The Load Slice Core Microarchitecture

Trevor E. Carlson, Uppsala University
Wim Heirman, Intel
Osman Allam, Ghent University
Stefanos Kaxiras, Uppsala University
Lieven Eeckhout, Ghent University
LSC: Improving Energy Efficiency

- All systems are power-limited
- OoO cores are inefficient

Performance Through MHP

- **Goal**
  - Out-of-order-like performance with in-order efficiency

- **Opportunity for in-order processors:**
  - Applications wait for the memory hierarchy
  - Stalls in-order processors

- **How to fix and keep efficiency?**
  - Identify **Memory Hierarchy Parallelism** (MHP)
  - Prioritize MHP-critical instructions
The Load Slice Core

- Restricted out-of-order core
- **Learn critical instruction slices**
  - Iterative Backwards Dependency Analysis (IBDA) to find loads and address generating instructions
- **Bypass critical instructions**
  - Expose MHP for performance

**Prior work**
- Dyn./spec. precomp., Continual flow, slipstream: OoO as a starting point
- Complexity effective: focuses on ILP, not MHP
- SLTP, iCFP, flea-flicker two-pass: use extensive structures for slices
- Runahead execution: re-executes instructions
- DAE, braid, OUTRIDER, flea-flicker multi-pass: require recompilation

**LSC**: hardware-only, does not re-execute
Optimization Example

- SPEC CPU2006 leslie3d
- Two load instructions are long-latency
- First use by add
- Key address generating instructions
- Branch instructions left out for clarity

```
label:
ld (r9+r8*8), r1
mov r6, r8
add r1, r1
mul r7, r8
add rdx, r8
mul (r9+r8*8), r2
test r8, $0x8000
bne label
```
Optimization Example

SPEC CPU2006 leslie3d

Two load instructions

First use by add

Branch instructions left out for clarity

In-order will stall here

Could get more MHP if we prioritize these instead

label:
ld (r9+r8*8), r1
mov r6, r8
add r1, r1
mul r7, r8
add rdx, r8
mul (r9+r8*8), r2
test r8, $0x8000
bne label
Two key LSC techniques:
1) Identify critical instruction slices
2) Bypass to increase MHP and performance
#1: Identifying critical instruction slices
#1: Identifying critical instruction slices

Identification of all dependencies is complex. Instead, focus on the critical slice.
Iterative Backward Dependency Analysis

- Learning critical slices
  1. Start with load and store addresses
  2. IBDA to learn address generating instructions

- IST – Instruction Slice Table
  - Tracks critical instructions
  - Enables bypassing for MHP

- RDT – Register Dependency Table
  - Maps registers to instruction producers
  - Enables backwards dependency analysis
Iterative Backwards Dependency Analysis

IBDA learns critical backwards slice

Marked this cycle
Marked previously/critical
Bypassing

- Restricted out-of-order core

- Bypass queue:
  - Execute critical slice instructions earlier
  - Out-of-order with respect to regular queue
  - *In-order* within each queue
  - Loads can bypass store data (great for MHP)

- Do we have memory dependence violations?
  - Address computations always marked for bypass
  - Address computations *execute in program order*
  - Guarantees correct memory ordering
    (store buffer knows addresses)
#2: Bypassing to increase MHP

The critical slice can now bypass the compute (blue) instructions.

First learned address gen. in bypass.
Experimental Setup

- Sniper multi-core simulator
  - ARM Cortex-A7-like configuration
  - 32KB L1s, 512KB L2, L1D prefetcher
  - 28nm (CACTI 6.5), 2.0GHz

- SPEC CPU2006 representative 750M instruction SimPoints and SPEC OMP and NPB representatives used

<table>
<thead>
<tr>
<th>In-order</th>
<th>LSC</th>
<th>OOO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stall on use</td>
<td>Restricted out-of-order</td>
<td>Full out-of-order</td>
</tr>
<tr>
<td>16-entry queue</td>
<td>32-entry IQ bypass queue, scoreboard</td>
<td>32-entry ROB and scheduler/issue queue</td>
</tr>
<tr>
<td>2-wide issue</td>
<td>2-wide issue</td>
<td>2-wide dispatch</td>
</tr>
<tr>
<td>ARM Cortex-A7-like</td>
<td>15% area overhead</td>
<td>155% area overhead (ARM Cortex-A9-like)</td>
</tr>
</tbody>
</table>
LSC Performance

**mcf**

- **MHP:**
  - DRAM

**soplex**

- **MHP:**
  - DRAM + L2

**h264ref**

- **MHP:**
  - L1

**calculix**

- **MHP:**
  - L1

Average: 53% improvement in-order, within 25% of OoO;

Better energy/area efficiency overall
LSC Many-Core Performance

<table>
<thead>
<tr>
<th></th>
<th>Power (W)</th>
<th>Area (mm²)</th>
<th>Cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max</td>
<td>45.0</td>
<td>350</td>
<td>-</td>
</tr>
<tr>
<td>In-order</td>
<td>25.5</td>
<td>344</td>
<td>105</td>
</tr>
<tr>
<td>LSC</td>
<td>25.3</td>
<td>322</td>
<td>98</td>
</tr>
<tr>
<td>Out-of-Order</td>
<td>44.0</td>
<td>140</td>
<td>32</td>
</tr>
</tbody>
</table>

LSC has almost a 2x performance benefit over an out-of-order design.
Conclusion

- MHP: an opportunity for better in-order performance

Load Slice Core

- Identify critical slices:
  - Backwards with IBDA
  - Learn across iterations

- Bypass critical instructions
  - Simple queue

- More performance through increased MHP:
  - Single-core: within 25% of OoO
  - Multicore: nearly 2x for area/power-limited designs
The Load Slice Core Microarchitecture

Trevor E. Carlson, Uppsala University
Wim Heirman, Intel
Osman Allam, Ghent University
Stefanos Kaxiras, Uppsala University
Lieven Eeckhout, Ghent University