Optimal versus Heuristic Global Code Scheduling

Sebastian Winkel
Intel® Compiler Lab
Introduction

• Importance of wide-issue in-order architectures
  – traditionally dominant in embedded VLIW DSPs
  – gaining some momentum in the high-end server segment
    (Intel® Itanium® 2, IBM® Power6™)

• Global instruction scheduling is crucial to extracting
  instruction-level parallelism on such architectures

• In comparison to local scheduling, global scheduling includes
  code motion between basic blocks
  – Many variants: upward, downward, compensation copies, etc.
  – On Itanium intertwined with EPIC optimizations
    (control and data speculation, predication, etc.)
Global Code Scheduling Example

(a) Incoming region before scheduling

(b) Schedule with cycle annotation

Length of hot path (B1 → B3 → B4) is reduced from 6 to 3 cycles

Speculative upward code motion

Compare speculation

Block collapsing

Multiway branch generation

Speculative upward code motion

Predicated downward code motion with compensation copies

Predicted upward code motion with compensation copies

Block collapsing

Multiway branch generation

Compare speculation

Speculative upward code motion

Predicated downward code motion with compensation copies

Block collapsing

Multiway branch generation

Compare speculation
**Global Scheduling Heuristics**

- Many heuristics have been developed
  - E.g., trace, selective, hyperblock, Bernstein/Rodeh [1]
  - Wavefront scheduling [Micro-32] used in the Intel compiler is among the most comprehensive methods

- Challenges
  - Complex interdependences between individual transformations
    - Hard for heuristics to weigh cost and benefit
  - Restrictions with respect to scheduling regions, supported code motion classes
  - No formal validation of correctness or quality of the results
Our ILP Scheduler

• **Optimal global scheduler based** on integer linear programming (ILP), implemented experimentally in the Intel® Itanium® product compiler

• **Goals:**
  - Find **performance headroom** (in EPIC and in our compiler)
  - Gain **insights into global scheduling** trade-offs – independently of any heuristic scheduling method

• **Contribution of this research in comparison with previous work** [Wilken00, Kästner00, Winkel04]:
  - Large optimization scope: Arbitrary scheduling regions, includes virtually all known EPIC scheduling optimizations
  - Efficiency: Still permits relatively large problem instances
  - Extensive experimental study
Overview

• Brief integer linear programming summary
• ILP scheduler overview
  – Optimization scope
  – Optimality notion
  – Region scheduling
• Experiments
  – Implementation and methodology
  – Results
• Conclusion

ILP formulations not covered in the talk, but described in the paper.
**Integer Linear Programming (ILP)**

- Proven combinatorial optimization method
  - Many applications in research and industry
- An ILP is described by a system of linear inequalities and a linear objective function
- Constraints can be thought to describe a **polytope**
- **Optimal solution** is an integer point contained in this polytope for which the objective function is minimal
- **Polyhedral efficiency**: It helps the solver if as many vertices of the polytope as possible are integral

7/19  12/2007  MICRO-40
Overview of Modeled Optimizations

- Global code motion:
  - Directions: upward, downward
  - Control conditions: predicated, speculative
  - Boundaries: across, into, and out of loops (*cyclic*)
  - Enablers: renaming, compensation copies
  - Global propagation of non-unit latencies

- Supported speculation features:
  - Control- and data-speculative loads
  - Partial-ready code motion, compare speculation

- Block model:
  - Block emptying and collapsing
  - Resulting *multiway branch generation*
  - Choose fall-through edges and block order

(Highlighted: new or significantly improved parts vs. previous work)

ILP scheduler can resolve all interdependences between these optimizations and deliver a global optimum
**Optimality Notion**

- Objective function minimizes *global schedule length (GSL)*
  - defined as the sum of the schedule lengths of the basic blocks, each weighted by the execution frequency of the block
- GSL reductions directly translate into *uninstalled execution time reductions*

- Objective function is “blind” to all other efficiency criteria
- **Second scheduling pass:**
  - Add constraints to the solved ILP that *fix the block lengths*
  - Change objective function so that it *minimizes global code motion and speculation*
  - Run solver again
  - Solvable within a few seconds because the GSL is fixed
Region Scheduling

- Instances > 1000 instructions often cannot be solved in acceptable time
- Newly developed region scheduling allows to schedule routines of arbitrary size
- Forms and schedules regions iteratively
  - First select largest and hottest loop within region size limits
  - Grow the region within the next-outer loop nest
- Grow regions one block deep into already scheduled “territory”
- Conceptually, generate and solve an ILP for the entire routine, but set all out-of-region decision variables to constants
Overview

• Brief integer linear programming summary
• ILP scheduler overview
  – Optimization scope
  – Optimality notion
  – Region scheduling
• Experiments
  – Implementation and methodology
  – Results
• Conclusion
Heuristic Scheduler GCS in Comparison

- Implements wavefront scheduling
  - Schedules blocks in an order defined by the downward movement of the wavefront
  - Scheduling decisions made based on priority and veto functions
  - No backtracking

- Only supported by GCS:
  - Integrated postincrement generation and redundancy elimination

- Only supported by ILP:
  - Cyclic code motion, downward code motion
Experimental Methodology

- Compared ILP scheduler on **five SPEC® CPU2006 integer benchmarks** against GCS
  - Did not test the entire suite for compile time reasons
  - Focused on those benchmarks with a relatively large percentage of unstalled execution time (optimization target), not dominated by pipelined loops
  - Applied to the hottest routines that capture 90% of the execution time

- Overall **104 routines were tested**
  - Tested each routine individually, measured speedup using HP Caliper IP sampling

- 10.0 compiler, highest optimization level (-O3, IPO, PGO)
- ILPs solved with (nonparallel) ILOG® CPLEX 10.0
ILP Solvability

