


CSE 582 – Compilers

Instruction Scheduling

Hal Perkins
Autumn 2002

11/21/2002 © 2002 Hal Perkins & UW CSE O-1




Agenda

- n Instruction scheduling issues – latencies
- n List scheduling

Credits: Adapted from slides by Keith Cooper, Rice University

11/21/2002 © 2002 Hal Perkins & UW CSE O-2



Issues

- n Many operations have non-zero latencies
- n Modern machines can issue several operations per cycle
- n Loads & Stores may or may not block
 - n ∴ may be slots after load/store for other work
- n Branch costs vary
- n Branches on modern processors typically have delay slots
- n GOAL: Scheduler should reorder instructions to hide latencies and take advantage of delay slots

11/21/2002 © 2002 Hal Perkins & UW CSE O-3

Some Idealized Latencies

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

11/21/2002 © 2002 Hal Perkins & UW CSE O-4

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


```

11/21/2002 © 2002 Hal Perkins & UW CSE O-5

Instruction Scheduling

- Problem
 - Given a code fragment for some machine and latencies for each operation, reorder to minimize execution time
- Constraints
 - Produce correct code
 - Minimize wasted cycles
 - Avoid spilling registers
 - Do this efficiently


11/21/2002 © 2002 Hal Perkins & UW CSE O-6



Precedence Graph

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


11/21/2002 © 2002 Hal Perkins & UW CSE O-7



Example Graph

- n 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


11/21/2002 © 2002 Hal Perkins & UW CSE O-8



Schedules (1)

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

11/21/2002 © 2002 Hal Perkins & UW CSE O-9




Schedules (2)

- n The *length* of a schedule S , denoted $L(S)$ is

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


11/21/2002 © 2002 Hal Perkins & UW CSE O-10



Constraints

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


11/21/2002 © 2002 Hal Perkins & UW CSE O-11



Algorithm Overview

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

11/21/2002 © 2002 Hal Perkins & UW CSE O-12




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
  
```

11/21/2002 © 2002 Hal Perkins & UW CSE O-13




Example

n 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
  
```

11/21/2002 © 2002 Hal Perkins & UW CSE O-14



Variations

- n Backward list scheduling
 - n Work from the root to the leaves
 - n Schedules instructions from end to beginning of the block
- n In practice, try both and pick the result that minimizes costs
 - n Little extra expense since the precedence graph and other information can be reused
- n Global scheduling and loop scheduling
 - n Extend basic idea in more aggressive compilers

11/21/2002 © 2002 Hal Perkins & UW CSE O-15
