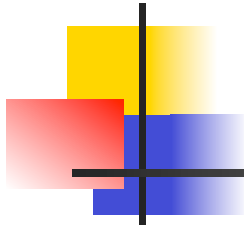


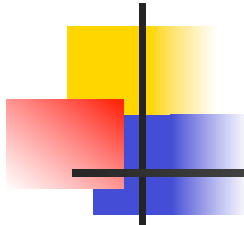
A Time Predictable Instruction Cache for a Java Processor

Martin Schoeberl



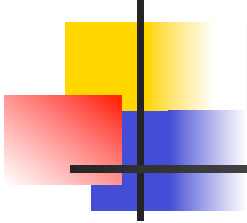
Overview

- Motivation
- Cache Performance
- Java Properties
- Method Cache
- WCET Analysis
- Results
- Conclusion, Future Work



Motivation

- CPU speed – memory access
- Caches are mandatory
- Caches improve average execution time
- Hard to predict WCET values
- Cache design for WCET analysis



Execution Time

$$t_{exe} = (CPU_{clk} + MEM_{clk}) \times t_{clk}$$

$$CPU_{clk} = IC \times CPI_{exe}$$

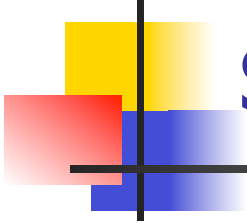
$$MEM_{clk} = Misses \times MP_{clk}$$

$$= IC \times Misses / Instruction \times MP_{clk}$$

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

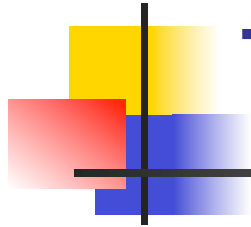
$$CPI = CPI_{exe} + CPI_{IM} + CPI_{DM}$$

H&P: CA:AQA



Misses per Instruction is too simple

- Architecture dependent (RISC vs. JVM)
 - Different instruction length
 - Different load/store frequencies
- Block size dependent
 - Lower for larger blocks
- Memory access time
 - Latency
 - Bandwidth



Two Cache Properties

- MBIB and MTIB

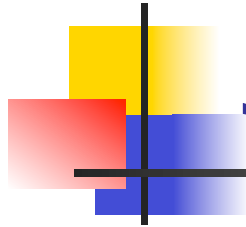
MBIB = Memory bytes read / Instruction byte

MTIB = Memory transactions / Instruction byte

- Reflects main memory properties

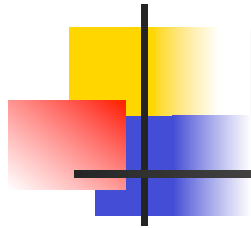
$$IM_{clk} / IB = MTIB \times Latency + MBIB / Bandwidth$$

