JOP: A Java Optimized Processor

Martin Schoeberl

JOP.design, Strausseng. 2-10/2/55, A-1050 Vienna, AUSTRIA
martin@jopdesign.com

Abstract. Java is still not a common language for embedded systems. It posses language features, like thread support, that can improve embedded system development, but common implementations as interpreter or just-in-time compiler are not practical. JOP is a hardware implementation of the Java Virtual Machine with focus on real-time applications. This paper describes the architecture of JOP and proposes a simple real-time extension of Java for JOP. First application in an industrial system showed that JOP is one way to use Java in the embedded world.

1 Introduction

Current software design practice for embedded systems is still archaic compared to software development for desktop systems. C and even Assembler is used on top of a small RTOS. The variety of embedded operating systems is large and this fragmentation of the market leads to high cost. Java [1] can be a way out of this dilemma and possess language features not found in C:

- Object-oriented
- Memory management with a garbage collector
- Implicit memory protection
- Threads

Memory management and threads are (besides device drivers) the main components of embedded operating systems. Finding these features in the language embedded systems can be programmed in Java without the need of an operating system.

Java on desktop systems comes with a large library. However, if Java is stripped down to the core components it has a very small memory footprint. With careful programming (like using only immortal memory as in [2]) the garbage collector can be avoided. Without a GC, Java can be used even in hard real-time systems.

The definition of the language includes also the definition of the binary and the Java Virtual Machine (JVM) [3] to execute these programs. The JVM is a stack machine and can be implemented in several ways:

Interpreter: A simple solution with low memory requirements, but lacks in performance.

Just-in-Time Compilation: Got very popular on desktop systems, but has two main disadvantages in embedded systems: A compiler is necessary on the target and due to compilation during runtime execution times are not predictable.

Batch Compilation: Java can be compiled in advance to the native instruction set of the target. Dynamic loading of classes is no longer possible (not a main concern in embedded systems).

Hardware Implementation: A Java Processor with JVM bytecodes as native instruction set.

JOP is a hardware implementation of the JVM targeted for small embedded systems with real-time constraints. It shall help to increase the acceptance of Java for those systems.

JOP is implemented as a soft core in an FPGA (Field Programmable Gate Array). Using an FPGA as processor for embedded systems is uncommon due to high cost compared to a micro controller. However, if the core is small enough, unused FPGA resources can be used to implement periphery in the FPGA resulting in a lower chip count and hence lower overall cost.

The main features of JOP are summarized below:

- Fast execution of Java bytecodes without JIT-Compiler.
- Predictable execution time of Java bytecodes.
- Small core that fits in a low cost FPGA.
- Configurable resource usage through HW/SW co-design.
- Flexibility for embedded systems through FPGA implementation.

2 Architecture

Every design is influenced by the available tools. In architecture, the constraints will be different whether we use wood, stone or steel. The same is true for CPU architecture. The first and primary implementation of JOP is in an FPGA.

2.1 FPGA Implementation

An FPGA has two basic building blocks: logic elements and memory. A logic element (LE) consists of a 4-bit LUT (Look Up Table) and a flip-flop. Memory blocks (ESB) are usually small (e.g. 0.5 KB) with independent read and write ports of configurable size. With these constraints, a stack machine is an attractive architecture in an FPGA:

- The stack can be implemented in internal memory.
- A register file in a RISC CPU needs two read ports and one write port for single cycle instructions. A stack needs only one read and one write port (common in current FPGAs).
- Instruction set is simpler and can be reduced to 8 bit.
- No data forwarding is necessary.

2.2 Micro Code

There is a great variation in complexity of Java bytecodes, the instructions of the JVM. There are simple instructions like arithmetic and logic operations on the stack. However, the semantics of instructions like *new* or *invokestatic* can result in class loading and verification. Because of this variation, not every JVM instruction can be implemented in hardware. One common solution, used in Suns picoJava-II [5], is to execute a subset of the bytecode native and trap on the more complex ones. This solution has a constant overhead for the software trap.

