Design Rationale of a Processor Architecture for Predictable Real-Time Execution of Java Programs

Martin Schoeberl

JOP.design, Vienna, Austria
martin@jopdesign.com

Abstract. Many of the benefits of Java, such as safe object references, 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 systems due to the lack of acceptable real-time performance. This paper provides a short overview of the issues with Java in real-time systems and the Real-Time Specification of Java (RTSJ) that addresses most of these problems. A simple real-time profile is presented and the implementation of this profile on top of a Java processor, designed for real-time systems, is described in detail. Performance comparison between this solution and the reference implementation of RTSJ on top of Linux show that a dedicated Java processor, without an underlying operating system, is more time predictable than an adoption of a general purpose OS for real-time systems.

1 Introduction

Java's first use was in an embedded system. In the early 1990s Java, whose original name was Oak, was created as programming tool for a wireless PDA. The device, called *7, was a small SPARC based hardware with a tiny embedded OS. *7 was not released as a product. Java however, was officially released 1995 as a new language for the Internet (to be integrated in Netscape's browser). Over the years, Java technology has become a programming tool for desktop applications and web services. The library, defined as part of the language, grew with each new release of Java.

Java for embedded systems was clearly out of focus for Sun. With the arrival of mobile phones, Sun again became interested in the embedded market. Sun defined different subsets of Java, which are collectively called J2ME (Java Micro Edition).

As the language became more popular, with easier object oriented programming than C++ and threads defined as part of the language, usage in real-time systems was considered. Two competing groups started to define how to convert Java for these systems.

Nilsen published the first paper on this topic in November of 1995 [1] and formed the Real-Time Working Group. The other group, known as the Real-Time Expert Group, which included Bollela and Gosling (the original designer of Java), published

the Real-Time Specification for Java (RTSJ) [2]. RTSJ was the first specification request under Sun's Java Community Process and received a great deal of attention from academic and industrial researchers.

Real-time Java presents different challenges for the Java Virtual Machine (JVM). Just-In-Time compilation is usually avoided and interpreting bytecodes leads to a slow execution model. Running a JVM on top of a Real Time Operating System (RTOS) adds extra overhead. JOP (Java Optimized Processor) [3] is the presented solution in hardware to build a solid basis for a real-time aware JVM. Tight interaction between the processor design and the design of the real-time language extensions can result in a performant and Worst Case Execution Time (WCET) analyzable real-time system.

This paper is organized as follows: Section 2 and Section 3 describe issues with the definition of Java to support real-time systems, followed by an overview of the Real-Time Specification for Java. Section 4 proposes a simple profile for real-time Java. Section 5 provides an overview of JOP, a Java processor for real-time systems. Design decisions and implementation details for a real-time enabled JVM are described in Section 6. This implementation is compared with the reference implementation of the RTSJ in Section 7. Section 8 is the Conclusion. In this paper, the words *task* and *thread* are interchangeable. *Task* is used when the context is more general and *thread* for Java specific aspects.

2 Java for Real-Time Systems

Although Java has language features that simplify concurrent programming the definition of these features is too vague for real-time systems. In the following section, some problematic aspects of Java for embedded real-time systems are described.

Threads and Synchronization: Java, as described in [4], defines a very loose behavior of threads and scheduling. For example, 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. Even an implementation without preemption is allowed. Wakeup of a single thread with notify() is not exactly defined: *The choice is arbitrary and occurs at the discretion of the implementation.* It is not mandatory for a JVM to deal with the priority inversion problem.

Garbage Collector: Garbage collection greatly simplifies programming and helps to avoid classic programming errors (e.g. memory leaks). Although real-time garbage collectors evolve, they are usually avoided in hard real-time systems. A more conservative approach to memory allocation is necessary.

WCET on Interfaces (OOP): Method overriding and Interfaces, the simplified concept of multiple inheritance in Java, are the key concepts in Java to support object oriented programming. Like function pointers in C, the dynamic selection of the actual function at runtime complicates WCET analysis. Implementation of interface

look up usually requires a search of the class hierarchy at runtime or very large dispatch tables.

Dynamic Class Loading: Dynamic class loading requires resolution and verification of classes. A function that is usually too complex (and consumes too much memory) for embedded devices. Upper bound of execution time for this function is almost impossible to predict (or too large).

Standard Library: For a Java conformant implementation, the full library (JDK) has to be part of it. The JAR files for this library constitute about 15 MB (in JDK 1.3, without native libraries), which is far too large for many embedded systems. Since Java was designed to be a safe language with a safe execution environment, no classes are defined for low-level access of hardware features. The standard library was not defined and coded with real-time applications in mind.