$$CPI_{IM} = IM_{clk} / IB \times Instruction\ length$$

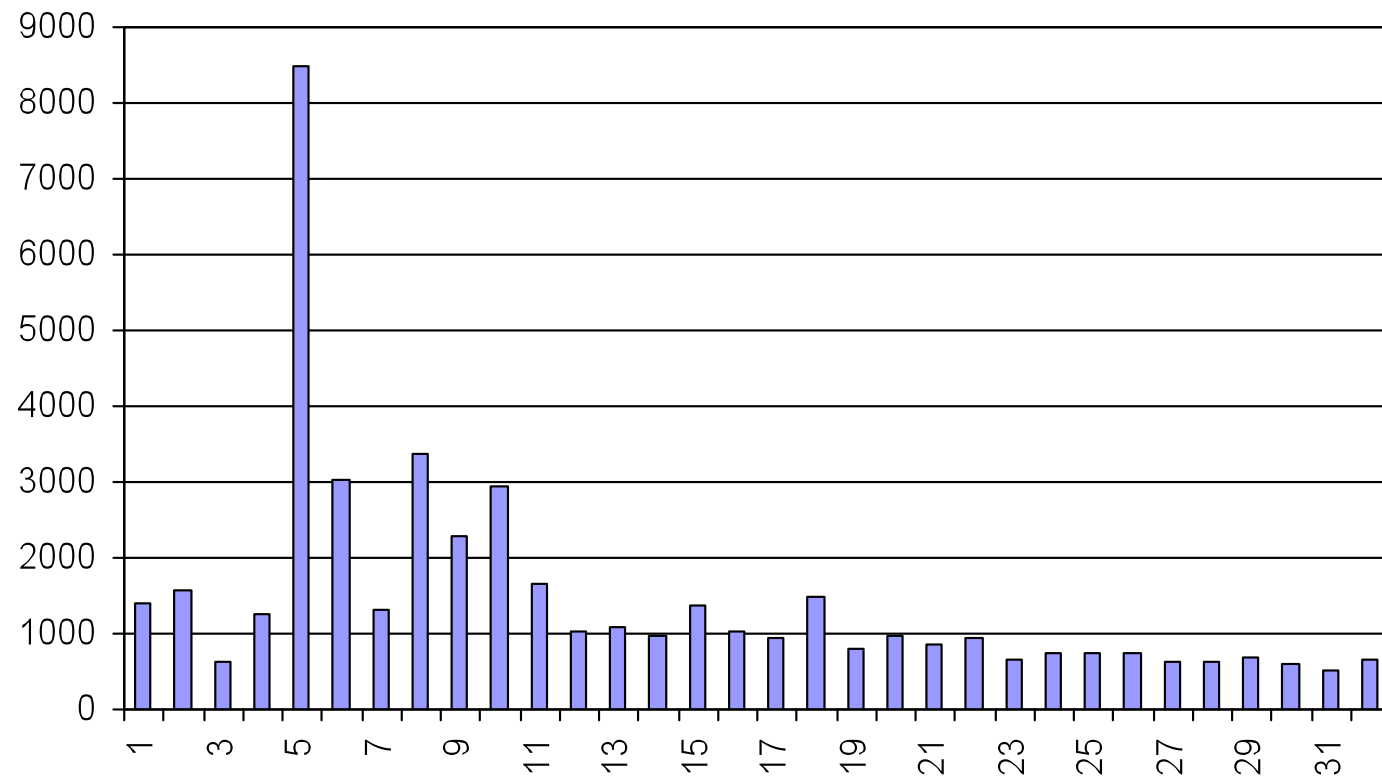


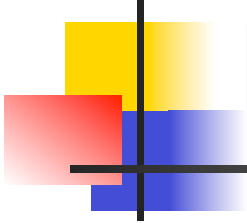
JVM Properties

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



Method Sizes (rt.jar)





Bytecodes for a Getter Method

```
private int val;
```

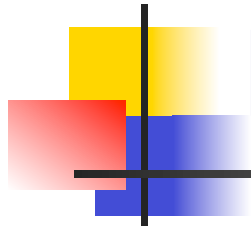
```
public int getVal() {  
    return val;  
}
```

```
public int getVal();  
Code:
```

