Chapter 1
Hardware Support for Embedded Java

Martin Schoeberl

1.1 Introduction
1.2 Java Processors
1.3 Support Technology for Java
1.4 Conclusions

The general Java runtime environment is resource hungry and unfriendly for real-time systems. To reduce the resource consumption of Java in embedded systems, direct hardware support of the language is a valuable option. Furthermore, an implementation of the Java virtual machine in hardware enables worst-case execution time analysis of Java programs. This chapter gives an overview of current approaches to hardware support for embedded and real-time Java.

1.1 Introduction

Embedded systems are usually resource constraint systems and often have to perform computations under time constraints. Although the first version of Java has been designed for an embedded system, current implementations of Java are quite resource hungry and the execution time of a Java application is hard to predict statically. One approach to reduce resource consumption for Java and enable worst-case execution time (WCET) analysis is to provide hardware support for embedded Java. In this chapter hardware implementations of the Java virtual machine (JVM), Java processors, and hardware support for Java specific idioms (e.g., garbage collection) are described.

Standard Java is not the best fit for embedded systems. Sun introduced the Java micro edition and the Connected Limited Device Configuration (CLDC) [62] for embedded systems. CLDC is a subset of Java resulting in a lower memory footprint. Based on CLDC, the real-time specification for Java (RTSJ) [8] defines a new memory model and strengthens the scheduling guarantees to enable Java for real-time systems. Safety-critical Java (SCJ) [30] defines a subset of RTSJ for safety-critical

Martin Schoeberl
Department of Informatics and Mathematical Modeling, Technical University of Denmark e-mail: masca@imm.dtu.dk

Table 1.1: Relevant Java processors for embedded Java

	Target technology	Size Logic	Size Memory	Speed (MHz)	Year (pub.)
picoJava [38, 60]	ASIC, Altera FPGA	27500 LC	38 KB	40	1997
Komodo [29, 28]	Xilinx FPGA	2600 LC		33	1999
aJile aJ-100 [1, 18]	ASIC 0.25μ	25 Kgates	48 KB	100	2000
Cjip [17, 26]	ASIC 0.35μ	70 Kgates	55 KB	80	2000
jHISC [31, 63]	Xilinx FPGA	15600 LC	14 KB	30	2002
FemtoJava [6]	Xilinx FPGA	2700 LC	0.5 KB	56	2003
JOP [46, 50]	Altera, Xilinx FPGA	3000 LC	4 KB	100	2003
jamuth [65]	Altera FPGA			33	2007
BlueJEP [15]	Xilinx FPGA	6900 LC	0 KB	85	2007
SHAP [72, 73]	Altera, Xilinx FPGA	5600 LC	22 KB	80	2007
aJile aJ-102/200 [2]	ASIC 0.18μ		80 KB	180	2009

systems. Most Java processors base the Java library on CLDC or a subset of it. Although the RTSJ would be a natural choice for embedded Java processors, none of the available processors support the RTSJ.

Embedded Java can be supported by a hardware implementation of the JVM – a Java processor, or by extending a RISC processor with Java specific support hardware. In the following section we provide an overview of current and most influential Java processors for embedded systems. Hardware support for low-level I/O in Java, object oriented caches, and garbage collection is described in Section 1.3. Most of the techniques for Java hardware support have been introduced in the context of Java processors. However, they can also be used to enhance a standard RISC processor, which executes compiled Java. The chapter is concluded in Section 1.4

1.2 Java Processors

In the late 90's, when Java became a popular programming language, several companies developed Java processors to speedup the execution of Java. Besides commercial processors, research on Java processor architectures was also very active at this time. With the introduction of advanced JIT compilers, the speed advantage of Java processors diminished, and many products for general purpose computing were cancelled. In the embedded domain, Java processors and coprocessors are still in use and actively developed. In this section an overview of products and research projects that survived or are still relevant are described. A description of some of the disappeared Java processors can be found in [48].

Table 1.1 lists the relevant Java processors available to date. The entries are listed in the order the processors became available. The last column presents the year where the first paper on the processor has been published. The references in the

first column are to the first published paper and to the most relevant paper for the processor.

The resource usage is given either in ASIC gates for an implementation in silicon or in logic cells (LC) when implemented in a field-programmable gate array (FPGA). The memory column gives the size of on-chip memory usage for caches and microcode store. Cache sizes are usually configurable. The column lists the default configuration. When an entry is missing, there is no information available. One design, the BlueJEP, uses Xilinx LCs for distributed RAM and therefore the LC count is high, but no dedicated on-chip memory is used. The listed resource consumptions of the processors are based on latest publications (most research projects grew in size over time). However, most FPGA based projects are configurable and the resource consumption depends on the concrete configuration.

Note, that the clock frequency does not give a direct indication of the performance of the processors, it is more an indication of the pipeline organization. E.g., the implementation of picoJava in an Altera FPGA [44] clocked at 40 MHz performs better than JOP in the same FPGA at 100 MHz.

1.2.1 picoJava

Sun introduced the first version of picoJava [38] in 1997. The processor was targeted at the embedded systems market as a pure Java processor with restricted support of C. picoJava-I contains four pipeline stages. A redesign followed in 1999, known as picoJava-II that is now freely available with a rich set of documentation [60, 61].

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.

The architecture of picoJava is a stack-based CISC processor implementing 341 different instructions [38] and is the most complex Java processor available. The processor can be implemented in about 440K gates [12]. An implementation of picoJava in an Altera FPGA was performed by Puffitsch and Schoeberl [44]. As seen in Table 1.1, picoJava is the biggest Java processor implemented in an FPGA. Nevertheless, it provides a baseline for comparison with other research processors.

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. To access memory, internal registers and for cache management, picoJava implements 115 extended instructions with 2-byte opcodes. These instructions are necessary to write system-level code to support the JVM.

Traps are generated on interrupts, exceptions, and for instruction emulation. 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

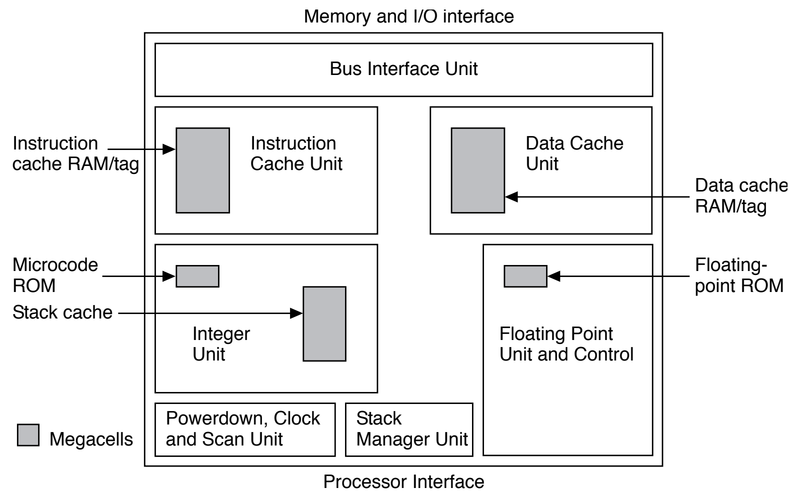

Figure 1.1: Block diagram of picoJava-II (from [60])

first instruction of the trap routine is in the instruction cache. The worst-case interrupt latency is 926 clock cycles [61]. This great variation in execution times for a trap hampers tight WCET estimates.