Execution Model: The first execution model for the JVM was an interpreter. The interpreter is now enhanced with Just-In-Time (JIT) compilation. Interpreting Java bytecodes is too slow and JIT compilation is not applicable in real-time systems. The time for the compilation process had to be included in the WCET, resulting in impracticable values.

Implementation Issues: The problems mentioned in this section are not *absolute* problems for real-time systems. However, they result in a slower execution model with a more pessimistic WCET.

According to [5] the static initializers of a class C are executed immediately before one of the following occurs: (a) an instance of C is created; (b) a static method of C is invoked or (c) a static field of C is used or assigned. Fig. 1 shows an example of this problem:

```
public class Problem {

    private static Abc a;
    public static int cnt; // implizit set 0

    static {
        // do some class initializaion
        a = new Abc();  //even this is ok.
    }

    public Problem() {
        ++cnt;
    }
}
// anywhere in some other class, without any
// instance of Problem this can lead to
// the execution of the initializer

    int nrOfProblems = Problem.cnt;
```

Fig. 1. Class initialization can occur very late

It follows that bytecodes *getstatic*, *putstatic, invokestatic* and *new,* can lead to class initialization and possible high WCET values. In the implementation of a JVM, it is necessary to check every execution of these bytecodes if the class is already initialized. This leads to a performance loss and is violated in some existing implementa-

tions of the JVM. For example in CACAO [6] the static initializer is called at compilation time.

Synchronization is possible with methods and on code blocks. Each object has a monitor associated with it and there are two different ways to gain and release ownership of a monitor. Bytecodes *monitorenter* and *monitorexit* explicitly handle synchronization. Otherwise, synchronized methods are marked in the class file with access flags to be synchronized. This means that all bytecodes for method invocation and return must check this access flag. This results in an unnecessary overhead on methods without synchronization. A better way would be to encapsulate the bytecode of synchronized methods with bytecodes *monitorenter* and *monitorexit*. This solution is used in Suns picoJava-II [7]. The code is manipulated in the class loader. The two different ways to express synchronization, in the bytecode stream and as access flags, are inconsistent.

3 Real-Time Specification for Java

To overcome some of the issues mentioned, the Real-Time Specification for Java (RTSJ) was created under the Sun Community Process. RTSJ defines a new API with support from the JVM [2]. The following guiding principles led to the definition:

- No restriction of the Java runtime environment.
- Backward compatibility for non-real-time Java programs.
- No syntactic extension to the Java language or new keywords.
- Predictable execution.
- Current practice and allow future implementations to add advanced features.

A Reference Implementation (RI) of RTSJ forms part of the specification [9]. RTSJ shall be backward compatible with existing non-real-time Java programs, which implies that RTSJ is intended to run on top of J2SE (and not on J2ME).

3.1 Threads and Scheduling

The behavior of the scheduler is clearer defined as in standard Java. A priority-based, preemptive scheduler with at least 28 real-time priorities is defined as base scheduler. Additional levels (ten) for the traditional Java threads need to be available. Threads with the same priority are queued in FIFO order.

The RTSJ introduces the concept of *schedulable objects*. Any instances of classes that implement the interface Schedulable, such as RealtimeThread, NoHeapRealtimeThread, and AsyncEventHandler, are managed by the scheduler. NoHeapRealtimeThread has, and AsyncEventHandler can have a higher priority than that of the garbage collector. As the available release-parameters indicate, threads are either periodic or asynchronous events.

The implementation of synchronized has to include an algorithm to prevent priority inversion. Priority inheritance protocol is the default and priority ceiling emulation

can be used on request. Threads waiting to enter a synchronized block are priority ordered and FIFO ordered within priority. Wait free queues are provided for communication between instances of java.lang.Thread and RealtimeThread.

3.2 Memory

As garbage collection is problematic in real-time applications, RTSJ defines additional memory areas:

Scoped memory is a memory area with bound lifetime similar to stack based memory. When a scope is entered (with a new thread or through enter()), all new objects are allocated in this memory area. Scoped memory areas can be nested and shared among threads. On exit of the last thread from a scope, all finalizers of the allocated objects are invoked and the memory area is freed.

Physical memory is used to control allocation in memories with different access time.

Raw memory allows byte-level access to physical memory or memory-mapped I/O.

Immortal memory is a memory shared between all threads without a garbage collector. All objects created in this memory area have the same live time as the application (a new definition of *immortal*).

3.3 Time and Timers

Classes to represent relative and absolute time with nanosecond accuracy are defined. All time parameters are split to a long for milliseconds and an int for nanoseconds within those milliseconds. A new type, rationale time, can be used to describe periods with a requested resolution over a longer period (i.e. allowing release jitter between the points of the *outer* period). Timer classes can generate time-triggered events (one shot and periodic).