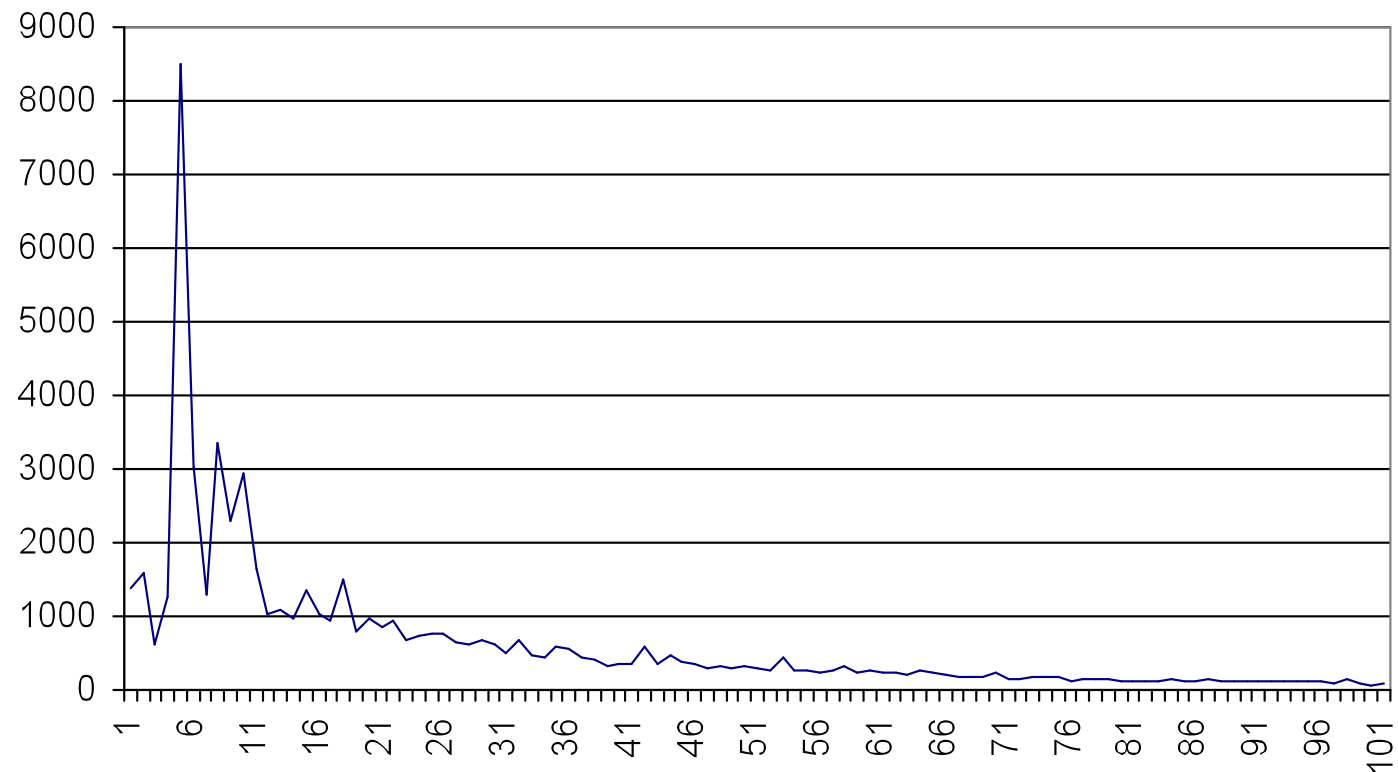
```
0: aload 0
```

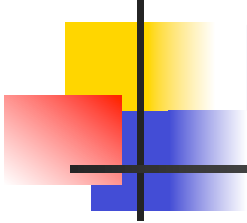
```
1: getfield #2;    //Field val:I
```

```
4: ireturn
```



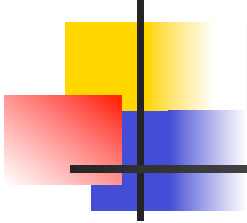
Method Sizes (rt.jar)





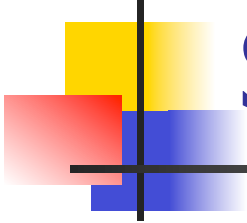
Method Sizes cont.

- Runtime library rt.jar (1.4):
 - 71419 methods
 - Largest: 16706 Bytes
 - 99% $\leq$ 512 Bytes
 - Larger methods are class initializer
- Application - javac: 98% $\leq$ 512 Bytes



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

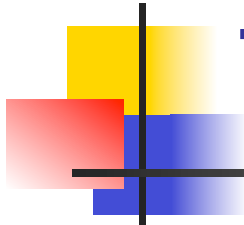


Single Method Cache

- Very simple WCET analysis
- High overhead:
 - Partially executed method
 - Fill on every call and return

```
foo() {  
    a();  
    b();  
}
```

	Block 1	Cache
foo()	foo	load
a()	a	load
return	foo	load
b()	b	load
return	foo	load

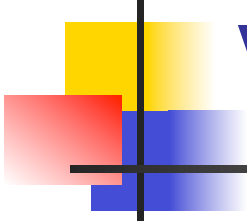


Two Block Cache

- One method per block
- Simple WCET analysis
- LRU replacement
- 2 word tag memory

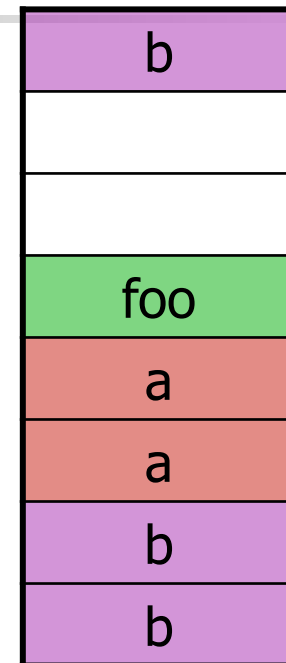
```
foo() {  
    a();  
    b();  
}
```

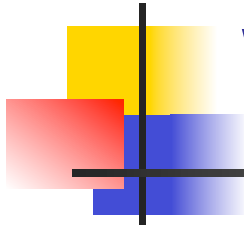
	Block 1	Block 2	Cache
foo()	foo	-	load
a()	foo	a	load
return	foo	a	hit
b()	foo	b	load
return	foo	b	hit



Variable Block Cache

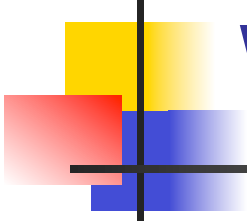
- Whole method loaded
- Cache is divided in blocks
- Method can span several blocks
- Continuous blocks for a method
- Replacement
 - LRU not useful
 - *Free* running next block counter
 - Stack oriented next block
- Tag memory: One entry per block





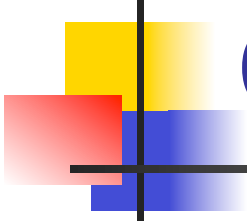
WCET Analysis

- Single method
 - Trivial – every call, return is a miss
 - Simplification: combine call and return load
- Two blocks:
 - Hit on call: Only if the same method as the last called – loop
 - Hit on return: Only when the method is a leave in the call tree – always a hit



