Just-in-Time HW/ISA/SW Co-optimization Techniques for SoC

Murakami, Kazuaki
Computing and Communication Center, Kyushu University | Faculty of Information Science and Electrical Engineering, Kyushu University

Mauro Goulart Ferreira, Victor
Faculty of Information Science and Electrical Engineering, Kyushu University

http://hdl.handle.net/2324/9111
“Just-in-Time HW/ISA/SW Co-optimization Techniques for SoC” (Part 1)

Kazuaki J. Murakami (*1)
Victor Mauro Goulart Ferreira (*2)

*1: Director of Computing & Communications Center, Kyushu University
*2: Ph.D. Candidate, Dept. of Informatics, Kyushu University

E-mail: arch@i.kyushu-u.ac.jp or kjm@acm.org
Tutorial Outline

- Part 1 (8:00-9:30)
  - Overview of JIT HW/ISA/SW Co-optimization
  - Functionality Morphing
- Part 2 (17:30-19:00 ➔ 15:00-16:30)
  - On-demand Recomputation
What is Just-in-Time HW/ISA/SW Co-optimization?

- Just-in-time?
  - **Dynamic**: Optimize SoC,
    - After SoC’s are shipped to the market
    - While SoC’s are used in the field
  - **Online**: Optimize SoC,
    - In parallel with the execution of application programs
    - In advance of the completion of the execution
  - **Adaptive**: Optimize SoC repeatedly,
    - In the form of a feedback loop
    - Until the system reaches some stable state

- HW/ISA/SW Co-optimization?
  - Optimize HW/ISA/SW of SoC coordinately
JIT HW/ISA/SW Co-optimization would work like ...

Application programs are running...

Application programs are under optimization...

Hints for Optimization

Binary Rewriting

HW/ISA/SW Co-optimization

Online Profiling

Instruction Execution

Profiler

Accelerator

Processor Core

Target Programs

Runtime Software

ISA

Instruction Execution

Profiler

Accelerator

Processor Core

Target Programs

Runtime Software

Instruction Execution

ISA Redefinition

Mode Control

Hardware Reconfiguration
Analogy: Formula 1

The car (=application program) is running.

The pit crew (=SysteMorph software) is monitoring the behavior of the car.

Once the pit crew finds any hints for optimization, the car pits in.

After the optimization, the car returns to the race.

The car is now under optimization.
JIT HW/ISA/SW Co-optimization and Other Optimization Techniques

When?

Runtime

Compile Time

Design Time

What?

Online Profiling & Optimization

CO: Compiler Optimization
DCO: Dynamic Compilation/Optimization
EH: Evolvable Hardware
RC: Reconfigurable Computing

Offline Profiling & Optimization

SW

HW

HW&SW

Dynamic HW Reconfiguration

JIT HW/ISA/SW Co-optimization

and Other Optimization Techniques

Kazuaki Murakami & Victor Mauro Goulart Ferreira ©2000-2004
Some References

- RC (Reconfigurable Computing)
  - FPL (http://fpl.org)
  - RAW (http://www.ece.lsu.edu/vaidy/raw05/)
- EH (Evolvable Hardware)
  - EH (http://ic.arc.nasa.gov/ic/eh2000)
- DCO (Dynamic Compilation/Optimization)
  - Dynamo (HP)
  - Code Morphing Software (Transmeta)
  - JIT Compilers
  - CGO (http://www.cgo.org)
- JIT HW/ISA/SW Co-optimization
  - SysteMorph (Kyushu Univ.)
Why JIT HW/ISA/SW Co-optimization?

**Motivations**

- "Time-To-Market": Need to reduce the TAT of SoC design
  - No extra time for optimization
- "Time-In-Market": Need to extend the product lifetime of SoC's
  - Extend the application areas of a single SoC design

**Goals**

- Enable SoC's to optimize and customize themselves in the field according as the users' behavior or favorite
  - Move the optimization phase from the SoC design time to the SoC operation time
- Allow SoC's to modify their functionality in the field
  - Make them tolerant of any specification changes in future
Tutorial Outline

- Part 1 (8:00-9:30)
  - Overview of JIT HW/ISA/SW Co-optimization
    - Definition and motivations
    - Implementation frameworks
    - Implementation examples
  - Functionality Morphing

- Part 2 (17:30-19:00)
  - On-demand Recomputation
How Can You Implement JIT HW/ISA/SW Co-optimization?

Application programs are running...