3.4 Asynchrony

Program logic representing external world events is scheduled and dispatched by the scheduler. An AsyncEvent object represents an external event (such as a POSIX signal or a hardware interrupt) or an internal event (through call of fire()). Event handlers are associated to these events and can be bound to a regular real-time thread or represent something *similar* to a thread. The relationship between events and handlers can be many-to-many. Release of handlers can be restricted to a minimum interarrival time.

4 A Profile for Real-Time Java

The RTSJ is a complex specification and provides features not necessary, or even contradictory to high-integrity real-time systems [10]. To evaluate JOP as a real-time processor, a simple specification, not compatible with the RTSJ, is proposed. It is possible, and has been done, to implement a subset of the RTSJ, such as Ravenscar-Java [11], on top of this specification. The guidelines of this specification are:

- High-integrity profile.
- Easy syntax, simplicity.
- Easy to implement.
- Low runtime overhead.
- No syntactic extension of Java.
- Minimum change of Java semantics.
- Support for time measurement if WCET analysis tools are not available.
- Known overhead (Documentation of runtime behavior and memory requirement of every JVM operation and all provided methods).

4.1 Application Structure

The application is divided in an *initialization* and a *mission* phase. All non time-critical operations, such as creation of the real-time threads and allocation of objects in the heap are performed during the initialization phase. In the mission phase, entered by calling startMission(), all real-time threads are scheduled to perform the time-critical operations.

For a simpler and faster scheduler, the number of threads has to be fixed at one point of the execution. During the transition to the mission phase, all dynamic data structures of the threads are moved to a fixed, priority ordered list for the scheduler. The 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.
- No garbage collection. The heap is implicit *immortal* memory.

4.2 Real-Time Threads

Threads and events are defined as schedulable objects similar to the RTSJ:

RtThread represents a periodic task. As usual, task work is coded in run() which gets called on missionStart(). A scoped memory object can be attached to an RtThread at creation. The thread is blocked with waitForNextPeriod() till the next period.

HwEvent represents an interrupt with a minimum interarrival time. If the hardware generates more interrupts, they are delayed or lost.

SwEvent represents a software-generated event. It is triggered by fire() and needs to override handle().

A Memory object represents scoped memory and can be used with enterMemory() and exitMemory() when dynamic memory is needed in the mission phase. Scoped memory is allocated in the initialization phase, attached to a schedulable object at creation and cannot be shared between these objects. Fig. 2 shows the definition of the basic classes. The time values for period, offset and minTime are in microseconds.

```
package joprt;

public class RtThread {

    public RtThread(int priority, int period)
    public RtThread(int priority, int period, int offset)
    public RtThread(int priority, int period, Memory mem)
    public RtThread(int priority, int period, int offset, Memory mem)

    public void enterMemory()
    public void exitMemory()

    public void run()
    public boolean waitForNextPeriod()

    public static void startMission()
}

public class HwEvent extends RtThread {

    public HwEvent(int priority, int minTime, int number)
    public HwEvent(int priority, int minTime, Memory mem, int number)

    public void handle()
}

public class SwEvent extends RtThread {

    public SwEvent(int priority, int minTime)
    public SwEvent(int priority, int minTime, Memory mem)

    public final void fire()
    public void handle()
}
```

Fig. 2. Schedulable objects

4.3 Scheduling

Threads and Events are scheduled with fixed priority. No real-time thread or event is scheduled during the initialization phase. Threads with the same priority receive an implicit priority order by their creation. To avoid confusion and since the number of priority levels is not restricted, every real-time thread should get a unique priority assigned.

Synchronized blocks are executed with priority ceiling emulation protocol. With objects for which the priority is not set, a top priority is assumed. This avoids priority inversions on objects that are not accessible from the application (e.g. objects inside a library).

4.4 Restriction of Java

Some restrictions of language features for WCET analyzable real-time threads and bound memory usage are listed below:

WCET: Only analyzable language constructs are allowed (i.e. no unbound loops or recursions).

Static class initialization: This code has to be moved to a static method (e.g. init()) and called in the initialization phase.

String concatenation: In immortal memory scope only String concatenation with string literals is allowed.

Finalization: finalize() has a weak definition in Java. Because real-time systems run *forever*, objects in the heap, that is implicit *immortal* memory in this specification, will never be finalized. Objects in scoped memory are released on exitMemory(). However, finalizations on these objects complicate WCET analysis of exitMemory().

Dynamic Class Loading: Due to the implementation and WCET analysis complexity dynamic class loading is avoided.

A program analysis tool can help to enforce these restrictions.

