

CSE P 501 – Compilers

Instruction Scheduling

Hal Perkins

Winter 2016

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

Operation	Cycles
LOAD	3
STORE	3
ADD	1
MULT	2
SHIFT	1
BRANCH	0 TO 8

Example: $w = w * 2 * x * y * z;$

Simple schedule

```
1 LOAD    r1 <- w
4 ADD     r1 <- r1,r1
5 LOAD    r2 <- x
8 MULT    r1 <- r1,r2
9 LOAD    r2 <- y
12 MULT   r1 <- r1,r2
13 LOAD   r2 <- z
16 MULT   r1 <- r1,r2
18 STORE  w <- r1
21 r1 free
```

2 registers, 20 cycles

Loads early

```
1 LOAD    r1 <- w
2 LOAD    r2 <- x
3 LOAD    r3 <- y
4 ADD     r1 <- r1,r1
5 MULT    r1 <- r1,r2
6 LOAD    r2 <- z
7 MULT    r1 <- r1,r3
9 MULT    r1 <- r1,r2
11 STORE  w <- r1
14 r1 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
```

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
$$L(S) = \max_n (S(n) + \text{delay}(n))$$
- 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  $\cup$  op;
  Cycle++;
  for each op in Active
    if (S(op) + delay(op)  $\leq$  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