As an example for a low-end embedded device we use the RCX robot controller from the LEGO MindStorms series. It contains a 16-bit Hitachi H8300 microcontroller [30], running at 16MHz. leJOS [31] is a tiny interpreting JVM for the RCX.

TINI is an enhanced 8051 clone running a software JVM. The results were taken from a custom board with a 20MHz crystal, and the chip's PLL is set to a factor of 2.

KVM is a port of the Sun's KVM that is part of the Connected Limited Device Configuration (CLDC) [32] to Alteras NIOS II processor on MicroC Linux. NIOS is implemented on a Cyclone FPGA and clocked with 50MHz. Besides the different clock frequency this is a good comparison of an interpreting JVM running in the same FPGA as JOP.

The benchmark results of **Komodo** were obtained by Matthias Pfeffer [33] on a cycle-accurate simulation of Komodo.

aJile's JEMCore is a direct-execution Java processor that is available in two different versions: the **aJ80** and the **aJ100** [7]. A development system, the JStamp [34], contains the aJ80 with an 8-bit memory, clocked at 74MHz. The SaJe board from Systronix contains an aJ100 that is clocked with 103MHz and contains 10ns 32-bit SRAM.

The **EJC** (Embedded Java Controller) platform [35] is a typical example of a JIT system on a RISC processor. The system is based on a 32-bit ARM720T processor running at 74MHz. It contains up to 64 MB SDRAM and up to 16 MB of NOR flash.

gcj is the GNU compiler for Java. This configuration represents the batch compiler solution, running on a 266MHz Pentium under Linux.

MB is the realization of Java on a RISC processor for an FPGA (Xilinx MicroBlaze [36]). Java is compiled to C with a Java compiler for real-time systems [37] and the C program is compiled with the standard GNU toolchain.

In Figure 4, the geometric mean of the two application benchmarks is shown. The unit used for the result is iterations per second. Note that the vertical axis is logarithmic, in order to obtain useful figures to show the great variation in performance. The top diagram shows absolute performance, while the bottom diagram shows the same results scaled to a 1MHz clock frequency. The results of the application benchmarks and the geometric mean are shown in Table IV.

It should be noted that scaling to a single clock frequency could prove problematic. The relation between processor clock frequency and memory access time cannot always be maintained. To give an example, if we were to increase the results of the 100MHz JOP to 1GHz, this would also involve reducing the memory access time from 15ns to 1.5ns. Processors with 1GHz clock frequency are already available, but the fastest asynchronous SRAM to date has an access time of 10ns.

A. Discussion

When comparing JOP and the aJile processor against leJOS, TINI, and KVM, we can see that a Java processor is up to 500 times faster than an interpreting JVM on a standard processor for an embedded system. The average performance of JOP is even better than a JIT-compiler solution on an embedded system, as represented by the EJC system.

Even when scaled to the same clock frequency, the compiling JVM on a PC (gcj) is much faster than either embedded solution. However, the kernel of the application is smaller than 4KB [20]. It therefore fits in the level one cache of the Pentium

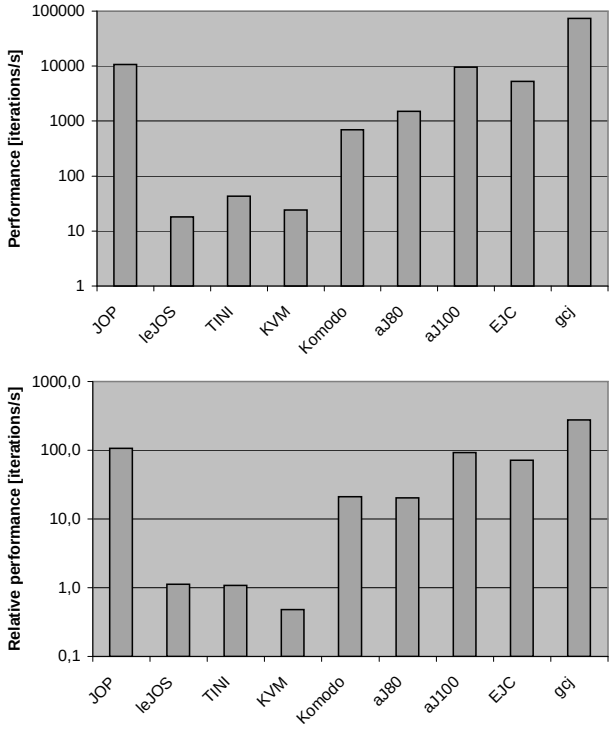

Fig. 4. Performance comparison of different Java systems with application benchmarks. The diagrams show the geometric mean of the two benchmarks in iterations per second – a higher value means higher performance. The top diagram shows absolute performance, while the bottom diagram shows the result scaled to 1MHz clock frequency.