5 Overview of JOP

JOP (Java Optimized Processor) [3] is the implementation of the JVM in hardware. JOP is intended for applications in embedded real-time systems and the primary implementation technology is in a Field Programmable Gate Array (FPGA), which results in the following design constraints:

- Every aspect of the architecture has to be time predictable for WCET (Worst Case Execution Time) analysis and predictable execution of real-time tasks.
- Low worst-case execution time is favored over average execution speed.
- The processor has to be small enough to fit in a low cost FPGA device to compete with traditional microcontrollers.

JOP is a full-pipelined architecture with single cycle execution of microcode instructions and a novel approach to map Java bytecode to these microcode instructions.

5.1 Architecture

Fig. 3 shows the datapath of JOP. In the first pipeline stage Java bytecodes, the instructions of the JVM, are fetched. These bytecodes are translated to addresses in the microcode. Bytecode branches are also decoded and executed in this stage. A fetched bytecode results in an absolute jump in the microcode (the second stage). The second pipeline stage fetches JOP instructions from the internal microcode memory and executes microcode branches.

The third pipeline stage performs, besides the usual decode function, address generation for the stack ram. Since every instruction of a stack machine has either *pop* or *push* characteristics, it is possible to generate the address for fill or spill for the *following* instruction in this stage.

In the execution stage operations are performed with two discrete registers: TOS and TOS-1. Data between stack ram and TOS-1 is also moved (fill or spill) in this stage. A stack machine with two explicit registers for the two topmost stack elements and automatic fill/spill does not need an extra write back stage, or any data forwarding.

A method cache, microcode ROM and stack RAM are implemented in internal memories of the FPGA with single cycle access.

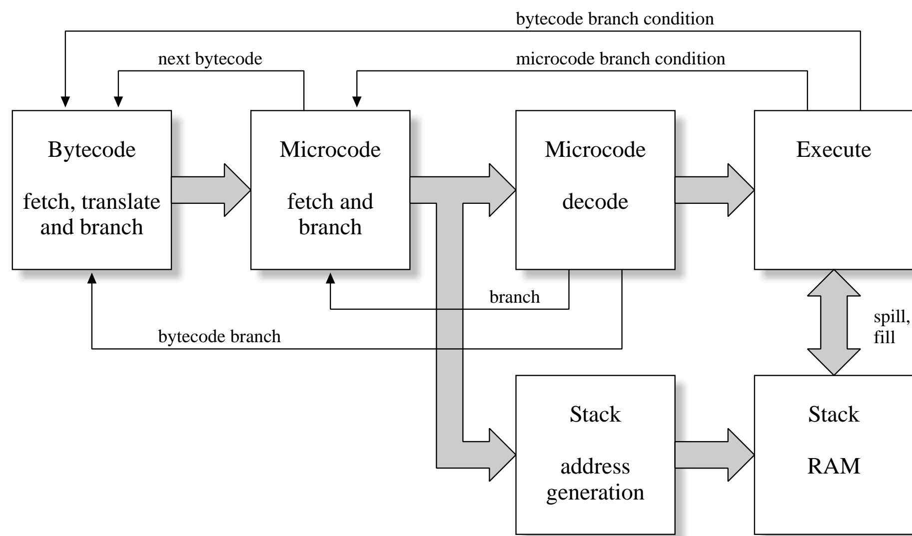

Fig. 3. Datapath of JOP

5.2 Microcode

There is a great variation of Java bytecodes. Simple instructions like arithmetic and logic operations on the stack are easy to implement in hardware. However, the semantic of bytecodes like *new* or *invoke* are too complex for hardware implementation. These bytecodes have to be implemented in a subroutine. Suns

These bytecodes have to be implemented in a subroutine. Suns picoJava-II [7] solves this problem by implementing only a subset of the bytecodes and generating a software trap on the more complex. This solution results in a constant execution overhead for the trap.

A different approach is used in JOP. JOP has its own instruction set (the so called microcode). Every bytecode is translated to an address in the microcode that implements the JVM. If the bytecode has a 1 to 1 mapping with a JOP instruction, it is executed in one cycle and the next bytecode is fetched and translated. For more complex bytecodes, JOP just continues to execute microcode in the following cycles. At the end of this instruction sequence the next bytecode is requested. This translation needs an extra pipeline stage but has zero overheads for complex JVM instructions.

The example in Fig. 4 shows the implementation of single cycle bytecodes and a bytecode as a sequence of JOP instructions. In this example, *ineg* takes 4 cycles to execute and after the last `add` the first instruction for the next bytecode is executed.

```
iadd:    add nxt    // 1 to 1 mapping

isub:    sub nxt

ineg:    ldi -1     // there is no -val
         xor        // function in the
         ldi 1      // ALU
         add nxt    // fetch next bc
```

Fig. 4. Implementation of iadd, isub and ineg

The microcode is translated with an assembler to a memory initialization file, which is downloaded during configuration. No further hardware is needed to implement loadable microcode.

