A Java Processor Architecture for Embedded Real-Time Systems

Martin Schoeberl

Institute of Computer Engineering, Vienna University of Technology, Austria

Abstract

Architectural advancements in modern processor designs increase average performance with features such as pipelines, caches, branch prediction, and out-of-order execution. However, these features complicate worst-case execution time analysis and lead to very conservative estimates. JOP (Java Optimized Processor) tackles this problem from the architectural perspective – by introducing a processor architecture in which simpler and more accurate WCET analysis is more important than average case performance.

This paper presents a Java processor designed for time-predictable execution of real-time tasks. JOP is the implementation of the Java virtual machine in hardware. JOP is intended for applications in embedded real-time systems and the primary implementation technology is in a field programmable gate array. This paper demonstrates that a hardware implementation of the Java virtual machine results in a small design for resource-constrained devices.

Key words: Real-time system; Time predictable architecture; Java processor

1. Introduction

Compared to software development for desktop systems, current software design practice for embedded real-time systems is still archaic. C/C++ and even assembly language are used on top of a small real-time operating system. Many of the benefits of Java, such as safe object references, the notion of concurrency as a first-class language construct, and its portability, have the potential to make embedded systems much safer and simpler to program. However, Java technology is seldom used in embedded real-time systems, due to the lack of acceptable real-time performance.

Traditional implementations of the Java virtual machine (JVM) as interpreter or just-in-time compiler are not practical. An interpreting virtual machine is too slow and therefore waste of processor resources. Just-in-time compilation has several disadvantages for embedded systems, notably that a compiler (with the intrinsic memory overhead) is necessary on the target system. Due to compilation during runtime, execution times are practically not predictable [1].

This paper introduces the concept of a Java processor [51] for embedded real-time systems, in particular the design of a small processor for resource-constrained devices with time-predictable execution of Java programs. This Java processor is called JOP – which stands for Java Optimized Processor –, based on the assumption that a full native implementation of all Java bytecode instructions [30] is not a useful approach.

Worst-case execution time (WCET) estimates of tasks are essential for designing and verifying real-time systems. Static WCET analysis is necessary for hard real-time systems. In order to obtain a low WCET value, a good processor model is necessary. Traditionally, only simple processors can be analyzed using practical WCET boundaries. Architectural advance-

[1] One could add the compilation time of a method to the WCET of that method. However, in that case we need a WCET analyzable compiler and the WCET gets impractical high.

ments in modern processor designs tend to abide by the rule: '*Make the average case as fast as possible*'. This is orthogonal to '*Minimize the worst case*' and has the effect of complicating WCET analysis. This paper tackles this problem from the architectural perspective – by introducing a processor architecture in which simpler and more accurate WCET analysis is more important than average case performance.

JOP is designed from ground up with time predictable execution of Java bytecode as major design goal. All function units, and especially the interaction between them, are carefully designed to avoid any time dependency between bytecodes. The architectural highlights are:

(i) Dynamic translation of the CISC Java bytecodes to a RISC, stack based instruction set (the microcode) that can be executed in a 3 stage pipeline.
(ii) The translation takes exactly one cycle per bytecode and is therefore pipelined. Compared to other forms of dynamic code translation the proposed translation does not add any variable latency to the execution time and is therefore time predictable.
(iii) Interrupts are inserted in the translation stage as special bytecodes and are transparent to the microcode pipeline.
(iv) The short pipeline (4 stages) results in short conditional branch delays and a hard to analyze branch prediction logic or branch target buffer can be avoided.
(v) Simple execution stage with the two topmost stack elements as discrete registers. No write back stage or forwarding logic is needed.
(vi) Constant execution time (one cycle) for all microcode instructions. No stalls in the microcode pipeline. Loads and stores of object fields are handled explicitly.
(vii) No time dependencies between bytecodes result in a simple processor model for the low-level WCET analysis.
(viii) Time predictable instruction cache that caches whole methods. Only invoke and return instruction can result in a cache miss. All other instructions are guaranteed cache hits.
(ix) Time predictable data cache for local variables and the operand stack. Access to local variables is a guaranteed hit and no pipeline stall can happen. Stack cache fill and spill is under microcode control and analyzable.
(x) No prefetch buffers or store buffers that can introduce unbound time dependencies of instructions. Even simple processors can contain an instruction prefetch buffer that prohibits exact WCET values. The design of the method cache and the translation unit avoids the variable latency of a prefetch buffer.
(xi) Good average case performance compared to other non real-time Java processors.
(xii) Avoidance of hard to analyze architectural features results in a very small design. Therefore an available real estate can be used for a chip multiprocessor solution.

In this paper, we will present the architecture of the real-time Java processor and the evaluation results for JOP, with respect to WCET, size and performance. We will show that the execution time of Java bytecodes can be exactly predicted in terms of the number of clock cycles. We will also evaluate the general performance of JOP in relation to other embedded Java systems. Although JOP is intended as a processor with a low WCET for all operations, its general performance is still important. We will see that a real-time processor architecture does not need to be slow.

In the following section, related work on real-time Java, Java processors, and issues with the low-level WCET analysis for standard processors is presented. In Section 3 a brief overview of the architecture of JOP is given, followed by a more detailed description of the microcode. In Section 4 it is shown that our objective of providing an easy target for WCET analysis has been achieved. Section 5 compares JOP's resource usage with other soft-core processors. In the Section 6, a number of different solutions for embedded Java are compared at the bytecode level and at the application level.

2. Related Work

In this section we present arguments for Java in real-time systems, various Java processors from industry and academia, and an overview of issues in the low-level WCET analysis that can be avoided by the proposed processor design.

2.1. *Real-Time Java*

Java is a strongly typed, object oriented language, with safe references, and array bounds checking. Java shares those features with Ada, the main language for safety critical real-time systems. It is even possible, and has been done [60,8], to compile Ada 95 for the JVM. In contrast to Ada, Java has a large user and open-source code base. In [64] it is argued that Java will become the better technology for real-time systems.

The object references replace error-prone C/C++ style pointers. Type checking is enforced by the compiler and performed at runtime. Those features greatly help to avoid program errors. Therefore Java is an attractive choice for safety critical and real-time systems [56,26]. Furthermore, threads and synchronization, common idioms in real-time programming, are part of the language.

An early document published by the NIST [33] defines the requirements for real-time Java. Based on those requirements the Real-Time Specification for Java (RTSJ) [7] started as first Java Specification Request (JSR). In the expert group of the RTSJ garbage collection was considered as the main issue of Java in real-time systems. Therefore the RTSJ defines, besides other idioms, new memory areas (scoped memory) and a `NoHeapRealtimeThread` that can interrupt the garbage collector. However, real-time garbage collection is an active research area (e.g., [5]). In [44] and [52] it is shown that a correctly scheduled garbage collector can be used even in hard real-time systems.

Discussion of the RTSJ, platforms for embedded Java and the definition and implementation of a real-time profile for embedded Java on JOP can be found in [48].

Java bytecode generation has to follow stringent rules [30] in order to pass the class file verification of the JVM. Those restrictions lead to an *analysis friendly* code, e.g. the stack size is known at each instruction. The control flow instructions are well defined. Branches are relative and the destination is within the same method. In Java class files there is more information available than in compiled C/C++ executables. All links are symbolic and it is possible to reconstruct the class hierarchy from the class files. Therefore, a WCET analysis tool can statically determine all possible targets for a virtual method invocation.

2.2. *Java Processors*

Table 1 lists the relevant Java processors available to date. Sun introduced the first version of picoJava [36] 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 [58,59]. The architecture of picoJava is a stack-based CISC processor implementing 341 different instructions [36] and is the most complex Java processor available. The processor can be implemented in about 440K gates [11]. Simple Java bytecodes are directly implemented in hardware, most of them execute in one to three cycles. Other performance critical instructions, for instance invoking a method, are implemented in microcode. picoJava traps on the remaining complex instructions, such as creation of an object, and emulates this instruction. A trap is rather expensive and has a minimum overhead of 16 clock cycles. This minimum value can only be achieved if the trap table entry is in the data cache and the first instruction of the trap routine is in the instruction cache. The worst-case trap latency is 926 clock cycles [59]. This great variation in execution times for a trap hampers tight WCET estimates.