Figure 1.1 shows the major function units of picoJava. The integer unit decodes and executes picoJava instructions. The instruction cache is direct-mapped, while the data cache is two-way set-associative, both with a line size of 16 bytes. The caches can be configured between 0 and 16 KB. An instruction buffer decouples the instruction cache from the decode unit. The floating-point unit (FPU) is organized as a microcode engine with a 32-bit datapath, supporting single- and double-precision operations. Most single-precision operations require four cycles. Double-precision operations require four times the number of cycles as single-precision operations. For low-cost designs, the FPU can be removed and the core traps on floating-point instructions to a software routine to emulate these instructions. picoJava provides a 64-entry stack cache as a register file. The core manages this register file as a circular buffer, with a pointer to the top of stack. The stack management unit automatically performs spill to and fill from the data cache to avoid overflow and underflow of the stack buffer. To provide this functionality the register file contains five memory ports. Computation needs two read ports and one write port, the concurrent spill and fill operations need additional read and write ports. The processor core consists of following six pipeline stages:

Fetch: Fetch 8 bytes from the instruction cache or 4 bytes from the bus interface to the 16-byte-deep prefetch buffer.

Decode: Group and precode instructions (up to 7 bytes) from the prefetch buffer. Instruction folding is performed on up to four bytecodes.

Register: Read up to two operands from the register file (stack cache).

A Java statement

```
c = a + b;
```

translates to the following bytecodes:

```
iload_1
iload_2
iadd
istore_3
```

Figure 1.2: A common folding pattern that is executed in a single cycle with instruction folding

Execute: Execute simple instructions in one cycle or microcode for multi-cycle instructions.
Cache: Access the data cache.
Writeback: Write the result back into the register file.

The integer unit together with the stack unit provides a mechanism, called instruction folding, to speed up common code patterns found in stack architectures, as shown in Figure 1.2. Instruction folding is a technique to merge several stack oriented bytecode instructions into fewer RISC type instructions dynamically. When all entries are contained in the stack cache, the picoJava core can fold these four instructions to one RISC-style single cycle operation.

Instruction folding was implemented in picoJava and proposed in several research papers. However, none of the current Java processors implements instruction folding. The theoretical papers on instruciton folding ignored the complexity of the folding pattern detection and the influence on the maximum clock frequency (besides the larger chip space). Gruian and Westmijze evaluated the instruction folding by implementing the proposed algorithms in an FPGA [16]. Their result shows that the reduced cycle count due to folding is more than offset by the decreased clock frequency.

picoJava contains a simple mechanism to speed-up the common case for monitor enter and exit. The two low order bits of an object reference are used to indicate the lock holding or a request to a lock held by another thread. These bits are examined by `monitorenter` and `monitorexit`. For all other operations on the reference, these two bits are masked out by the hardware. Hardware registers cache up to two locks held by a single thread.

To efficiently implement a generational or an incremental garbage collector picoJava offers hardware support for write barriers through memory segments. The hardware checks all stores of an object reference if this reference points to a different segment (compared to the store address). In this case, a trap is generated and the garbage collector can take the appropriate action. Additional two reserved bits in the object reference can be used for a write barrier trap to support incremental collection. The hardware support for write barriers can also be used for assignment

checks of RTSJ based memory areas [21]. A combination of GC support and RTSJ assignment checks with the picoJava hardware support is presented in [22].

The distribution of picoJava does not contain a complete JVM and no Java libraries. It is expected that picoJava is just the base platform for different variants of an embedded JVM.

1.2.2 aJile's JEMCore

aJile's JEMCore is a Java processor that is available as both an IP core and a stand alone processor [1, 18]. It is based on the 32-bit JEM2 Java chip developed by Rockwell-Collins. JEM2 is an enhanced version of JEM1, created in 1997 by the Rockwell-Collins Advanced Architecture Microprocessor group. Rockwell-Collins originally developed JEM for avionics applications by adapting an existing design for a stack-based embedded processor. Rockwell-Collins decided not to sell the chip on the open market. Instead, it licensed the design exclusively to aJile Systems Inc., which was founded in 1999 by engineers from Rockwell-Collins, Centaur Technologies, Sun Microsystems, and IDT.

The core contains 24 32-bit wide registers. Six of them are used to cache the top elements of the stack. The datapath consists of a 32-bit ALU, a 32-bit barrel shifter, and support for floating point operations (disassembly/assembly, overflow and NaN detection). The control store is a 4K by 56 ROM to hold the microcode that implements the Java bytecode. An additional RAM control store can be used for custom instructions. This feature is used to implement the basic synchronization and thread scheduling routines in microcode. This results in low execution overheads with thread-to-thread yield of less than one μs (at 100 MHz). An optional Multiple JVM Manager (MJM) supports two independent, memory protected JVMs. The two JVMs execute time-sliced on the processor. According to aJile, the processor can be implemented in 25K gates (without the microcode ROM). The MJM needs additional 10K gates.

The first two silicon versions of JEM where the aJ-80 and the aJ-100. Both versions comprise a JEM2 core, the MJM, 48 KB zero wait state RAM and peripheral components, such as timer and UART. 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 aJ-100 provides a generic 8-bit, 16-bit or 32-bit external bus interface, while the aJ-80 only provides an 8-bit interface. The aJ-100 can be clocked up to 100 MHz and the aJ-80 up to 66 MHz. The power consumption is about 1mW per MHz.

The third generation of aJile's Java processor (JEMCore-II) is enhanced with a fixed-point MAC unit and a 32 KB, 2-way set-associative, unified instruction and data cache. The latest versions of the aJile processor (aJ-102 [2] and aj-200 [3]), based on the JEMCore-II, are system-on-chips, including advanced I/O devices, such as an Ethernet controller, a LCD panel interface, an USB controller, and a hardware accelerator for encryption/decryption. Both processors are implemented in 0.18μ and can be clocked up to 180 MHz. The aJ-102 is intended as a network

processor, whereas the aJ-200, with hardware support for image capturing and a media codec, targets the real-time multimedia market.

The aJile processor is intended for real-time systems with an on-chip real-time thread manager. The RTOS and all device drivers are written entirely in Java. Furthermore, aJile Systems was part of the original expert group for the RTSJ. However, the aJile runtime system does not support the RTSJ, but implements their own version of real-time threads. The aJile processor could be a reasonable platform for WCET analysis, but no information about the bytecode execution times is disclosed.

