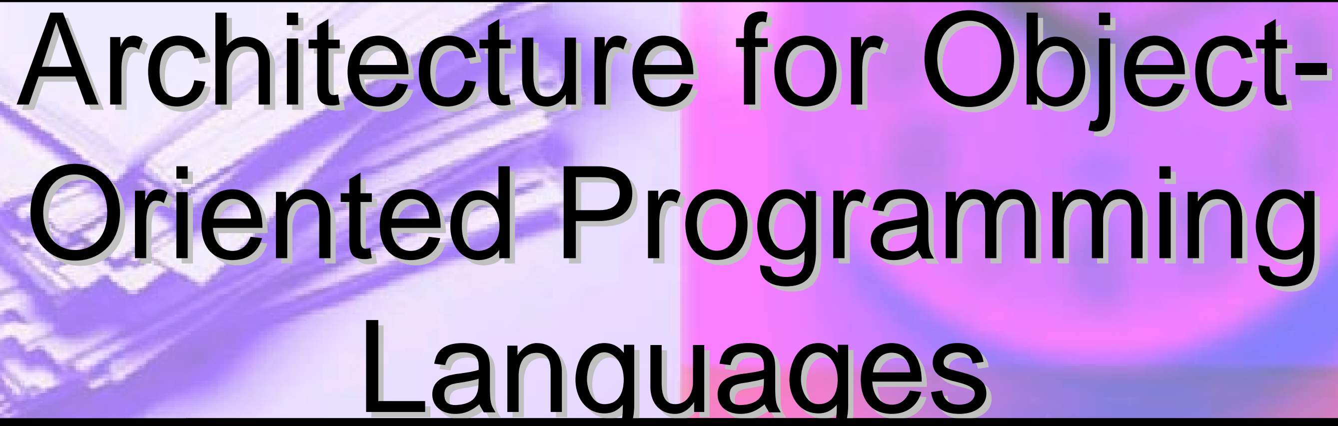

Architecture for Object-Oriented Programming Languages

Martin Schoeberl
Vienna University of Technology, Austria

- OO Instructions in Java
- Amdahl’s law
- Execution time measurements
- Array access in HW
- Measurements
- Conclusion, future work

OO Instructions

- Complex
- Several machine instructions on a RISC
- Microcode in a Java processor
- Dominate the execution time
- HW support can help

Java OO Instructions

- Object and array creation
- Method invocation
- Field access
- Array access

OO Instructions

- Need object (or class) reference
  - Depends on runtime layout
  - Compacting GC needs pointer forwarding or indirection
- Null pointer check
- Array bounds check
- Constant pool access

Possible Optimizations

- Null pointer check w. MMU
  - Not in all embedded processors
- Cache of array size
  - Compiler or hardware
- Method inlining
  - Undo for dynamic class loading
- Hardware implementation

- Execution time of bytecode pairs
  - Almost all bytecode manipulate the stack
  - Additional *undo* bytecode
- Loop with bytecodes under test
- Compensation loop
- Adapts till 1s measured

Micro Benchmark for iaload

- Test loop:

```
public int test(int cnt) {

    int a = 0;
    int i;

    for (i=0; i<cnt; ++i) {
        a += arr[i&0x3ff];
    }
    return a;
}
```

- Overhead loop:

```
public int ovh(int cnt) {

    int a = 0;
    int i;

    for (i=0; i<cnt; ++i) {
        a += abc&0x3ff;
    }
    return a;
}
```

Bytecode for iaload Test

- test loop:

```
9:   iload_2
10:  getstatic   #2;
13:  iload_3
14:  sipush  1023
17:  iand
18:  iaload
19:  iadd
20:  istore_2
```

- overhead loop:

```
9:   iload_2
10:  getstatic   #3;
13:  sipush  1023
16:  iand
17:  iadd
18:  istore_2
```

Bytecode Execution Time

Instruction	JOP	aJile100
iload iadd	2	8
if_icmplt taken	6	18
if_icmplt n/taken	6	14
getfield	22	23
getstatic	15	15
iaload	36	13
invokevirtual	138	115
invokestatic	100	95
invokeinterface	144	153