Table 1
JOP and various Java processors

	Target	Size		Speed
	technology	Logic	Memory	(MHz)
JOP	Altera, Xilinx FPGA	2050 LCs	3 KB	100
picoJava [58,59]	No realization	128 Kgates	38 KB	
aJile [1,19]	ASIC 0.25μ	25 Kgates	48 KB	100
Cjip [18,25]	ASIC 0.35μ	70 Kgates	55 KB	80
Moon [62,63]	Altera FPGA	3660 LCs	4 KB	
Lightfoot [9]	Xilinx FPGA	3400 LCs	4 KB	40
Komodo [27]	Xilinx FPGA	2600 LCs		33
FemtoJava [6]	Xilinx FPGA	2710 LCs	0.5 KB	56

aJile's JEMCore is a direct-execution Java processor that is available as both an IP core and a stand alone processor [1,19]. It is based on the 32-bit JEM2 Java chip developed by Rockwell-Collins. Two silicon versions of JEM exist today: the aJ-80 and the aJ-100. Both versions comprise a JEM2 core, 48 KB zero wait state RAM and peripheral components. 16 KB of the RAM is used for the writable control store. The remaining 32 KB is used for storage of the processor stack. The aJile processor is intended for real-time systems with an on-chip real-time thread manager. aJile Systems was part of the expert group for the RTSJ [7]. However, no information is available about bytecode execution times.

The Cjip processor [18,25] supports multiple instruction sets, allowing Java, C, C++ and assembler to coexist. Internally, the Cjip uses 72 bit wide microcode instructions, to support the different instruction sets. At its core, Cjip is a 16-bit CISC architecture with on-chip 36 KB ROM and 18 KB RAM for fixed and loadable microcode. Another 1 KB RAM is used for eight independent register banks, string buffer and two stack caches. Cjip is implemented in 0.35-micron technology and can be clocked up to 80 MHz. The JVM is implemented largely in microcode (about 88% of the Java bytecodes).

Java thread scheduling and garbage collection are implemented as processes in microcode. Microcode instructions execute in two or three cycles. A JVM bytecode requires several microcode instructions. The Cjip Java instruction set and the extensions are described in detail in [24]. For example: a bytecode `nop` executes in 6 cycles while an `iadd` takes 12 cycles. Conditional bytecode branches are executed in 33 to 36 cycles. Object oriented instructions such `getfield`, `putfield` or `invokevirtual` are not part of the instruction set.

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 [62], 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 [63] is available as an encrypted HDL source for Altera FPGAs or as VHDL or Verilog source code.

The Lightfoot 32-bit core [9] 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. The core is provided as an EDIF netlist for dedicated Xilinx devices.

Komodo [27] 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. Komodos multi-threading is similar to hyper-threading in modern processors that are trying to hide latencies in instruction fetching. However, this feature leads to very pessimistic WCET values (in effect rendering this performance gain useless in hard real-time systems). The fact that the pipeline clock is only a quarter of the system clock also wastes a considerable amount of potential performance.

FemtoJava [6] 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. The resource usage is very high, compared to the minimal Java subset implemented and the low performance of the processor.

Besides the *real* Java processors a FORTH chip (PSC1000 [38]) is marketed as Java processors. Java coprocessors (e.g. JSTAR [32]) provide Java execution speedup for general-purpose processors. Jazelle [4] is an extension of the ARM 32-bit RISC processor. It introduces a third instruction set (bytecode), besides the Thumb instruction set (a 16-bit mode for reduced memory consumption), to the processor. The Jazelle coprocessor is integrated into the same chip as the ARM processor.

So far, all processors described (except Cjip) perform weakly in the area of time-predictable execution of Java bytecodes. However, a low-level analysis of execution times is of primary importance for WCET analysis. Therefore, the main objective is to define and implement a processor architecture that is as predictable as possible. However, it is equally important that this does not result in a low performance solution. Performance shall not suffer as a result of the time-predictable architecture. In Section 6, the overall performance of various Java systems, including the aJile processor, Komodo, and Cjip, is compared with JOP.

2.3. *WCET Analysis*

WCET Analysis can be divided in high-level and low-level analysis (see also Section 4). The high-level analysis is a mature research topic [29,43,40]. The main issues to be solved are in the low-level analysis. The processors that can be analyzed are usually several generations behind actual architectures [14,34,20]. An example: Thesing models in his 2004 PhD thesis [61] the PowerPC 750 (the MPC755 variant). The PowerPC 750 was introduced 1997 and the MPC755 is now (2006) *not recommended for new designs*.

The main issues in low-level analysis are many features of modern processors that increase average performance. All those features, such as multi-level caches, branch target buffer, out-of-order (OOO) execution, and speculation, include a lot of state that depends on a large execution history. Modeling this history for the WCET analysis leads to a state explosion for the final WCET calculation. Therefore low-level WCET analysis usually performs simplifications and uses conservative estimates. One example of this conservative estimate is to classify a cache access, if the outcome of the cache access is unknown, as a miss to be on the safe side. In [31] it is shown that this intuitive assumption can be wrong on dynamically scheduled microprocessors. An example is provided where a cache hit can cause a longer execution than a cache miss. In [28] a hypothetical OOO microprocessor is modeled for the analysis.

However, verification of the proposed approach on a real processor is missing. Another issue is the missing or sometimes wrong documentation of the processor internals [13]. From a survey of the literature we found that modeling a new version of a microprocessor and finding all undocumented details is usually worth a full PhD thesis.

We argue that trying to catch up on the analysis side with the growing complexity of modern computer architectures is not feasible. A paradigm shift is necessary, either on the hardware level or on the application level. Puschner argues for a single-path programming style [41] that results in a constant execution time. In that case execution time can be simply measured. However, this programming paradigm is quite unusual and restrictive. We argue in this paper that the computer architecture has to be redefined or adapted for real-time systems. Predictable and *analyzable* execution time is of primary importance for this computer architecture.

3. 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."

The name JOP stands for *Java Optimized Processor* to enforce that the microcode instructions are optimized for Java bytecode. A direct implementation of all bytecodes [30] in hardware is not a useful approach. Some bytecodes are very complex (e.g., `new` has to interact with the garbage collector) and the dynamic instruction frequency is low [36,16]. All available Java processors implement only a subset of the instructions in hardware.

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 module extension provides the link between the processor core, and the memory and IO modules.

The processor core contains the three microcode pipeline stages *microcode fetch*, *decode* and *execute* and an additional translation stage *bytecode fetch*. The ports to the other modules are the two top elements of the stack (A and B), input to the top-of-stack (Data), bytecode cache address and 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

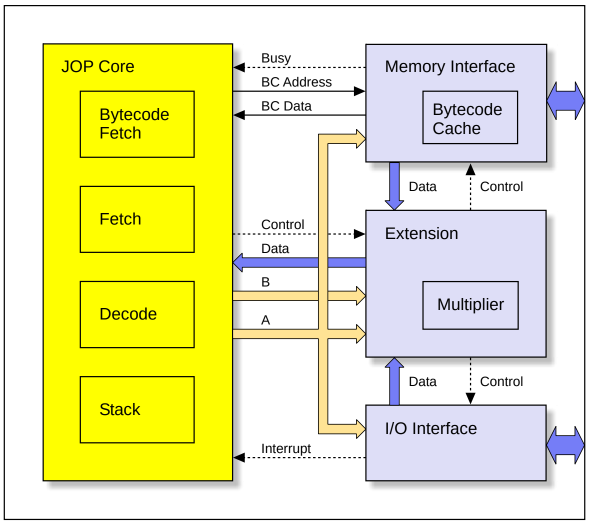

Fig. 1. Block diagram of JOP

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`[2] 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 request for a method to be placed in the cache is performed through the extension module, but the cache hit detection and load is performed by the memory interface independently of the processor core (and therefore concurrently).

The I/O interface contains peripheral devices, such as the system time and timer interrupt for real-time thread scheduling, a serial interface and application-specific devices. Read and write to and from this module are controlled by the extension module. All external devices are connected to the I/O interface.

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.

The division of the processor into those four modules greatly simplifies the adaptation of JOP for different application domains or hardware platforms. Porting JOP

[2] The busy signal can also be used to stall the whole processor pipeline. This was the change made to JOP by Flavius Gruian [17]. However, in this synchronization mode, the concurrency between the memory access module and the main pipeline is lost.