Online Profiling

Target Programs

Runtime Software

Profiler

Instruction Execution

Binary Rewriting

Target Programs

Runtime Software

Profiler

Mode Control

HW/ISA/SW Co-optimization

Application programs are under optimization...

Hints for Optimization

Instruction Execution

ISA Redefinition

Hardware Reconfiguration

Processor Core

Accelerator

Processor Core

Accelerator

ISA

How Can You Implement JIT HW/ISA/SW Co-optimization?
When Can You Optimize Your SoC?

- Optimize in idle time
- Optimize in sleep time

Zzzzzz...
Where Can You Optimize Your SoC?

- Optimize on your hand
- Optimize at some remote “Pit Center”
Tutorial Outline

- Part 1 (8:00-9:30)
  - Overview of JIT HW/ISA/SW Co-optimization
    - Definition and motivations
    - Implementation frameworks
  - Implementation examples
  - Functionality Morphing
- Part 2 (17:30-19:00)
  - On-demand Recomputation
How Can You Optimize Your SoC?
- Some Examples -

- For higher performance
  - Functionality Morphing
    - Identify some frequently executed things (e.g., hot spots, hot paths, hot instruction sequence), and then offload the execution of these from the main processor to attached accelerator (e.g., wide-issue VLIW, dynamic reconfigurable fabric)
  - On-demand Recomputation
    - Identify some frequently cache-miss causing LOAD instructions (called delinquent loads or critical loads), then replace these LOADs with the corresponding recomputation codes

- For lower power consumption
  - Dynamic Scaling of:
    - Power-supply voltage (DVS: dynamic voltage scaling)
    - Hardware size to use (# of FU's, cache size, and so on)
(1) Monitor program path on 1-way scalar processor (SP)

(2) Detect and predict hot program path

(3) Transform the scalar code of the hot path into a software-pipelined code for n-way VLIW

(4) Load the software-pipelined code into the I$ of the n-way VLIW

(5) Replace the hot path with a co-processor call
Functionality Morphing - Another Case for Hot-Path Offloading -

Application programs are running...

(1) Monitor program path

(2) Detect and predict hot program path

(3) Transform the function of the hot path into a logic function

(4) Reconfigure the hardware of a reconfigurable co-processor (RCP)

(5) Replace the hot path with a co-processor call

Application programs are under optimization...

Processor Core

Profiler

Reconfigurable Fabric

Instruction Execution

Profiler

Reconfigurable Fabric

Instruction Execution

Target Programs

Runtime Software

Target Programs

Runtime Software

ISA

(5) Replace the hot path with a co-processor call

(4) Reconfigure the hardware of a reconfigurable co-processor (RCP)
Application programs are running...

(1) Monitor D-cache misses
(2) Detect and identify critical LOADs
(3) Monitor STORE execution
(4) Identify STOREs corresponding to critical LOADs
(5) Generate the recomputation (RC) code computing the values of the data which the delinquent LOADs would load
(6) Replace the delinquent LOADs with the RC code

Application programs are under optimization...

Instruction Execution

Target Programs
Runtime Software

Processor Core
Cache Miss Profiler
STORE Filter

Instruction Execution

Target Programs
Runtime Software

Processor Core
Cache Miss Profiler
STORE Filter
Tutorial Outline

- **Part 1 (8:00-9:30)**
  - ✔ Overview of JIT HW/ISA/SW Co-optimization
  - Functionality Morphing
    - Concept
    - Online hot-path profiling
    - Dynamic trace-based software pipelining
    - Hyperscalar processor (as an accelerator)
    - Performance issues

- **Part 2 (17:30-19:00)**
  - On-demand Recomputation
Functionality Morphing
- A Case for Hot-Path Offloading -

Application programs are running...

1. Monitor program path on 1-way scalar processor (SP)
2. Detect and predict hot program path

Application programs are under optimization...

3. Transform the scalar code of the hot path into a software-pipelined code for n-way VLIW
4. Load the software-pipelined code into the I$ of the n-way VLIW
5. Replace the hot path with a co-processor call

Functionality Morphing -- A Case for Hot-Path Offloading --

-- A Case for Hot-Path Offloading --

Kazuaki Murakami & Victor Mauro Goulart Ferreira ©2000-2004
Program Paths and Hot Paths

Path: signature
ABDG: A.0101
ABDGJ: A.01001
ABDHJ: A.01111
ACEIJ: A.10111
ACFIJ: A.11111
Functionality Morphing
- Offload Hot Paths from Main Processor to Co-Processor -

- **BEFORE**: Original binary code

  ```
  : ...
  load
  load
  mul
  add
  store
  br ...
  : ...
  ```

  Replace the code sequence of a hot path with a single CP call instruction.