TABLE IV
APPLICATION BENCHMARKS ON DIFFERENT JAVA SYSTEMS IN ITERATIONS PER SECOND – A HIGHER VALUE IS BETTER.

	Frequency [MHz]	Kfl	UDP/IP	Geom. Mean [Iterations/s]	Scaled
JOP	100	16,582	6,849	10,657	107
leJOS	16	25	13	18	1.1
TINI	40	64	29	43	1.1
KVM	50	36	16	24	0.5
Komodo	33	924	520	693	21
aJ80	74	2,221	1,004	1,493	20
aJ100	103	14,148	6,415	9,527	92
EJC	74	9,893	2,822	5,284	71
gcj	266	139,884	38,460	73,348	276
MB	100	3,792			

TABLE V
EXECUTION TIME IN CLOCK CYCLES FOR VARIOUS JVM BYTECODES

	JOP	leJOS	TINI	Komodo	aJ80	aJ100
iload iadd	2	836	789	8	38	8
iinc	11	422	388	4	41	11
ldc	9	1,340	1,128	40	67	9
if_icmplt taken	6	1,609	1,265	24	42	18
if_icmplt n/taken	6	1,520	1,211	24	40	14
getfield	23	1,879	2,398	48	142	23
getstatic	15	1,676	4,463	80	102	15
iaload	29	1,082	1,543	28	74	13
invoke	126	4,759	6,495	384	349	112
invoke static	100	3,875	5,869	680	271	92
invoke interface	142	5,094	6,797	1617	531	148

MMX (16KB + 16KB). For a comparison with a Pentium class processor we would need a larger application.

JOP is about 7 times faster than the aJ80 Java processor on the popular JStamp board. However, the aJ80 processor only contains an 8-bit memory interface, and suffers from this bottleneck. The SaJe system contains the aJ100 with 32-bit, 10ns SRAMs and is a about 10% slower than JOP with its 15ns SRAMs.

The MicroBlaze system is a representation of a Java batch-compilation system for a RISC processor. MicroBlaze is configured with the same cache[6] as JOP and clocked at the same frequency. JOP is about four times faster than this solution, thus showing that native execution of Java bytecodes is faster than batch-compiled Java on a similar system. However, the results of the MicroBlaze solution are at a preliminary stage[7], as the Java2C compiler [37] is still under development.

The micro-benchmarks are intended to give insight into the implementation of the JVM. In Table V, we can see the execution time in clock cycles of various bytecodes. As almost all bytecodes manipulate the stack, it is not possible to measure the execution time for a single bytecode. As a minimum requirement, a second instruction is necessary to reverse the stack operation. For compiling versions of the JVM, these micro-benchmarks do not produce useful results. The compiler performs optimizations that make it impossible to measure execution times at this fine a granularity.

For JOP we can deduce that the WCET for simple bytecodes is also the average execution time. We can see that the combination of iload and iadd executes in two cycles, which means that each of these two operations is executed in a single cycle. The iinc bytecode is one of the few instructions that do not manipulate the stack and can be measured alone. As iinc is not implemented in hardware, we have a total of 11 cycles that are executed in microcode. It is fair to assume that this comprises too great an overhead for an instruction that is found in every iterative loop with an integer index. However, the decision to implement this instruction in microcode was derived from the observation that the dynamic instruction count for iinc is only 2% [1].

The sequence for the branch benchmark (if_icmplt) contains the two load instructions that push the arguments onto the stack. The arguments are then consumed by the branch instruction. This benchmark verifies that a branch requires a constant four cycles on JOP, whether it is taken or not.

During the evaluation of the aJile system, unexpected behavior was observed. The aJ80 on the JStamp board is clocked at 7.3728MHz and the internal frequency can be set with a PLL. The aJ80 is rated for 80MHz and the maximum PLL factor that can be used is therefore ten. Running the benchmarks with different PLL settings gave some strange results. For example,

[6]The MicroBlaze with a 8KB data and 8KB instruction cache is about 1.6 times faster than JOP. However, a 16KB memory is not available in low-cost FPGAs and is an unbalanced system with respect to the LC/memory relation.