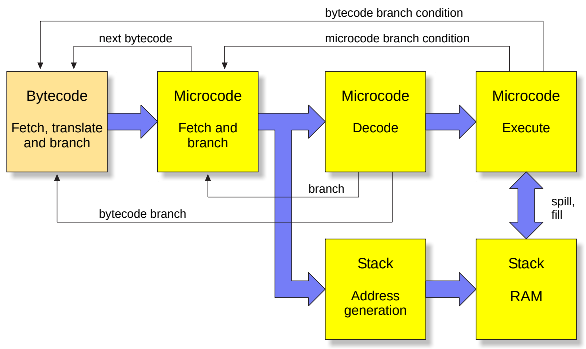

Fig. 2. Datapath of JOP

to a new FPGA board usually results in changes in the memory module alone. Using the same board for different applications only involves making changes to the I/O module. JOP has been ported to several different FPGAs and prototyping boards and has been used in different real-world applications, but it never proved necessary to change the processor core.

3.1. *The Processor Pipeline*

JOP is a fully pipelined architecture with single cycle execution of microcode instructions and a novel approach of translation from Java bytecode to these instructions. Figure 2 shows the datapath for JOP, representing the pipeline from left to right. Blocks arranged vertically belong to the same pipeline stage.

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 (the stack cache). As every stack machine microcode instruction (except `nop`, `wait`, and `jbr`) 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 to the stack cache needs neither an extra write-back stage nor any data forwarding. Figure 3 shows the architecture

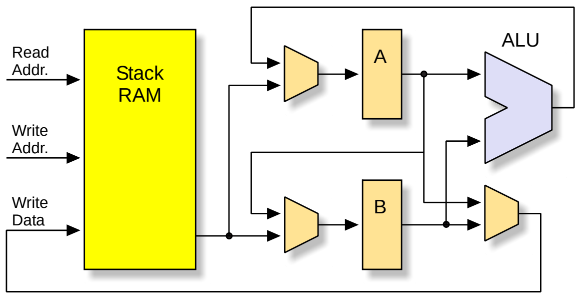

Fig. 3. The execution stage with the two-level stack cache

of the execution stage with the two-level stack cache. The operands for the ALU operation reside in the two registers. The result is written in the same cycle into register A again. That means execute and write back is performed in a single pipeline stage.

We will show that all operations can be performed with this architecture. Let A be the top-of-stack (TOS) and B the element below TOS. The memory that serves as the second level cache is represented by the array *sm*. Two indices in this array are used: p points to the logical third element of the stack and changes as the stack grows or shrinks, v points to the base of the local variables area in the stack and n is the address offset of a variable. *op* is a two operand stack operation with a single result (i.e. a typical ALU operation).

Case 1: ALU operation

$A \leftarrow A\ op\ B$

$B \leftarrow sm[p]$

$p \leftarrow p - 1$

The two operands are provided by the two top level registers. A single read access from *sm* is necessary to fill B with a new value.

Case 2: Variable load (*Push*)

$A \leftarrow sm[v+n]$

$B \leftarrow A$

$sm[p+1] \leftarrow B$

$p \leftarrow p + 1$

One read access from *sm* is necessary for the variable read. The former TOS value moves down to B and the data previously in B is written to *sm*.

Case 3: Variable store (*Pop*)

$sm[v+n] \leftarrow A$

$A \leftarrow B$

$B \leftarrow sm[p]$

$p \leftarrow p - 1$

The TOS value is written to *sm*. A is filled with B and B is filled in an identical manner to Case 1, needing a single read access from *sm*.

We can see that all three basic operations can be performed with a stack memory with one read and one write port. Assuming a memory is used that can handle concurrent read and write access, there is no structural access conflict between *A*, *B* and *sm*. That means that all operations can be performed concurrently in a single cycle. Further details of this two-level stack architecture, and that there are no RAW conflicts, are described in [50].

The short pipeline results in a short delay for a conditional branch. Therefore, a hard to analyze (with respect to WCET) branch prediction logic can be avoided. One question remains: Is the pipeline well balanced? Compared to other FPGA designs (see Section 5) the maximum frequency is quite high. To evaluate if we could do better we performed some experiments by adding pipeline stages in the critical path. In the 4-stage pipeline the critical path is in the first stage, the bytecode fetch and translation stage (100 MHz). Pipelining this unit increased the maximum frequency to 106 MHz and moved the critical path to the execution stage (the barrel shifter). Pipelining this barrel shifter resulted in 111 MHz and the critical path moved to the feedback of the branch condition (located in the microcode fetch stage). Pipelining this path moved the critical path to the microcode decode stage. That means that not a single stage dominates the critical path. From these experiments we conclude that the design with four pipeline stages result in a well balanced design.

3.2. *Interrupt Logic*

Interrupts and (precise) exceptions are considered hard to implement in a pipelined processor [21], 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 and exceptions (e.g., stack overflow) 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 3.4.3) 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. This feature is used to implement the preemptive, fixed priority real-time scheduler in Java [47].

3.3. *Cache*

A pipelined processor architecture calls for higher memory bandwidth. A standard technique to avoid processing bottlenecks due to the lower available memory bandwidth is caching. However, standard cache organizations improve the average execution time but are difficult to predict for WCET analysis [20]. 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 [50]. The *stack cache* is organized in two levels: the two top elements are implemented as registers, the lower level as a large on-chip memory. Fill and spill between these two levels is done in hardware. Fill and spill between the on-chip memory and the main memory is subjected to microcode control and therefore time-predictable. The exchange of the on-chip stack cache with the main memory can be either done on method invocation and return or on a thread switch.

In [49], a novel way to organize an instruction cache, as *method cache*, is given. The idea is to cache complete methods. A cache fill from main memory is only performed on a miss on method invocation or return. Therefore, all other bytecodes have a guaranteed cache hit. That means no instruction can stall the pipeline.

The cache is organized in blocks, similar to cache lines. However, the cached method has to span continuous[3] blocks. The *method cache* can hold more than one method. 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. The average case performance of this *method cache* is similar to a direct mapped cache [49]. The

[3] The cache addresses wrap around at the end of the on-chip memory. Therefore, a method is also considered continuous when it spans from the last to the first block.

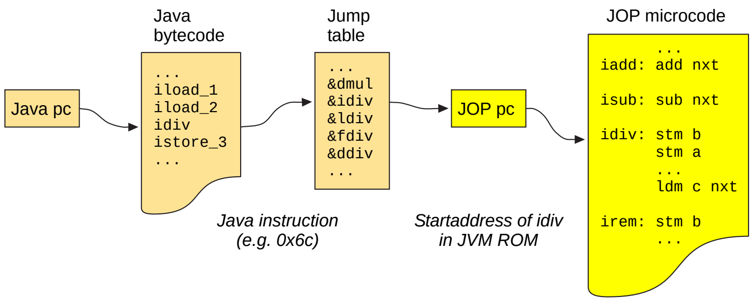

Fig. 4. Data flow from the Java program counter to JOP microcode

maximum method size is restricted by the size of the method cache. The pre-link tool verifies that the size restriction is fulfilled by the application.

3.4. *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[4] stack machine.

3.4.1. *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.

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.

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

Another common solution, used in Sun's picoJava [58], 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 (except for a bytecode branch that takes 4 instead of 3 cycles to execute). The area overhead of the translation stage is 290 LCs, or about 15% of the LCs of a typical JOP configuration. 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 4 gives an example of the data flow from the Java program counter to JOP microcode. The figure represents the two pipeline stages bytecode fetch/translate and microcode fetch. The fetched bytecode acts as an index for the jump table. The jump table contains the start addresses for the bytecode implementation in microcode. This address is loaded into the JOP program counter for every bytecode executed. JOP executes the sequence of microcode until the last one. The last one is marked with *nxt* in microcode assembler. This *nxt* bit in the microcode ROM triggers a new translation i.e., a new address is loaded into the JOP program counter. In Figure 4 the implementation of bytecode `idiv` is an example of a longer sequence that ends with microcode instruction `ldm c nxt`.

Some bytecodes, such as ALU operations and the short form access to *locals*, are directly implemented by an equivalent microcode instruction. Additional instructions are available to access internal registers, main memory and I/O devices. A relative conditional branch (zero/non zero of the top-of-stack) performs control

```
dup:    dup nxt    // 1 to 1 mapping

//  a and b are scratch variables at
//  the microcode level.

dup_x1: stm a      // save TOS
        stm b      // and TOS-1
        ldm a      // duplicate former TOS
        ldm b      // restore TOS-1
        ldm a nxt  // restore TOS and fetch
               // the next bytecode
```

Fig. 5. Implementation of `dup` and `dup_x1`

flow decisions at the microcode level. A detailed description of the microcode instructions can be found in [51].

