picoJava-II in an FPGA

Wolfgang Puffitsch

Vienna University of Technology

September 28, 2007

Outline

1. Motivation
2. Architecture
3. Hardware
4. Software
5. Evaluation
6. Conclusion

Outline

1 Motivation

2 Architecture

3 Hardware

4 Software

5 Evaluation

6 Conclusion

Motivation

- picoJava often referenced in papers
- Only one known implementation
  - Data about performance missing
- picoJava designed for ASICs, competitors for FPGAs
  - How to compare them?

Outline

1. Motivation
2. Architecture
   - Overview
   - Features
   - Instruction Folding
3. Hardware
4. Software
5. Evaluation
6. Conclusion

Overview

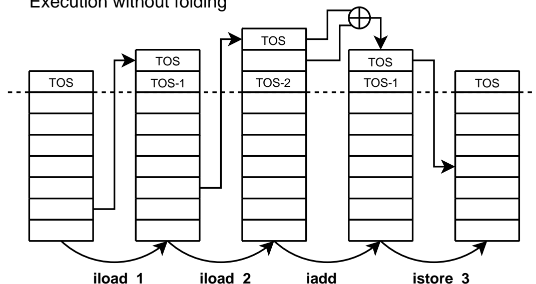
- Created by Sun Microsystems
- Released under the Sun Community Source License
- Well documented
- Flexible, configurable
- Not the whole JVM in silicon
  - Complex instructions are executed via traps
  - Only static class loader
  - No class library
- Extended bytecodes (memory access, special registers, etc.)
- Some hardware has to be designed

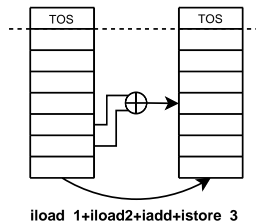
Features

- 6-stage RISC pipeline
- 32-bit integer unit, optional 32-bit FPU
- 64 entry register stack cache
- 0 to 16 Kbyte instruction cache
- 0 to 16 Kbyte data cache
- Native execution of Java bytecodes
- Claimed: 120 K gates, 100 MHz

Instruction Folding

- Top 7 bytes in instruction buffer are examined
- Instructions are classified (loads, stores, ops, nonfoldable)
- Certain patterns can be combined to single instruction, e.g. `iload_1; iload_2; iadd; istore_3`





Outline

1 Motivation

2 Architecture

3 Hardware
- Stack Cache
- Memory and I/O

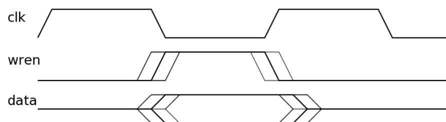
4 Software

5 Evaluation

6 Conclusion

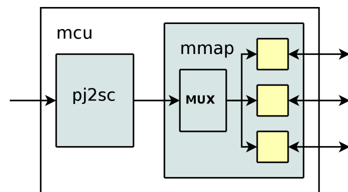
Stack Cache

- 64 32-bit entries
- Conceptually a direct mapped cache / circular buffer
- Must be enabled to ensure correct program behaviour
- Asynchronous RAM (not available in modern FPGAs)
- Five-port RAM (three read ports, two write ports)
- Write pulse only high during low period of clock
  - Both level and edge of write pulse are used
- Data available while write pulse is high
- $\Rightarrow$ Race condition inevitable, tool support needed



Memory and I/O

- picoJava communicates via *bus interface unit* with the environment
- Wrote bridge picoJava ↔ SimpCon
  - JOP uses SimpCon
  - Reuse of components
- xml-schema and tool for generation of memory map
- 8 KB on-chip RAM as boot ROM
- 512 KB SRAM on Altera DE2 board
- UART, LEDs, Timer



Outline

1 Motivation

2 Architecture

3 Hardware

4 Software
- Software
- Bootstrapping
- Loader
- Traps

5 Evaluation

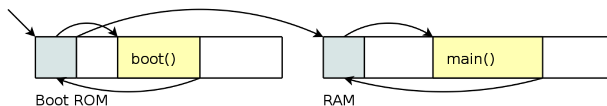
6 Conclusion

Software

- Java assembler and disassembler
- Instruction Accurate Simulator *ias*
- Static class loader
  - Written in C
  - Relies on SPARC as host processor
  - JOP has free class loader written in Java
- Code to handle traps
  - Provided code specific to simulation environment
- Bootstrap program to enable caches etc.
- Java class library
- Classes to access I/O modules