[7]As not all language constructs can be compiled, only the Kfl benchmark was measured. Therefore, the performance bar for MicroBlaze is missing in Figure 4

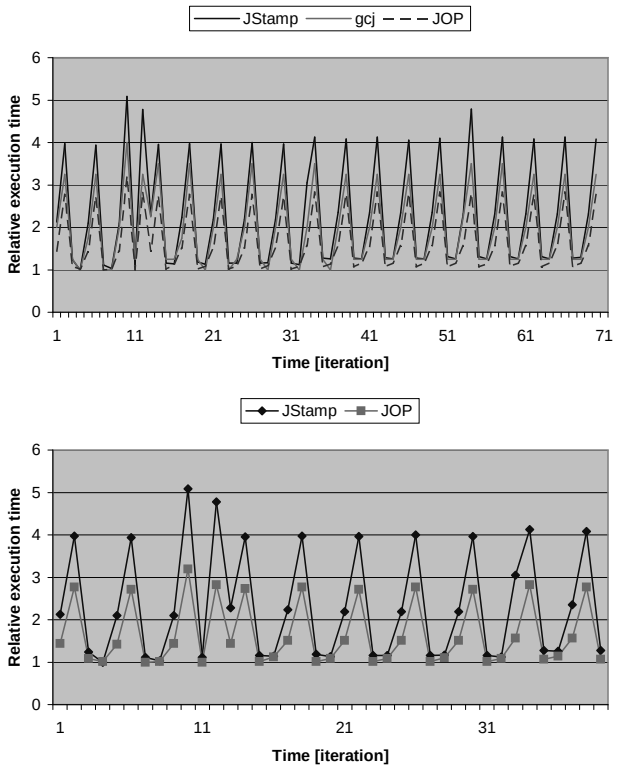

Fig. 5. Execution time of the main function for the Kfl benchmark. The values are scaled to the minimum execution time. The bottom figure shows a detail of the top figure.

with a PLL multiplier setting of ten, the aJ80 was about 12.8 times faster! Other PLL factors also resulted in a greater than linear speedup. The only explanation we could find was that the internal time, System.currentTimeMillis(), used for the benchmarks depends on the PLL setting. A comparison with the wall clock time showed that the internal time of the aJ80 is 23% faster with a PLL factor of 1 and 2.4% faster with a factor of ten – a property we would not expect on a processor that is marketed for real-time systems. The SaJe board can also suffer from the problem described.

B. Execution Time Jitter

For real-time systems, the worst-case of the execution time is of primary importance. We have measured the execution times of several iterations of the main function from the Kfl benchmark. Figure 5 shows the measurements, scaled to the minimum execution time.

A period of four iterations can be seen. This period results from simulating the commands from the base station that are executed every fourth iteration. At iteration 10, a command to start the motor is issued. We see the resulting rise in execution time at iteration 12 to process this command. At iteration 54, the simulation triggers the end sensor and the motor is stopped.

The different execution times in the different modes of the application are inherent in the design of the simulation. However, the ratio between the longest and the shortest period is five for the JStamp, four for the gcj system and only three

for JOP. Therefore, a system with an aJile processor needs to be 1.7 times faster than JOP in order to provide the same WCET for this measurement. At iteration 33, we can see a higher execution time for the JStamp system that is not seen on JOP. This variation at iteration 33 is not caused by the benchmark.

The execution time under gcj on the Linux system showed some very high peaks (up to ten times the minimum, not shown in the figures). This observation was to be expected, as the gcj/Linux system is not a real-time solution. The Sun JIT-solution was also measured, but is omitted from the figure. As a result of the invocation of the compiler at some point during the simulation, the worst-case ratio between the maximum and minimum execution time was 1313 – showing that a JIT-compiler is impractical for real-time applications.

It should be noted that execution time measurement is not a safe method for obtaining WCET estimates. However, in situations where no WCET analysis tool is available, it can give some insight into the WCET behavior of different systems.

V. Conclusion

In this paper, we presented a brief overview of the concepts for a real-time Java processor, called JOP, and the evaluation of this architecture. We have seen that JOP is the smallest hardware realization of the JVM available to date. Due to the efficient implementation of the stack architecture, JOP is also smaller than a *comparable* RISC processor in an FPGA. Implemented in an FPGA, JOP has the highest clock frequency of all known Java processors.

We compared JOP against several embedded Java systems and, as a reference, with Java on a standard PC. A Java processor is up to 500 times faster than an interpreting JVM on a standard processor for an embedded system. JOP is about seven times faster than the aJ80 Java processor and about 10% faster than the aJ100. Preliminary results using compiled Java for a RISC processor in an FPGA, with a similar resource usage and maximum clock frequency to JOP, showed that native execution of Java bytecodes is faster than compiled Java.

The proposed processor has been used with success to implement several commercial real-time applications. JOP is open-source and all design files are available at http://www.jopdesign.com/.