The difference to other forms of instruction translation in hardware is that the proposed solution is time predictable. The translation takes one cycle (one pipeline stage) for each bytecode, independent from the execution history. Instruction folding, e.g., implemented in picoJava [36,58], is also a form of instruction translation in hardware. Folding is used to translate several (stack oriented) bytecode instructions to a RISC type instruction. This translation needs an instruction buffer and the fill level of this instruction buffer depends on the execution history. The length of this history that has to be considered for analysis is not bounded. Therefore this form of instruction translation is not exactly time predictable.

3.4.2. *Bytecode Example*

The example in Figure 5 shows the implementation of a single cycle bytecode and an infrequent bytecode as a sequence of JOP instructions. The suffix `nxt` marks the last instruction of the microcode sequence. In this example, the `dup` bytecode is mapped to the equivalent `dup` microcode and executed in a single cycle, whereas `dup_x1` takes five cycles to execute, and after the last instruction (`ldm a nxt`), the first instruction for the next bytecode is executed. The scratch variables, as shown in the second example, are stored in the on-chip memory that is shared with the stack cache.

Some bytecodes are followed by operands of between one and three bytes in length (except `lookupswitch` and `tableswitch`). Due to pipelining, the first operand byte that follows the bytecode instruction is available when the first microcode instruction enters the execution stage. If this is a one-byte long operand, it is ready to be accessed. The increment of the Java program counter after the read of an operand byte is coded in the JOP instruction (an *opd* bit similar to the *nxt* bit).

In Listing 6, the implementation of `sipush` is shown.

```
sipush: nop opd       // fetch next byte
        nop opd      // and one more
        ld_opd_16s nxt // load 16 bit operand
```

Fig. 6. Bytecode operand load

The bytecode is followed by a two-byte operand. Since the access to bytecode memory is only one[5] byte per cycle, *opd* and *nxt* are not allowed at the same time. This implies a minimum execution time of $n+1$ cycles for a bytecode with n operand bytes.

3.4.3. *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. That means bytecodes (e.g., `new` or floating point operations) can be implemented in Java. This feature also allows for the easy configuration of resource usage versus performance.

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. 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. A single empty static method consumes three 32-bit words in memory. Therefore, the overhead of this special class is 3 KB, which is 9% of a minimal *hello world* program (34 KB memory footprint).

3.5. *Architecture Summary*

In this section, we have introduced JOP's architecture. In order to handle the great variation in the complexity of Java bytecodes we have proposed a translation to a different instruction set, the so-called microcode. This microcode is still an instruction set for a stack machine, but more RISC-like than the CISC-like JVM bytecodes. The core of the stack machine constitutes a three-stage pipeline. An additional pipeline stage in front of this core pipeline stage performs bytecode fetch

[5] The decision is to avoid buffers that would introduce time dependencies over bytecode boundaries.

and the translation to microcode. This organization has no execution time overheads for more complex bytecodes and results in the short pipeline that is necessary for any processor without branch prediction. The additional translation stage also presents an elegant way of incorporating interrupts virtually *for free*. Only a multiplexor is needed in the path from the translation stage to the microcode decode stage. The microcode scratch variables are only valid during a microcode sequence for a bytecode and need not be saved on an interrupt.

At the time of this writing 43 of the 201 different bytecodes are implemented by a single microcode instruction, 93 by a microcode sequence, and 40 bytecodes are implemented in Java.

4. Worst-Case Execution Time

Worst-case execution time (WCET) estimates of tasks are essential for designing and verifying real-time systems. WCET estimates can be obtained either by measurement or static analysis. The problem with using measurements is that the execution times of tasks tend to be sensitive to their inputs. As a rule, measurement does not guarantee safe WCET estimates. Instead, static analysis is necessary for hard real-time systems. Static analysis is usually divided into a number of different phases:

Path analysis generates the control flow graph (a directed graph of basic blocks) of the program and annotates (manual or automatic) loops with bounds.

Low-level analysis determines the execution time of basic blocks obtained by the path analysis. A model of the processor and the pipeline provides the execution time for the instruction sequence.

Global low-level analysis determines the influence of hardware features such as caches on program execution time. This analysis can use information from the path analysis to provide less pessimistic values.

WCET Calculation collapses the control flow graph to provide the final WCET estimate. Alternative paths in the graph are collapsed to a single value (the largest of the alternatives) and loops are collapsed once the loop bound is known.

For the low-level analysis, a good timing model of the processor is needed. The main problem for the low-level analysis is the execution time dependency of instructions in modern processors that are not designed for real-time systems. JOP is designed to be an easy target for WCET analysis. The WCET of each bytecode can be predicted in terms of number of cycles it requires. There are no dependencies between bytecodes.

Each bytecode is implemented by microcode. We can obtain the WCET of a single bytecode by performing WCET analysis at the microcode level. To prove that there are no time dependencies between bytecodes, we have to show that no processor states are *shared* between different bytecodes.

WCET analysis has to be done at two levels: at the microcode level and at the bytecode level. The microcode WCET analysis is performed only once for a processor configuration and described in the next sections. The result from this microcode analysis is the timing model of the processor. The timing model is the input for the WCET analysis at the bytecode level (i.e. the Java application) as shown in the example in Section 4.5.1 and in the WCET tool description in Section 4.5.2.

It has to be noted that we cannot provide WCET values for the other Java systems from Section 6, e.g. the aJile Java processor, as there is no information on the instruction timing available.

4.1. *Microcode Path Analysis*

To obtain the WCET values for the individual bytecodes we perform the path analysis at the microcode level. First, we have to ensure that a number of restrictions (from [42]) of the code are fulfilled:

- Programs must not contain unbounded recursion. This property is satisfied by the fact that there exists no call instruction in microcode.
- Function pointers and computed `gotos` complicate the path analysis and should therefore be avoided. Only simple conditional branches are available at the microcode level.
- The upper bound of each loop has to be known. This is the only point that has to be verified by inspection of the microcode.

To detect loops in the microcode we have to find all backward branches (e.g. with a negative branch offset)[6]. The branch offsets can be found in a VHDL file (`offtbl.vhd`) that is generated during microcode assembly. In the current implementation of the JVM there are ten different negative offsets. However, not each offset represents a loop. Most of these branches are used to share common code. Three branches are found in the initialization code of the JVM. They are not part of a bytecode implementation and can be ignored. The only loop that is found in a regular bytecode is in the

[6] The loop branch can be a forward branch. However, the basic blocks of the loop contain at least one backward branch. Therefore we can identify all loops by searching for backward branches only.

implementation of `imul` to perform a fixed delay. The iteration count for this loop is constant.

A few bytecodes are implemented in Java [7] and can be analyzed in the same way as application code. The bytecodes `idiv` and `irem` contain a constant loop. The bytecodes `new` and `anewarray` contain loops to initialize (with zero values) new objects or arrays. The loop is bound by the size of the object or array. The bytecode `lookupswitch` [8] performs a linear search through a table of branch offsets. The WCET depends on the table size that can be found as part of the instruction.

As the microcode sequences are very short, the calculation of the control flow graph for each bytecode is done manually.

4.2. *Microcode Low-level Analysis*

To calculate the execution time of basic blocks in the microcode, we need to establish the timing of microcode instructions on JOP. All microcode instructions except `wait` execute in a single cycle, reducing the low-level analysis to a case of merely counting the instructions.

The `wait` instruction is used to stall the processor and wait for the memory subsystem to finish a memory transaction. The execution time of the `wait` instruction depends on the memory system and, if the memory system is predictable, has a known WCET. A main memory consisting of SRAM chips can provide this predictability and this solution is therefore advised. The predictable handling of DMA, which is used for the instruction cache fill, is explained in [49]. The `wait` instruction is the only way to stall the processor. Hardware events, such as interrupts (see [46]), do not stall the processor.

Microcode is stored in on-chip memory with single cycle access. Each microcode instruction is a single word long and there is no need for either caching or prefetching at this stage. We can therefore omit performing a low-level analysis. No pipeline analysis [13], with its possible unbound timing effects, is necessary.

4.3. *Bytecode Independency*

We have seen that all microcode instructions except `wait` take one cycle to execute and are therefore independent of other instructions. This property directly translates to independency of bytecode instructions.

The `wait` microcode instruction provides a convenient way to hide memory access time. A memory read or write can be triggered in microcode and the processor can continue with microcode instructions. When the data from a memory read is needed, the processor explicitly waits, with the `wait` instruction, until it becomes available.