The approach to this problem in JOP is different. JOP has its own instruction set (the so called micro code). Some bytecodes have a 1 to 1 mapping to JOP instructions, for the more complex a sequence of JOP instructions is necessary. Every bytecode is translated to an address in the micro code that implements the JVM. Fig. 1 shows an example of this indirection.

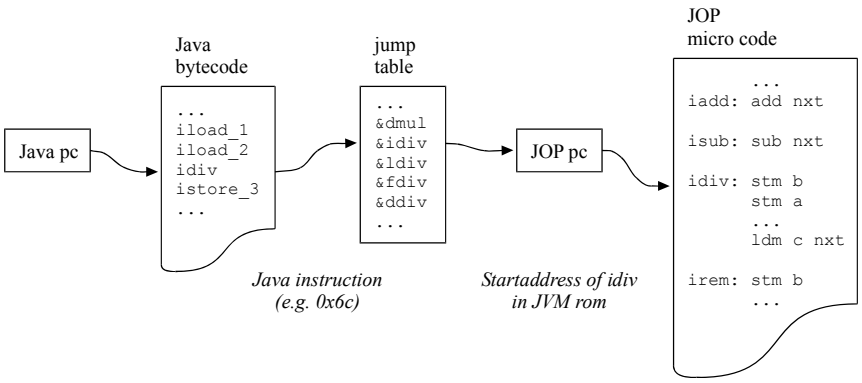

Fig. 1. Data flow from the Java program counter to JOP micro code. The fetched bytecode is used as an index into the jump table. The jump table contains the start addresses of the JVM implementation in micro code. This address is loaded into the JOP program counter for every executed bytecode.

If the bytecode has an equivalent JOP instruction, it is executed in one cycle and the next bytecode is translated. For more complex bytecodes JOP just continues to execute micro code in the following cycles. The end of this sequence is coded in the instruction (as the *nxt* bit). This translation needs an extra pipeline stage but has zero overheads for complex JVM instructions.