1.2.3 Komodo and jamuth

Komodo [28] 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.

Komodo's multi-threading is similar to hyper-threading in modern processors that are trying to hide latencies due to cache misses and branch misspredictions. However, this feature leads to very pessimistic WCET values if all threads are considered. For a real-time setting one thread can be given top priority in the hardware scheduler. The other threads can use the stall cycles (e.g., due to a memory access) of the real-time thread. Therefore, a single real-time thread can provide timing guarantees and the other hardware threads can be used for soft real-time tasks or for interrupt service threads. Multiple real-time threads are supported by a software based real-time scheduler.

The Java processor jamuth is the follow-up project to Komodo [65], and is targeted for commercial embedded applications with Altera FPGAs. jamuth is well integrated in the Alter's System-on-a-Programmable-Chip builder. The memory and peripheral devices are connected via the Avalon bus. The standard configuration of jamuth uses a scratchpad memory for trap routines and the garbage collector. An additional instruction cache is shared between all hardware threads.

1.2.4 Java Optimized Processor (JOP)

JOP [50] is an implementation of the JVM in hardware, especially designed for real-time systems. To support hard timing constraints, the main focus of the development of JOP has been on time-predictable bytecode execution. All function units, and especially the interactions between them, are carefully designed to avoid any time dependencies between bytecodes. This feature simplifies the low-level part of WCET analysis, a mandatory analysis for hard real-time systems.

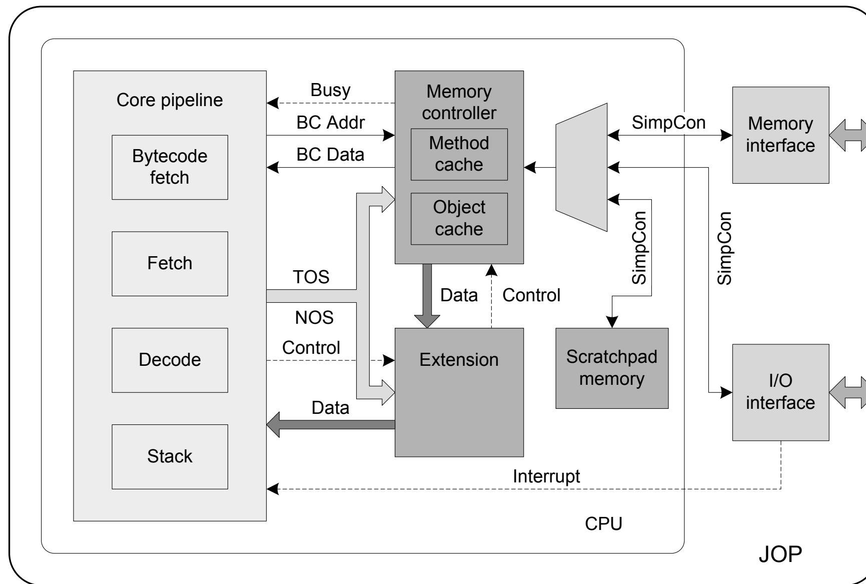

Figure 1.3: Block diagram of JOP (from [51])

Figure 1.3 shows the block diagram of JOP. The main components are: the 4-stage pipeline, the memory controller with the method cache [47] and object cache [53], the extension module for hardware accelerators, and a scratchpad memory. Main memory and I/O devices are connected via the SimpCon interface to the CPU core.

JOP dynamically translates the CISC Java bytecodes to a RISC, stack based instruction set (the microcode) that can be executed in a 3-stage pipeline. The translation takes exactly one cycle per bytecode and is therefore pipelined (adding a fourth pipeline stage). Compared to other forms of dynamic code translation, the translation in JOP does not add any variable latency to the execution time and is therefore time-predictable. Interrupts are inserted in the translation stage as special instructions and are transparent to the microcode pipeline. All microcode instructions have a constant execution time of one cycle. No stalls are possible in the microcode pipeline. Loads and stores of object fields are handled explicitly. The absence of time dependencies between bytecodes results in a simple processor model for the low-level WCET analysis [58].

JOP contains a simple execution stage with the two topmost stack elements as discrete registers. No write back stage or data forwarding logic is needed. The short pipeline (4 stages) results in short conditional branch delays and therefore helps to avoid any hard-to-analyze branch prediction logic or branch target buffer.

JOP introduced a special instruction cache, the method cache [47], which caches whole methods. With a method cache, only invoke and return bytecodes can result in a cache miss. All other bytecodes are guaranteed cache hits. The idea to cache

whole methods is based on the assumption that WCET analysis at the call graph level is more practical than performing cache analysis for each bytecode. Furthermore, loading whole methods also leads to better average case execution times for a memory with long latency but high bandwidth. The load time depends on the size of the method. However, on JOP, the cache loading is done in parallel with microcode execution in the core pipeline. Therefore, small methods do not add any additional latency to the invoke or return bytecodes. The method cache is also integrated in the embedded Java processor SHAP [42] and considered for jamuth [65] as a time-predictable caching solution.[1]

JOP has it's own notion of real-time threads that are scheduled with a preemptive, priority based scheduler. In the multi-processor configuration of JOP [40], the scheduling is partitioned. A thread is pinned to a core and each core has it's own scheduler based on a core local timer.

JOP is available in open-source under the GNU GPL. The open-source approach with JOP enabled several research projects to build upon JOP: a hardware GC implementation [13]; the BlueJEP processor architecture [15] and the first version of SHAP [43][2] are based on JOP; and JOP has been combined with the LEON processor for a real-time Agent computing platform targeting distributed satellite systems [9]. JOP is also in use in several industrial applications [49, 35]. Furthermore, the WCET friendly architecture of JOP enabled development of several WCET analysis tools for Java [19, 7, 23]. The WCET analysis tool WCA is part of the JOP source distribution [58].

1.2.5 BlueJEP

BlueJEP is a Java processor specified in Bluespec System Verilog (BSV) to evaluate BSV for system design [15]. BlueJEP's design starting point was the architecture of JOP [50]. With respect to the Java build process and microcode, BlueJEP is JOP compatible. However, the pipeline has a different structure. Six stages are connected via searchable FIFOs and are as follows: fetch bytecode, fetch microcode, decode and fetch register, fetch stack, execute, and write back. The last stages contain a forwarding network. Quite unconventional is the bypass option where the execution stage can be skipped by individual instructions. BlueJEP can be configured to use a memory management unit that performs mark-compact based garbage collection in hardware [14].

[1] Personal communication with Sascha Uhrig.

[2] The similarity can be found when comparing the microcode instructions of JOP and SHAP. The microcode instructions of SHAP are described in the appendix of the technical report [43].