For a memory store, this `wait` could be deferred until the memory system is used next (similar to a write buffer). It is possible to initiate the store in a bytecode such as `putfield` and continue with the execution of the next bytecode, even when the store has not been completed. In this case, we introduce a dependency over bytecode boundaries, as the state of the memory system is *shared*. To avoid these dependencies that are difficult to analyze, each bytecode that accesses memory waits (preferably at the end of the microcode sequence) for the completion of the memory request.

Furthermore, if we would not wait at the end of the store operation we would have to insert an additional `wait` at the start of every read operation. Since read operations are more frequent than write operations (15% vs. 2.5%, see [51]), the performance gain from the hidden memory store is lost.

4.4. *WCET of Bytecodes*

The control flow of the individual bytecodes together with the basic block length (that directly corresponds with the execution time) and the time for memory access result in the WCET (and BCET) values of the bytecodes. These exact values for each bytecode can be found in [51].

Simple bytecode instructions are executed by either one microinstruction or a short sequence of microinstructions. The execution time in cycles equals the number of microinstructions executed. As the stack is on-chip it can be accessed in a single cycle. We do not need to incorporate the main memory timing into the instruction timing. Table 2 shows examples of the execution time of such bytecodes.

Object oriented instructions, array access, and invoke instructions access the main memory. Therefore we have to model the memory access time. We assume a simple SRAM with a constant access time. Access time that exceeds a single cycle includes additional wait states (r_{ws} for a memory read and w_{ws} for a memory write). The following example gives the execution time for `getfield`, the read access of an object field:

$$t_{getfield} = 10 + 2r_{ws}$$

[7] The implementation can be found in the class `com.jopdesign.sys.JVM`.

[8] `lookupswitch` is one way of implementing the Java `switch` statement. The other bytecode, `tableswitch`, uses an index in the table of branch offsets and has therefore a constant execution time.

Table 2
Execution time of simple bytecodes in cycles

Opcode	Instruction	Cycles	Funtion
3	iconst_0	1	load constant 0 on TOS
4	iconst_1	1	load constant 1 on TOS
16	bipush	2	load a byte constant on TOS
17	sipush	3	load a short constant on TOS
21	iload	2	load a local on TOS
26	iload_0	1	load local 0 on TOS
27	iload_1	1	load local 1 on TOS
54	istore	2	store the TOS in a local
59	istore_0	1	store the TOS in local 0
60	istore_1	1	store the TOS in local 1
89	dup	1	duplicate TOS
90	dup_x1	5	complex stack manipulation
96	iadd	1	integer addition
153	ifeq	4	conditional branch

However, the memory subsystem performs read and write parallel to the execution of microcode. Therefore, some access cycles can be hidden. The following example gives the exact execution time of bytecode `ldc2_w` in clock cycles:

$$t_{ldc2\_w} = 17 + \begin{cases} r_{ws} - 2 & : \quad r_{ws} > 2 \\ 0 & : \quad r_{ws} \leq 2 \end{cases} + \begin{cases} r_{ws} - 1 & : \quad r_{ws} > 1 \\ 0 & : \quad r_{ws} \leq 1 \end{cases}$$

Thus, for a memory with two cycles access time ($r_{ws} = 1$), as we use it for a 100 MHz version of JOP with a 15 ns SRAM, the wait state is completely hidden by microcode instructions for this bytecode.

Memory access time also determines the cache load time on a miss. For the current implementation the cache load time is calculated as follows: the wait state c_{ws} for a single word cache load is:

$$c_{ws} = \begin{cases} r_{ws} - 1 & : \quad r_{ws} > 1 \\ 0 & : \quad r_{ws} \leq 1 \end{cases}$$

On a method invoke or return the bytecode has to be loaded into the cache on a cache miss. The load time l is:

$$l = \begin{cases} 6 + (n+1)(2 + c_{ws}) & : \quad \text{cache miss} \\ 4 & : \quad \text{cach hit} \end{cases}$$

where n is the length of the method in number of 32-bit words. For short methods the load time of the method on a cache miss, or part of it, is hidden by microcode execution. As an example the exact execution time for the bytecode `invokestatic` is:

$$t = 74 + r + \begin{cases} r_{ws} - 3 & : \quad r_{ws} > 3 \\ 0 & : \quad r_{ws} \leq 3 \end{cases} + \begin{cases} r_{ws} - 2 & : \quad r_{ws} > 2 \\ 4 & : \quad r_{ws} \leq 2 \end{cases}$$
$$+ \begin{cases} l - 37 & : \quad l > 37 \\ 0 & : \quad l \leq 37 \end{cases}$$

For `invokestatic` a cache load time l of up to 37 cycles is completely hidden. For the example SRAM timing the cache load of methods up to 36 bytes long is hidden. The WCET analysis tool, as described in the next section, knows the length of the invoked method and can therefore calculate the time for the invoke instruction cycle accurate.

```
final static int N = 5;

static void sort(int[] a) {

   int i, j, v1, v2;
   // loop count = N-1
   for (i=N-1; i>0; --i) {
      // loop count = (N-1)*N/2
      for (j=1; j<=i; ++j) {
         v1 = a[j-1];
         v2 = a[j];
         if (v1 > v2) {
            a[j] = v1;
            a[j-1] = v2;
         }
      }
   }
}
```

Fig. 7. Bubble Sort test program for the WCET analysis

4.5. *WCET Analysis of the Java Application*

We conclude this section with a worst-case analysis (now at the bytecode level) of Java applications. First we provide manual analysis on a simple example and than a brief description of the automation through a WCET analyzer tool.

4.5.1. *An Example*

In this section we perform manually a worst and best case analysis of a classic example, the Bubble Sort algorithm. The values calculated are compared with the measurements of the execution time on JOP on all permutations of the input data. Figure 7 shows the test program in Java. The algorithm contains two nested loops and one condition. We use an array of five elements to perform the measurements for all permutations (i.e. $5! = 120$) of the input data. The number of iterations of the outer loop is one less than the array size: $c_1 = N - 1$, in this case four. The inner loop is executed $c_2 = \sum_{i=1}^{c_1} i = c_1(c_1+1)/2$ times, i.e. ten times in our example.

The annotated control flow graph (CFG) of the example is shown in Figure 8. The edges contain labels showing how often the path between two nodes is taken. We can identify the outer loop, containing the blocks B2, B3, B4 and B8. The inner loop consists of blocks B4, B5, B6 and B7. Block B6 is executed when the

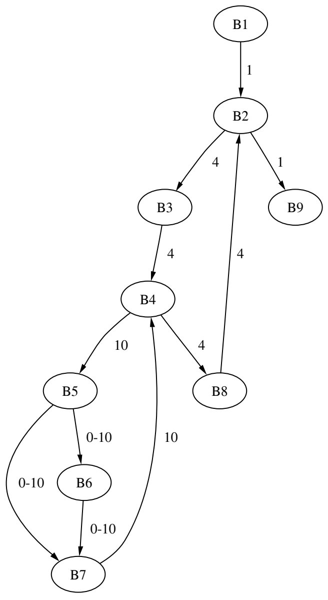

Fig. 8. The control flow graph of the Bubble Sort example

Table 3
WCET and BCET in clock cycles of the basic blocks

			WCET		BCET	
Block	Addr.	Cycles	Count	Total	Count	Total
B1	0:	2	1	2	1	2
B2	2:	5	5	25	5	25
B3	6:	2	4	8	4	8
B4	8:	6	14	84	14	84
B5	13:	74	10	740	10	740
B6	30:	73	10	730	0	0
B7	41:	15	10	150	10	150
B8	47:	15	4	60	4	60
B9	53:		1		1	
Execution time calculated				1799		1069
Execution time measured				1799		1069

condition of the `if` statement is true. The path from B5 to B7 is the only path that depends on the input data.

In Table 3 the basic blocks with the start address (Addr.) and their execution time (Cycles) in clock cycles and the worst and best case execution frequency (Count) is given. The values in the forth and sixth columns (Count) of Table 3 are derived from the CFG and show how often the basic blocks are executed in the worst and best cases. The WCET and BCET value for each block is calculated by multiplying the clock cycles by the execution frequency. The overall WCET and BCET values are calculated by summing the values of the individual blocks B1 to B8. The last block (B9) is omitted, as the measurement does not contain the return statement.

