Increasing the Efficiency of an Embedded Multi-Core Bytecode Processor Using an Object Cache

Martin Zabel,
Thomas B. Preußer,
Rainer G. Spallek

JTRES’12 25.10.2012

Martin.Zabel@tu-dresden.de http://vlsi-eda.inf.tu-dresden.de
Outline

1. Motivation
2. Related Work
3. Heap-Access Analysis
4. Implementation & Results
5. Conclusion
Motivation

Why Java?
- Object orientation, portability
- Automatic memory management, security
- Support for thread parallelization

Why Java-(bytecode-)processor?
- Native execution of Java-bytecode
  - no OS, no interpretation, no re-compilation
- Real-time
- Suited for embedded systems with limited resources

Why multi-core processors?
- Power consumption increases over-proportional with clock frequency.
- Use thread-level parallelism instead.
Java Multi-Core Processor

**Examples:** JopCMP, jamuth, REALJava and SHAP

**Common property:** central shared heap for all cores

**SHAP Multi-Core:**
- Local stack-memory per core
- Method-cache per core
- Pipelining of heap-accesses
- Concurrent GC for real-time apps
- Maximum speed-up of 8 for programs with an above-average number of memory accesses [1]
- CLDC, constant-time interface method dispatch, …

---

Martin Zabel

Increasing the Efficiency of an Embedded Multi-Core Bytecode Processor using an Object Cache
Further reduce the demands on the heap memory interface to achieve higher speed-ups through thread-level parallelism.
Outline

1 Motivation
2 Related Work
3 Heap-Access Analysis
4 Implementation & Results
5 Conclusion
Related Work

Common solution for object-oriented processors:
Cache for objects in analogy to data-caches [2] [3]

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object Reference</td>
<td>Offset</td>
</tr>
<tr>
<td></td>
<td>Object Content</td>
</tr>
</tbody>
</table>

Especially for real-time systems:
Separate Caches for different data areas [4]:
- Classic data cache for data at static addresses (e.g. class data)
- Object-cache for data at dynamic addresses (e.g. objects)
Indirect Object-Addressing

**Problem** of JopCMP and SHAP:
- Object-table stored in external memory.
- Additional latency for each heap-access.
- Additional demand on memory bandwidth.

**Solution:**
- Translation look-aside buffer (TLB) [1]
- Virtually indexed object-cache [2]
Cache Coherence

**Problem:** Coherence of distributed caches

**Advantage of the Java Virtual Machine:**
Synchronization only when [5]
- entering a critical section, or
- accessing a “volatile” variable.
Outline

1 Motivation
2 Related Work
3 Heap-Access Analysis
4 Implementation & Results
5 Conclusion
Heap-Access Analysis

Evaluation:

- Benchmark suite JemBench (Version 2.0) [9], all except microkernel benchmarks
- SHAP Multi-Core with 1 core and trace unit [1]
- Recording of executed bytecodes and memory accesses
Heap-Access Analysis

Memory Bandwidth Utilization [%]

- **Data Accesses (Core Itself)**
- **Bytecode Fetches (Method Cache)**
- **Accesses for Memory Management**

**Benchmarks**:
- AES
- Bubble Sort
- Kfl
- Lift
- Matrix Mul
- N-Queens
- Sieve
- Udplp

11% highlighted for Matrix Mul benchmark.
Heap-Access Analysis

Evaluation:

- Benchmark suite JemBench (Version 2.0) [9]
- SHAP Multi-Core with 1 core and trace unit [1]
- Recording of executed bytecodes and memory accesses

Results:

1. Most frequently object-accesses are reads on arrays and member variables.
2. 80% of all object accesses concentrate on 6 objects.
3. Frequent access onto the first user-specific object offsets (-2 and 1)

→ Further evaluation: small full-associative cache for each core

1 (already accounts for implicit reads of array length)
Small Full-associative Local Cache

Storing only invariant data:
- Would require no extra logic for cache coherency.
- In general, only class information pointer and array size are invariant.

→ Significant reduction only for array-intensive programs.