- **AFTER**: Modified binary code

  ```
  : ...
  call CP
  nop
  nop
  nop
  nop
  br ...
  : ...
  ```

  Copy the code sequence in another memory region for a "housekeeping" purpose.

  - Activate the CP
  - Return from the CP
Tutorial Outline

- **Part 1 (8:00-9:30)**
  - Overview of JIT HW/ISA/SW Co-optimization
  - Functionality Morphing
    - Concept
    - Online hot-path profiling
  - Dynamic trace-based software pipelining
  - Hyperscalar processor (as an accelerator)
  - Performance issues

- **Part 2 (17:30-19:00)**
  - On-demand Recomputation
Functionality Morphing
- How Do You Predict Hot Paths? -

Application programs are running...

(1) Monitor program path on 1-way scalar processor (SP)
(2) Detect and predict hot program path

Instruction Execution

ISA

Target Programs

Runtime Software

1-way SP

Hot Path Profiler

n-way VLIW

Application programs are under optimization...

(3) Transform the scalar code of the hot path into a software-pipelined code for n-way VLIW

(4) Load the software-pipelined code into the I$ of the n-way VLIW

(5) Replace the hot path with a co-processor call

Instruction Execution

Target Programs

Runtime Software

1-way SP

Hot Path Profiler

n-way VLIW

Functionality Morphing -- How Do You Predict Hot Paths? --
How Do You Predict Hot Paths?
- Online Hot-Path Profiling -

- **Offline Profiling**
  - Summary of program behavior based on whole program trace
  - Good for:
    - CO (Compiler Optimization)
    - RC (Reconfigurable Computing)

- **Online Profiling**
  - Prediction based on current execution window of program
  - Good for:
    - DCO (Dynamic CO)
    - JIT HW/ISA/SW Co-optimization

How Do You Predict Hot Paths?
- Online Hot-Path Profiling -

- **Offline Profiling**
  - Summary of program behavior based on whole program trace
  - Good for:
    - CO (Compiler Optimization)
    - RC (Reconfigurable Computing)

- **Online Profiling**
  - Prediction based on current execution window of program
  - Good for:
    - DCO (Dynamic CO)
    - JIT HW/ISA/SW Co-optimization

Current Execution Window

Whole Program Trace
Online Hot-Path Profiling Algorithms

- (Offline Profiling)
  - Ball-Larus’s path profiling [Ball: MICRO1996]

- Online Profiling
  - Dynamo’s NET prediction [Duesterwald: ASPLOS2000]
  - SysteMorph’s branch-history based hot-path prediction [Yoshimatsu: HPC Asia2004]
Branch-History Based Hot-Path Prediction

- Profile the history of branch instruction’s behaviors (taken or not-taken, branch target)
- If the execution frequency at a path head exceeds the threshold, select the path head (“A” in the figure) as a candidate of the hot path head
- Traverse the object code, starting with the candidate (“A”), based on the branch history, and predict the hot path

[Yoshimatsu et al., HPC Asia 2004]
How to Reduce Runtime Overhead - HW Assist & Runtime SW -

**Selector**
- Buffers all the executed branch instructions
- Sends them as tuples to HW compressor

**Tuple**
- `<bia, bta>`
  - `bia`: branch instruction address
  - `bta`: branch target address

**HW Compressor**
- Counts the tuples, and compresses them in the message form
- Sends the messages to SW profiler

**Messages**
- `(tuple, count)`

**SW Profiler**
- Accumulates the messages
- Predicts hot paths
How Many Hot Paths Are Detected?

Number of Identified Hot Path

Number of Different Hot Paths

Benchmarks:
- automotive/basemud
- automotive/bitcount
- automotive/flight
- consumer/peg
- network/netfix
- network/dijkstra
- security/diablo
- security/ripndad
- security/java
- telecom/fft
How Do Hot Paths Appear?

- Application: MP3 encode

![Graphs showing executed instruction count and number of instructions on each hot path.](image)
Tutorial Outline

- **Part 1 (8:00-9:30)**
  - Overview of JIT HW/ISA/SW Co-optimization
  - Functionality Morphing
    - Concept
    - Online hot-path profiling
    - Dynamic trace-based software pipelining
  - Hyperscalar processor (as an accelerator)
  - Performance issues

- **Part 2 (17:30-19:00)**
  - On-demand Recomputation
Functionality Morphing
- How Do You Speedup Hot-Paths? -

Application programs are running...