JOP supports all bytecodes of the JVM specification in CLDC 1.0 [8] by Sun. A special feature of JOP allows missing bytecodes to be implemented in Java itself. All non-implemented bytecodes result in a jump to one microcode sequence. In this sequence, a static method (the method table is indexed by the bytecode value) from a system class is invoked. If, for example, hardware resources are too constraint to include a floating-point unit, these functions can be implemented in Java.

6 Implementation Details

General-purpose processors are optimized for average throughput and non real-time operating systems are responsible for fair and efficient scheduling of resources. Real-time systems need a processor with low and known WCET of instructions. Real-time operating systems have properties, such as fast interrupt time, rapid context switch, short blocking times and a scheduler that implements a simple, in most cases strict priority driven, scheduling algorithm. This section describes design decisions for JOP to support such real-time systems.

6.1 Interrupts

Interrupts are usually associated with low-level programming of device drivers. In a typical RTOS the priorities of interrupts and their handler functions are above task priorities and yield to immediate context switch. In this form, interrupts cannot be integrated in a schedule with *normal* tasks. The execution time of the interrupt handler has to be integrated in the schedulability analysis as additional blocking time. A better solution is to handle interrupts, that represent external events, as schedulable objects with priority levels in the range of real-time tasks, as in the RTSJ suggested.

The Timer Interrupt: The timer or clock interrupt has a different semantic than other interrupts. The main purpose of the timer interrupt is representation of time and release of periodic or time triggered tasks. One common implementation is a clock tick. The interrupt occurs at a regular interval (e.g. 10 ms) and a decision has to be taken whether a task has to be released. This approach is simple to implement, but there are two major drawbacks: The resolution of timed events is bound by the resolution of the clock tick and clock ticks without a task switch are a waste of execution time.

A better approach, used in JOP, is to generate timer interrupts at the release times of the tasks. Time is represented by a system counter. The timer interrupt can be programmed to occur at a specified value of this system counter. This allows generation of jitter free events. The scheduler is now responsible to reprogram the timer after each occurrence of a timer interrupt. The list of sleeping threads has to be searched to find the nearest release time in the future of a higher priority thread than the one that will be released now. This time is used for the next timer interrupt.

External Events: Hardware interrupts, other than the timer interrupt, are represented as asynchronous events with an associated thread. This means that the event is a *normal* schedulable object under the control of the scheduler. With a minimum interarrival time, enforced by hardware, these events can be incorporated in the priority assignment and schedulability analysis like periodic tasks.

Software Interrupts: The common software generated interrupts, such as illegal memory access or divide by zero, are represented by Java runtime exceptions and need no special handler. They can be detected with a try-catch block. Care has to be taken by the application to change to a safe state, as these exceptions are allocated in immortal memory and result in a memory leak.

Asynchronous notification from the program is supported like an external event as a schedulable object with an associated thread. The event is triggered through the call of `fire()`. The thread with the handler is placed in the runnable state and scheduled according to priority.

Hardware Failures: Serious hardware failures, such as illegal opcode or parity error from the memory systems, lead to a shutdown of the system. However, a *last try* to

call a handler that changes the state of the system to a fail-safe state and signal an upper level system, can improve the integrity of the overall system.

6.2 Scheduling

An important issue in real-time systems is the time for a task switch. A task switch consists of two actions:

- **Scheduling:** Selection of the task order and timing.
- **Dispatching:** Context switch between tasks.

Scheduling: Most real-time systems use a fixed-priority preemptive scheduler. Tasks with the same priority are usually scheduled in a FIFO order. Two common ways to assign priorities are rate monotonic or, in a more general form, deadline monotonic assignment. When two tasks get the same priority, we can choose one of them and assign a higher priority to that task and the task set is still schedulable. We get a strictly monotonic priority order and do not have to deal with FIFO order. This eliminates queues for each priority level and results in a single, priority ordered task list.

Strictly fixed priority schedulers suffer from a problem called *priority inversion* [12]. The problem where a low priority task blocks a high priority task on a shared resource is solved by raising the priority of the low priority task. Two standard priority inversion avoidance protocols are common:

Priority Inheritance Protocol: A lock assigns the priority of the highest-priority waiting task to the task holding the lock until that task releases the resource.

Priority Ceiling Emulation Protocol: A lock gets a priority assigned above the priority of the highest-priority task that will ever acquire the lock. Every task will be immediately assigned the priority of that lock when acquiring it.

Priority inheritance protocol is more complex to implement and the time when the priority of a task is raised is not so obvious. It is not raised because the task does anything, but because another task reaches some point in its execution path.

Using priority ceiling emulation with unique priorities, different from task priorities, the priority order is still strict monotonic. The priority ordered task list is expanded with slots for each lock. If a task acquires a lock, it is placed in the corresponding slot. With this extension to the task list, scheduling is still simple and can be efficiently implemented.