<table>
<thead>
<tr>
<th>Function</th>
<th>% of all memory accesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>BubbleSort</td>
<td>33%</td>
</tr>
<tr>
<td>Sieve</td>
<td>20%</td>
</tr>
<tr>
<td>MatrixMul</td>
<td>17%</td>
</tr>
</tbody>
</table>

Write-through instead of write-back:
- No special GC interaction required.
- Simple cache coherence: Invalidate cache when
  • entering a critical section, or
  • accessing a “volatile” variable.
Outline

1 Motivation
2 Related Work
3 Heap-Access Analysis
4 Implementation & Results
5 Conclusion
Cache Design (1)

Cache integration: into memory manager port

Core modifications:
- Bytecode to access volatile variables
- Microcode to invalidate cache
Cache Design (2)

Features:
- Address & Offset-Cache (AOC) with write-through, LRU-strategy
- 1 valid-bit per cached word
- Configurable: # of cache lines, cached offsets

Disadvantage:
Additional latency of 1 clock cycle during cache miss
Cache Configuration

Huge configuration space:
- Cache lines: $l > 0, l \in N$
- Offsets: no (address only), range $[-x, y]$ with $x, y \in N, x > 0$

But synthesis for $n = 1, 2, ..., 18$ cores too expensive.

Search for good initial configuration.

Configuration space exploration:
- Baseline design for comparison: TLB with 2 entries
- Benchmarks:
  - JemBench
  - JavaGrande Framework [7]: HeapSort, SparseMatMult (with integers)
- Platform:
  - SHAP on Virtex-5 FPGA XC5VLX110T
  - Same clock frequency of 80 MHz as baseline design
Setup: 1 Core, Cache Address Only

![Graph showing relative performance of different benchmarks with varying cache configurations.](image)

- Use 8 lines

Martin Zabel
Increasing the Efficiency of an Embedded Multi-Core Bytecode Processor using an Object Cache
Setup: 1 Core, 8 Lines

Relative Performance

Cache Configuration


SparseMatmultInt
MatrixMul (N=20)
Lift
AES
UdpIp
NQueens
Kfl
HS
Sieve
BS

Cache Offsets -2 & 1

Martin Zabel
Increasing the Efficiency of an Embedded Multi-Core Bytecode Processor using an Object Cache

Folie 20 von 28
Speed-Up of JemBench MatrixMul

Matrix multiplication:
Default with 20x20 matrix

Reference value:
Single-core performance

Ideal speed-up:
10 for \( n \geq 10 \) cores

Speed-up maximum:
8,3 \( \rightarrow \) 9,4
@ 10 cores
Speed-Up of JemBench MatrixMul (N=90)

**Matrix multiplication:**
Extended to 90x90 matrix

**Reference value:**
Single-core performance

**Ideal speed-up:**
15 for 15--17 cores

**Speed-up maximum:**
9.5 ⇒ 14.0 @ 15 cores
Speed-Up of JGF SparseMatMult with Integer

**Matrix multiplication:**
Reduced to 4.000x4.000 matrix with 16.000 non-zero integer elements

**Reference value:**
Single-core performance

**Speed-up maximum:**
7.6 $\rightarrow$ 14.7 / 15.5
@ 17 cores
FPGA Resource Usage

Synthesis results for 8 cache lines, offsets -2 & 1:
- Up to 17 cores at 80 MHz (same as baseline design) fit on XC5VLX110T
- Additional hardware resources **per core**:
  - For up to 15 cores:

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Register</th>
<th>LUTs</th>
<th>RAMB36</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address Only</td>
<td>+8%</td>
<td>+3%*</td>
<td>-</td>
</tr>
<tr>
<td>Offsets -2 &amp; 1</td>
<td>+9%</td>
<td>+8%*</td>
<td>-</td>
</tr>
<tr>
<td>Offsets [-8,7]</td>
<td>+17%</td>
<td>+9%</td>
<td>+1</td>
</tr>
</tbody>
</table>

* = includes distributed RAM

- 16 and 17 cores: over-proportional due to high FPGA utilization.
Outline

1 Motivation
2 Related Work
3 Heap-Access Analysis
4 Implementation & Results
5 Conclusion
Conclusion

**Baseline design:** SHAP Multi-Core with 2-lines TLB

**Realized object-cache:**
- Full-associative combined address and offset cache
- Configurable in # of lines and cached offsets

**Results:**
- With 4 cache-lines: Amortization of additional latency
- With 8 cache-lines:
  - Speed-up maximum increased from formerly 8—9 to now 14.
  - Single-core performance increased by 12 to 21%.

⇒ Up to twice absolute compute performance.
⇒ Requires only 9% more hardware resources.
Thank you for your attention!

Questions?
Selected Literature

1. Zabel, Martin: Effiziente Mehrkernarchitektur für eingebettete Java-Bytecode-Prozessoren, Dresden, Technische Universität, Diss., 2012


