Illusionist: Transforming Lightweight Cores into Aggressive Cores on Demand

A. Ansari, S. Feng, S. Gupta, J. Torrellas, S. Mahlke

HPCA - 2013

University of Illinois         University of Michigan

June 28, 2013
Motivation

- CMPs (w/ many lightweight cores) achieve good energy efficiency and throughput
  - Single-thread performance stagnates or even becomes worse

- Asymmetric CMPs (w/ aggressive cores) can accelerate threads
  - Threads needs to be mapped/migrated efficiently to these cores
  - Fixed in design time so less flexible and less adaptive in runtime

- Proposed solution: Rather than using aggressive cores to accelerate threads, use the aggressive core to accelerate large number of lightweight cores and provide an illusion as if the whole chip is full of aggressive cores, if needed
  - Aggressive core executes distilled version of programs to provide cache and branch hints to respective lightweight core

A. Ansari, S. Feng, S. Gupta, J. Torrellas, S. Mahlke
Illusionist: Transforming Lightweight Cores into Aggressive Cores on Demand
To maximize single-thread performance, aggressive cores should not be used to accelerate individual threads, but rather should redundantly execute threads running on the lightweight cores.

- Single-thread: 35% over LWC; Throughput: 2x over AC
Acceleration Opportunities

- Lightweight core: EV4, 2-issue in-order
- Aggressive core: EV6, 6-issue out-of-order
- With perfect hints, EV4(OoO) can outperform EV6(6-issue OoO)

*Figure 2: IPC of different DEC Alpha microprocessors, normalized to EV4’s IPC. In most cases, by providing perfect hints for the simpler cores (EV5, and EV4 (OoO)), these cores can achieve a performance comparable to that achieved by a 6-issue OoO EV6.*
Illusionist: Core Coupling Architecture

- Base case architecture: connecting an aggressive and a lightweight core together.
Illusionist: System Overview

- CMP level architecture: in a 44-core system one AC is shared among 10 LWC in two ring based network.
Program Distillation Techniques

- Aggressive distillation: so that AC can generate hints for multiple LWC through multithreading-based, time-multiplexed execution
- Instruction removal:
  - Keep back-slices of branches and memory instructions
  - Sliding window: top inst is removed if it doesn’t produces any value which drives branches or memory insts inside window (?)
  - Highly biased branches: 90% biased are not helped by AC and LWC uses its own branch predictor for such branches
  - Unnecessary cache hints: If the loads and stores access the addresses in the same cache line then stores are removed
- This analysis can be done offline; they do it using DynamicRIO – a dynamic, just-in-time compiler with overhead of less than 1%
Phase-Based Program Selection

- Idea is to predict the phases w/o actually running the program on both lightweight and aggressive cores
  - Then limit the dual-core execution to the most useful phases
- A linear regression model is used to predict the IPC improvement in next epoch based on L1 misses, and branch mispredictions
  - The LWC which benefits most gets the AC to generate its hints
Example of Distilled Program

- Code from 179.art: First statement is rare, else clause is highly biased so turned into unconditional

Original code:
```
if (high<=low)
return;

srand(10);
for (i=low;i<high;i++) {
    for (j=0;j<numf1s;j++) {
        if (i<high)
            tds[i][j] = tds[i][0];
        else
            tds[i][j] = bus[i][1];
        tds[i][j] = bus[i][1];
    }
}
```

Distilled code:
```
for (i=low;i<high;i++) {
    for (j=0;j<numf1s;j++) {
        if (i<high)
            noise1 = (double)(rand()&0xffff);
        else
            noise2 = noise1/(double)0xffff;
        tds[i][j] += noise2;
    }
}
```
Experimental Setup

- **Performance:** Modified SimAlpha
  - Spec-2k with SimPoint
- **Power:** Wattch, HotLeakage, and CACTI
- **Area:** Synopsys toolchain + 90nm TSMC

**Table 1:** Configuration of the underlying lightweight and aggressive cores.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>A lightweight core</th>
<th>An aggressive core</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch/issue/commit width</td>
<td>2 per cycle</td>
<td>6 per cycle</td>
</tr>
<tr>
<td>Reorder buffer</td>
<td>32 entries</td>
<td>128 entries</td>
</tr>
<tr>
<td>Load/store queue entries</td>
<td>8/8</td>
<td>32/32</td>
</tr>
<tr>
<td>Issue queue</td>
<td>16 entries</td>
<td>64 entries</td>
</tr>
<tr>
<td>Instruction fetch queue</td>
<td>8 entries</td>
<td>32 entries</td>
</tr>
<tr>
<td>Branch predictor</td>
<td>tournament (bimodal + Illusionist BP)</td>
<td>tournament (bimodal + 2-level)</td>
</tr>
<tr>
<td>Branch target buffer size</td>
<td>256 entries, direct-map</td>
<td>1024 entries, 2-way</td>
</tr>
<tr>
<td>Branch history table</td>
<td>1024 entries</td>
<td>4096 entries</td>
</tr>
<tr>
<td>Return address stack</td>
<td>-</td>
<td>32 entries</td>
</tr>
<tr>
<td>L1 data cache</td>
<td>8KB direct-map, 3 cycles access latency, 2 ports</td>
<td>64KB, 4-way, 5 cycles access latency, 4 ports</td>
</tr>
<tr>
<td>L1 instr. cache</td>
<td>4KB direct-map, 2 cycles access latency, 2 ports</td>
<td>64KB, 4-way, 5 cycles access latency, 1 port</td>
</tr>
<tr>
<td>L2 cache</td>
<td>1MB per core, unified and shared, 8-way, 16 cycles access latency</td>
<td></td>
</tr>
<tr>
<td>Main memory</td>
<td></td>
<td>250 cycles access latency</td>
</tr>
</tbody>
</table>
Instructions Removed

- Percentage of instructions removed from the original programs
- On an average, 76% of the instructions are removed when performing the analysis on a sliding window of size 100K insts.
Accuracy of Generated Hints

- Larger window results into more aggressive instruction removal
  - This comes at a higher loss of accuracy
- For a 10K window size (used in the final evaluation) the accuracy is 79% compared to perfect program execution
In most applications, the breakdowns are similar

Breakdown of Instructions

- Memory
- Branch
- Integer
- Float
- Misc.
On an average, 35% better single-thread performance
Area-Neutral Comparison of Alternative

- Final system: After phase-based pruning
- 35% better single-thread performance compared to LWC
- 2X better throughput compared to AC
Instead of using aggressive cores to execute the bottleneck threads, they are used to enable lightweight cores to achieve high single-thread performance w/o sacrificing throughput.

Aggressive core is shared among multiple lightweight cores
- This requires aggressive program distillation
- Programs are distilled by static, dynamic and phase based pruning

Finally, it is argued that using the AC to generate hints for LWC is better option than off-loading LWC’s work to AC.