Dispatching: The time for a context switch depends on how *large* the state of a task is. For a stack machine it is not so obvious what belongs to the state of a task. If the stack resides in main memory, only a few registers (e.g. program counter and stack pointer) have to be saved and restored. However, the stack is a frequently accessed memory region of the JVM. The stack can be seen as data cache and should be placed *near* the execution unit. This means that the stack is part of the execution context and has to be saved and restored on a context switch.

In JOP the stack is placed in local memory with single cycle access time. With this configuration, the next question is how much of the stack is placed there. The complete stack of a thread can reside local or only the stack frame of the current method. If the complete stack of a thread is stored in local memory invocation of methods and returns are fast, but the context is large. For fast context switch, it would be preferable to have only a short stack in local memory. This results in fewer data to be transferred to and from main memory but more memory transfers on method call and return. The local stack can further be divided to small pieces, each holding only one stack frame of one thread. During the context switch only the stack pointer has to be saved and restored. The outcome of this is a very fast context switch but the size of the local memory limits the maximum number of threads.

Since JOP is a soft-core processor, these different solutions can be configured for different application requirements. Even a mix of these policies is possible. A simple way to allocate the stacks would be to dedicate one stack slot to each *important* thread, with one stack slot shared by the remaining threads. Only a switch *to* such a less important thread requires save and restore of this stack slot with main memory.

6.3 Architectural Design Decisions

In hard real-time systems, meeting temporal requirements is of the same importance as functional correctness. This results in different architectural constraints than a design for a non real-time system. Upper bound of execution time is of premium importance. Good average execution time is useless for a pure hard real-time system.

Common architectural components, found in general purpose processors to enhance average performance, are usually problematic for WCET analysis. A pragmatic approach to this problem is to ignore these features for the analysis. With a processor designed for real-time applications, these useless features have to be substituted by predictable architecture enhancements.

Branch Prediction: As the pipelines of current general-purpose processors get longer to support higher clock rates the penalty of branches get too high. This is compensated by branch prediction logic with branch target buffers. However, the upper bound of branch execution time is the same as without this feature. In JOP, branch prediction is avoided. This results in pressure on the pipeline length. The core processor has a minimal pipeline length of three stages resulting in a branch delay of three cycles in microcode. The two slots in the branch delay can be filled with instructions or *nop*. With the additional bytecode fetch and translation stage, the overall pipeline is four stages and results in a four cycle execution time for a bytecode branch.

Caches and Instruction Prefetch: To reduce the growing gap between clock frequency of the processor and memory access times multi level cache architectures are commonly used. Since even a single level cache is problematic for WCET analysis, more levels in the memory architecture are almost not analyzable. The additional levels also increase the latency of memory access on a cache miss.

In a stack machine, the stack is a frequently accessed memory area. This makes the stack an ideal candidate to be placed near the execution unit in the memory hierarchy. In JOP the stack is implemented as internal memory with the two top elements as explicit registers. This single cycle memory can be seen as a data cache. However, unlike in picoJava, this limited memory is not automatically spilled and filled to/from main memory. Automatically spill and fill introduces unpredictable access to the main memory. Data exchange between internal stack and main memory is under program control and can be done on method call/return or on a thread switch.

The next most accessed memory area is the code area. A simple prefetch queue, like in older processors, could increase instruction throughput after executing a multi cycle bytecode. For a stream of single cycle bytecodes, prefetching is useless and the frequent occurrence of branches and method invocations, about 20% [13] in typical Java programs, reduce the performance gain. The prefetch queue also results in execution time dependencies over a stream of instructions, which complicates timing analysis.

JOP has a method cache with a novel replace policy. Since typical methods in Java programs are short and there are only relative branches in a method, a complete method is loaded in the cache on invocation and on return. This cache fill strategy lumps all cache misses together and is very simple to analyze. It also simplifies the hardware of the cache since no tag memory or address translation is necessary. The *romizer* tool JavaCodeCompact checks the maximum allowed method size.

Memory areas for the heap and class description with the constant pool are not cached in JOP.

Superscalar Processors: A superscalar processor consists of several execution units and tries to extract instruction level parallelism (ILP) with out of order execution. Again, this is a nightmare for timing analysis. The code for a stack machine has less implicit parallelism than a register machine.

One form of enhancement, usually implemented in stack machines, is instruction folding. The instruction stream is scanned to find frequent patterns like load-load-add-store and substitutes these four instructions with one RISC like operation. There are two problems with instruction folding in JOP: The combined instruction needs two read and one write access to the stack in a single cycle. This would result in doubling of the internal memory usage in the FPGA. It also needs, at minimum, four bytes read access to the method cache. To overcome word boundaries, prefetching has to be introduced after the method cache. This results in an additional pipeline stage, time dependency of instructions with a more complex analysis and much hardware resources for the multiplexers.