1. Monitor program path on 1-way scalar processor (SP)

2. Detect and predict hot program path

3. Transform the scalar code of the hot path into a software-pipelined code for n-way VLIW

(1-way SP) Instruction Execution

Target Programs

Runtime Software

Instruction Execution

(1-way SP) Instruction Execution

Target Programs

Runtime Software

1. Monitor program path on 1-way scalar processor (SP)

2. Detect and predict hot program path

3. Transform the scalar code of the hot path into a software-pipelined code for n-way VLIW

4. Load the software-pipelined code into the I$ of the n-way VLIW

5. Replace the hot path with a co-processor call

Application programs are under optimization...

Functionality Morphing

-- How Do You Speedup Hot-Paths? --
How Do You Speedup Hot Paths?
- Dynamic Trace-Based Software Pipelining-

**BEFORE** software pipelining

**AFTER** software pipelining

---

Kazuaki Murakami & Victor Mauro Goulart Ferreira ©2000-2004
Dynamic Trace-Based Software Pipelining (1)

Basic Blocks

Hot Path

Other Paths

Detect & extract hot path

Generate software-pipelined VLIW code

A if(1)

B
d if(2)

C if(3)

D

E

F

G

H

I

J

Trace

A if(1)

B

d if(2)

D

if(2)

H

J

J

H

d if(2)

D

B

A

if(1)
Dynamic Trace-Based Software Pipelining (2)

Add housekeeping code

housekeeping code
Tutorial Outline

- **Part 1 (8:00-9:30)**
  - Overview of JIT HW/ISA/SW Co-optimization
    - Functionality Morphing
      - Concept
      - Online hot-path profiling
      - Dynamic trace-based software pipelining
      - Hyperscalar processor (as an accelerator)
    - Performance issues

- **Part 2 (17:30-19:00)**
  - On-demand Recomputation
Functionality Morphing
- What Architecture Enables It? -

Application programs are running...

1. Monitor program path on 1-way scalar processor (SP)
2. Detect and predict hot program path
3. Transform the scalar code of the hot path into a software-pipelined code for n-way VLIW
4. Load the software-pipelined code into the I$ of the n-way VLIW
5. Replace the hot path with a co-processor call

Application programs are under optimization...

Functionality Morphing -- What Architecture Enables It? --

ISA
Instruction Execution

ISA
Instruction Execution

Kazuaki Murakami & Victor Mauro Goulart Ferreira ©2000-2004
What Architecture Enables Functionality Morphing? - Hyperscalar Processor -

Main Processor
(1-way Scalar Processor)

I Cache

Instruction Fetch

Instruction Decode & Issue

FU

RF

D Cache

Co-processor
(n-way VLIW Processor)

n-way VLIW Instruction Register

FU

FU

...

FU

Register File
Tutorial Outline

- **Part 1 (8:00-9:30)**
  - Overview of JIT HW/ISA/SW Co-optimization
  - Functionality Morphing
    - Concept
    - Online hot-path profiling
    - Dynamic trace-based software pipelining
    - Hyperscalar processor (as an accelerator)
    - Performance issues

- **Part 2 (17:30-19:00)**
  - On-demand Recomputation
How Much Can You Improve the Performance?  
- Amdahl’s Law -

\[ S = \frac{1}{(1 - \text{OFF} \times \text{HP}) + \frac{\text{OFF} \times \text{HP}}{N}} \]

- **S**: Speedup
- **HP**: Fraction of the hot paths’ total execution time over the original program execution time (\(0 \leq \text{HP} \leq 1\))
- **OFF**: Fraction of hot paths detected and offloaded to the accelerator over the entire hot paths (\(0 \leq \text{OFF} \leq 1\))
- **N**: Number of FU’s of n-way VLIW co-processor (accelerator)
How Much Is the Fraction of Hot Path’s Execution Time (HP)?

Executed Instructions Covered by Hot Path

- Fraction of Hot Paths’ Execution Time over the Original Program Execution Time [%]

Benchmarks

*Hot Path Threshold=1000
How Many Hot Paths Are Detected?

