JOP: A Java Optimized Processor for Embedded Real-Time Systems

Martin Schoeberl
JOP Research Targets

- Java processor
- Time-predictable architecture
- Small design
- Working solution (FPGA)
Overview

- Motivation
- Research objectives
- Java and the JVM
- Related work
- JOP architecture
- Results
- Conclusions, future work
Current Praxis

- C and assembler
- Embedded systems are RT systems
- Different RTOS
- JIT is not possible
- JVM interpreter are slow
- => Java processor
Why Java?

- Safe OO language
  - No pointers
  - Type-safety
  - Garbage collection
- Built in model for concurrency
- Platform \textit{independent}
- Very rich \textit{standard} library
Research Objectives

- Primary objectives:
  - Time-predictable Java platform
  - Small design
  - A working processor

- Secondary objectives:
  - Acceptable performance
  - A flexible architecture
  - Real-time profile for Java
Java and the JVM

- Java language definition
- Class library
- The Java virtual machine (JVM)
  - An instruction set – the *bytecodes*
  - A binary format – the *class file*
  - An algorithm to *verify* the class file
The JVM instruction set

- 32 (64) bit stack machine
- Variable length instruction set
- Simple to very complex instructions
- Symbolic references
- Only relative branches
Memory Areas for the JVM

- Stack
  - Most often accessed
  - On-chip memory as cache
- Code
  - Novel instruction cache
- Class description and constant pool
- Heap
Implementations of the JVM

- Interpreter
- Just-in-time compilation
- Batch compilation
- Hardware implementation
Related Work

- picoJava
  - SUN, never released
- aJile JEMCore
  - Available, RTSJ, two versions
- Komodo
  - Multithreaded Java processor
- FemtoJava
  - Application specific processor
## Research Objectives

<table>
<thead>
<tr>
<th></th>
<th>picoJava</th>
<th>aJile</th>
<th>Komodo</th>
<th>FemtoJava</th>
<th>JOP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Predictability</td>
<td>- -</td>
<td>.</td>
<td>-</td>
<td>.</td>
<td>+ +</td>
</tr>
<tr>
<td>Size</td>
<td>- -</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+ +</td>
</tr>
<tr>
<td>Performance</td>
<td>+ +</td>
<td>+</td>
<td>-</td>
<td>- -</td>
<td>+</td>
</tr>
<tr>
<td>JVM conf.</td>
<td>+ +</td>
<td>+</td>
<td>-</td>
<td>- -</td>
<td>.</td>
</tr>
<tr>
<td>Flexibility</td>
<td>- -</td>
<td>- -</td>
<td>+</td>
<td>+ +</td>
<td>+ +</td>
</tr>
</tbody>
</table>
JOP Architecture

- Overview
- Microcode
- Processor pipeline
- An efficient stack machine
- Instruction cache
JOP Block Diagram

- JOP Core
  - Bytecode Fetch
  - Fetch
  - Decode
  - Stack

- Memory Interface
  - Bytecode Cache

- Extension
  - Multiplier

- I/O Interface

Connections:
- Busy
- BC Address
- BC Data
- Control
- Data
- Interrupt

JOP Overview
JVM Bytecode Issue

- Simple and complex instruction mix
- No bytecodes for *native* functions
- Common solution (e.g. in picoJava):
  - Implement a subset of the bytecodes
  - SW trap on complex instructions
  - Overhead for the trap – 16 to 926 cycles
  - Additional instructions (115!)
JOP Solution

- Translation to microcode in hardware
- Additional pipeline stage
- No overhead for complex bytecodes
  - 1 to 1 mapping results in single cycle execution
  - Microcode sequence for more complex bytecodes
- Bytecodes can be implemented in Java
Microcode

- Stack-oriented
- Compact
- Constant length
- Single cycle
- Low-level HW access

An example

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 TOS
            ldm b // restore TOS–1
            ldm a nxt // restore TOS
            // and fetch next bytecode
Processor Pipeline

- Bytecode: Fetch, translate and branch
- Microcode: Fetch and branch
- Microcode: Decode
- Microcode: Execute
- Stack: Address generation
- Stack: RAM
- Spill, fill
- Bytecode branch condition
- Microcode branch condition
- Next bytecode
- Branch
- Bytecode branch
Interrupts

- Interrupt logic at bytecode translation
  - Special bytecode
  - Transparent to the core pipeline
- Interrupts under scheduler control
  - Priority for device drivers
  - No additional blocking time
  - Integration in schedulability analysis
  - Jitter free timer events
  - Bound to a thread
An Efficient Stack Machine

- JVM stack is a logical stack
  - Frame for return information
  - Local variable area
  - Operand stack
- Argument-passing regulates the layout
- Operand stack and local variables need caching
Stack access

- Stack operation
  - Read TOS and TOS-1
  - Execute
  - Write back TOS

- Variable load
  - Read from deeper stack location
  - Write into TOS

- Variable store
  - Read TOS
  - Write into deeper stack location
Two-Level Stack Cache

- Dual read only from TOS and TOS-1
- Two register (A/B)
- Dual-port memory
- Simpler Pipeline
- No forwarding logic