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

//  a and b are scratch variables for the JVM code.
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 next bytecode
```

This example shows the implementation of a single cycle bytecode (*dup*) and an unusual bytecode (*dup_x1*) as a sequence of JOP instructions that take 5 cycles to execute.

2.3 Pipeline Overview

The stack architecture allows a short pipeline resulting in short branch delays. Fig. 2 shows an overview of the pipeline. Three stages form the core of JOP, executing JOP instructions. An additional stage in the front of the core pipeline translates bytecodes to addresses in micro code. Every JOP instruction takes one cycle. Conditional branches have an implicit delay of two cycles. This branch delay can be filled with instructions or *nop*.

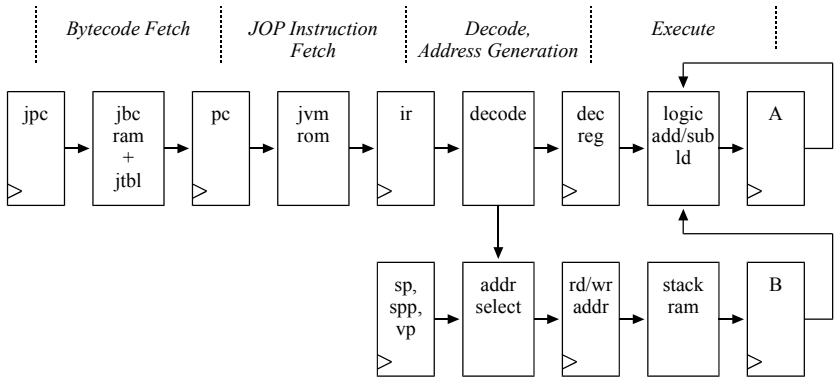

Fig. 2. Pipeline of JOP

2.4 Java Bytecode Fetch

The first pipeline stage can be seen in Fig. 3. All bytecodes are fetched from internal memory (*bytecode ram*). This memory, the instruction cache, is filled on function call and return. Every byte is mapped through *jtbl* to an address for the micro code rom (*jpaddr*). It is also stored in a register for later use as operand. Since *jpc* is also used to read operands, the program counter is stored in *jpcbr* during an instruction fetch. *jinstr* is used to decode the type of a branch and *jpcbr* to calculate the target address.

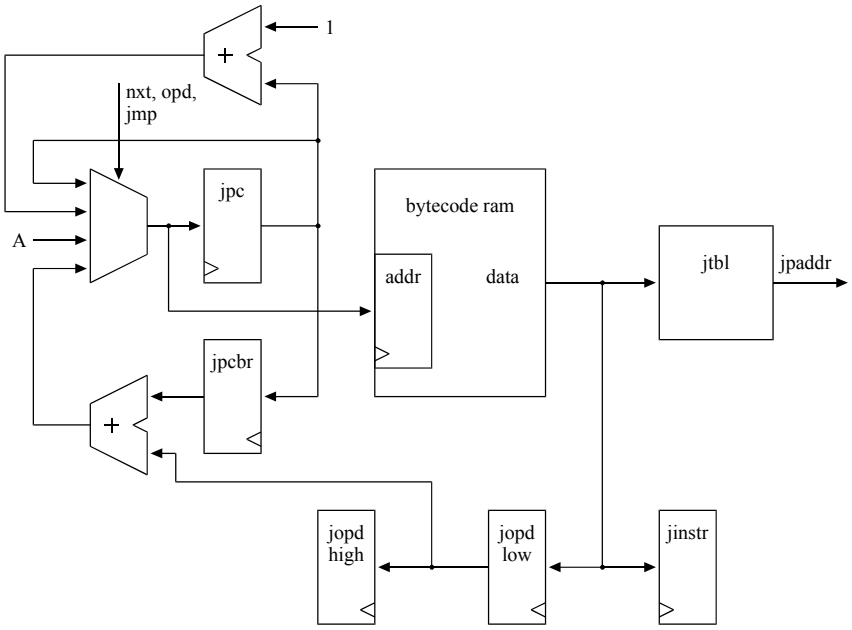

Fig. 3. Java bytecode fetch

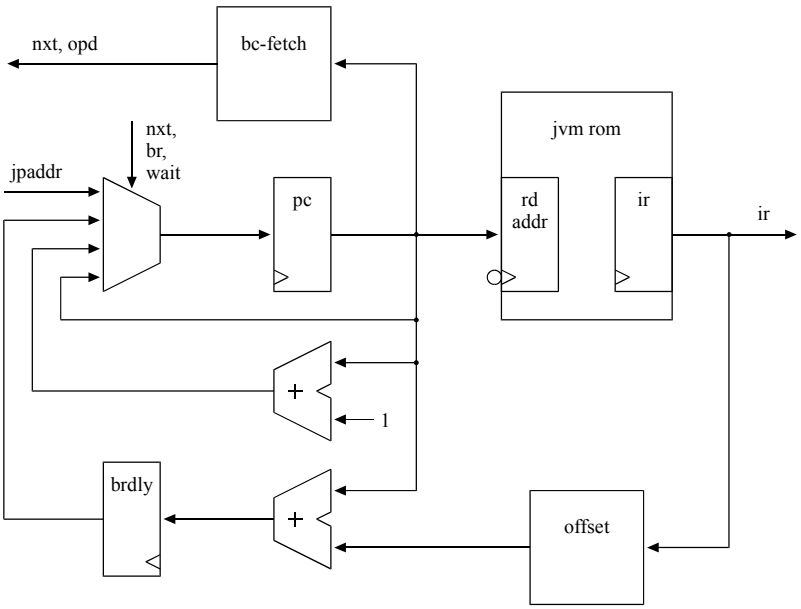

Fig. 4. JOP instruction fetch

2.5 JOP Instruction Fetch

Fig. 4 shows the second pipeline stage. JOP micro code that implements the JVM is stored in the memory labeled *jvm rom*. The program counter *pc* is incremented during normal execution. If the instruction is labeled with *nxt* a new bytecode is requested from the first stage and *pc* is loaded with *jpaddr*. *jpaddr* is the starting address for the implementation of that bytecode. This label and the one for a bytecode operand load (*opd*) are stored in *bc-fetch*.

brdly holds the target for a taken conditional branch. Many branch destinations share the same offset. A table (*offset*) is used to store these relative offsets. This indirection makes it possible to use only five bits in the instruction coding for branch targets and allow larger offsets. The three tables *bc-fetch*, *offset* and *jtbl* (from the bytecode fetch stage) are generated during assembly of the JVM code. The outputs are VHDL files. For an implementation in an FPGA it is no problem to recompile the design after changing the JVM implementation. For an ASIC with loadable JVM a different solution is necessary.

Current FPGAs don't allow asynchronous memory access. They force us to use the registers in the memory blocks. However, the output of these registers is not accessible. To avoid an additional pipeline stage just for a register-register move the read address register is clocked on the negative edge.

2.6 Decode and Address Generation

The third pipeline stage shown in Fig. 5 provides two functions. JOP instructions are decoded for the execution stage and addresses for read and write accesses of the stack ram are generated.