Programs for embedded and real-time systems are usually multi threaded. In future work, it will be investigated if the additional hardware resources needed for ILP can be better used with additional processor cores utilizing this implicit parallelism.

Garbage Collection: As use of the heap is avoided in hard real-time systems, no garbage collector is implemented. Without a garbage collector, the memory layout of objects can be simplified. Every reference points direct to the object. No indirection

through a handle, which would simplify memory compaction in the garbage collector, is needed. This reduces access time to object fields and methods.

Time Predictable Instructions: A good model of a processor with accurate timing information is essential for a tight WCET analysis. The architecture of JOP and the microcode are designed with this in mind. Execution time of bytecodes is known cycle accurate. It is possible to analyze WCET on a bytecode level [14] without the uncertainties of an interpreting JVM [15] or generated native code from ahead-of-time compilers for Java.

7 Results

In this section, the implementation of the simple real-time profile with JOP is compared with the RI of RTSJ on top of Linux. The RI is an interpreting implementation of the JVM not optimized for performance. A commercial version of the RTSJ, JTime by TimeSys, should perform better. However, it was not possible to get a license of JTime for research purpose. JOP is implemented in a low cost FPGA (Cyclone EP1C6) from Altera clocked with 100 MHz. The test results for the RI where obtained on an Intel Pentium MMX 266 MHz, running Linux with two different kernels: a generic kernel version 2.4.20 and the real-time kernel from TimeSys [16] as recommended for the RI. For each test 500 measurements where made. Time was measured with a hardware counter in JOP and the Time Stamp Counter of the Pentium processor under Linux.

7.1 Periodic Threads

Many activities in real-time systems must be performed periodically. Low release jitter is of major importance for tasks such as control loops. The test setting is similar to the periodic thread test in [17]. A single real-time thread only calls waitForNextPeriod() in a loop and records the time between following calls. A second idle thread, with lower priority, just consumes processing time. This test setting results in two context switches per period. Table 1 shows average, standard deviation and extreme values for different period times on JOP. The same values are shown in Table 2 for the RI.

Usage of microsecond accurate timer interrupts, programmed by the scheduler, result in excellent performance of periodic threads in JOP. No jitter from the scheduler can be seen with a single thread at periods above 80 us.

The measurement for the RI is without the first values. The first values are a little bit misleading since the RI behaves unpredictable at *startup*. The RI performs inaccurate at periods below 20 ms. This effect has also been observed in [18]. Larger periods, that are multiples of 10 ms, have very low jitter. However, using a period such as 35 ms shows a standard deviation of five ms. A detailed look in the collected samples shows only values of 30 and 40 ms. This implies a timer tick of 10 ms in the underly-

ing operating system. There is no real difference when running this test on the generic Linux kernel and on the TimeSys kernel. Table 2 represents the measurements on the generic kernel. This comparison shows the advantage of an adjustable timer interrupt over a fixed timer tick.

Table 1. Jitter of Periodic Threads with JOP.

	Avg.	*Std. Dev.*	*Min.*	*Max.*
T=80 us	80 us	28 us	52 us	115 us
T=100 us	100 us	0 us	100 us	100 us
T=500 us	500 us	0 us	500 us	500 us
T=1 ms	1 ms	0 ms	1 ms	1 ms
T=5 ms	5 ms	0 ms	5 ms	5 ms
T=10 ms	10 ms	0 ms	10 ms	10 ms

Table 2. Jitter of Periodic Threads with RI/RTSJ.

	Avg.	*Std. Dev.*	*Min.*	*Max.*
T=500 us	315 us	93 us	18 us	569 us
T=1 ms	1.00 ms	0.01 ms	0.946 ms	1.055 ms
T=5 ms	4.00 ms	7.92 ms	0.017 ms	19.90 ms
T=10 ms	6.64 ms	9.34 ms	0.019 ms	19.94 ms
T=20 ms	20.0 ms	0.015 ms	19.87 ms	20.14 ms
T=30 ms	30.0 ms	0.031 ms	29.69 ms	30.31 ms
T=35 ms	35.0 ms	5.001 ms	29.75 ms	40.25 ms
T=50 ms	50.0 ms	0.018 ms	49.95 ms	50.06 ms
T=100 ms	100 ms	0.002 ms	99.94 ms	100.1 ms

7.2 Context Switch

The test setting consists of two threads. A low priority thread continuously stores the current time in a shared variable. A high priority, periodic thread measures the time difference between this value and the time immediately after waitForNextPeriod(). Table 3 reports the time for the context switch in processor clock cycles.