References

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

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

[3] J. M. O'Connor and M. Tremblay, "picoJava-I: The Java virtual machine in hardware," *IEEE Micro*, vol. 17, no. 2, pp. 45–53, 1997.

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

[5] Sun, *picoJava-II Programmer's Reference Manual*. Sun Microsystems, March 1999.

[6] S. Dey, P. Sanchez, D. Panigrahi, L. Chen, C. Taylor, and K. Sekar, "Using a soft core in a SOC design: Experiences with picoJava," *IEEE Design and Test of Computers*, vol. 17, pp. 60–71, July 2000.

[7] aJile, "aj-100 real-time low power Java processor." preliminary data sheet, 2000.

[8] D. Hardin, "Real-Time Objects on the Bare Metal: An Efficient Hardware Realization of the JavaTM Virtual Machine," in *Proceedings of the Fourth International Symposium on Object-Oriented Real-Time Distributed Computing*, p. 53, IEEE Computer Society, 2001.

[9] Vulcan, "Moon v1.0." data sheet, January 2000.

[10] Vulcan, "Moon2 - 32 bit native Java technology-based processor." product folder, 2003.

[11] DCT, "Lightfoot 32-bit Java processor core." data sheet, September 2001.

[12] J. Kreuzinger, U. Brinkschulte, M. Pfeffer, S. Uhrig, and T. Ungerer, "Real-time event-handling and scheduling on a multithreaded Java microcontroller," *Microprocessors and Microsystems*, vol. 27, no. 1, pp. 19–31, 2003.

[13] A. C. Beck and L. Carro, "Low power java processor for embedded applications," in *Proceedings of the 12th IFIP International Conference on Very Large Scale Integration*, December 2003.

[14] T. R. Halfhill, "Imsys hedges bets on Java," *Microprocessor Report*, August 2000.

[15] PTSC, "Ignite processor brochure, rev 1.0." Available at http://www.ptsc.com.

[16] ARM, "Jazelle – arm architecture extensions for Java applications." white paper.

[17] Nazomi, "JA 108 product brief." Available at http://www.nazomi.com.

[18] F. Gruian, P. Andersson, K. Kuchcinski, and M. Schoeberl, "Automatic generation of application-specific systems based on a micro-programmed java core," in *Proceedings of the 20th ACM Symposium on Applied Computing, Embedded Systems track*, (Santa Fee, New Mexico), March 2005.

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

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

[21] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach, 3rd ed.* Palo Alto, CA 94303: Morgan Kaufmann Publishers Inc., 2002.

[22] Altera, "Cyclone FPGA Family Data Sheet, ver. 1.2," April 2003.

[23] Altera, "Nios soft core embedded processor, ver. 1." data sheet, June 2000.

[24] M. Delvai, W. Huber, P. Puschner, and A. Steininger, "Processor support for temporal predictability – the spear design example," in *Proceedings of the 15th Euromicro International Conference on Real-Time Systems*, Jul. 2003.

[25] P. Puschner, "Experiments with wcet-oriented programming and the single-path architecture," in *Proc. 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, Feb. 2005.

[26] Xilinx, "Spartan-3 FPGA family: Complete data sheet, ver. 1.2," 2005.

[27] M. Eden and M. Kagan, "The pentium processor with mmx technology," in *Proceedings of Compcon '97*, pp. 260–262, IEEE Computer Society, 1997.

[28] SPEC, "The spec jvm98 benchmark suite." Available at http://www.spec.org/, August 1998.

[29] M. Schoeberl, "Using a Java optimized processor in a real world application," in *Proceedings of the First Workshop on Intelligent Solutions in Embedded Systems (WISES 2003)*, (Austria, Vienna), pp. 165–176, June 2003.

[30] Hitachi, "Hitachi single-chip microcomputer h8/3297 series." Hardware Manual.

[31] J. Solorzano, "leJOS: Java based os for lego RCX." Available at: http://lejos.sourceforge.net/.

[32] Sun, "Java 2 platform, micro edition (j2me)." Available at: http://java.sun.com/j2me/docs/.

[33] M. Pfeffer, *Ein echtzeitfähiges Java-System für einen mehrfädigen Java-Mikrocontroller*. PhD thesis, University of Augsburg, 2000.

[34] Systronix, "Jstamp real-time native Java module." data sheet.

[35] EJC, "The ejc (embedded java controller) platform." Available at http://www.embedded-web.com/index.html.

[36] Xilinx, "Microblaze processor reference guide, edk v6.2 edition." data sheet, December 2003.

[37] A. Nilsson, "Compiling java for real-time systems," licentiate thesis, Dept. of Computer Science, Lund University, May 2004.