CSE P 501 – Compilers

Instruction Scheduling
Hal Perkins
Spring 2018

UW CSE P 501 Spring 2018
Agenda

• Instruction scheduling issues – latencies

• List scheduling
Issues (1)

- Many operations have non-zero latencies
- Modern machines can issue several operations per cycle
  - Want to take advantage of multiple function units on chip
- Loads & Stores may or may not block
  - may be slots after load/store for other useful work
Issues (2)

- Branch costs vary
- Branches on some processors have delay slots
- Modern processors have heuristics to predict whether branches are taken and try to keep pipelines full

- GOAL: Scheduler should reorder instructions to hide latencies, take advantage of multiple function units and delay slots, and help the processor effectively pipeline execution
Latencies for a Simple Example Machine

<table>
<thead>
<tr>
<th>Operation</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD</td>
<td>3</td>
</tr>
<tr>
<td>STORE</td>
<td>3</td>
</tr>
<tr>
<td>ADD</td>
<td>1</td>
</tr>
<tr>
<td>MULT</td>
<td>2</td>
</tr>
<tr>
<td>SHIFT</td>
<td>1</td>
</tr>
<tr>
<td>BRANCH</td>
<td>0 TO 8</td>
</tr>
</tbody>
</table>

UW CSE P 501 Spring 2018
Example: \( w = w \cdot 2 \cdot x \cdot y \cdot z; \)

Simple schedule

- 1 LOAD \( r_1 \leftarrow w \)
- 4 ADD \( r_1 \leftarrow r_1, r_1 \)
- 5 LOAD \( r_2 \leftarrow x \)
- 8 MULT \( r_1 \leftarrow r_1, r_2 \)
- 9 LOAD \( r_2 \leftarrow y \)
- 12 MULT \( r_1 \leftarrow r_1, r_2 \)
- 13 LOAD \( r_2 \leftarrow z \)
- 16 MULT \( r_1 \leftarrow r_1, r_2 \)
- 18 STORE \( w \leftarrow r_1 \)

21 \( r_1 \) free

2 registers, 20 cycles

Loads early

- 1 LOAD \( r_1 \leftarrow w \)
- 2 LOAD \( r_2 \leftarrow x \)
- 3 LOAD \( r_3 \leftarrow y \)
- 4 ADD \( r_1 \leftarrow r_1, r_1 \)
- 5 MULT \( r_1 \leftarrow r_1, r_2 \)
- 6 LOAD \( r_2 \leftarrow z \)
- 7 MULT \( r_1 \leftarrow r_1, r_3 \)
- 9 MULT \( r_1 \leftarrow r_1, r_2 \)
- 11 STORE \( w \leftarrow r_1 \)
- 14 \( r_1 \) is free

3 registers, 13 cycles
Instruction Scheduling

• Problem
  – Given a code fragment for some machine and
    latencies for each operation, reorder to minimize
    execution time

• Constraints
  – Produce correct code (required)
  – Minimize wasted cycles
  – Avoid spilling registers if possible
  – Do this efficiently
Precedence Graph

- Nodes $n$ are operations
- Attributes of each node
  - type – kind of operation
  - delay – latency
- If node $n_2$ uses the result of node $n_1$, there is an edge $e = (n_1, n_2)$ in the graph
Example Graph

- Code
  a LOAD r1 <- w
  b ADD r1 <- r1, r1
  c LOAD r2 <- x
  d MULT r1 <- r1, r2
  e LOAD r2 <- y
  f MULT r1 <- r1, r2
  g LOAD r2 <- z
  h MULT r1 <- r1, r2
  i STORE w <- r1

UW CSE P501 Spring 2018

Cycle x2 y3 z3 w2
Ready y x w z
Active x y w z
Schedules (1)

- A correct schedule $S$ maps each node $n$ into a non-negative integer representing its cycle number, and
  - $S(n) \geq 0$ for all nodes $n$ (obvious)
  - If $(n_1, n_2)$ is an edge, then $S(n_1) + \text{delay}(n_1) \leq S(n_2)$
  - For each type $t$, there are no more operations of type $t$ in any cycle than the target machine can issue
Schedules (2)

- The *length* of a schedule $S$, denoted $L(S)$ is
  $$L(S) = \max_n \left( S(n) + \text{delay}(n) \right)$$
- The goal is to find the shortest possible correct schedule
  - Other possible goals: minimize use of registers, power, space, ...
Constraints

• Main points
  – All operands must be available
  – Multiple operations can be ready at any given point
  – Moving operations can lengthen register lifetimes
  – Moving uses near definitions can shorten register lifetimes
  – Operations can have multiple predecessors
• Collectively this makes scheduling NP-complete
• Local scheduling is the simpler case
  – Straight-line code
  – Consistent, predictable latencies
Algorithm Overview

• Build a precedence graph $P$
• Compute a priority function over the nodes in $P$
  (typical: longest latency-weighted path)
• Use list scheduling to construct a schedule, one cycle at a time
  – Use queue of operations that are ready
  – At each cycle
    • Chose a ready operation and schedule it
    • Update ready queue
• Rename registers to avoid false dependencies and conflicts
List Scheduling Algorithm

Cycle = 1; Ready = leaves of P; Active = empty;
while (Ready and/or Active are not empty)
    if (Ready is not empty)
        remove an op from Ready;
        S(op) = Cycle;
        Active = Active ∪ op;
        Cycle++;
    for each op in Active
        if (S(op) + delay(op) <= Cycle)
            remove op from Active;
            for each successor s of op in P
                if (s is ready – i.e., all operands available)
                    add s to Ready
Example

• Code
  a LOAD r1 <- w
  b ADD r1 <- r1,r1
  c LOAD r2 <- x
  d MULT r1 <- r1,r2
  e LOAD r2 <- y
  f MULT r1 <- r1,r2
  g LOAD r2 <- z
  h MULT r1 <- r1,r2
  i STORE w <- r1
Forward vs Backwards

• Backward list scheduling
  – Work from the root to the leaves
  – Schedules instructions from end to beginning of the block

• In practice, compilers try both and pick the result that minimizes costs
  – Little extra expense since the precedence graph and other information can be reused
  – Different directions win in different cases
Beyond Basic Blocks

- List scheduling dominates, but moving beyond basic blocks can improve quality of the code. Some possibilities:
  - Schedule extended basic blocks
    - Watch for exit points – limits reordering or requires compensating
  - Trace scheduling
    - Use profiling information to select regions for scheduling using traces (paths) through code