The execution time of the program is measured using the cycle counter in JOP. The current time is taken at

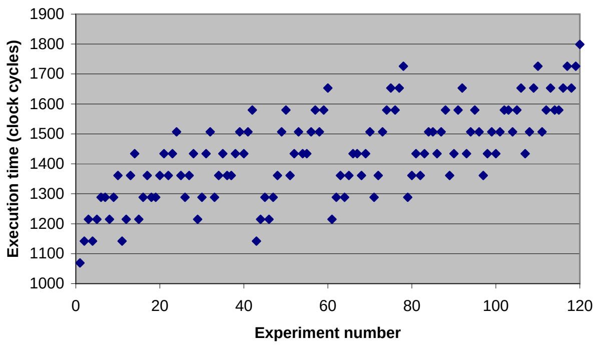

Fig. 9. Execution time in clock cycles of the Bubble Sort program for all 120 permutations of the input data

both the entry of the method and at the end, resulting in a measurement spanning from block B1 to the beginning of block B9. The last statement, the `return`, is not part of the measurement. The difference between these two values (less the additional 8 cycles introduced by the measurement itself) is given as the execution time in clock cycles (the last row in Table 3). The measured WCET and BCET values are exactly the same as the calculated values.

In Figure 9, the measured execution times for all 120 permutations of the input data are shown. The vertical axis shows the execution time in clock cycles and the horizontal axis the number of the test run. The first input sample is an already sorted array and results in the lowest execution time. The last sample is the worst-case value resulting from the reversely ordered input data. We can also see the 11 different execution times that result from executing basic block B6 (which performs the element exchange and takes 73 clock cycles) between 0 and 10 times.

4.5.2. *WCET Analyzer*

In [53] we have presented a static WCET analysis tool for Java. During the high-level analysis the the relevant information is extracted from the class files. The control flow graph (CFG) of the basic blocks [9] is extracted from the bytecodes. Annotations for the loop counts are extracted from comments in the source. Furthermore, the class hierarchy is examined to find all possible targets for a method invoke.

The tool performs the low-level analysis at the bytecode level. The behavior of the method cache is integrated for a simpler form (a two block cache). The well known execution times of the different bytecodes (see Section 4.4) simplifies this part of the WCET analysis,

[9] A basic block is a sequence of instructions without any jumps or jump targets within this sequence.

Table 4
WCET benchmark examples

Program	Description	LOC
`crc`	CRC calculation for short messages	8
`robot`	A simple line follower robot	111
`Lift`	A lift controler	635
`Kfl`	*Kippfahrleitung* application	1366
`UdpIp`	UDP/IP benchmark	1297

Table 5
Measured and estimated WCETs with results in clock cycles

Program	Measured (cycle)	Estimated (cycle)	Pessimism (ratio)
`crc`	1552	1620	1.04
`robot`	736	775	1.05
`Lift`	7214	11249	1.56
`Kfl`	13334	28763	2.16
`UdpIp`	11823	219569	18.57

which is usually the most complex one, to a great extent. As there are no pipeline dependencies the calculation of the execution time for a basic block is merely just adding the individual cycles for each instruction.

The actual calculation of the WCET is transformed to an integer linear programming problem, a well known technique for WCET analysis [43,29]. We performed the WCET analysis on several benchmarks (see Table 4). We also *measured* the WCET values for the benchmarks. It has to be noted that we actually cannot measure the real WCET. If we could measure it, we would not need to perform the WCET analysis at all. The measurement gives us an idea of the pessimism of the analyzed WCET. The benchmarks `Lift` and `Kfl` are real-world examples that are in industrial use. `Kfl` and `UdpIp` are also part of an embedded Java benchmark suit that is used in Section 6.

Table 5 shows the measured execution time and the analyzed WCET. The last column gives an idea of the pessimism of the WCET analysis. For very simple programs, such as `crc` and `robot`, the pessimism is quite low. For the `Lift` example it is in an acceptable range. The difference between the measurement and the analysis in the `Kfl` example results from the fact that our measurement does not cover the WCET path. The large conservatism in `UdpIp` results from the loop bound in the IP and UDP checksum calculation. It is set for a 1500 byte packet buffer, but the payload in the benchmark is only 8 bytes. The last two examples also show the issue when a real-time application is developed without a WCET analysis tool available.

The WCET analysis tool, with the help of loop annotations, provides WCET values for the schedulability analysis. Besides the calculation of the WCET the tool provides user feedback by generating bytecode listings with timing information and a graphical representation of the CFG with timing and frequency information. This representation of the WCET path through the code can guide the developer to write WCET aware real-time code.

4.6. *Discussion*

The Bubble Sort example and experiments with the WCET analyzer tool have demonstrated that we have achieved our goal: JOP is a simple target for the WCET analysis. Most bytecodes have a single execution time (WCET = BCET), and the WCET of a task (the analysis at the bytecode level) depends only on the control flow. No pipeline or data dependencies complicate the low-level part of the WCET analysis.

The same analysis is not possible for other Java processors. Either the information on the bytecode execution time is missing [10] or some processor features (e.g., the high variability of the latency for a trap in picoJava) would result in very conservative WCET estimates. Another example that prohibits exact analysis is the mechanism to automatically fill and spill the stack cache in picoJava. The time when the memory (cache) is occupied by this spill/fill action depends on a long instruction history. Also the fill level of the 16-byte-deep prefetch buffer, which is needed for instruction folding, depends on the execution history. All this automatically buffering features have to be modeled quite conservative. A pragmatic solution is to assume empty buffers at the start of a basic block. As basic blocks are quite short most of the buffering/prefetching does not help to lower the WCET.

Only for the Cjip processor the execution time is well documented [24]. However, as seen in Section 6.2, the *measured* execution time of some bytecodes is *higher* than the documented values. Therefore the documentation is not complete to provide a safe processor model of the Cjip for the WCET analysis.

5. Resource Usage

Cost 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]). Processors for embedded systems are therefore optimized for minimum chip size. In this

[10] We tried hard to get this information for the aJile processor.

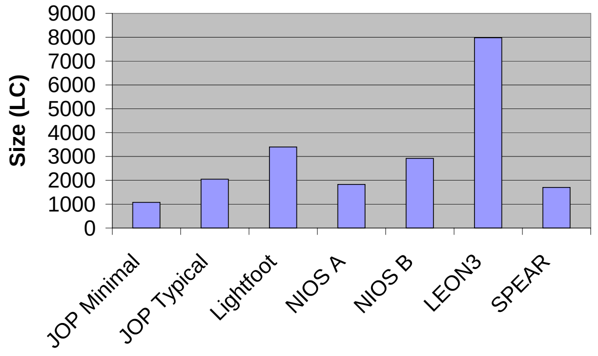

Fig. 10. Size in logic cells (LC) of different soft-core processors

Table 6
Size and maximum frequency of FPGA soft-core processors

Processor	Resources (LC)	Memory (KB)	fmax (MHz)
JOP Minimal	1077	3.25	98
JOP Typical	2049	3.25	100
Lightfoot [11] [9]	3400	4	40
NIOS A [2]	1828	6.2	120
NIOS B [2]	2923	5.5	119
LEON3 [15]	7978	10.9	35
SPEAR [12] [10]	1700	8	80

section, we will compare JOP with different processors in terms of size.

One major design objective in the development of JOP was to create a small system that can be implemented in a low-cost FPGA. Figure 10 and Table 6 show the resource usage for different configurations of JOP and different soft-core processors implemented in an Altera EP1C6 FPGA [3]. 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. The maximum frequency for all soft-core processors is in the same technology or normalized (SPEAR) to the technology.

All configurations of JOP contain the on-chip microcode memory, the 1 KB stack cache, a 1 KB method cache, a memory interface to a 32-bit static RAM, and an 8-bit FLASH interface for the Java program and the FPGA configuration data. The minimum configuration implements multiplication and the shift operations in microcode. In the typical 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 consumes about 30% of the LCs in a Cyclone EP1C6, thus leaving enough resources free for application-specific logic.

As a reference, NIOS [2], Altera's popular RISC soft-core, is also included in Table 6. 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.

LEON3 [15], the open-source implementation of the SPARC V8 architecture, has been ported to the exact same hardware that was used for the JOP numbers. LEON3 is a representative of a RISC processor that is used in embedded real-time systems (e.g., by ESA for space missions).

SPEAR [10] (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 [39]. 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 [66]. Only the instantiation and initialization code for 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 (83 MHz).