1.2.6 Java accelerators

Another approach to speedup Java programs in an embedded system is to enhance a RISC core with a Java accelerator. The main idea is to support (legacy) C code *and* Java code in a single chip. The accelerator can work as translation unit or as a Java coprocessor. The translation unit substitutes the switch statement of an interpreting JVM (bytecode decoding) through hardware and/or translates simple bytecodes to a sequence of RISC instructions on the fly. A coprocessor is placed in the instruction fetch path of the main processor and translates Java bytecodes to sequences of instructions for the host CPU or directly executes basic Java bytecodes. The complex instructions are emulated by the main processor.

Nozomi's JA108 [36], previously known as JSTAR, Java accelerator sits between the native processor and the memory subsystem. JA108 fetches Java bytecodes from memory and translates them into native microprocessor instructions. JA108 acts as a pass-through when the core processor's native instructions are being executed. The JA108 is targeted for use in mobile phones to increase performance of Java multimedia applications. The coprocessor is available as standalone package or with included memory and can be operated up to 104 MHz. The resource usage for the JSTAR is known to be about 30K gates plus 45 Kbits for the microcode.

Jazelle [4] is an extension of the ARM 32-bit RISC processor, similar to the Thumb state (a 16-bit mode for reduced memory consumption). A new ARM instruction puts the processor into Java state. Bytecodes are fetched and decoded in two stages, compared to a single stage in ARM state. Four registers of the ARM core are used to cache the top stack elements. Stack spill and fill is handled automatically by the hardware. Additional registers are reused for the Java stack pointer, the variable pointer, the constant pool pointer, and locale variable 0 (the `this` pointer in methods). Keeping the complete state of the Java mode in ARM registers simplifies its integration into existing operating systems. The Jazelle coprocessor is integrated into the same chip as the ARM processor. The hardware bytecode decoder logic is implemented in less than 12K gates. It accelerates, according to ARM, some 95% of the executed bytecodes. 140 bytecodes are executed directly in hardware, while the remaining 94 are emulated by sequences of ARM instructions. This solution also uses code modification with `quick` instructions to substitute certain object-related instructions after link resolution. All Java bytecodes, including the emulated sequences, are re-startable to enable a fast interrupt response time.

The Cjip processor [17, 26] 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 of Cjip 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 [25]. 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.

1.2.7 Further Java processor projects

Several past and current research projects address execution of Java applications in hardware. This section briefly introduces those projects.

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 jHISC project proposes a high-level instruction set architecture for Java [63]. The processor consumes 15600 LCs and the maximum frequency in a Xilinx Virtex FPGA is 30 MHz. The prototype can only run simple programs and the performance is estimated with a simulation.

The SHAP Java processor [72] contains a memory management unit for hardware assisted garbage collection. An additional RISC processor, the open-source processor ZPU, is integrated with SHAP and performs the GC work [41].

The research project JPOR [10] aims for a Java processor design that is optimized for the execution of RTSJ programs. It is a typical implementation of a Java processor combining direct support for simple bytecodes and microcode instructions for more complex bytecodes.

A Java processor (probably JOP) has been extended to support dual issue of bytecodes [27]. To simplify the runtime on the Java processor, the system is extended with a RISC processor [59]. The interface between the RISC processor and the Java processor is based on cross-core interrupts. A few micro benchmarks provide a comparison of the Java processor with an interpreting JVM (Sun's CVM).

1.2.8 Chip-multiprocessors

Several of the active research projects on Java processors explore chip-multiprocessor (CMP) configurations. The main focus of the CMP version of JOP [39, 40] is to keep the CMP time-predictable, even with access to shared main memory. Pitter implemented a TDMA based memory arbiter and integrated the timing model of the arbiter into the WCET analysis tool for JOP. Therefore, the JOP CMP is the first time-predictable CMP that includes a WCET analysis tool.

The multi-threaded jamuth processor uses the Avalon switch fabric to build a CMP system of a small number of cores [64]. As the jamuth core is already multi-threaded, the effective supported concurrent threads if 4 times the number of cores. The evaluation shows that increasing the number of cores provides a better per-

formance gain than increasing the number of threads in one core. For the jamuth system, the optimum number of threads in one core is two. The second thread can utilize the pipeline when the first thread stalls in a memory access. Adding a third thread results only in a minor performance gain.

The CMP version of SHAP contains a pipelined memory arbiter and controller to optimize average case performance [73]. By using a pipelined, synchronous SRAM as main memory, the memory delivers one word per clock cycle. On a CMP system the bottleneck is the memory bandwidth and not the memory latency. Therefore, this pipelined memory delivers the needed memory bandwidth for the SHAP CMP system.

With true concurrence on a CMP the pressure on efficient synchronization primitives increases. A promising approach to simplify synchronization at the language level and provide more true execution concurrency is transactional memory (TM) [20]. A real-time TM (RTTM) has been implemented in the CMP version of JOP [55]. The Java annotation `@atomic` on a method is used to mark the transaction. The code within the transaction is executed concurrent to other possible transactions. All writes to the memory are kept in a core local transaction buffer during the transaction. At the end of the transaction that buffer is written atomically to main memory. When a conflict between transactions occurs, a transaction is aborted and the atomic section is retried. For real-time systems this retry count must be bounded [54]. To enable this bound, RTTM is designed to avoid any false conflict detections that can occur on a cache-based hardware TM.

1.2.9 Discussion

The basic architecture of all Java processors is quite similar: simple bytecodes are supported in hardware and more complex bytecodes are implemented in microcode or even in Java. JVM bytecode is stack oriented and almost all instructions operate on the stack. Therefore, all Java processor provide a dedicated cache for stack content. The main difference between the architectures is the amount of bytecodes that are implemented in hardware and the caching system for instructions and heap allocated data. As the market for embedded Java processors is still small, only the JEMCore is available in silicon. picoJava is discontinued by Sun/Oracle and the other processor projects target FPGAs.

Standard Java is too big for embedded systems. The subset defined in the CLDC is a starting point for the Java library support of embedded Java. JEMCore, jamuth, JOP, and SHAP support the CLDC. As the CLDC is missing an API for low-level I/O, all vendors provide their own Java classes or mechanism for low-level I/O (see next section). CLDC is based on Java 1.1 and is missing important features, such as collection classes. Therefore, most projects add some classes from standard Java to the CLDC base. The results in libraries somewhere between CLDC and standard Java, without being compatible. A new library specification for (classic) embedded systems, based on the current version of Java, would improve this situation.

Most processors (JEMCore, jamuth, and JOP) target real-time systems. They provide a real-time scheduler with tighter guarantees than a standard JVM thread scheduler. As this real-time scheduling does not fit well with standard Java threads, special classes for (periodic) real-time threads are introduced. None of the runtime systems supports the RTSJ API. We assume that the RTSJ is too complex for such resource constraint devices. It might also be the case that the effort to implement a RTSJ JVM is too high for the small development teams behind the processor projects. However, with the simpler SCJ specification there is hope that the embedded Java processors will adapt their runtime to a common API.