A Quantitative Approach

- First measure the possible speedup
- Amdahl's Law

$$s = \frac{t_{old}}{t_{new}} = \frac{1}{1 - f_i + \frac{f_i}{s_i}}$$

- Overall speedup s depends on individual speedup s_i and fraction f_i used
  - Rule of diminishing returns

- Instruction count *IC*
- Clocks per instruction *CPI*

$$t_{exe} = IC \times CPI \times t_{clock}$$

  - Keep *IC* and t_{clock} constant
  - Just need CPI_i and IC_i

$$CPI = \frac{\sum_{i=1}^{n} IC_i \times CPI_i}{IC}$$

CPU Performance Equation

- Complete equation
  - CPU pipeline stall cycles
    - Depends on instruction stream
  - Cache miss cycles
- Not easy to measure

$$CPI = \frac{\sum_{i=1}^{n} IC_i \times CPI_i}{IC} + CPI_{stall} + CPI_{miss}$$

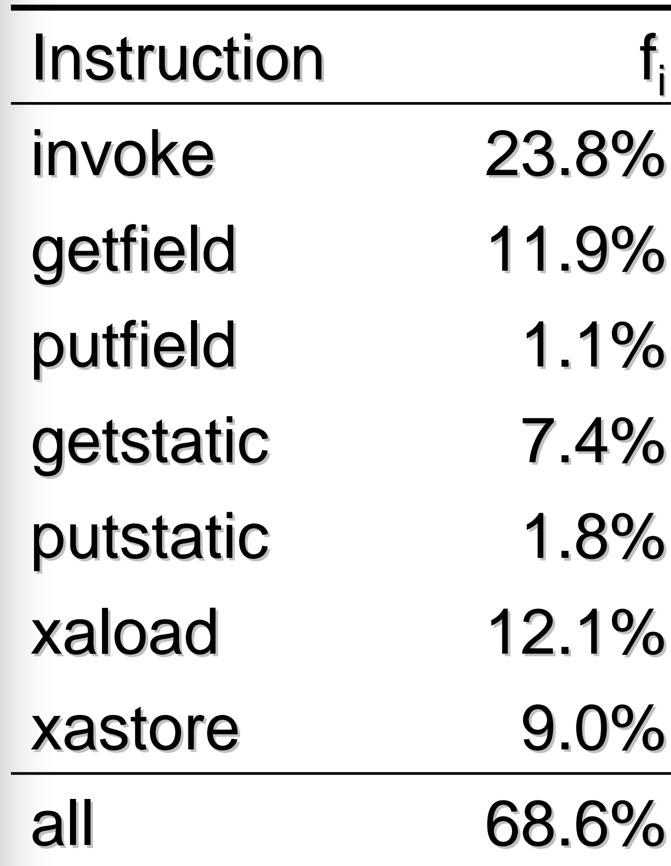
Indirect Approach

- Use Amdahl's Law
  - Increase execution time
  - Easy on FPGA architecture
  - Calculate fraction

$$s = \frac{t_{old}}{t_{new}} = \frac{1}{1 - f_i + \frac{f_i}{s_i}}$$

$$f_i = \frac{s_i}{1 - s_i}\left(\frac{1}{s} - 1\right)$$

Execution Time Fractions



Instruction	f_i
invoke	23.8%
getfield	11.9%
putfield	1.1%
getstatic	7.4%
putstatic	1.8%
xaload	12.1%
xastore	9.0%
all	68.6%

- On JOP
  - 4 KB I cache
  - 512 B D cache
  - 2 cycle mem.
- App. benchmarks

Evaluation

- Implementation of array access
- On a Java processor - JOP
  - FPGA implementation
  - Small core – HW overhead counts
  - 100 MHz at low-cost FPGA
  - 200 MHz at high-perf./cost FPGA
- Possible also on RISC

Array Access Optimization

- Array access
  - Check reference against *null*
  - Check array bounds
  - Read or write data
- Three memory accesses (JOP)
  - Load pointer from handle
  - Load array size
  - Load/store the data