- 625 (first pass) ILPs solved on a 1.6 GHz Itanium® 2
- Standard scheduling region size limit of 500 instructions
  - For hard-to-solve routines, decremented in steps of 50 until the ILPs can be solved within 4 hours

382 ILPs, average sol. time 6s

88 ILPs, average sol. time 20 minutes, average ILP size 8257 constraints x 5225 variables

Solved with optimality gap:

Closeness to optimality that can be proven by the solver
Main Results
for Itanium 2

Average speedup of 10%
  = 1/3-1/2 of GSL gain due to dynamic stalls
  - Benefit of schedule length reductions could be higher on processors with fine-grain SoEMT

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Number of scheduled Routines</th>
<th>Instructions</th>
<th>GCS w-IPC</th>
<th>ILP w-IPC</th>
<th>(Static) GSL Gains excl. SWP</th>
<th>Post GRA</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>400.perlbench</td>
<td>32</td>
<td>25702</td>
<td>3.02</td>
<td>4.41</td>
<td>32%</td>
<td>30%</td>
<td>12%</td>
</tr>
<tr>
<td>401.bzip2</td>
<td>11</td>
<td>11729</td>
<td>3.14</td>
<td>4.64</td>
<td>30%</td>
<td>19%</td>
<td>10%</td>
</tr>
<tr>
<td>445.gobmk</td>
<td>44</td>
<td>27263</td>
<td>2.89</td>
<td>4.20</td>
<td>27%</td>
<td>23%</td>
<td>7%</td>
</tr>
<tr>
<td>458.sjeng</td>
<td>14</td>
<td>9362</td>
<td>3.00</td>
<td>4.51</td>
<td>32%</td>
<td>28%</td>
<td>11%</td>
</tr>
<tr>
<td>473.astar</td>
<td>3</td>
<td>725</td>
<td>3.01</td>
<td>4.69</td>
<td>40%</td>
<td>37%</td>
<td>10%</td>
</tr>
<tr>
<td>Total</td>
<td>104</td>
<td>74781</td>
<td>3.0</td>
<td>4.5</td>
<td>32%</td>
<td>27%</td>
<td>10%</td>
</tr>
</tbody>
</table>
ILP: Different Modeled Target Microarchitectures

• Narrow machine with halved issue width (3 instr./cycle)
  – Half the number of execution units of each type
  – Two-cycle L1 cache latency
Results: GSLs 51% larger, w-IPC of 2.7

• Wide machine with double issue width (12 instr./cycle)
  – Twice the number of execution units of each type
Results: 8% GSL reduction, w-IPC of 5.1

Six-wide design of Itanium® 2 seems to match the available instruction-level parallelism best
Quantitatively, upward (speculative) code motion outnumbers all other classes.

ILP scheduler achieves shorter schedules with less code motion and speculation:
- Spill/fill percentage GCS vs. ILP: 0.8% vs. 0.2%
**ILP: Impact of Code Motion Classes**

- (Static) GSL gains over optimal *local* scheduling when enabling optimizations in the shown order.
- Overall *91% GSL gain*, demonstrating the tremendous importance of global instruction scheduling on wide-issue in-order architectures.
Conclusion

- **Substantial performance headroom** in global instruction scheduling on IPF
  - 10% over GCS at the highest optimization levels
  - Static weighted IPC increases from 3 to 4.5, demonstrating significant available instruction-level parallelism

- Experiments identified **three optimizations** with an outstanding GSL impact:
  - Speculative upward motion, cyclic code motion, block collapsing

- Solution times reasonable for targeted optimizations and research, yet still too large for the product compiler
  - May change in the long term because ILP solving is well parallelizable
Acknowledgments

• Kalyan Muthukumar

• Dan Lavery, Howard Chen, Gerolf Hoflehner, Darshan Desai

Questions?
Backup
**ILP: Impact of Scheduling Region Sizes**

- We have scheduled each routine with different *region size thresholds*, studied GSL impact

<table>
<thead>
<tr>
<th>#Instructions</th>
<th>GSL loss</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>-2%</td>
</tr>
<tr>
<td>250</td>
<td>-4%</td>
</tr>
<tr>
<td>125</td>
<td>-9%</td>
</tr>
<tr>
<td>62</td>
<td></td>
</tr>
</tbody>
</table>

- GCS scheduling regions are significantly *smaller* than those of the ILP scheduler (average 56 vs. 126)
- GCS regions are acyclic
  - loops nested away
- ILP regions can be cyclic
  - loops are first-class citizens
- Most of the benefit from the larger ILP regions comes from code motion into/out of loops
Cyclic Code Motion (CCM)

- Speculative upward code motion out of loop entry blocks
- Requires that compensation copies are moved across the back edge as well
  - These cyclic copies are subject to different, loop-carried dependences
  - Implementation stores possible cyclic copies in a common tail block, moves them upward synchronously with the other copies (  )

```c
ld V2=[V1] ;
chk.s V2
cmp.ge p1,p2 = V2,V3
ld.s V2=[V1]

st [V4]=V2,8
cmp.eq p3,p4 = 0,V4
ld.s V2=[V1]
```
Solution Time Optimizations

- Design of a functionally correct ILP model is comparably easy
- The challenge is to make the method scale well on larger problem instances
  - Solution time optimizations took at least half of the entire R&D effort
- Two approaches used
  1. Reduce ILP sizes:
     - Detect infeasible/definitely unprofitable code motion in advance, exclude from search space
  2. Improve polyhedral efficiency:
     - Main approach: Search for maximum cliques of mutually exclusive decision variables; extend clique constraints:
       \[ X_1 + X_2 + \ldots + X_n + X_{n+1} \leq 1 \]