Illusionist: Transforming Lightweight Cores into Aggressive Cores on Demand

Amin Ansari¹, Shuguang Feng², Shantanu Gupta³, Josep Torrellas¹, and Scott Mahlke⁴

¹ University of Illinois, Urbana-Champaign
² Northrop Grumman Corp.
³ Intel Corp.
⁴ University of Michigan, Ann Arbor
Adapting to Application Demands

- Number of threads to execute is not constant
  - Many threads available
    - System with many lightweight cores achieves a better throughput
  - Few threads available
    - System with aggressive cores achieves a better throughput
  - Single-thread performance is always better with aggressive cores

- Asymmetric Chip Multiprocessors (ACMPs):
  - Adapt to the variability in the number of threads
  - Limited in that there is no dynamic adaptation

- To provide dynamic adaptation:
  - We use core coupling
Core Coupling

- Typically configured as leader/follower cores where the leader runs ahead and attempts to accelerates the follower

  - Slipstream
  - Master/slave Speculation

  The leader runs ahead by executing a “pruned” version of the application

  - Flea Flicker
  - Dual-core Execution

  The leader speculates on long-latency operations

  - Paceline

  The leader is aggressively frequency scaled (reduced safety margins)

  - DIVA

  A smaller follower core simplifies the design/verification of the leader core
Extending Core Coupling

A 9 Core ACMP System

Hints

Aggressive Core (AC)

Lightweight Core (LWC)

Lightweight Core

Lightweight Core

Lightweight Core

Lightweight Core

Lightweight Core

Lightweight Core

9 core ACMP

7 LWCs + a coupled cores

Illusionist

Throughput

Configuration
• Higher single-thread performance for all LWCs
  o By using a single aggressive core
  o Giving the appearance of 8 semi-aggressive cores
Illusionist vs Prior Work

Hints

Master Slave Parallelization [Zilles’02]

- Higher single-thread performance for only a single aggressive core
- By using an army of LWCs (slave cores)
- Pushing the ILP limit
- Spawning threads for the slave cores to work on and also check the speculative computation on the master core
Providing Hints for Many Cores

- Original IPC of the aggressive core ~2X of that of a LWC

- We want an AC to keep up with a large number of LWCs
  - We need to substantially reduce the amount of work that the aggressive core needs to do per each thread running on a LWC

- We need to run lower num of instructions per each thread
  - We distill the program that the aggressive core needs to run
  - We limit the execution of the program only to most fruitful parts

- The main challenge here is to
  - Preserve the effectiveness of the hints while removing instructions
Program Distillation

- **Objective:** reduce the size of program while preserving the effectiveness of the original hints (branch prediction and cache hits)

- **Distillation techniques**
  - **Aggressive instruction removal (on average, 77%)**
    - Remove instructions which do not contribute to hint generation
    - Remove highly biased branches and their back slice
    - Remove memory inst. accessing the same cache line
  - **Select the most promising program phases**
    - Predictor that uses performance counters
    - Regression model based on IPC, $ and BP miss rates
**Example of Instruction Removal**

```c
if (high<=low)
    return;

srand(10);
for (i=low;i<high;i++) {
    for (j=0;j<numf1s;j++) {
        if (i%low) {
            tds[j][i] = tds[j][0];
            tds[j][i] = bus[j][0];
        } else {
            tds[j][i] = tds[j][1];
            tds[j][i] = bus[j][1];
        }
    }
}
for (i=low;i<high;i=i+4) {
    for (j=0;j<numf1s;j++) {
        noise1 = (double)(rand()&0xffff);
        noise2 = noise1/(double)0xffff;
        tds[j][i] += noise2;
        bus[j][i] += noise2;
        tds[j][i+1] += noise2;
        bus[j][i+1] += noise2;
    }
    for (j=0;j<numf1s;j++) {
        noise1 = (double)(rand()&0xffff);
        noise2 = noise1/(double)0xffff;
        tds[j][i+2] += noise2;
        bus[j][i+2] += noise2;
    }
    for (j=0;j<numf1s;j++) {
        noise1 = (double)(rand()&0xffff);
        noise2 = noise1/(double)0xffff;
        tds[j][i+3] += noise2;
        bus[j][i+3] += noise2;
    }
}
```
If we can predict these phases without actually running the program on both lightweight and aggressive cores, we can limit the dual core execution only to the most useful phases.
Phase Prediction

- Phase predictor:
  - does a decent job predicting the IPC trend
  - can sit either in the hypervisor or operating system and reads the performance counters while the threads running

- Aggressive core runs the thread that will benefit the most
Illusionist: Core Coupling Architecture

Aggressive Core
- L1-Data
- Cache Fingerprint
- Hint Gathering

Lightweight Core
- L1-Inst
- L1-Data
- Hint Distribution
- Hint Disabling

Resynchronization signal and hint disabling information

Queue
- head
- tail

Shared L2 cache

Memory Hierarchy

Read-Only

FET DEC REN DIS EXE MEM COM

DE RE DI EX ME CO
Illusionist System
Experimental Methodology

- **Performance**: Heavily modified SimAlpha
  - Instruction removal and phase-based program pruning
  - SPEC-CPU-2K with SimPoint
- **Power**: Wattch, HotLeakage, and CACTI
- **Area**: Synopsys toolchain + 90nm TSMC

<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>250 cycles access latency</td>
</tr>
<tr>
<td>Main memory</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Performance After Acceleration

On average, 43% speedup compared to a LWC
In most benchmarks, the breakdowns are similar.
Area-Neutral Comparison of Alternatives

System Throughput | Power | Average Single-Thread Performance | Total Energy

More Lightweight Cores

2X

34%

Normalized to All Aggressive Cores

0.25
0.5
0.75
1
1.25
1.5
1.75
2

All Aggressive Cores (ACs) | 1 AC + 1 LWC | After Instruction Removal | After Phase-Based Pruning | All Lightweight Cores (LWCs)
Conclusion

- On-demand acceleration of lightweight cores
  - using a few aggressive cores

- Aggressive core keeps up with many LWCs by
  - Aggressive inst. removal with a minimal impact on the hints
  - Phase-based program pruning based on hint effectiveness

- Illusionist provides an interesting design point
  - Compared to a CMP with only lightweight cores
    - 35% better single thread performance per thread
  - Compared to a CMP with only aggressive cores
    - 2X better system throughput
BACKUP SLIDES
Comparison with Alternatives

Number of available threads = 60% of the number of lightweight cores
Comparison with Alternatives

More Lightweight Cores

number of available threads = 30% of the number of lightweight cores
Percentage of Instruction Removed

![Bar chart showing percentage of instructions removed for different benchmarks and instruction counts. The chart compares the removal of instructions for 100, 1K, 10K, and 100K instructions across various benchmarks and shows the average across all benchmarks.]
Hint Accuracy after Instruction Removal