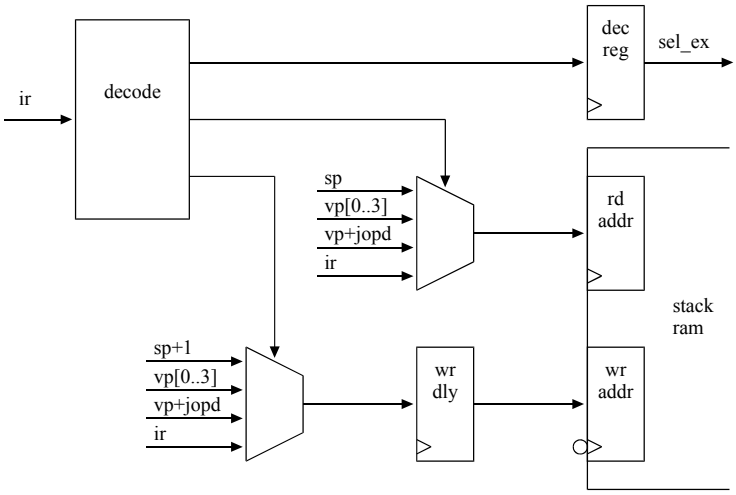

Fig. 5. Decode and address generation

Instructions of a stack machine can be categorized with respect to stack manipulation in *pop* or *push*. This allows us to generate the addresses for fill or spill of TOS-1 for the *following* instruction during the decode stage, saving one extra pipeline stage.

2.7 Execute

As can be seen in Fig. 6 TOS and TOS-1 are implemented as register *A* and *B*. Every arithmetic/logical operation is performed with *A* and *B* as source and *A* as destination. All load operations (local variables, internal register, external memory and periphery) result in the value loaded in *A*. Therefore no write back pipeline stage is necessary. *A* is also the source for store operations. Register *B* is never accessed directly. It is read as implicit operand or for stack spill on push instructions and written during stack spill and fill.

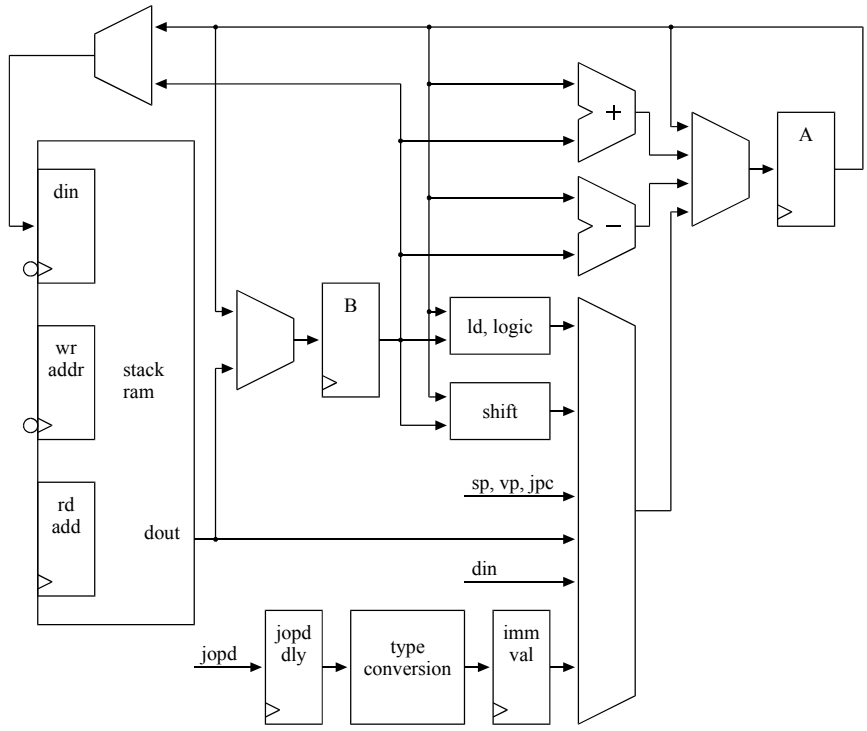

Fig. 6. Execution stage

3 HW/SW Co-Design

Using a hardware description language and loading the design in an FPGA, the traditional strict border between hardware and software gets blurred. Is configuring an FPGA not more like loading a program for execution?

This looser distinction makes it possible to move functions easily between hardware and software resulting in a highly configurable design. If speed is an issue, more functions are realized in hardware. If cost is the primary concern these functions are moved to software and a smaller FPGA can be used. Let us examine these possibilities on a relatively expensive function: multiplication. In Java bytecode *imul* performs a 32 bit signed multiplication with a 32 bit result. There are no exceptions on overflow.

Since single cycle multiplications for 32 bits are far beyond the possibilities of current FPGAs, we can implement *imul* with a sequential booth multiplier in VHDL. Three JOP instructions are used to access this function: *stopa* stores the first operand and *stpob* stores the second operand and starts the sequential multiplier. After 33 cycles, the result is loaded with *ldmul*.

If we run out of resources in the FPGA, we can move the function to micro code. The implementation of *imul* needs 73 JOP instructions and has an almost constant execution time.

JOP micro code is stored in an embedded memory block of the FPGA. This is also a resource of the FPGA. We can move the code to external memory by implementing *imul* in Java bytecode. Bytecodes not implemented in micro code result in a static method call from a special class (com.jopdesign.sys.JVM). The class has prototypes for every bytecode ordered by the bytecode value. This allows us to find the right method by indexing the method table with the value of the bytecode. The additional overhead for this implementation is a call and return with the cache refills.

Table 1 lists the resource usage and execution time for the three implementations. Executions time is measured with both operands negative, the worst-case execution time for the software implementations. The implementation in Java loads bytecodes from a slow memory interface (8 bit, 3 cycle per byte) and execution time depends on the caching policy.