Original Implementation

- In microcode

$$n_{aload} = 32 + 3r_{ws}$$

$$n_{astore} = 35 + 2r_{ws} + w_{ws}$$

- With 2 cycle memory
  - xaload 35 cycles
  - xastore 38 cycles

HW Implementation

- Memory access in HW
  - Still 3 times
- Perform null check in parallel
  - Load of handle does not hurt
- EA calculation parallel to size load
  - Size is part of the handle
- xaload in parallel to bounds check
  - Wrong load does not hurt

- In hardware

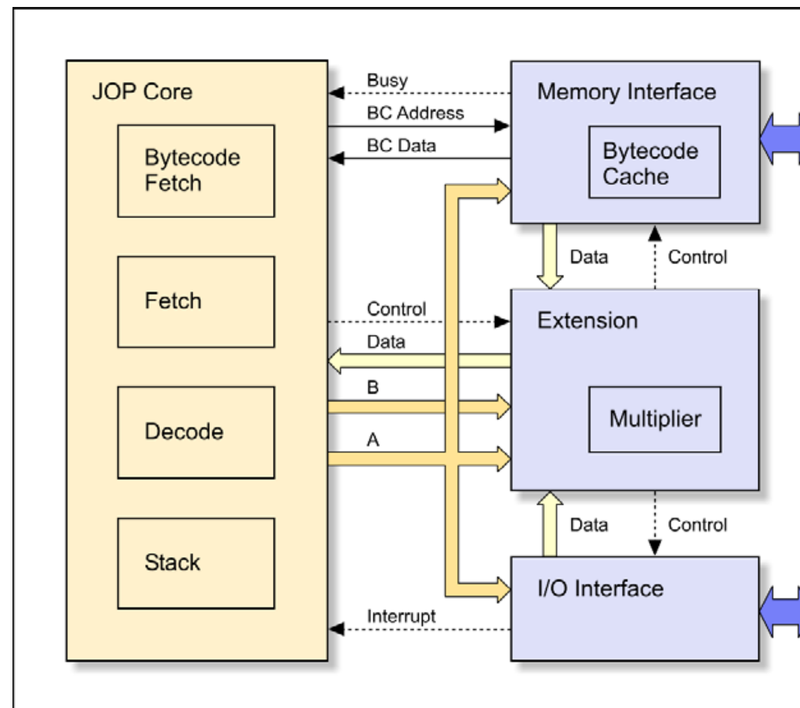
$$n_{aload} = 7 + 3r_{ws}$$

$$n_{astore} = 9 + 2r_{ws} + w_{ws}$$

- With 2 cycle memory
  - xaload 10 cycles
    - Speedup 3.5
  - xastore 12 cycles
    - Speedup 3.17

Implementation

- Change memory interface only
  - Unchanged core pipeline
- Just a few more states and additional microcode instruction



Benchmark Speedup / Area

Benchmark	Speedup / Area
Kfl	1.07
Udplp	1.26
Lift	1.21
geo. Mean	1.18
HW Size	1.08

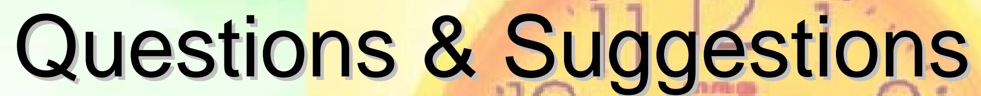
Y-axis: 0.95, 1, 1.05, 1.1, 1.15, 1.2, 1.25, 1.3

Conclusion

- OO instructions are complex
  - Instruction frequency is high
  - Execution time fraction is very high
- Some HW support can help
  - Indirection for moving GC
  - Null pointer check (MMU)
  - Array bounds check
- Can help compiled Java on a RISC

- HW support for invoke
  - Complete in HW too expensive
  - Find a good tradeoff
  - Change of core pipeline
- Try the array HW on a RISC
  - LEON (SPARC V8)
  - MIPS
  - Challenge: find a JIT or AOT that supports it (CACAO?, your JVM?)

Thank You


Questions & Suggestions