For real-time systems the WCET needs to be known as input for schedulability analysis. WCET is usually derived by static program analysis. As the presented processor architectures are relative simple (e.g., no dynamic scheduling in the pipeline), they should be an easy target for WCET analysis. However, only for three processors the execution time of bytecodes is documented (picoJava, Cjip, and JOP). JOP is the only processor that is supported by WCET analysis tools.

1.3 Support Technology for Java

Several techniques to support the execution of Java programs have been proposed and implemented. Most techniques have been introduced in the context of a Java processor. However, the basic concepts can also be applied to a RISC processor to enhance execution performance or time predictability of Java applications. In this section we review support for low-level IO and interrupts, special cache organizations for heap allocated data, and support for garbage collection.

1.3.1 Low-level I/O and interrupts

Java, as a platform independent language and runtime system, does not support direct access to low-level I/O devices. However, embedded Java systems often consist only of a JVM without an underlying operations system. In that case, the JVM acts as the operating system. The different solutions to interface I/O devices and implement interrupt handlers directly in Java on Java processors and embedded JVMs are described in the following section.

The RTSJ defines an API for direct access to physical memory, including hardware registers. Essentially one uses `RawMemoryAccess` at the level of primitive data types. Although the solution is efficient, this representation of physical memory is not object oriented. A type-safe layer with support for representing individual registers can be implemented on top of the RTSJ API. The topic of RTSJ support for low-level I/O is discussed in detail in Chapter **??**.

The RTSJ specification suggests that asynchronous events are used for interrupt handling. Yet, it neither specifies an API for interrupt control nor semantics of the

handlers. Second level interrupt handling can be implemented within the RTSJ with an `AsyncEvent` that is bound to a *happening*. The happening is a string constant that represents an interrupt, but the meaning is implementation dependent.

The aJile Java processor [1] uses native functions to access devices. Interrupts are handled by registering a handler for an interrupt source (e.g., a GPIO pin). Systronix suggests[3] to keep the handler short, as it runs with interrupts disabled, and delegate the real handling to a thread. The thread waits on an object with ceiling priority set to the interrupt priority. The handler just notifies the waiting thread through this monitor. When the thread is unblocked and holds the monitor, effectively all interrupts are disabled.

On top of the multiprocessing pipeline of Komodo [28] the concept of interrupt service threads is implemented. For each interrupt one thread slot is reserved for the interrupt service thread. It is unblocked by the signaling unit when an interrupt occurs. A dedicated thread slot on a fine-grain multithreading processor results in a very short latency for the interrupt service routine. No thread state needs to be saved. However, this comes at the cost to store the complete state for the interrupt service thread in the hardware. In the case of Komodo, the state consists of an instruction window and the on-chip stack memory. Devices are represented by Komodo specific I/O classes.

One option to access I/O registers directly in an embedded JVM is to access them via C functions using the Java native interface. Another option is to use so called hardware objects [56], which represent I/O devices as plain Java objects. The hardware objects are platform specific (as I/O devices are), but the mechanism to represent I/O devices as Java objects can be implemented in any JVM. Hardware objects have been implemented so far in six different JVMs: CACAO, OVM, SimpleRTJ, Kaffe, HVM, and JOP. Three of them are running on a standard PC, two on a microcontroller, and one is a Java processor.

In summary, access to device registers is handled in both aJile and Komodo by abstracting them into library classes with access methods. This leaves the implementation to the particular JVM and does not give the option of programming them at the Java level. Exposing hardware devices as Java objects, as implemented with the hardware objects, allows safe access to device registers directly from Java. Interrupt handling in aJile is essentially first level, but with the twist that it may be interpreted as RTSJ event handling, although the firing mechanism is atypical. Komodo has a solution with first level handling through a full context shift.

1.3.2 Cache organizations for Java

Java releys heavily on objects allocated on the heap and the automatic memory management with a garbage collector. The resulting data usage patterns are different

[3] A template can be found at `http://practicalembeddedjava.com/tutorials/aJileISR.html`

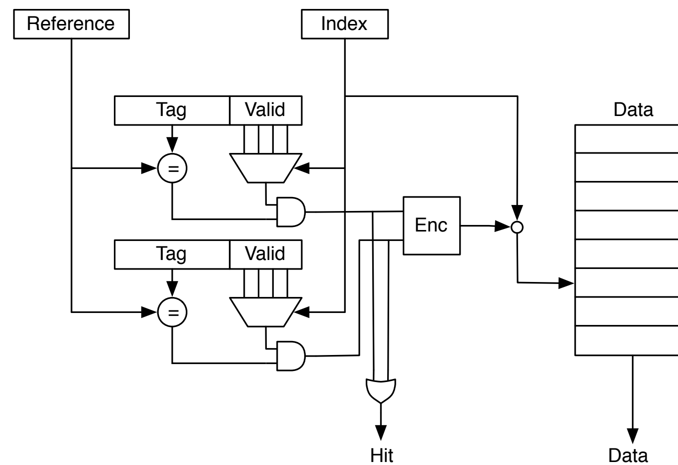

Figure 1.4: Object cache with associativity of two and four fields per object (from [47])

from e.g., C programs. Therefore, a system optimized for the execution of Java programs can benefit from a specialized cache, an object cache.

Figure 1.4 shows one possible organization of an object cache. The cache is accessed via the object reference and the field index. In this example figure the associativity is two and each cache line is four fields long. All tag memories are compared in parallel with the object reference. Parallel to the tag comparison, the valid bits for the individual fields are checked. The field index performs the selection of the valid bit multiplexer. The output of the tag comparisons and valid bit selection is fed into the encoder, which delivers the selected cache line. The line index and the field index are concatenated and build the address of the data cache.

One of the first proposals of an object cache [68] appeared within the Mushroom project [69]. The Mushroom project investigated hardware support for Smalltalk-like object oriented systems. The cache is indexed by a combination of the object identifier (the handle in the Java world) and the field offset. Different combinations, including xoring of the two fields, are explored to optimize the hit rate. The most effective generation of the hash function for the cache index was the xor of the upper offset bits (the lower bits are used to select the word in the cache line) with the lower object identifier bits. Considering only the hit rate, caches with a block size of 32 and 64 bytes perform best. However, under the assumption of realistic miss penalties caches with 16 and 32 bytes lines size result in lower average access times per field access.

With an indirection based access to an object two data structures need to be cached: the actual object and the indirection to that object. In [68], a common cache for both data structures and a split cache are investigated. As the handle indirection cache is only accessed when the object cache results in a miss, a medium hit rate on

the handle indirection cache is sufficient. Therefore, the best configuration is a large object cache and a small handle indirection cache.