Number of Identified Hot Path

Benchmarks

Number of Different Hot Paths
Tutorial Outline

✓ Part 1 (8:00-9:30)
  ✓ Overview of JIT HW/ISA/SW Co-optimization
  ✓ Functionality Morphing
    ✓ Concept
    ✓ Online hot-path profiling
    ✓ Dynamic trace-based software pipelining
    ✓ Hyperscalar processor (as an accelerator)
    ✓ Performance issues

■ Part 2 (17:30-19:00)
  ■ On-demand Recomputation
“Just-in-Time HW/ISA/SW Co-optimization Techniques for SoC” (Part 2)

Kazuaki J. Murakami (*1)
Victor Mauro Goulart Ferreira (*2)

*1: Director of Computing & Communications Center, Kyushu University
*2: Ph.D. Candidate, Dept. of Informatics, Kyushu University

E-mail: arch@i.kyushu-u.ac.jp or kjm@acm.org
JIT HW/ISA/SW Co-optimization would work like ...

Application programs are running...

Online Profiling

Hardware Reconfiguration

Mode Control

Hints for Optimization

Binary Rewriting

ISA Redefinition

Instruction Execution

Target Programs

Runtime Software

Profiler

Processor Core

Accelerator

HW/ISA/SW Co-optimization

Application programs are under optimization...

Instruction Execution

Target Programs

Runtime Software

Profiler

Processor Core

Accelerator

JIT HW/ISA/SW Co-optimization

would work like ...

Application programs are running...

Online Profiling

Hardware Reconfiguration

Mode Control

Hints for Optimization

Binary Rewriting

ISA Redefinition

Instruction Execution

Target Programs

Runtime Software

Profiler

Processor Core

Accelerator

HW/ISA/SW Co-optimization

would work like ...
Tutorial Outline

- **Part 1 (8:00-9:30)**
  - Overview of JIT HW/ISA/SW Co-optimization
  - Functionality Morphing

- **Part 2 (17:30-19:00 ➔ 15:00-16:30)**
  - On-demand Recomputation
Tutorial Outline: Part 2

• On-demand Recomputation
  – Computing Centric Computation (CCC)
  – Memory Wall Problem
  – Performance-Critical Instructions
  – On-demand Recomputation
  – Performance Issues
  – Summary and Conclusions
Computing Centric Computation

“Change the paradigm of Memory Centric Computation”
Memory is a precious resource, so use it scrupulously!
Background and Motivation

• Semiconductor improvements and trends (ITRS 2003)
  – SoC, CMP, MT…
  – Embedded memories
  – Wireless communication and devices with multimedia

• Memory-processor performance gap
  ➔ Memory wall problem (MWP)

• Performance-critical load instructions (or delinquent loads or troublesome loads)
Memory Paradigm

Long Time Ago (50–70’s)
- Slow Processors
- Magnetic Memories
  - Small
  - Very slow
- Trend → reduce program size

Recent Past (80–90’s)
- Transistor technology
- DRAM
  - Much bigger and denser memories
  - Fast access
- Trend → Use memory to improve Performance (caching)

The Future is now! (2000 and beyond)
- Higher transistor miniaturization
- Even bigger memories
- Even faster processors!!
  - but too fast and too hot
- Trends → reduce “visible” memory latency, memory accesses, improve hit rate of caches, and so on…
Tutorial Outline: Part 2

• On-demand Recomputation
  ✓ Computing Centric Computation (CCC)
  – Memory Wall Problem
  – Performance-Critical Instructions
  – On-demand Recomputation
  – Performance Issues
  – Summary and Conclusions
Processor vs. Memory Performance

Processor Performance Improvement Rate (60%/year)

DRAM Performance Improvement Rate (9%/year)

Performance Discrepancy (~50%/year and growing!)
Memory Wall Problem in 1995


**Trends for a Current Cache Miss/Hit Cost Ratio of 4**

**Trends for a Current Cache Miss/Hit Cost Ratio of 16**
Dealing with MWP

• Memory latency reduction
  – Caching
  – Prefetching
  – Speculative load
  – Memory compression
  – Embedded DRAM (e.g., PPRAM, IRAM, PIM, etc.)

• Memory latency hiding
  – Non-blocking caches
  – MT (multithreading)
  – SMT (simultaneous multithreading)
Latency Impact on Computation Time

Latencies L1, L2, M

WHY IS THAT?