Table 1. Different implementations of *imul*

	Hardware [LE]	Micro code [Byte]	Time [Cycle]
VHDL	300	12	37
Micro code	0	73	750
Java	0	0	~2300

Only a few lines of code have to be changed to select one of the three implementations. The showed principle can also be applied to other expensive bytecodes like: *idiv*, *ishr*, *iushr* and *ishl*. As a result, the resource usage of JOP is highly configurable and can be selected for every application. The possibility to call Java methods from micro code also allows us to code part of the JVM (like thread scheduling) in Java.

4 Real-Time Predictability

In real-time systems, especially hard real-time systems, meeting time constraints is of the same importance as functional correctness. One way to prove that all timing requirements are met is to calculate WCET (Worst-Case Execution Time) of all tasks.

These WCET values are the input for schedulability analysis. High-level WCET analysis, analyzing possible program flows, is a well-established research area [6]. A framework for portable Java bytecode WCET analysis can be found in [7]. At the low-level analysis execution time of bytecodes are derived from a specific VM model. The accuracy of the model has a major impact on the tightness of the WCET. Caches and pipeline effects are hard to model and can lead to an overestimation. Pipeline effects of common pairs of bytecodes are modeled in [8] to get tighter WCET.

These problems can be avoided with a well-known timing behavior for each bytecode and the cache. In JOP the execution time of all bytecodes is known cycle accurate. Most instructions have a constant execution time. Although JOP is full pipelined, resulting in some single cycle bytecodes, there are no timing dependencies between successive bytecodes. Even the conditional branch bytecodes have constant 4-cycle execution time whether the branch is taken or not.

4.1 Time Predictable Caches

The memory areas of the JVM can be classified as follows:

- Class description with method table and constant pool
- Code area
- Stack
- Heap for objects

We can decide which memory areas are cached. The two most frequent accessed areas are stack and code. The stack is implemented as internal memory in JOP resulting in a single cycle cache with independent read and write ports.

The same is true for code memory. Every executed bytecode is read from an internal memory (i.e. instruction cache). However, there is a big difference between JOPs instruction cache and instruction caches found in typical processors: No cache misses are allowed! This allows absolute predictable modeling of the cache behavior.

When is this cache filled? Typical Java programs consist of short methods. There are no branches out of the method and all branches inside are relative. In JOP the full code of a method has to be loaded in the cache before execution. The cache is filled on calls and returns. This means that all cache fills are lumped together with a known execution time.

The full loaded method and the relative addressing inside a method results in a simpler cache. No tag memory and no address translation are necessary.

