MLP-Aware Dynamic Instruction Window Resizing for Adaptively Exploiting Both ILP and MLP

Yuya Kora
Kyohei Yamaguchi
Hideki Ando

Nagoya University
Single-Thread Performance

• In the past: Pollack’s law
  – Performance improvement $\propto \sqrt{\text{area}}$

• Recently: little improve
  – Despite ever increasing transistor budget

• Use increased transistor budget effectively to improve single-thread performance
Memory Wall

• Large speed gap between processor and main memory

• Conventional solutions:
  – Large cache
    • Expensive: several MB is not enough
  – Hardware prefetcher
    • Effective only for regular access patterns
Aggressive Out-of-Order Execution

• Significantly increase #in-flight instructions through extensive instruction window resources
  – Issue queue, reorder buffer, load/store queue

• Allow parallel memory accesses by executing cache-miss loads as early as possible
  – Reduce effective memory access latency
  – MLP: memory-level parallelism
Advantage and Disadvantage

• Advantage
  – Data fetch is accurate
  – Simple extension of conventional processor

• Disadvantage
  – Large resources lengthen clock cycle time
Outline

• Tradeoff Involved in Enlarging Window Resources for Aggressive Out-of-Order Execution

• Dynamic Instruction Window Resizing

• Evaluation

• Conclusion
Performance Problem in Enlarged Window Resources

- Large window resources lengthen the clock cycle time
  - Can be solved by pipelining resources

- Prevent ILP from being exploited
  - Pipelined issue queue cannot issue dependent instructions back-to-back
  - Reduce IPC

- Tradeoff
  - Large window: beneficial for MLP, but harmful for ILP
  - Small window: beneficial for ILP, but cannot exploit MLP
Example Illustrating Tradeoffs

- Large window is beneficial, even if it is pipelined
- MLP is exploited

- Large window is harmful, because IQ is pipelined
Outline

• Tradeoff Involved in Enlarging Window Resources for Aggressive Out-of-Order Execution

• Dynamic Instruction Window Resizing

• Evaluation

• Conclusion
Dynamic Instruction Window Resizing

• Adapt window size to available parallelism
  – ILP or MLP

• As more exploitable MLP is predicted
  – Window resources are enlarged and pipeline depth is increased

• If prediction indicates less MLP is exploited (= ILP is more valuable)
  – Window resources are shrunk and pipeline depth is decreased
Prediction when MLP is Exploitable

• If an LLC miss occurs once
  – Predict that MLP is exploitable for a while

• If memory latency has lapsed after the last LLC miss
  – Predict that MLP will not be exploitable
Rationale of Prediction

• LLC misses are typically clustered
Term: *level*

- Instruction window resource *level*
  - {size, pipeline-depth} of the resource

- As level number increases
  - Size increases
  - Each resource is pipelined so that it does not increase the clock cycle time
What is Level Transition?

When the current size is 64 entries with 1-stage pipeline, and we enlarge to 160 entries with 2-stage pipeline, we say that the level is increased or transits from 1 to 2.
Algorithm: Put it Together

- If LLC miss occurs:
  - Increase level of the resources

- If memory latency has lapsed after last LLC miss:
  - Check if resources are all shrinkable
  - If so, it decreases level
  - Otherwise, it stops resource allocation to increase vacancies in resources, and postpones level decrease until shrinkable
Outline

• Tradeoff Involved in Enlarging Window Resources for Aggressive Out-of-Order Execution

• Dynamic Instruction Window Resizing

• Evaluation
  – Environment
  – IPC
  – Energy Efficiency
  – Cost Efficiency
  – Comparison with Runahead Execution

• Conclusion
## Configuration of Base Processor

<table>
<thead>
<tr>
<th>Feature</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture type</td>
<td>Intel P6</td>
</tr>
<tr>
<td>Issue width</td>
<td>4</td>
</tr>
<tr>
<td>ROB</td>
<td>128 entries</td>
</tr>
<tr>
<td>IQ</td>
<td>64 entries</td>
</tr>
<tr>
<td>LSQ</td>
<td>64 entries</td>
</tr>
<tr>
<td>L1 I-cache</td>
<td>64KB, 2-way, 32B line</td>
</tr>
<tr>
<td>L1 D-cache</td>
<td>64KB, 2-way, 32B line, 2 ports, 2-cycle hit latency</td>
</tr>
<tr>
<td>L2 cache</td>
<td>LLC, 2MB, 4-way, 64B line, 12-cycle hit latency</td>
</tr>
<tr>
<td>Main memory</td>
<td>300-cycle minimum latency</td>
</tr>
<tr>
<td>Data prefetcher</td>
<td>stride-based, 4K-entry, 4-way table, 16-data on L2 miss</td>
</tr>
</tbody>
</table>
## Size and Pipeline Depth of Resources

<table>
<thead>
<tr>
<th>resource</th>
<th>parameter</th>
<th>level</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>IQ</td>
<td>entries</td>
<td>64</td>
<td>160</td>
<td>256</td>
</tr>
<tr>
<td></td>
<td>pipe depth</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>ROB</td>
<td>entries</td>
<td>128</td>
<td>320</td>
<td>512</td>
</tr>
<tr>
<td></td>
<td>pipe depth</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>LSQ</td>
<td>entries</td>
<td>64</td>
<td>160</td>
<td>256</td>
</tr>
<tr>
<td></td>
<td>pipe depth</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