<table>
<thead>
<tr>
<th>Latency</th>
<th>gcc95</th>
<th>hydro2d95</th>
<th>swim00</th>
<th>vortex00</th>
</tr>
</thead>
<tbody>
<tr>
<td>2_20_100</td>
<td>1</td>
<td>1.455739644</td>
<td>1.243039904</td>
<td>1.459929397</td>
</tr>
<tr>
<td>4_40_200</td>
<td></td>
<td>1.455739644</td>
<td>1.243039904</td>
<td>1.459929397</td>
</tr>
<tr>
<td>8_80_400</td>
<td></td>
<td></td>
<td>1.068616711</td>
<td>2.395756543</td>
</tr>
</tbody>
</table>

gcc95
hydro2d95
swim00
vortex00

L1 = 128KB
L2 = 4MB

Latency Impact on Computation Time

Normalized Total Sim. Time in Cycles
Memory System Overhead

• The impact of the memory system on execution time is composed of:
  – Number of memory accesses / Cache behavior
    • Can be improved by modern compiler techniques; however, the primary limit on compiler’s ability to improve memory behavior is its imperfect knowledge about the run-time behavior of the program
  – Memory access latency
    • Physical limitation but can be dealt with memory latency reduction (caching – temporal locality) and hiding, or tolerating schemes (change the “perceived” memory latency)
Tutorial Outline: Part 2

- On-demand Recomputation
  - Computing Centric Computation (CCC)
  - Memory Wall Problem
    - Performance-Critical Instructions
    - On-demand Recomputation
    - Performance Issues
    - Summary and Conclusions
Critical Instructions

• Performance critical instructions [Roth: HPCA2001] (delinquent and troublesome inst. is also seen in the literature)
  – Loads that are likely to miss in the cache
  – Branches that are likely to be mispredicted

• In DDMT (data-driven multithreading), the computation of these critical instructions are computed by another thread in advance to the main thread so the data is “prefetched” by the sub-thread and stays ready to be used by the main thread

• Data-driven pre-execution scheme as a unified general-purpose performance engine

• Memory latency effects on computation time appear mainly on these critical LOADs!!
Simulation Environment

- Simplescalar v.3.0 and SPEC2000 benchmarks
  - Distinct IL1 and DL1 and unified UL2
  - Five distinct cache configurations (L1, L2):
    - (32K,128K), (64K,256K), (128K,512K), (128K,1M), (128K,4M)
    - L1 (32B line size, 4 way); L2 (64B line size, 8 way)
    - Latencies: L1=8; L2=80; M=400 (units are clock cycles)

- Measurement of:
  - Cache miss count
  - Cache miss rate
  - Number of instructions (PC) and addresses (EA – effective address) grouped by miss counts

Profiling format

<cache ID {il1,dl1,ul2{d/i}}, R/W, PC, EA, time (clock #), inst. mnemonic (assembly)>
Cache Miss Count and Miss Rate for SPECint2000 and SPECfp2000
Results

• For DL1:
  – For integer benchmarks, gzip, mcf, vortex, bzip2 had $\approx 8\%$ miss rate the others, less than 1%
  – mgrid had about 25% miss rate and apsi a bit more than 14%, applu, about 4% while other FP benchmarks less than 2%.

• UL2 results as follows…
ul2 misses (fp)

number of cache misses

cache configurations

il2 misses
dl2 misses

(32k, 128k)
(64k, 256k)
(128k, 512k)
(128k, 1024k)
ul2 miss rate (fp)

cache configurations

cache miss rate

(32k, 128k)
(64k, 256k)
(128k, 512k)
(128k, 1024k)
How Much Do Top 10 Critical LOADs Occupy L2 Cache Misses

@ SPEC CPU 2000 benchmarks on Simplescalar 3.0d. (50M Forward, 10M Exec, L1=32k, L2=128k.)
How Much Can You Improve the Performance by Eliminating Top 10 Critical LOADs

Performance Improvement [%]

FP benchmarks

Integer benchmarks

f168
f171
f173
f177
f179
f183
f188
f301
i164
i175
i176
i181
i197
i253
i255
i256
i300

0.0%
0.0%
3.2%
0.7%
78.9%
40.0%
7.0%
7.3%
2.9%
16.8%
0.9%
8.9%
3.2%
2.0%
Qualitative and Quantitative Study of Critical LOADs in SPEC2000 Benchmarks
### Miss Count of Critical LOADs for L2 Cache