Bootstrapping

- Initialize location of trap table
- Invoke `<clinit>` methods via trap
- Invoke boot()
  - Enable caches (especially stack cache)
  - Enable instruction folding
  - Read program via UART and write to RAM
- Jump to RAM
- Invoke main() via trampoline
- Loop forever



Loader

- Based on loader for JOP
- Statically resolves references
  - Includes all used classes
- Replaces complex instructions with *quick* counterparts
  - Removing overhead for traps
- Create trampolines for booting
- Has to handle instructions which are specific to picoJava

Traps

- Emulate complex instructions, interrupts, exceptions
- Call frame not compatible with regular functions
  - At least parts in assembler
- Located in boot ROM
  - Ensures fast execution
  - Can be overridden if wanted
- Not all traps implemented yet
  - Exceptions, casts not supported
  - No support for floating point operations
- Interrupt latency between 6 and 926 cycles

Outline

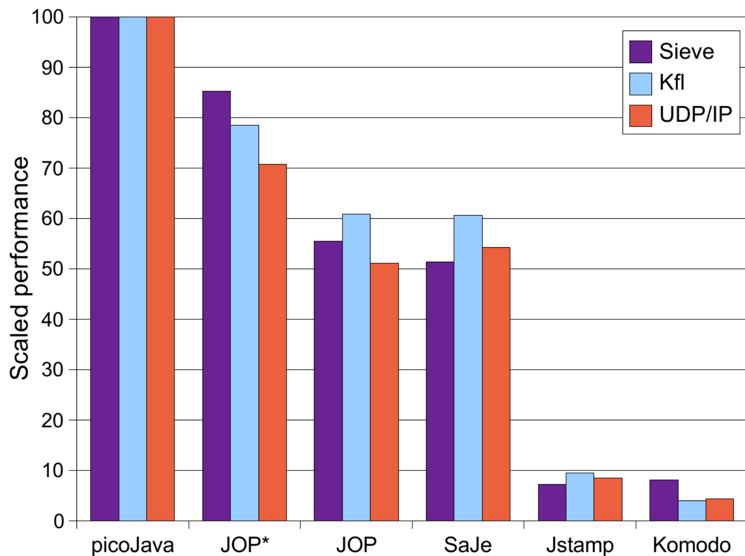
1. Motivation
2. Architecture
3. Hardware
4. Software
5. Evaluation
    - Comparison
    - Benchmarks
6. Conclusion

Comparison of Platforms

picoJava	JOP*	JOP	SaJe	JStamp	Komodo
27.8K LCs	2.9K LCs	1.8K LCs	25K gates	25K gates	2.6K LCs
40 MHz	100 MHz	100 MHz	103 MHz	74 MHz	33 MHz

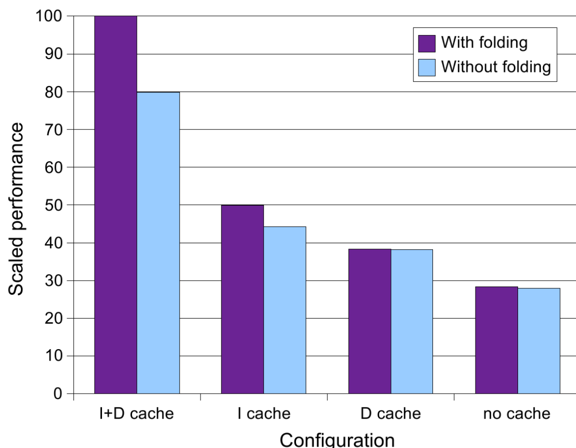
- JOP* refers to the most recent version of JOP
- SaJe and JStamp use aJ-100 and aJ-80 processors, respectivly
- Both 16 KB data and instruction cache
- picoJava uses far more resources than other processors
- picoJava is relatively slow

Application Benchmarks



Different configurations

- Based on geometric mean of Sieve, Kfl and UDP/IP



Outline

1. Motivation
2. Architecture
3. Hardware
4. Software
5. Evaluation
6. Conclusion
   - Status Quo
   - Summary

Status Quo

- Hardware developed
  - Stack cache
  - picoJava $\leftrightarrow$ SimpCon bridge
  - Memory modules and UART
  - Cache memories
- Adapted loader
- Developed bootstrap code
- Implemented most important traps
- Only basic library yet
- No threads, garbage collection, interrupts

Summary

- Complex architecture
  - Instruction folding
  - Stack cache
- Need to write support software
  - Code for bootstrapping
  - Loader
  - Traps
  - Java class library
- High resource consumption
- High performance - Do *fast* and *real-time* match up in this case?

Thank you for your attention!