The remaining two memory areas are not cached in JOP. Currently only one method is cached resulting in a refill on every method return. This solution has a predictable execution time but the average performance is worse than a traditional in-

struction cache. Keeping more methods in the cache with an efficient and predictable replace policy is a topic for further research.

4.2 Simple Real-Time Extension for Java

Tasks or threads are vital program constructs in embedded programming. Since threads and synchronization are defined as part of the language, Java can greatly simplify concurrent programming. Java, as described in [1], defines a very loose behavior of threads and scheduling. E.g. the specification allows even low priority threads to preempt high priority threads. This prevents threads from starvation in general purpose applications, but is not acceptable in real-time programming. To compensate for this under-specification, extensions to Java have been published. The Real-Time Specification for Java (RTSJ) [2] developed under the Sun Community Process addresses this problem.

RTSJ is complex to implement and applications developed with RTSJ are (due to some sophisticated features of the RTSJ) difficult to analyze. Different profiles with restrictions of the RTSJ have been suggested. In [9] a subset of the RTSJ for high-integrity application domain, with hard real-time constraints, is proposed. It is inspired by the Ravenscar profile for Ada [10] and the focus is on exact temporal predictability. Ravenscar-Java profile [11], based on previous mentioned work, restricts RTSJ even further. It claims to be compatible with RTSJ in the sense that programs written according to the profile are valid RTSJ programs. However, mandatory usages of new classes like `PeriodicThread` need an emulation layer to run on an RTSJ system. In this case, it is better to define completely new classes for a subset and provide the mapping to RTSJ. This leads to clearer distinction between the two definitions.

This real-time extension resembles the ideas from [9] and [11] but is not compatible with RTSJ. Its main purpose is to provide a framework for the development of JOP. If only a subset of RTSJ is implemented and allowed it is harder for programmers to find out what is available and what not. Use of different classes for a different specification is less error prone and restrictions can be enforced (e.g. setting thread priority only in the constructor of a real-time thread).

Application Structure. Following restrictions apply to the application:

- Initialization and mission phase.
- Fixed number of threads.
- Threads are created at initialization phase.
- All shared objects are allocated at initialization.

Threads. Three schedulable objects are defined: *RtThread* represents a periodic task. As usual task work is coded in `run()` which gets called on `missionStart()`. *HwEvent* represents an interrupt with a minimum inter-arrival time. If the hardware generates more interrupts, they get lost. A software event (*SwEvent*) is scheduled after a call of `fire()`.