Object oriented architecture support for a Java processor is proposed in [67], including an object cache, extended folding, and a virtual dispatch cache. The object cache is indexed by (part of) the object reference and the field offset. The virtual method cache is motivated by avoiding a virtual dispatch table (a very common implementation approach in OO languages) to save memory for embedded devices. This cache assumes monomorphic call sites. The cache is indexed by some lower bits of the PC at the call site. As polymorphic call sites trash the cache, an extension as a hybrid polymorphic cache is proposed. The whole proposal is based on a very high level simulation – running some Java applications in a modified, interpreting JVM (Sun JDK 1.0.2). No estimates on the hardware complexity are given.

A dedicated cache for heap allocated data is proposed in [66]. The object layout is handle based. The object reference with the field index is used to address the cache – it is called virtual address object cache. Cache configurations are evaluated with a simulation in a Java interpreter and the assumption of 10 ns cycle time of the Java processor and a memory latency of 70 ns. For different cache configurations (up to 32 KB) average case field access times between 1.5 and 5 cycles are reported. For most benchmarks, the optimal block size was found to be 64 bytes, which is quite high for the medium latency (7 cycles) of the memory system. The proposed object cache is also used to cache arrays. Therefore, the array accesses favor a larger block size to benefit from spatial locality.

Wright et al. propose a cache that can be used as object cache and as conventional data cache [71]. To support the object cache mode the instruction set is extended with a few object-oriented instructions such as load and store of object fields. The object layout is handle based and the cache line is addressed with a combination of the object reference (called object id) and part of the offset within the object. The main motivation of the object cache mode is in-cache garbage collection [70]. The youngest generation of objects in a generational GC is allocated in the object cache and can be collected without main memory access.

The possible distinction between different data areas of the JVM (e.g., constant pool, heap, method area) enables unique cache configurations for a Java processor. It is argued that a split-cache design can simplify WCET analysis of data caches [52]. For heap allocated objects a time-predictable object cache has been implemented in JOP [53]. The object cache is organized to cache single objects in a cache line. The cache is highly associative to track object field accesses in the WCET analysis via the symbolic references instead of the actual addresses. WCET analysis based evaluation of the object cache shows that even a small cache with a high associativity provides good hit rate analyzability [24].

1.3.3 Garbage collection

A real-time garbage collector has to fulfill two basic properties: ensure that programs with bounded allocation rates do not run out of memory and provide short blocking times. Even for incremental garbage collectors, heap compaction can be source of considerable blocking time. Chapter **??** discusses real-time garbage collection in detailed. Here the focus is on hardware support for garbage collection

Nielsen and Schmidt [37] propose hardware support, the object-space manager (OSM), for real-time garbage collector on a standard RISC processor. The concurrent garbage collector is based on [5], but the concurrency is of finer grain than the original Baker algorithm as it allows the mutator to continue during the object copy. The OSM redirects field access to the correct location for an object that is currently being copied. [45] extends the OSM to a GC memory module where a local microprocessor performs the GC work. In the paper the performance of standard C++ dynamic memory management is compared against garbage collected C++. The authors conclude that C++ with the hardware supported garbage collection performs comparable with traditional C++.

One argument against hardware support for GC might be that standard processors will never include GC specific instructions. However, Azul Systems has included a read barrier in their RISC based chip-multiprocessor system [11]. The read barrier looks like a standard load instruction, but tests the TLB if a page is a GC-protected page. GC-protected pages contain objects that are already moved. The read barrier instruction is executed after a reference load. If the reference points into a GC-protected page a user-mode trap handler corrects the stale reference to the forwarded reference.

Meyer proposes in [32, 33] a new RISC processor architecture for exact pointers. The processor contains a set of pointer registers and a set of data registers. The instruction set guarantees correct handling of pointers. Furthermore, the object layout and the stack are both split to pointer containing and data containing regions (similar to the split stack). A microprogrammed GC unit is attached to the main processor [34]. Close interaction between the RISC pipeline and the GC coprocessor allow the redirection for field access in the correct semi-space with a concurrent object copy. The hardware cost of this feature is given as an additional word for the back-link in every pointer register and every attribute cache line. It is not explicitly described in the paper when the GC coprocessor performs the object copy. We assume that the memory copy is performed in parallel with the execution of the RISC pipeline. In that case, the GC unit *steals* memory bandwidth from the application thread. The GC hardware uses an implementation of Baker's read-barrier [5] for the incremental copying algorithm. The cost of the read-barrier is between 5 and 50 clock cycles. The resulting minimum mutator utilization for a time quantum of 1 ms was measured to be 55%. For a real-time task with a period of 1 kHz the resulting overhead is about a factor of 2.

The Java processor SHAP [72] contains a memory management unit with a hardware garbage collector. That unit redirects field and array access during a copy operation of the GC unit.

During heap compaction, objects are copied. Copying is usually performed atomically to avoid interference with application threads, which could render the state of an object inconsistent. Copying of large objects and especially large arrays introduces long blocking times that are unacceptable for real-time systems. In [57] an interruptible copy unit is presented that implements non-blocking object copy. The unit can be interrupted after a single word move. The evaluation showed that it is possible to run high priority hard real-time tasks at 10 kHz parallel to the garbage collection task on a 100 MHz Java processor.

1.4 Conclusions

To enable Java in resource constraint embedded systems, several projects implement the Java virtual machine (JVM) in hardware. These Java processors are faster than an interpreting JVM, but use fewer resources than a JIT compiler. Furthermore, the direct implementation of the JVM enables WCET analysis at bytecode level.

Java also triggered research on hardware support for object-oriented languages. Special forms of caches, sometimes called object cache, are optimized for the data access pattern of the JVM. Garbage collection is an important feature of Java. As the overhead of GC, especially for an incremental real-time GC, can be quite high on standard processors, several mechanisms for hardware support of GC have been proposed.

So far, Java processors failed in the standard and server computing domain. However, with the Azul system, hardware support for Java within a RISC processor has now entered the server domain. It will be interesting to see whether hardware support for Java and other managed languages will be included in mainstream processor architectures.

With the introduction of the safety-critical Java specification the interest in Java processors might increase. Java processors execute bytecode, which itself is easier to analyze. Certification of safety-critical applications will benefit from the direct execution of Java bytecode as the additional translation steps to C and machine code are avoided.