Table 3. Time for a Thread Switch in Clock Cycles.

	Avg.	*Std. Dev.*	*Min.*	*Max.*
JOP	4088	10.29	4083	4116
RI Linux	4253	1239	3232	19628
RI TS Linux	12923	1145	11529	21090

This test did not produce the expected behavior on the RI on the generic Linux kernel. The high priority thread was not scheduled, when the low priority thread runs in this tight loop. After inserting a Thread.yield() and an operating system call, such as System.out.print(), in this loop, the test performed as expected. This indicates a major problem in the RI or the scheduler in the operating system. This problem disappeared with the RI on the TimeSys Linux kernel. However, the context switch time is three times longer than on the standard kernel.

7.3 Asynchronous Event Handler

In this test setting, a high priority event handler is triggered by a low priority periodic thread. As AsynchEventHandler performs poor [18], a BoundAsynchEventHandler is used for the RI test program. Time is measured between the invocation of fire() and the first statement of the event handler. Table 4 shows the elapsed time in clock cycles for JOP and the RTSJ RI.

Table 4. Dispatch Latency of Event Handlers.

	Avg.	*Std. Dev.*	*Min.*	*Max.*
JOP	4283	3.0	4283	4350
RI Linux	53685	7014	47400	87196
RI TS Linux	69273	7832	63060	101292

The time to dispatch an asynchronous event is similar to the context switch time in JOP. The maximum value occurred only on the first event, all following events where dispatched with the minimum time.

In the RI the dispatch time is about 12 times larger than a context switch with a significant variation in time. This indicates that the implementation of fire() and the communication of the event to the underlying operating system is not optimal. The time factor between a context switch and event handling on the TimeSys kernel is lower than on the standard kernel, but still significant.

8 Conclusion

Java possesses language features as safety and object orientation that can greatly improve development of embedded systems. However, most embedded systems impose timing constraints that are contradictory to the unpredictable performance of standard Java. The RTSJ is a specification that addresses this problem. However, the RTSJ is a specification to large and complex to be implemented in small embedded systems. A simpler profile for real-time Java is presented in this paper. This profile is implemented on top of a hardware JVM, i.e. a Java processor, specially designed for real-time systems. Although this profile restricts the Java programming model it has been used with success to implement several commercial real-time applications. Tight integration of the real-time scheduler with the supporting processor result in an efficient platform for Java in embedded real-time systems. Performance comparison between this implementation and the RTSJ on top of Linux show that a dedicated Java processor without an underlying operating system is better predictable than trying to adopt a general purpose OS for real-time systems. Time will show, if an implementation of the RTSJ on a *real* RTOS will outperform the presented solution.

References

[1] K. Nilsen. Issues in the Design and Implementation of Real-Time Java, July 1996. Published June 1996 in *Java Developers Journal*, republished in Q1 1998 *Real-Time Magazine*

[2] Bollela, Gosling, Brosgol, Dibble, Furr, Hardin and Trunbull. *The Real-Time Specification for Java*, Addison Wesley, 1st edition, 2000.

[3] M. Schoeberl. JOP: a Java Optimized Processor. In *Workshop on Java Technologies for Realtime and Embedded Systems (JTRES 2003)*, Catania, Sicily, Italy, November 2003.

[4] J. Gosling, B. Joy, G. Steele and G. Bracha. *The Java Language Specification*. Addison Wesley, 2nd edition, 1997.

[5] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 2nd edition, 2000.

[6] A. Krall and R. Graft. CACAO - A 64 bit JavaVM just-in-time compiler. In G. C. Fox and W. Li, editors, *PPoPP'97 Workshop on Java for Science and Engineering Computation*, Las Vegas, June 1997.

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

[8] Sun Microsystems. *Java 2 Platform, Micro Edition (J2ME)*, available at: http://java.sun.com/j2me/docs/

[9] TimeSys. Real-Time Specification for Java Reference Implementation. http://www.timesys.com/

[10] P. Puschner and A. J. Wellings. A Profile for High Integrity Real-Time Java Programs. In *Proceedings of the 4th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC)*, 2001.

[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] L. Sha, R. Rajkumar, and J.P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Transactions on Computers*, 1990. 39 (9): p. 1175-1185.
[13] R. Radhakrishnan, J. Rubio and L.K. John. Characterization of Java applications at byte-code and ultra-SPARC machine code levels. In *International Conference on Computer Design (ICCD 1999)*, pp. 281-284, October 1999.
[14] 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.
[15] 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.
[16] TimeSys. Linux RTOS Standard Edition available at: http://www.timesys.com/
[17] A. Corsaro, D. Schmidt. Evaluating Real-Time Java Features and Performance for Real-Time Embedded Systems. Appeared at *The 8th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2002.
[18] 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