- Instruction fetch
- Instruction decode
- Execute, load or store
JVM Properties

- Short methods
- Maximum method size is restricted
- No branches out of or into a method
- Only relative branches
Proposed Cache Solution

- Full method cached
- Cache fill on call and return
  - Cache misses only at these bytecodes
- Relative addressing
  - No address translation necessary
- No fast tag memory
- Simpler WCET analysis
Architecture Summary

- Microcode
- 1+3 stage pipeline
- Two-level stack cache
- Method cache

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

- Size
  - Compared to soft-core processors

- General performance
  - Application benchmark (KFL & UDP/IP)
  - Various Java systems

- Real-time performance
  - 100MHz JOP – 266MHz Pentium MMX
  - Simple RT profile – RTSJ/RT-Linux
## Size of FPGA processors

<table>
<thead>
<tr>
<th>Processor</th>
<th>Resources [LC]</th>
<th>Memory [KB]</th>
<th>$f_{\text{max}}$ [MHz]</th>
</tr>
</thead>
<tbody>
<tr>
<td>JOP min.</td>
<td>1077</td>
<td>3.25</td>
<td>98</td>
</tr>
<tr>
<td>JOP typ.</td>
<td>1831</td>
<td>3.25</td>
<td>101</td>
</tr>
<tr>
<td>Lightfoot</td>
<td>3400</td>
<td>1</td>
<td>40</td>
</tr>
<tr>
<td>Komodo</td>
<td>2600</td>
<td>?</td>
<td>33/4</td>
</tr>
<tr>
<td>FemtoJava</td>
<td>2000</td>
<td>?</td>
<td>4</td>
</tr>
<tr>
<td>NIOS</td>
<td>2923</td>
<td>5.5</td>
<td>119</td>
</tr>
<tr>
<td>SPEAR</td>
<td>1700</td>
<td>8</td>
<td>80</td>
</tr>
</tbody>
</table>
Application Benchmark

Performance (iterations/s)

- leOS
- KVM
- TiNi
- Cjip
- Komodo
- au80
- EJC
- au100
- JOP
- grj

JOP Overview
Benchmark Scaled
## Periodic Thread Jitter

<table>
<thead>
<tr>
<th>Period</th>
<th>JOP</th>
<th>RTSJ/Linux</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Min.</td>
<td>Max.</td>
</tr>
<tr>
<td>50 us</td>
<td>35 us</td>
<td>63 us</td>
</tr>
<tr>
<td>70 us</td>
<td>70 us</td>
<td>70 us</td>
</tr>
<tr>
<td>100 us</td>
<td>100 us</td>
<td>100 us</td>
</tr>
<tr>
<td>5 ms</td>
<td>5 ms</td>
<td>5 ms</td>
</tr>
<tr>
<td>10 ms</td>
<td>10 ms</td>
<td>10 ms</td>
</tr>
<tr>
<td>30 ms</td>
<td>30 ms</td>
<td>30 ms</td>
</tr>
<tr>
<td>35 ms</td>
<td>35 ms</td>
<td>35 ms</td>
</tr>
</tbody>
</table>
Context Switch

- Low priority thread records current time
- High priority periodic/event thread measures elapsed time after unblocking
- Time in cycles

<table>
<thead>
<tr>
<th></th>
<th>JOP</th>
<th>RTSJ/Linux</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Min.</td>
<td>Max.</td>
</tr>
<tr>
<td>Thread</td>
<td>2676</td>
<td>2709</td>
</tr>
<tr>
<td>SW Event</td>
<td>2773</td>
<td>2935</td>
</tr>
</tbody>
</table>
Applications

- Kippfahrleitung
  - Distributed motor control
- ÖBB
  - Vereinfachtes Zugleitsystem
  - GPS, GPRS, supervision
- TeleAlarm
  - Remote tele-control
  - Data logging
  - Automation
JOP in Research

- University of Lund, SE
  - Application specific hardware (Java->VHDL)
  - Hardware garbage collector
- Technical University Graz, AT
  - HW accelerator for encryption
- University of York, GB
  - Javamen – HW for real-time systems
- Institute of Informatics at CBS, DK
  - RT GC, Embedded RT Machine Learning
- University of California, Irvine, USA
  - WCET Analysis
JOP for Teaching

- Easy access – open-source
  - Computer architecture
  - Embedded systems

- UT Vienna
  - JVM in hardware course
  - Digital signal processing lab

- CBS
  - Distributed data mining (WS 2005)
  - Very small information systems (SS 2006)

- Wikiversity
Contributions

- Real-time Java processor
  - Exactly known execution time of the BCs
  - No mutual dependency between BCs
  - Time-predictable method cache

- Resource-constrained processor
  - RISC stack architecture
  - Efficient stack cache
  - Flexible architecture
Future Work

- HW for real-time garbage collector
- Instruction cache WC analysis
- Multiprocessor JVM
- Java computer
More Information

- JOP Thesis and source
  - http://www.jopdesign.com/download.jsp

- Various papers
  - http://www.jopdesign.com/docu.jsp