9. Code Scheduling for ILP-Processors

- Software compilers optimizing code for ILP-processors, including VLIW

- 9.1 Introduction
- 9.2 Basic block scheduling
- 9.3 Loop scheduling
- 9.4 Global scheduling

Levels of static scheduling

- Basic block scheduling
- Loop scheduling
- Global scheduling

Amount of extracted parallelism, performance

Typical layout of compiler:

- Traditional, optimizing, pre-pass parallel, post-pass parallel

Contrasting the performance of software/hardware static code schedulers and ILP-processor

9.1 Basic Block Schedulers:

- List scheduler: in each step
  - Basic Block:
    - straight-line code,
    - can only enter at beginning and left at end

- Eligible Instructions are:
  - dependency-free
  - no predecessors in the Data Dependency Graph
  - or data to be produced by a predecessor node is already available

  - timing: checking when the data will be ready
  - hardware required resources are available

- Selects the set of eligible items
- Looks for the 'best choice' from the eligible set
- Looks for the 'Best Choice' from the eligible set of instructions (critical path in Data Dependency Graph)

  - Choose First: Schedule instructions in the critical path
  - Choose Next: others

**Heuristic Rule for choosing the next instructions**

**Case Study: compiler for IBM Power1**

- Basic block approach: list scheduler
- Using Data Dependency Graph
- Critical path
  - the longest (in term of execution time)
- Earliest time
  - when check the data for the instruction will be available
- Latency value (on each arc of DDG)
  - how many time units the successors node have to wait before the result of the predecessor node becomes available

**Example program:**

The intermediate code and the corresponding DDG

**The first three steps in scheduling the example DDG using Warren's algorithm**
Successive steps in scheduling the example DDG

9.3 Loop Scheduling

- Loop Unrolling
  - Loop Unrolled: basic concept
    - \( b(1) = 2 \times a(1) \)
    - \( b(2) = 2 \times a(2) \)
    - \( b(3) = 2 \times a(3) \)
  - save execution time at the expense of code length
  - omitting inter-iteration loop updating code
  - performance improvement
  - enlarging the basic block size
  - can lead to more effective schedule with considerable speed up

Simply unrolling is not practicable
  - when a loop has to be executed a large number of times
    - solution:
      - unroll the loop a given number of time (e.g. 3)
  - larger basic block
  - Inter-iteration dependency
    - only feasible if the required data becomes available in due time

Speed-up produced by loop unrolling for the first 14 Lawrence Livermore loops

<table>
<thead>
<tr>
<th>Loop</th>
<th>Unrolled by 2 blocks</th>
<th>Unrolled by 4 blocks</th>
<th>Unrolled by 8 blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>LLL *1</td>
<td>1.82</td>
<td>2.68</td>
<td>3.92</td>
</tr>
<tr>
<td>LLL *2</td>
<td>1.48</td>
<td>1.72</td>
<td>1.91</td>
</tr>
<tr>
<td>LLL *3</td>
<td>1.62</td>
<td>2.21</td>
<td>2.90</td>
</tr>
<tr>
<td>LLL *4</td>
<td>1.14</td>
<td>1.20</td>
<td>1.32</td>
</tr>
<tr>
<td>LLL *5</td>
<td>1.20</td>
<td>1.30</td>
<td>1.36</td>
</tr>
<tr>
<td>LLL *6</td>
<td>1.28</td>
<td>1.30</td>
<td>1.35</td>
</tr>
<tr>
<td>LLL *7</td>
<td>1.62</td>
<td>1.67</td>
<td>1.45</td>
</tr>
<tr>
<td>LLL *8</td>
<td>0.95</td>
<td>0.94</td>
<td>0.97</td>
</tr>
<tr>
<td>LLL *9</td>
<td>1.26</td>
<td>1.35</td>
<td>1.32</td>
</tr>
<tr>
<td>LLL *10</td>
<td>1.49</td>
<td>1.59</td>
<td>1.35</td>
</tr>
<tr>
<td>LLL *11</td>
<td>1.74</td>
<td>2.45</td>
<td>3.83</td>
</tr>
<tr>
<td>LLL *12</td>
<td>1.74</td>
<td>2.43</td>
<td>2.73</td>
</tr>
<tr>
<td>LLL *13</td>
<td>1.63</td>
<td>0.93</td>
<td>0.95</td>
</tr>
<tr>
<td>LLL *14</td>
<td>1.03</td>
<td>0.65</td>
<td>0.94</td>
</tr>
<tr>
<td>Aggregate</td>
<td>1.24</td>
<td>1.20</td>
<td>1.36</td>
</tr>
<tr>
<td>Overall</td>
<td>1.37</td>
<td>1.56</td>
<td>1.42</td>
</tr>
</tbody>
</table>