From this comparison we can see that we have achieved our objective of designing a small processor. The commercial Java processor, Lightfoot, consumes 1.7 times the logic resources of JOP in the typical configuration (with a lower clock frequency). A typical 32-bit RISC processor (NIOS) consumes about 1.5 times (LEON about four times) the resources of JOP. However, the NIOS processor can be clocked 20% faster than JOP in the same technology. The vendor independent and open-source RISC processor LEON can be clocked only with 35% of JOP's frequency. The only processor that is similar in size is SPEAR. How-

[11] The data for the Lightfoot processor is taken from the data sheet [9]. The frequency used is that in a Vertex-II device from Xilinx. JOP can be clocked at 100 MHz in the Vertex-II device, making this comparison valid.

[12] 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 40 MHz in an APEX device and JOP can be clocked at 50 MHz in the same device.

ever, while SPEAR is a 16-bit processor, JOP contains a 32-bit datapath.

6. Performance

In this section, we will evaluate the performance of JOP in relation to other embedded Java systems. Although JOP is intended as a processor with a low WCET for all operations, its general performance is still important.

6.1. *General 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 [55], needs more functionality than what is usually available in a CLDC compliant device (e.g., a filesystem and `java.net`). Some benchmarks from the SPECjvm98 suits also need several MB of heap.

Due to the absence of a *standard* Java benchmark for embedded systems, a small benchmark suite 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 [45] (the first industrial application of JOP). The application is written as a cyclic executive. 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. The Kfl benchmark consists of 511 methods and 14 KB code, the UDP/IP benchmark of 508 methods and 13 KB code (including the supporting library).

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 and an 'overhead' loop that contains any overheads from the benchmark that should be subtracted from the result (this feature is designed for the micro-benchmarks). 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 accuracy of the measurement depends on the resolution of the system time. For the measurements under Linux, the system time has a resolution of 10ms, resulting in an inaccuracy of 1%. The accuracy of the system time on leJOS, TINI and the aJile is not known, but is considered to be in the same range. For JOP, a μs counter is used for time measurement.

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

JOP is implemented in a Cyclone FPGA [3], running at 100 MHz. The main memory is a 32-bit SRAM (15ns) with an access time of 2 clock cycles. The benchmarked configuration of JOP contains a 4 KB method cache 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 [22], running at 16 MHz. leJOS [54] is a tiny interpreting JVM for the RCX.

KVM is a port of the Sun's KVM that is part of the Connected Limited Device Configuration (CLDC) [57] to Alteras NIOS II processor on MicroC Linux. NIOS is implemented on a Cyclone FPGA and clocked with 50 MHz. Besides the different clock frequency this is a good comparison of an interpreting JVM running in the same FPGA as JOP.

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

Cjip The measured system [23] is a replacement of the TINI board and contains a Cjip [25] clocked with 80 MHz and 8 MB DRAM.

The benchmark results of **Komodo** were obtained by Matthias Pfeffer [37] on a cycle-accurate simulation of Komodo.

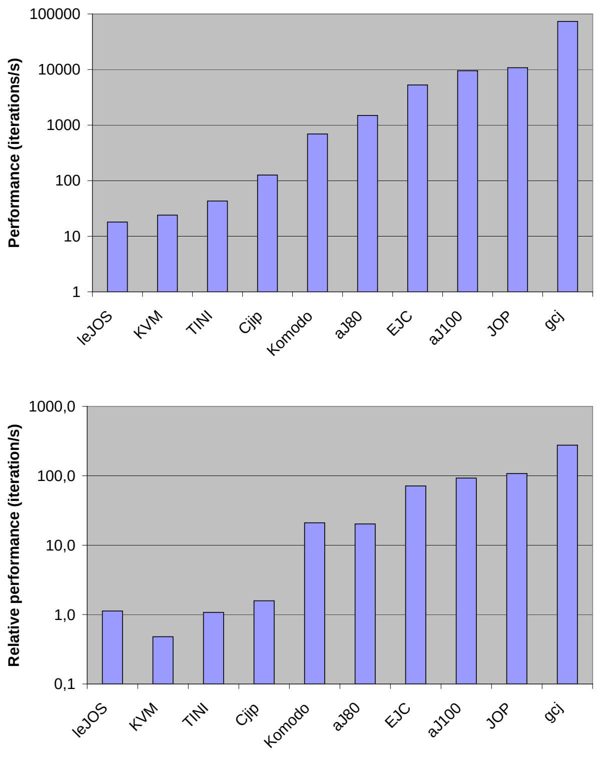

Fig. 11. Performance comparison of different Java systems with application benchmarks. The diagrams show the geometric mean of the two benchmarks in iterations per second – a higher value means higher performance. The top diagram shows absolute performance, while the bottom diagram shows the result scaled to 1 MHz clock frequency.

aJile's JEMCore is a direct-execution Java processor that is available in two different versions: the **aJ80** and the **aJ100** [1]. The aJ100 provides a generic 8-bit, 16-bit or 32-bit external bus interface, while the aJ80 only provides an 8-bit interface.

The **EJC** (Embedded Java Controller) platform [12] is a typical example of a JIT system on a RISC processor. The system is based on a 32-bit ARM720T processor running at 74 MHz. 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 266 MHz Pentium MMX under Linux.

MB is the realization of Java on a RISC processor for an FPGA (Xilinx MicroBlaze [65]). Java is compiled to C with a Java compiler for real-time systems [35] and the C program is compiled with the standard GNU toolchain.

It would be interesting to include the other soft-core Java processors (Moon, Lightfoot, and FemtoJava) in this comparison. However, it was not possible to obtain the benchmark data. The company that produced Moon seems to be disappeared and FemtoJava could not run all benchmarks.

Table 7
Application benchmarks on different Java systems. The table shows the benchmark results in iterations per second – a higher value means higher performance.

	Frequency (MHz)	Kfl	UDP/IP	Geom. Mean	Scaled
		(Iterations/s)			
JOP	100	17111	6781	10772	108
leJOS	16	25	13	18	1.1
TINI	40	64	29	43	1.1
KVM	50	36	16	24	0.5
Cjip	80	176	91	127	1.6
Komodo	33	924	520	693	21
aJ80	74	2221	1004	1493	20
aJ100	103	14148	6415	9527	92
EJC	74	9893	2822	5284	71
gcj	266	139884	38460	73348	276
MB	100	3792			

In Figure 11, 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 1 MHz clock frequency. The results of the application benchmarks and the geometric mean are shown in Table 7.

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 100 MHz JOP to 1 GHz, this would also involve reducing the memory access time from 15 ns to 1.5 ns. Processors with 1 GHz clock frequency are already available, but the fastest asynchronous SRAM to date has an access time of 10 ns.

6.2. *Discussion*

When comparing JOP and the aJile processor against leJOS, KVM, and TINI, 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 4 KB [49]. It therefore fits in the level one cache of the Pentium MMX. For a compar-

Table 8
Execution time in clock cycles for various JVM bytecodes

	JOP	leJOS	TINI	Cjip	Komodo	aJ80	aJ100
iload iadd	2	836	789	55	8	38	8
iinc	8	422	388	46	4	41	11
ldc	9	1340	1128	670	40	67	9
if_icmplt taken	6	1609	1265	157	24	42	18
if_icmplt n/taken	6	1520	1211	132	24	40	14
getfield	22	1879	2398	320	48	142	23
getstatic	15	1676	4463	3911	80	102	15
iaload	36	1082	1543	139	28	74	13
invoke	128	4759	6495	5772	384	349	112
invoke static	100	3875	5869	5479	680	271	92
invoke interface	144	5094	6797	5908	1617	531	148

ison 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, 10 ns SRAMs. JOP with its 15 ns SRAMs is about 12% faster than the aJ100 processor.

The MicroBlaze system is a representation of a Java batch-compilation system for a RISC processor. MicroBlaze is configured with the same cache[13] 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[14], as the Java2C compiler [35] is still under development.

The micro-benchmarks are intended to give insight into the implementation of the JVM. In Table 8, 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 in the benchmark loop. The single bytecode would trash the stack. As a minimum requirement, a second instruction is necessary in the loop to reverse the stack operation.

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 8 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% [51].

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.

The Cjip implements the JVM with a stack oriented instruction set. It is the only example (except JOP) where the instruction set is documented *including* the execution time [24]. We will therefore check some of the results with the numbers provided in the documentation. The execution time is given in ns, assuming a 66 MHz clock. The execution time for the basic integer add operation is given as 180 ns resulting in 12 cycles. The load of a local variable (when it is one of the first four) takes 35 cycles. In the micro-benchmark we measure 55 cycles instead of the theoretical 47 (`iadd` + `iload_n`). We assume that we have to add some cycles for the fetch of the bytecodes from memory.