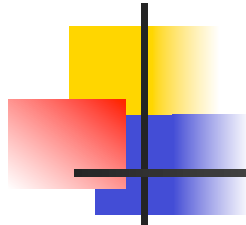
WCET Analysis Var. Blocks

- Part of the call tree
- Method length determines cache content
- Still simpler than direct-mapped
 - Call tree instead of instruction address
 - Analysis only at call and return
 - Independent of link addresses



Caches Compared

- Embedded application benchmark
 - Cyclic loop style
 - Simulation of external events
 - Simulation of a Java processor (JOP)
- Different memory systems:
 - SRAM: $L = 1$ cycle, $B = 2$ Bytes/cycle
 - SDRAM: $L = 5$ cycle, $B = 4$ Bytes/cycle
 - DDR: $L = 4.5$ cycle, $B = 8$ Bytes/cycle



Direct-Mapped Cache

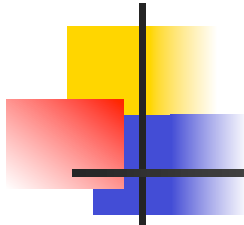
Plainest WCET target, size: 2KB

Line size	MBIB	MTIB	SRAM	SDR	DDR
8	0.17	0.022	0.11	0.15	0.12
16	0.25	0.015	0.14	0.14	0.10
32	0.41	0.013	0.22	0.17	0.11

MBIB = Memory bytes read / Instruction byte

Memory read in clock cycles / Instruction byte

MTIB = Memory transactions / Instruction byte



Fixed Block Cache

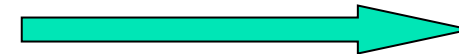
Cache size: 1, 2 and 4KB

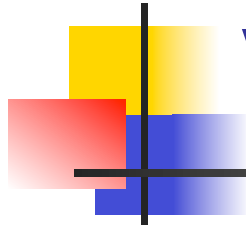
Type	MBIB	MTIB	SRAM	SDR	DDR
Single	2.31	0.021	1.18	0.69	0.39
Two	1.21	0.013	0.62	0.37	0.21
Four	0.90	0.010	0.46	0.27	0.16

MBIB = Memory bytes read / Instruction byte

MTIB = Memory transactions / Instruction byte

Memory read in clock cycles / Instruction byte

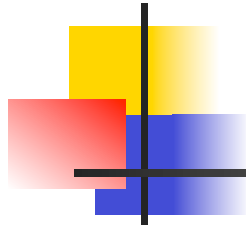




Variable Block Cache

Cache size: 2KB

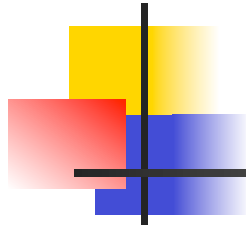
Block count	MBIB	MTIB	SRAM	SDR	DDR
8	0.73	0.008	0.37	0.22	0.13
16	0.37	0.004	0.19	0.11	0.06
32	0.24	0.003	0.12	0.08	0.04
64	0.12	0.001	0.06	0.04	0.02



Caches Compared

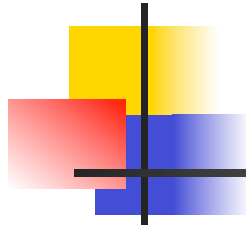
Cache size: 2KB

Type	MBIB	MTIB	SRAM	SDR	DDR
VB 16	0.37	0.004	0.19	0.11	0.06
VB 32	0.24	0.003	0.12	0.08	0.04
DM 8	0.17	0.022	0.11	0.15	0.12
DM 16	0.25	0.015	0.14	0.14	0.10



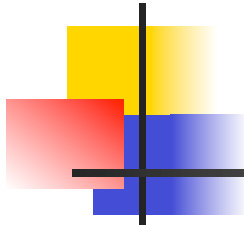
Summary

- Two cache properties: MBIB & MTIB
- JVM: short methods, relative branches
- Single Method cache
 - Misses only on call and return
- Caches compared
 - Embedded application
 - Different memory systems



Future Work

- WCET analysis framework
- Compare WCET values with a traditional cache
- Different replacement policies
- Don't keep short methods in the cache



Further Information

- Reading
 - JOP Thesis: p 103-119
 - Martin Schoeberl. A Time Predictable Instruction Cache for a Java Processor. In *Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2004)*, 2004.
- Simulation
 - .../com/jopdesign/tools
- Hardware
 - .../vhd1/core/cache.vhd
 - .../hdl/memory/mem_sc.vhd