References

1. aJile Systems. aj-100 real-time low power Java processor. preliminary data sheet, 2000.
2. aJile Systems. aj-102 technical reference manual v2.4. Available at `http://www.ajile.com/`, 2009.
3. aJile Systems. aj-200 technical reference manual v2.1. Available at `http://www.ajile.com/`, 2010.
4. ARM. Jazelle technology: ARM acceleration technology for the Java platform. white paper, 2004.
5. H.G. Baker. List processing in real time on a serial computer. *Commun. ACM*, 21(4):280–294, 1978.
6. A. C. Beck and L. Carro. Low power Java processor for embedded applications. In *Proceedings of the 12th IFIP International Conference on Very Large Scale Integration*, pages 213–228, Darmstadt, Germany, December 2003.
7. T. Bogholm, H. Kragh-Hansen, P. Olsen, B. Thomsen, and K.G. Larsen. Model-based schedulability analysis of safety critical hard real-time Java programs. In *Proceedings of the 6th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2008)*, pages 106–114, New York, NY, USA, 2008. ACM.
8. G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, and M. Turnbull. *The Real-Time Specification for Java*. Java Series. Addison-Wesley, June 2000.
9. C.P. Bridges and T. Vladimirova. Agent computing applications in distributed satellite systems. In *International Symposium on Autonomous Decentralized Systems, 2009. ISADS '09*, pages 1 –8, march 2009.
10. Z. Chai, W. Zhao, and W. Xu. Real-time Java processor optimized for RTSJ. In *Proceedings of the 2007 ACM symposium on Applied computing*, SAC '07, pages 1540–1544, New York, NY, USA, 2007. ACM.
11. C. Click, G. Tene, and M. Wolf. The pauseless GC algorithm. In Michael Hind and Jan Vitek, editors, *Proceedings of the 1st International Conference on Virtual Execution Environments, VEE 2005, Chicago, IL, USA, June 11-12, 2005*, pages 46–56. ACM, 2005.
12. 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.
13. F. Gruian and Z. Salcic. Designing a concurrent hardware garbage collector for small embedded systems. In *Proceedings of Advances in Computer Systems Architecture: 10th Asia-Pacific Conference, ACSAC 2005*, pages 281–294. Springer-Verlag GmbH, October 2005.
14. F. Gruian and M. Westmijze. Bluejamm: A bluespec embedded Java architecture with memory management. In *SYNASC '07: Proceedings of the Ninth International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 459–466, Washington, DC, USA, 2007. IEEE Computer Society.

15. F. Gruian and M. Westmijze. Bluejep: a flexible and high-performance Java embedded processor. In *JTRES '07: Proceedings of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems*, pages 222–229, New York, NY, USA, 2007. ACM.
16. F. Gruian and M. Westmijze. Investigating hardware micro-instruction folding in a Java embedded processor. In *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '10, pages 102–108, New York, NY, USA, 2010. ACM.
17. T.R. Halfhill. Imsys hedges bets on Java. *Microprocessor Report*, August 2000.
18. D.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.
19. T. Harmon. *Interactive Worst-case Execution Time Analysis of Hard Real-time Systems*. PhD thesis, University of California, Irvine, 2009.
20. M. Herlihy and J.E.B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture, 1993*, pages 289–300, 1993.
21. M.T. Higuera-Toledano. Hardware-based solution detecting illegal references in real-time Java. In *Proceedings. 15th Euromicro Conference on Real-Time Systems (ECRTS 2003)*, pages 229–337, july 2003.
22. M.T. Higuera-Toledano. Hardware support for detecting illegal references in a multiapplication real-time Java environment. *ACM Trans. Embed. Comput. Syst.*, 5:753–772, November 2006.
23. B. Huber. Worst-case execution time analysis for real-time Java. Master's thesis, Vienna University of Technology, Austria, 2009.
24. B. Huber, W. Puffitsch, and M. Schoeberl. Worst-case execution time analysis driven object cache design. *Concurrency and Computation: Practice and Experience*, doi: 10.1002/cpe.1763, 2011.
25. Imsys. ISAJ reference 2.0, January 2001.
26. Imsys. Im1101c (the Cjip) technical reference manual / v0.25, 2004.
27. H-J Ko and C-J Tsai. A double-issue Java processor design for embedded applications. In *IEEE International Symposium on Circuits and Systems, 2007. ISCAS 2007*, pages 3502 – 3505, May 2007.
28. J. Kreuzinger, U. Brinkschulte, M. Pfeffer, S. Uhrig, and T. Ungerer. Real-time event-handling and scheduling on a multithreaded Java microcontroller. *Microprocessors and Microsystems*, 27(1):19–31, 2003.
29. J. Kreuzinger, R. Marston, T. Ungerer, U. Brinkschulte, and C. Krakowski. The komodo project: thread-based event handling supported by a multithreaded Java microcontroller. In *EUROMICRO Conference, 1999. Proceedings. 25th*, volume 2, pages 122 –128 vol.2, 1999.
30. D. Locke, B.S. Andersen, B. Brosgol, M. Fulton, T. Henties, J.J. Hunt, J.O. Nielsen, K. Nilsen, M. Schoeberl, J. Tokar, J. Vitek, and A.J. Wellings. Safety-critical Java technology specification, public draft. Available at `http://www.jcp.org/en/jsr/detail?id=302`, 2011.
31. M.P. Lun and A.S. Fong. Introducing pipelining technique in an object-oriented processor. In *TENCON '02. Proceedings. 2002 IEEE Region 10 Conference on Computers, Communications, Control and Power Engineering*, volume 1, pages 301 – 305 vol.1, oct 2002.
32. M. Meyer. A novel processor architecture with exact tag-free pointers. In *2nd Workshop on Application Specific Processors*, pages 96–103, San Diego, CA, 2003.
33. M. Meyer. A novel processor architecture with exact tag-free pointers. *IEEE Micro*, 24(3):46–55, 2004.
34. M. Meyer. An on-chip garbage collection coprocessor for embedded real-time systems. In *RTCSA '05: Proceedings of the 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'05)*, pages 517–524, Washington, DC, USA, 2005. IEEE Computer Society.
35. G. Michel and J. Sachtleben. An integrated gyrotron controller. *Fusion Engineering and Design*, In Press, Corrected Proof:–, 2011.