```
public abstract class RtTask {
    public void enterMemory()
    public void exitMemory()
}

public class RtThread extends RtTask {

    public RtThread(int priority, int usPeriod)
    public RtThread(int priority, int usPeriod, Memory mem)
    public void run()
    public boolean waitForNextPeriod()
}

public class HwEvent extends RtTask {

    public HwEvent(int priority, int usMinTime, int number)
    public void handle()
}

public class SwEvent extends RtTask {

    public SwEvent(int priority, int usMinTime)
    public SwEvent(int priority, int usPeriod, Memory mem)
    public final void fire()
    public void handle()
}
```

Definition of the basic classes for the simple real-time extension.

Scheduling. The class `Scheduler` defines a preemptive fixed priority scheduler (with FIFO within priorities). Synchronized blocks are executed with priority ceiling protocol. The scheduler does not dispatch any `RtThread` until `startMission()` is called. Standard Java threads are scheduled during initialization phase, however usage is discouraged. The scheduler provides access methods to measured worst-case execution time of periodic work and the handler method. These values can be used during application development when no WCET analysis tool is available.

Memory. The profile does not support a garbage collector. All memory allocation have to be done in the initialization phase. For new objects during mission phase a scoped memory is provided. A scoped memory area is assigned to one `RtThread` on creation. It is not allowed to share a scoped memory between threads. No references from the heap to scoped memory are allowed. Scoped memory is explicit entered and left with calls from the application logic. As suggested in [12] memory areas are cleared on creation and when leaving the scope (call of `exitMemory`()) leading to a memory area with constant allocation time.

An Example. Following code shows the principle coding of a worker thread, creation of two real-time threads and an event handler:

```
public class Worker extends RtThread {

    private SwEvent event;

    public Worker(int p, int t, SwEvent ev) {
        super(p, t, new Memory(10000));
        event = ev;
        init();
    }

    private void init() {
        // All initialization has to be placed here
    }

    public void run() {

        for (;;) {
            work();        // do some work
            event.fire();  // and fire an event
            enterMemory(); // do some work in scoped memory
            workWithMem();
            exitMemory();
            if (!waitForNextPeriod()) {
                missedDeadline();
            }
        }
    }
}

// Application initializing:
// Create an event handler and worker threads with
// priorities according to their periods
Handler h = new Handler(RtThread.MAX_PRIORITY, 1000);
FastW fw = new FastW(RtThread.MAX_PRIORITY-1, 2000);
Worker w = new Worker(RtThread.MAX_PRIORITY-2, 10000, h);

// Change to mission phase for all
// periodic threads and event handlers
Scheduler.startMission();