**Data miss count for L2 (L1=32K, L2=128K)**

#### Miss counts

<table>
<thead>
<tr>
<th>Miss Count Range</th>
<th>apsi</th>
<th>bzip2</th>
<th>parser</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-499</td>
<td>50</td>
<td>18</td>
<td>21</td>
</tr>
<tr>
<td>500-999</td>
<td>53</td>
<td>197</td>
<td>256bzip2</td>
</tr>
<tr>
<td>1000-1499</td>
<td>301</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1500-1999</td>
<td>256</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2000-2499</td>
<td>1026</td>
<td>7280</td>
<td></td>
</tr>
<tr>
<td>2500-2999</td>
<td>7338</td>
<td>7280</td>
<td></td>
</tr>
<tr>
<td>3000-3499</td>
<td>1030</td>
<td>7280</td>
<td></td>
</tr>
<tr>
<td>3500-3999</td>
<td>0</td>
<td>7280</td>
<td></td>
</tr>
<tr>
<td>4000-4499</td>
<td>0</td>
<td>7280</td>
<td></td>
</tr>
<tr>
<td>4500-4999</td>
<td>1026</td>
<td>7280</td>
<td></td>
</tr>
<tr>
<td>over 5000</td>
<td>21552</td>
<td>65345</td>
<td>197</td>
</tr>
</tbody>
</table>

**Number of Critical Instructions**

- **apsi:** 403385 0x0042d2f8 103012 0x0042d380
- **bzip2:** 905465 0x0041a300 65345 0x004144a8
- **parser:** 21552 0x004375d0
Tutorial Outline: Part 2

- On-demand Recomputation
  - Computing Centric Computation (CCC)
  - Memory Wall Problem
  - Performance-Critical Instructions
    - On-demand Recomputation
    - Performance Issues
    - Summary and Conclusions
On-demand Recomputation

“If *that data* is not in the cache…
No problem, recompute it!”
No more data starvation due to long memory latencies*…
On-demand Recomputation

- Alleviate the negative impact of memory latency by regenerating or recomputing the data to load when a cache miss occurs.
- Reduce memory accesses and visible latency of a cache miss. Instead of going to a far slow memory, try to regenerate the missed data on-the-fly.
- Critical LOADs are good candidates to apply.

In order to re-execute, we need an RC (Recomputation Code).

Access “reduction”
Latency hiding.
On-demand Recomputation

Before CCC

Source Code

\[ c = a + b; \]

\[ z = x + c; \]

\text{Reuse of Value } c

Object Code

Load a
Load b
Add c, a, b
Store c

\ldots

After CCC

Object Code

Load a
Load b
Add c, a, b
Store c

\ldots

Generate RC code Dynamically

Substitute ‘Load c’ by respective RC code

\[ [\text{RC Code}] \]

Load x
Add z, x, c
Store z

\text{Frequently missing LOAD, or critical LOAD}
How Can You Implement On-demand Recomputation

Application programs are running...
(1) Monitor D-cache misses
(2) Detect and identify critical LOADs
(4) Identify STOREs corresponding to critical LOADs

Application programs are under optimization...
(3) Monitor STORE execution
(5) Generate the recomputation (RC) code computing the values of the data which the critical LOADs would load
(6) Replace the critical LOADs with the RC code

Kazuaki Murakami & Victor Mauro Goulart Ferreira ©2000-2004
How to Detect and Identify Critical LOADs

- **Trigger**
  - When a cache miss occurs, builds up tuples
- **Tuple**
  - \(<\text{lia}, \text{la}>\)
    - lia: load instruction address
    - la: load address
- **Cache Miss Profiler**
  - Counts the tuples, and compresses them in the message form
  - Sends the messages to critical LOAD finder
- **Messages**
  - \((\text{tuple}, \text{count})\)
- **Critical LOAD Finder**
  - Accumulates the messages
  - Classify critical loads according to a given threshold
How to Detect Last Instance STORE Corresponding to Critical LOADs

- **Selector**
  - Buffers all the executed store instructions
  - Sends them as tuples to STORE filter

- **Tuple**
  - `<sia, sa>`
    - sia: store instruction address
    - sa: store address

- **STORE Filter**
  - Keep track of critical load instructions in the list of `<lia, la>`
  - Get the tuples, and then searches the list associatively with the key (sa)
  - Sends the matched tuples to last-instance STORE finder

- **Last-Instance STORE Finder**
  - Accumulates the filtered tuples
  - Finds the last instance of STOREs corresponding a critical load
How Can You Build Recomputation (RC) Code

... 00419e80  addu $v0[2], $v0[2], $a0[4]
00419e88  sll $v0[2], $v0[2], 0x1
00419e90  subu $v0[2], $s4[20], $v0[2]
00419e98  addiu $v0[2], $v0[2], 48
00419ea0  lw $v1[3], 4($v0[2])
...