36. Nazomi. JA 108 product brief. Available at `http://www.nazomi.com`.
37. K. Nilsen and W.J. Schmidt. Cost-effective object space management for hardware-assisted real-time garbage collection. *ACM Letters on Programming Languages and Systems*, 1(4):338–354, December 1992.
38. J. M. O'Connor and M. Tremblay. picoJava-I: The Java virtual machine in hardware. *IEEE Micro*, 17(2):45–53, 1997.
39. C. Pitter and M. Schoeberl. Towards a Java multiprocessor. In *Proceedings of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2007)*, pages 144–151, Vienna, Austria, September 2007. ACM Press.
40. C. Pitter and M. Schoeberl. A real-time Java chip-multiprocessor. *ACM Trans. Embed. Comput. Syst.*, 10(1):9:1–34, 2010.
41. T. B. Preusser, P. Reichel, and R.G. Spallek. An embedded GC module with support for multiple mutators and weak references. In Christian Müller-Schloer, Wolfgang Karl, and Sami Yehia, editors, *Architecture of Computing Systems - ARCS 2010, 23rd International Conference, Hannover, Germany, February 22-25, 2010. Proceedings*, volume 5974 of *Lecture Notes in Computer Science*, pages 25–36. Springer, 2010.
42. T. B. Preusser, M. Zabel, and R.G. Spallek. Bump-pointer method caching for embedded Java processors. In *Proceedings of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2007)*, pages 206–210, New York, NY, USA, 2007. ACM.
43. T.B. Preusser, M. Zabel, and P. Reichel. The SHAP microarchitecture and Java virtual machine. Technical Report TUD-FI07-02, Fakultaet Informatik, TU Dresden, April 2007.
44. W. Puffitsch and M. Schoeberl. picoJava-II in an FPGA. In *Proceedings of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2007)*, pages 213–221, Vienna, Austria, September 2007. ACM Press.
45. W.J. Schmidt and K. Nilsen. Performance of a hardware-assisted real-time garbage collector. In *ASPLOS-VI: Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*, pages 76–85, New York, NY, USA, 1994. ACM Press.
46. M. Schoeberl. JOP: A Java optimized processor. In *On the Move to Meaningful Internet Systems 2003: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2003)*, volume 2889 of *LNCS*, pages 346–359, Catania, Italy, November 2003. Springer.
47. M. Schoeberl. A time predictable instruction cache for a Java processor. In *On the Move to Meaningful Internet Systems 2004: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2004)*, volume 3292 of *LNCS*, pages 371–382, Agia Napa, Cyprus, October 2004. Springer.
48. M. Schoeberl. *JOP: A Java Optimized Processor for Embedded Real-Time Systems*. PhD thesis, Vienna University of Technology, 2005.
49. M. Schoeberl. Application experiences with a real-time Java processor. In *Proceedings of the 17th IFAC World Congress*, pages 9320–9325, Seoul, Korea, July 2008.
50. M. Schoeberl. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture*, 54/1–2:265–286, 2008.
51. M. Schoeberl. *JOP Reference Handbook: Building Embedded Systems with a Java Processor*. CreateSpace, August 2009. Available at `http://www.jopdesign.com/doc/handbook.pdf`.
52. M. Schoeberl. Time-predictable cache organization. In *Proceedings of the First International Workshop on Software Technologies for Future Dependable Distributed Systems (STFSSD 2009)*, pages 11–16, Tokyo, Japan, March 2009. IEEE Computer Society.
53. M. Schoeberl. A time-predictable object cache. In *Proceedings of the 14th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2011)*, pages 99–105, Newport Beach, CA, USA, March 2011. IEEE Computer Society.
54. M. Schoeberl, F. Brandner, and J. Vitek. RTTM: Real-time transactional memory. In *Proceedings of the 25th ACM Symposium on Applied Computing (SAC 2010)*, pages 326–333, Sierre, Switzerland, March 2010. ACM Press.

55. M. Schoeberl and P. Hilber. Design and implementation of real-time transactional memory. In *Proceedings of the 20th International Conference on Field Programmable Logic and Applications (FPL 2010)*, pages 279–284, Milano, Italy, August 2010. IEEE Computer Society.
56. M. Schoeberl, S. Korsholm, T. Kalibera, and A. P. Ravn. A hardware abstraction layer in Java. *ACM Trans. Embed. Comput. Syst.*, accepted, 2010.
57. M. Schoeberl and W. Puffitsch. Non-blocking real-time garbage collection. *ACM Trans. Embedded Comput. Syst.*, 10(1), 2010.
58. M. Schoeberl, W. Puffitsch, R.U. Pedersen, and B. Huber. Worst-case execution time analysis for a Java processor. *Software: Practice and Experience*, 40/6:507–542, 2010.
59. K.N. Su and C.J. Tsai. Fast host service interface design for embedded Java application processor. In *IEEE International Symposium on Circuits and Systems, 2009. ISCAS 2009*, pages 1357 –1360, May 2009.
60. Sun. *picoJava-II Microarchitecture Guide*. Sun Microsystems, March 1999.
61. Sun. *picoJava-II Programmer's Reference Manual*. Sun Microsystems, March 1999.
62. Sun Microsystems. Connected limited device configuration 1.1. Available at `http://jcp.org/aboutJava/communityprocess/final/jsr139/`, March 2003.
63. Y.Y. Tan, C.H. Yau, K.M. Lo, W.S. Yu, P.L. Mok, and A.S. Fong. Design and implementation of a Java processor. *Computers and Digital Techniques, IEE Proceedings-*, 153:20–30, 2006.
64. S. Uhrig. Evaluation of different multithreaded and multicore processor configurations for soPC. In Koen Bertels, Nikitas J. Dimopoulos, Cristina Silvano, and Stephan Wong, editors, *Embedded Computer Systems: Architectures, Modeling, and Simulation, 9th International Workshop, SAMOS*, volume 5657 of *Lecture Notes in Computer Science*, pages 68–77. Springer, 2009.
65. S. Uhrig and J. Wiese. jamuth: an IP processor core for embedded Java real-time systems. In *Proceedings of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2007)*, pages 230–237, New York, NY, USA, 2007. ACM Press.
66. N. Vijaykrishnan and N. Ranganathan. Supporting object accesses in a Java processor. *Computers and Digital Techniques, IEE Proceedings-*, 147(6):435–443, 2000.
67. N. Vijaykrishnan, N. Ranganathan, and R. Gadekarla. Object-oriented architectural support for a Java processor. In Eric Jul, editor, *ECOOP*, volume 1445 of *Lecture Notes in Computer Science*, pages 330–354. Springer, 1998.
68. I. Williams and M. Wolczko. An object-based memory architecture. In *Proceedings of the Fourth International Workshop on Persistent Object Systems*, pages 114–130, Martha's Vineyard, MA (USA), September 1990.
69. I.W. Williams. *Object-Based Memory Architecture*. PhD thesis, Department of Computer Science, University of Manchester, 1989.
70. G. Wright, M.L. Seidl, and M. Wolczko. An object-aware memory architecture. Technical Report SML–TR–2005–143, Sun Microsystems Laboratories, February 2005.
71. G. Wright, M.L. Seidl, and M. Wolczko. An object-aware memory architecture. *Sci. Comput. Program*, 62(2):145–163, 2006.
72. M. Zabel, T.B. Preusser, P. Reichel, and R. G. Spallek. Secure, real-time and multi-threaded general-purpose embedded Java microarchitecture. In *Prceedings of the 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD 2007)*, pages 59–62, Lübeck, Germany, Aug. 2007.
73. M. Zabel and R.G. Spallek. Application requirements and efficiency of embedded Java bytecode multi-cores. In *JTRES '10: Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, pages 46–52, New York, NY, USA, 2010. ACM.