// Do some non real-time work
for (;;) {
    watchdogBlink();
    Thread.sleep(500);
}
```

5 Results

Table 2 compares the resource usage of different soft-core processors:

- Nios: [13] Alteras configurable load/store RISC processor.
- SPEAR: [14] Scalable Processor for Embedded Applications in Real-time Environments with 16-bit instruction set and 3-stage pipeline.

- Lightfoot: [15] Xilinxs Java processor core, stack-based, 3-stage pipeline.
- JOP: with multiplier, single cycle shift, 8 bit memory interface, UART and timer.

Table 2. Different FPGA soft cores

Processor	LEs	ESB	Data Path
Nios	1700	2.5 KB	32-bit
SPEAR	1700	8 KB	16-bit
Lightfoot	3400	1 KB	32-bit
JOP	2100	3 KB	32-bit

Table 3 shows the result from a small benchmark. JOP runs at 24 MHz for this test and is compared with JVM version 1.1 on a Compaq notebook with Intel 486SX25. The 486SX has 1.2M transistors i.e. 300k gates. The FPGA where JOP is implemented for this comparison is (according to Altera) equivalent of 30k gates.

Table 3. JVM performance compared

Processor		Execution time	Relative performance
486SX25	Interpreting	19.55 s	1.00
486SX25	JVM with JIT	5.00 s	3.91
JOP		1.73 s	11.3

5.1 Applications

Balfour Beatty Austria has developed a *Kippfahrleitung* to speed up loading and unloading of goods wagons. The solution is to tilt the contact wire up on a line up to one kilometer. An asynchrony motor on each mast is used for this tilting. Nevertheless, it has to be done synchronic on the whole line. Technical this is a distributed embedded real-time control system with one processor board per mast communicating over an RS485 bus with a base station. The main challenge was to react to the sensors in real-time and control the bus access of this distributed system.

A second application of JOP is currently under development: The Austrian Railways adds a new security system for single-track lines. Every locomotive will be equipped with a GPS receiver and communication device. The position of the train, differential correction data for GPS and commands are exchanged with a server in the central station over a virtual private network. JOP is the heart of the communication device in the locomotive.

6 Conclusion

This paper presented the architecture of a hardware implementation of the JVM. The flexibility of FPGAs and HW/SW co-design makes it possible to adapt the resource usage of the processor for different applications. Predictable execution time of bytecodes, a predictable instruction cache and a simple extension of Java enable usage of

JOP in real-time applications. Although the full implementation of the JVM is still missing, one successful project showed that JOP is mature enough to be used in real-world applications. JOP encourages usage of Java in embedded systems.

Additional work has to be done to complete the JVM and port essential parts of the Java library. Further research will focus on the predictable instruction cache and hardware support for the real-time extensions of Java. More information and all VHDL and Java sources for JOP can be found in [16].

References

1. K. Arnold and J. Gosling. *The Java Programming Language*, Addison Wesley, 2nd edition, 1997.
2. Bollela, Gosling, Brosgol, Dibble, Furr, Hardin and Trunbull. *The Real-Time Specification for Java*, Addison Wesley, 1st edition, 2000.
3. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*, Addison Wesley, 2nd edition, 1999.
4. Altera Corporation. *ACEX Programmable Logic Family*, Data Sheet, ver. 1.01, April 2000.
5. Sun microsystems. *picoJava-II Processor Core*, Data Sheet, April 1999.
6. P. Puschner and C. Koza. Calculating the Maximum Execution Time of Real-Time Programs, *Real-Time Systems Journal*, 1(2): pp.159-176, September 1989
7. G. Bernat, A. Burns and A. Wellings. Portable Worst-Case Execution Time Analysis Using Java Byte Code, In *Proc. 6th Euromicro conference on Real-Time Systems*, pp.81-88, June 2000
8. I. Bate, G. Bernat, G. Murphy and P. Puschner. Low-Level Analysis of a Portable Java Byte Code WCET Analysis Framework, In *6th IEEE Real-Time Computing Systems and Applications (RTCSA2000),* pp.39-48, South Korea, December 2000
9. P. Puschner and A. J. Wellings. A Profile for High Integrity Real-Time Java Programs. In *Proc. of the 4th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC),* 2001
10. A. Burns and B. Dobbing. The Ravenscar Tasking Profile for High Integrity Real-Time Programs. In *Proc. of the 1998 annual ACM SIGAda international conference on Ada*, pp. 1-6, Washington, D.C., United States, 1998
11. J. Kwon, A. Wellings and S. King. Ravenscar-Java: a high integrity profile for real-time Java, In *Proc. of the 2002 joint ACM-ISCOPE conference on Java Grande*, pp. 131-140, Seattle, Washington, USA, 2002
12. A. Corsaro, D. Schmidt. The Design and Performance of the jRate Real-Time Java Implementation. Appeared at *the 4th International Symposium on Distributed Objects and Applications*, 2002
13. Altera Corporation. *Nios Soft Core Embedded Processor*, Data Sheet, ver. 1, June 2000.
14. M. Delvai, W. Huber, P. Puschner and A. Steininger. Processor Support for Temporal Predictability - The SPEAR Design Example. In *Proc. of Euromicro Conference on Real-Time Systems (ECRTS03)*, Porto, Portugal, July 2003.
15. Xilinx Corporation. *Lightfoot 32-bit Java Processor Core*, Data Sheet, September 2001.
16. Martin Schoeberl. JOP - a Java Optimized Processor, http://www.jopdesign.com.