* : run on workstations
# : run on workstations, where memory hazards could be resolved during scheduling
Software Pipelining

- Software pipelining is an analogous term to hardware pipelining.
- Each pipeline stage uses one EU.
- Each pipeline cycle executes one instruction.

E.G.:

for I = 1 to 7 do {
    b(I) = 2 * a(I)
}

Basic methods for software pipelining

Unrolling-based techniques

- Principle of the unrolling-based URPR software pipelining method

E.G. loop 7 times, 4 EU's VLIW, fmul takes 3 cycles

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Iteration number</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>load</td>
</tr>
<tr>
<td>c+1</td>
<td>fmul load</td>
</tr>
<tr>
<td>c+2</td>
<td>decr fmul load</td>
</tr>
<tr>
<td>c+3</td>
<td>nop decr fmul load</td>
</tr>
<tr>
<td>c+4</td>
<td>store nop decr fmul load</td>
</tr>
<tr>
<td>c+5</td>
<td>store nop decr fmul load</td>
</tr>
<tr>
<td>c+6</td>
<td>store nop decr fmul load</td>
</tr>
<tr>
<td>c+7</td>
<td>store nop decr</td>
</tr>
<tr>
<td>c+8</td>
<td>store nop</td>
</tr>
<tr>
<td>c+9</td>
<td>store</td>
</tr>
<tr>
<td>c+10</td>
<td>store</td>
</tr>
</tbody>
</table>

Modulo scheduling

- Find the repetitive character of the schedule.
- Guess the minimal required length of the new (partially unrolled) loop.
- Period of repetition.
- Try schedule for this interval [period].
  - Taking into account data and resource dependencies.
- If the schedule is not feasible.
  - Then the length [period] is increased.
- Try again.

9.4 Global Scheduling:
Scheduling of individual instructions beyond basic blocks

<table>
<thead>
<tr>
<th>Code scheduling</th>
<th>Loop scheduling</th>
<th>Global scheduling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic block</td>
<td>Slightly parallel</td>
<td>ILP-processors</td>
</tr>
<tr>
<td>scheduling</td>
<td>Moderately parallel</td>
<td></td>
</tr>
<tr>
<td>Loop scheduling</td>
<td>Highly parallel</td>
<td></td>
</tr>
</tbody>
</table>

| | Traditional pipelined | Superpipelined processors |
|-----------------------|--------------------------|
| * YUV processors      | Highly superscalar processors |
Moving up an independent instruction beyond a basic block boundary ‘along both paths’ of a conditional branch

Moving up an independent instruction beyond a basic block boundary ‘along a single path’ of a conditional branch [guess T]

Trace Scheduling: flow graph, guess 3

Bookkeeping Code [B]

Finite Resource Global Scheduling: solving long compilation times and code explosion

- For VLIW
  - 3.7 Speed-up over Power1
  - 49% Compilation time over PL/I
  - 2.1 Code explosion over RISC

Input code
Software window

Dependency free code for the processor

In the case of a supercomputer for a parallel processor

In the case of a VLIW processor