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.

7. 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 performed the WCET analysis of the implemented JVM at the microcode level. This analysis provides the WCET and BCET values for the individual bytecodes. We have also shown that there are no dependencies between individual bytecodes. This feature, in combination with the method cache [49], makes JOP an easy target for low-level WCET analysis of

[13] The MicroBlaze with a 8 KB data and 8 KB instruction cache is about 1.5 times faster than JOP. However, a 16 KB memory is not available in low-cost FPGAs and is an unbalanced system with respect to the LC/memory relation.

[14] As not all language constructs can be compiled, only the Kfl benchmark was measured. Therefore, the bars for MicroBlaze are missing in Fig. 11.

Java applications. As far as we know, JOP is the only Java processor for which the WCET of the bytecodes is known and documented.

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 12% 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/`.

Acknowledgment

The author thanks the anonymous reviewers for their insightful and detailed comments on the first version of the paper. The comments have helped to shape this paper and clarify ambiguous sections in the first version. The author also thanks Andreas Steininger and Peter Puschner for their support during the PhD thesis and comments on this paper.

References

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

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

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

[4] ARM. Jazelle technology: ARM acceleration technology for the Java platform. white paper, 2004.

[5] David F. Bacon, Perry Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 285–298, New York, NY, USA, 2003. ACM Press.

[6] Antonio Carlos Beck and Luigi Carro. Low power java processor for embedded applications. In *Proceedings of the 12th IFIP International Conference on Very Large Scale Integration*, December 2003.

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

[8] Cyrille Comar, Gary Dismukes, and Franco Gasperoni. Targeting GNAT to the Java virtual machine. In *TRI-Ada '97: Proceedings of the conference on TRI-Ada '97*, pages 149–161, New York, NY, USA, 1997. ACM Press.

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

[10] Martin Delvai, Wolfgang Huber, Peter Puschner, and Andreas Steininger. Processor support for temporal predictability – the spear design example. In *Proceedings of the 15th Euromicro International Conference on Real-Time Systems*, Jul. 2003.

[11] 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*, 17(3):60–71, July 2000.

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

[13] Jakob Engblom. *Processor Pipelines and Static Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, 2002.

[14] Jakob Engblom, Andreas Ermedahl, Mikael Södin, Jan Gustafsson, and Hans Hansson. Worst-case execution-time analysis for embedded real-time systems. *International Journal on Software Tools for Technology Transfer (STTT)*, V4(4):437–455, August 2003.

[15] Jiri Gaisler. A portable and fault-tolerant microprocessor based on the SPARC v8 architecture. In *DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*, page 409, Washington, DC, USA, 2002. IEEE Computer Society.

[16] David Gregg, James Power, and John Waldron. Benchmarking the java virtual architecture - the specjvm98 benchmark suite. In N. Vijaykrishnan and M. Wolczko, editors, *Java Microarchitectures*, pages 1–18. Kluwer Academic, 2002.

[17] Flavius Gruian, Per Andersson, Krzysztof Kuchcinski, and Martin 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.

[18] Tom R. Halfhill. Imsys hedges bets on Java. *Microprocessor Report*, August 2000.

[19] David S. Hardin. Real-time objects on the bare metal: An efficient hardware realization of the Java virtual machine. In *Proceedings of the Fourth International Symposium on Object-Oriented Real-Time Distributed Computing*, page 53. IEEE Computer Society, 2001.

[20] Reinhold Heckmann, Marc Langenbach, Stephan Thesing, and Reinhard Wilhelm. The influence of processor architecture on the design and results of WCET tools. *Proceedings of the IEEE*, 91(7):1038–1054, Jul. 2003.

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

[22] Hitachi. Hitachi single-chip microcomputer h8/3297 series. Hardware Manual.

[23] Imsys. Snap, simple network application platform. Available at http://www.imsys.se/.

[24] Imsys. ISAJ reference 2.0, January 2001.

[25] Imsys. Im1101c (the cjip) technical reference manual / v0.25, 2004.

[26] Java Expert Group. Java specification request JSR 302: Safety critical java technology. Available at http://jcp.org/en/jsr/detail?id=302.

[27] J. Kreuzinger, U. Brinkschulte, M. Pfeffer, S. Uhrig, and Th. Ungerer. Real-time event-handling and scheduling on a multithreaded Java microcontroller. *Microprocessors and Microsystems*, 27(1):19–31, 2003.

[28] Xianfeng Li, Abhik Roychoudhury, and Tulika Mitra. Modeling out-of-order processors for WCET analysis. *Real-Time Systems*, V34(3):195–227, November 2006.

[29] Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In *LCTES '95: Proceedings of the ACM SIGPLAN 1995 workshop on Languages, compilers, & tools for real-time systems*, pages 88–98, New York, NY, USA, 1995. ACM Press.

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

[31] Thomas Lundqvist and Per Stenström. Timing anomalies in dynamically scheduled microprocessors. In *RTSS '99: Proceedings of the 20th IEEE Real-Time Systems Symposium*, page 12, Washington, DC, USA, 1999. IEEE Computer Society.

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

[33] K. Nilsen, L. Carnahan, and M. Ruark. Requirements for real-time extensions for the Java platform. Available at http://www.nist.gov/rt-java/, September 1999.

[34] Kelvin D. Nilsen and Bernt Rygg. Worst-case execution time analysis on modern processors. *SIGPLAN Not.*, 30(11):20–30, 1995.

[35] Anders Nilsson. Compiling java for real-time systems. Licentiate thesis, Dept. of Computer Science, Lund University, May 2004.

[36] J. Michael O'Connor and Marc Tremblay. picoJava-I: The Java virtual machine in hardware. *IEEE Micro*, 17(2):45–53, 1997.

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

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

[39] Peter 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.

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

[41] Peter Puschner and Alan Burns. Writing temporally predictable code. In *Proceedings of the The Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002)*, page 85, Washington, DC, USA, 2002. IEEE Computer Society.

[42] Peter Puschner and Christian Koza. Calculating the maximum execution time of real-time programs. *Real-Time Syst.*, 1(2):159–176, 1989.

[43] Peter Puschner and Anton Schedl. Computing maximum task execution times – a graph-based approach. *Journal of Real-Time Systems*, 13(1):67–91, Jul. 1997.

[44] Sven Gestegard Robertz and Roger Henriksson. Time-triggered garbage collection: robust and adaptive real-time GC scheduling for embedded systems. In *LCTES '03: Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pages 93–102, New York, NY, USA, 2003. ACM Press.

[45] Martin 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)*, pages 165–176, Austria, Vienna, June 2003.

[46] Martin Schoeberl. Design rationale of a processor architecture for predictable real-time execution of Java programs. In *Proceedings of the 10th International Conference on Real-Time and Embedded Computing Systems and Applications (RTCSA 2004)*, Gothenburg, Sweden, August 2004.

[47] Martin Schoeberl. Real-time scheduling on a Java processor. In *Proceedings of the 10th International Conference on Real-Time and Embedded Computing Systems and Applications (RTCSA 2004)*, Gothenburg, Sweden, August 2004.

[48] Martin Schoeberl. Restrictions of Java for embedded real-time systems. In *Proceedings of the 7th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2004)*, pages 93–100, Vienna, Austria, May 2004.

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

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

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

[52] Martin Schoeberl. Real-time garbage collection for Java. In *Proceedings of the 9th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC 2006)*, pages 424–432, Gyeongju, Korea, April 2006.

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

[54] Jose Solorzano. leJOS: Java based os for lego RCX. Available at: http://lejos.sourceforge.net/.

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

[56] O. Strom, K. Svarstad, and E. J. Aas. On the utilization of Java technology in embedded systems. *Design Automation for Embedded Systems*, 8(1):87–106, 2003.

[57] Sun. Java 2 platform, micro edition (J2ME). Available at: http://java.sun.com/j2me/docs/.

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

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

[60] S. Tucker Taft. Programming the internet in Ada 95. In *Ada-Europe '96: Proceedings of the 1996 Ada-Europe International Conference on Reliable Software Technologies*, pages 1–16, London, UK, 1996. Springer-Verlag.

[61] Stephan Thesing. *Safe and Precise Worst-Case ExecutionTime Prediction by Abstract Interpretation of Pipeline Models*. PhD thesis, University of Saarland, 2004.

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

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

[64] Andy Wellings. Is Java augmented with the RTSJ a better real-time systems implementation technology than Ada 95? *Ada Lett.*, XXIII(4):16–21, 2003.

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

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