In case of a miss in `lw`, substitute it for the sequence of instructions in the data-flow graph with respect to the last instance store instruction!!
Recomputation (RC) Code - Higher Levels -

- Terms definition:
  - Critical Load: loads that miss in the cache (high memory latencies)
  - Last Instance Store: the store inst that lastly generated the data actually being referred by the critical load
  - 1-level RC code: the RC code (tree) from the last instance store to the first load inst
  - N-level RC code: beyond a given leaf load continue generating RC code for that data up to N-levels.

Recomputation code is generated by backtracking or traversing from the last instance store to the leaf load inst.
How Can You Execute RC Code?

• On the main processor
  – On the same thread
    • Replace the critical load instruction with the RC code by means of dynamic binary rewriting, and then execute it
  – On another thread (helper thread) in the case of SMT
    • Trigger the execution of the RC code on the helper thread if the critical load instruction causes a cache miss

• On another (co-)processor in the case of CMP
  • Trigger the execution of the RC code on the helper processor if the critical load instruction causes a cache miss
Possible Implementations: On the Same Thread

Possible Implementations:

On the Same Thread

STORE \( \alpha \)
LOAD \( \alpha \)
compute \( \alpha \)
exploit \( \alpha \)
LOAD \( \alpha \)
exploit \( \alpha \)

Caller

Callee

\( \times \) replace out

\( \times \) Reconfiguration Code (RC)
Possible Implementations:
On Another Thread

Cache  Main Thread  Sub Thread

<table>
<thead>
<tr>
<th>compute α</th>
<th>STORE α</th>
</tr>
</thead>
</table>

x replace out

LOAD α  exploit α

LOAD α  exploit α

Cache Miss Handling

LOAD α  exploit α

Compute α

Cache Miss

Invoke Sub-thread

If $t_s >> t_C$

Kazuaki Murakami & Victor Mauro Goulart Ferreira ©2000-2004
Tutorial Outline: Part 2

• On-demand Recomputation
  ✓ Computing Centric Computation (CCC)
  ✓ Memory Wall Problem
  ✓ Performance-Critical Instructions
  ✓ On-demand Recomputation
    – Performance Issues
    – Summary and Conclusions
Performance Issues (1/5):
How Many Critical LOADs Can Be Replaced by RC Codes?

- Fraction of critical load instructions which can be replaced by RC codes

**FP benchmarks**
- f168: 0.0%
- f171: 0.0%
- f173: 48.0%
- f177: 0.0%
- f179: 0.0%
- f183: 0.0%
- f188: 0.0%
- f301: 0.0%

**Integer Benchmarks**
- i164: 65.0%
- i175: 0.1%
- i176: 10.9%
- i181: 7.9%
- i197: 0.0%
- i253: 7.8%
- i255: 10.3%
- i256: 0.0%
- i300: 0.0%
Performance Issues (2/5):
Average RC Code Size

Average number of instructions

FP benchmarks

Integer Benchmarks

Kazuaki Murakami & Victor Mauro Goulart Ferreira ©2000-2004
Performance Issues (3/5):
Speedup Attained by Replacing Critical LOADs with RC Codes (RC Case)

Execution-time reduction rate [%]

FP benchmarks

Integer Benchmarks

Kazuaki Murakami & Victor Mauro Goulart Ferreira ©2000-2004
Performance Issues (4/5): Potential Effects of Critical LOADs’ Elimination (Ideal Case)

Performance Improvement [%]

FP benchmarks

Integer benchmarks

Kazuaki Murakami & Victor Mauro Goulart Ferreira ©2000-2004
Performance Issues (5/5):
Comparison between Ideal Case (Graph 4) and RC Case (Graph 3)
Related Work

• Prefetching
  – Intel’s SP (Speculative Prefetching)
  – Roth’s DDMT, and so on
• Value prediction
• Memoization (or Value reuse)
On-demand Recomputation

• Outline
  ✓ Computing Centric Computation (CCC)
  ✓ Memory Wall Problem
  ✓ Performance-Critical Instructions
  ✓ On-demand Recomputation
  ✓ Performance Issues
  – Summary and Conclusions
Tutorial Outline

• Part 1 (8:00-9:30)
  ✓ Overview of JIT HW/ISA/SW Co-optimization
  ✓ Functionality Morphing

• Part 2 (15:00-16:30)
  ✓ On-demand Recomputation
Thank you
References