- Has physically 4x larger window resources than base
- Can be configured to one of three levels
- Clock cycle time is determined by IQ delay in base processor
Environment for Performance Evaluation

• Simulator based on SimpleScalar 3.0a

• Alpha ISA

• SPEC2006 benchmark programs
  – Mem-intensive: AVG load latency ≥ 10 cycles
  – Comp-intensive: AVG load latency < 10 cycles
IPC: Evaluation Models

• **Fixed size model**: Size of window resources is **FIXED** during execution. At levels 2 and 3, resources are pipelined, which causes issue bubble and extra branch misprediction penalty.

• **Dynamic resizing model**: Size of window resources is **DYNAMICALLY RESIZED**. At levels 2 and 3, resources are pipelined, which causes issue bubble and extra branch misprediction penalty.

• **Ideal model**: Same as the fixed size model, but **NOT pipelined**. No penalty related to pipelining is imposed.
• **mem-intensive programs**: Level 3 achieves best performance. MLP is exploited aggressively with a large window.

• **comp-intensive programs**: Performance is not so sensitive to level, but level 1 is the best.

• Best resource level is different depending on program, when the size is fixed.
• Dynamic resizing model achieves as good as **best** performance for levels 1 to 3 of fixed size model.
• Imply good adaptability
• **21%** speedup for all programs
IPC: Ideal Model

- No significant degradation in dynamic resizing model
- Imply good adaptability
Samples from Memory-intensive Programs

**libquantum**

<table>
<thead>
<tr>
<th>Level</th>
<th>Fix</th>
<th>Res</th>
<th>Relative IPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>lev1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lev2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lev3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**soplex**

<table>
<thead>
<tr>
<th>Level</th>
<th>Fix</th>
<th>Res</th>
<th>Relative IPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>lev1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lev2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lev3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Samples from Compute-intensive Programs

**gcc**

<table>
<thead>
<tr>
<th>Level</th>
<th>Fix</th>
<th>Res</th>
</tr>
</thead>
<tbody>
<tr>
<td>lev1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lev2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lev3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**sjeng**

<table>
<thead>
<tr>
<th>Level</th>
<th>Fix</th>
<th>Res</th>
</tr>
</thead>
<tbody>
<tr>
<td>lev1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lev2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lev3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Energy Efficiency: Assumption

• Performance/Energy $\propto \frac{1}{EDP}$

• Consumed energy is derived using McPAT

• 32nm LSI technology

• Temperature of 350K
Energy Efficiency: Results

- Power is increased, but perf is improved $\Rightarrow$ Better energy efficiency
- Memory-intensive: 36% better
- Compute-intensive: 8% worse
- Overall: 8% better
Cost Efficiency: Assumption

• Calculate area using McPAT

• 32nm LSI technology
# Cost Efficiency: Results

<table>
<thead>
<tr>
<th>Additional cost</th>
<th>Value (per core)</th>
<th>1.6mm²</th>
</tr>
</thead>
<tbody>
<tr>
<td>vs. base core</td>
<td></td>
<td>6%</td>
</tr>
<tr>
<td>vs. Sandy Bridge core</td>
<td></td>
<td>8%</td>
</tr>
<tr>
<td>vs. Sandy Bridge chip</td>
<td></td>
<td>3%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Speedup</th>
<th>Achieved</th>
<th>21%</th>
</tr>
</thead>
<tbody>
<tr>
<td>expected by Pollack’s law</td>
<td></td>
<td>3%</td>
</tr>
<tr>
<td>augmented L2 cache</td>
<td></td>
<td>1%</td>
</tr>
</tbody>
</table>

2MB, 4-way → 2.5MB, 5-way
(increased cost is 1.3x greater than the additional cost)

Good cost/performance ratio, that far exceeds that based on Pollack’s law
Background on Runahead Execution

- **Features**
  - Exploit MLP by pre-execution
  - Can also exploit ILP because it requires only a small instruction window
- **When an L2 cache miss occurs**
  - Checkpoint architecture state and enter runahead mode
- **In runahead mode**
  - Instructions are speculatively executed
  - If another L2 miss occurs, MLP is exploited by overlapping main memory access with the triggered load access
- **Runahead mode ends when the original L2 miss is resolved, and normal execution restarts from the checkpoint**

- **Practical**
  - Simple
  - Accommodate existing architecture
  - Adopted in SunMicro Rock and IBM Power6
Comparison with RA: Results

- Runahead that has an **EXCLUSIVE** period for MLP exploitation achieves better performance in very mem-intensive programs.
- Resizing achieves better performance in moderately mem-intensive because it can exploit ILP and MLP **SIMULTANEOUSLY**.
Conclusion

• Dynamic instruction window resizing
  – Exploit ILP and MLP adaptively
  – Based on prediction of available parallelism

• Features
  – Very simple
  – Very practical

• Our scheme achieves
  – Performance level similar to the best performance achieved with fix-sized resources
  – 21% speedup
  – 6% extra cost of a core, or 3% of an entire proc chip
  – 8% better energy efficiency