

## Instruction Scheduling

### Motivation

- instruction latency (pipelining)  
several cycles to complete instruction
- instruction-level parallelism (VLIW, superscalar)  
execute multiple instructions per cycle

### Issues

- reorder instructions to reduce execution time
- static schedule → insert NOPs
- dynamic schedule → pipeline stalls
- preserve correctness, improve performance
- interactions with optimizations

### Sources of latency (hazards)

- data - operands depend on previous instruction
- structural - limited hardware resources
- control - targets of conditional branches

## Data Dependences

Dependences ⇒ memory locations instead of values

Statement  $b$  depends on statement  $a$  if there exists:

- true or flow dependence  
 $a$  writes a location that  $b$  later reads (read-after-write or RAW)
- anti-dependence  
 $a$  reads a location  $b$  later writes (write-after-read or WAR)
- output dependence  
 $a$  writes a location that  $b$  later writes (write-after-write or WAW)

Another dependence (doesn't constrain ordering)

- input dependence  
 $a$  reads a location that  $b$  later reads (read-after-read or RAR)

### Example

// true	// anti	// output	// input
a =	= a	a =	= a
= a	a =	a =	= a

## Instruction Scheduling

### Approach

- schedule after register allocation (postpass)
- model used to estimate execution time

### A legal schedule

- assume each  $i \in Instr$  has  $delay(i)$
- a legal schedule  $S$  maps each  $i \in Instr$  onto a non-negative integer representing its cycle number (i.e., time instruction is executed)
- if  $i_2$  "depends" on  $i_1$ ,  $S(i_1) + delay(i_1) \leq S(i_2)$
- the *length* of schedule  $S$ , denoted  $L(S)$ , is

$$L(S) = \max_{i \in Instr} (S(i) + delay(i))$$

- $S$  is optimal if  $L(S) \leq L(T), \forall$  legal schedules  $T$

### Scope

- basic blocks (list scheduling)
- branches (trace scheduling, percolation)
- loops (unrolling, software pipelining)

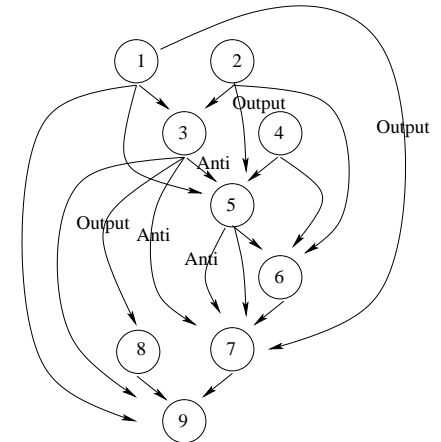
## Precedence Graph

### Construction

- instructions ⇒ nodes
- dependences ⇒ edges

Example <op> <dst, s1, s2>

```
1 load r1, X
2 load r2, Y
3 mult r3, r2, r1
4 load r4, A
5 mult r2, r4, r1
6 add r5, r2, r4
7 mult r1, r2, r5
8 load r3, B
9 add r7, r1, r3
```



Renaming can eliminate anti & output dependences

a =	a =
= a	= a
a =	b =
= a	= b

## List Scheduling

### Algorithm

1. rename to eliminate anti/output dependences (optional)
2. construct precedence graph
3. assign priorities to instructions
4. iteratively select & schedule instructions
  - (a) candidates  $\leftarrow$  roots of graph
  - (b) while candidates remaining
    - i. pick highest priority candidate
    - ii. schedule instruction
    - iii. add exposed instructions to candidates

### Two flavors of list scheduling

#### Forward list scheduling

- start with available ops
- work forward
- ready  $\Rightarrow$  all ops available

#### Backward list scheduling

- start with no successors
- work backward
- ready  $\Rightarrow$  latency covers uses

## Scheduling Example

### Compute critical path

- build precedence graph
- start from instruction(s) with no successors, work backwards
- weight of node = instruction latency + maximum successor weight

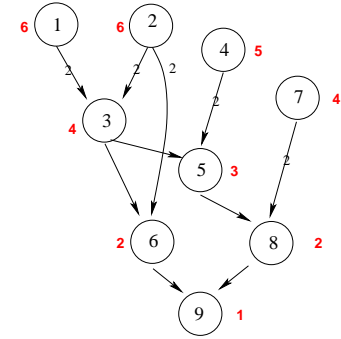
### Example machine model

- 2-cycle latency for load
- 1-cycle latency otherwise

### Example code

```

1 load r1, X
2 load r2, Y
3 mult r3, r2, r1
4 load r4, A
5 mult r5, r4, r3
6 add r6, r2, r3
7 load r7, B
8 mult r8, r5, r7
9 jmp
    
```



## Scheduling heuristics

### Problems

- how to choose between ready instructions?
- NP-hard for straight-line code

### Heuristics used to prioritize candidates

1. will not cause pipeline stall
2. longest weighted path to root (critical path)
3. highest latency (more overlap)
4. most immediate successors (create candidates)
5. most descendants (create more candidates)

### Approach

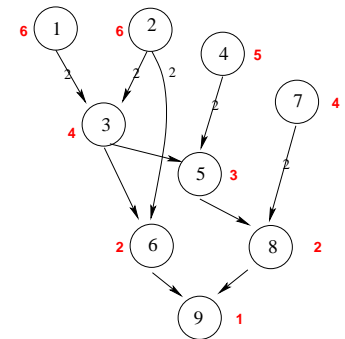
- use multiple heuristics (to help break ties)
- try multiple schedules (then take best result)

## Scheduling Example

Cycle	1	2	3	4	5	6	7	8	9
Candidates	1,2,4,7	1,4,7	4,7	3,7	5,6,7	5,6	6,8	8	9
Single-issue schedule	2	1	4	3	7	5	6	8	9

Cycle	1	2	3	4	5	6
Candidates	1,2,4,7	4,7	3	5,6,8	8	9
Dual-issue schedule	1	4	3	5	8	9
	2	7		6		

Cycle	1	2	3	4	5	6
Candidates	1,2	4	3,7	5	6,8	9
Dual-issue schedule	1	4	3	5	6	9
(backwards)	2		7		8	



## Trace Scheduling

### Overview

- a *trace* is a path through code
- examine branch probabilities, find trace representing most likely path
- schedule instructions for trace, emit *repair code* around trace
- repeat as necessary

### Example

```

code1                code1
code2  code4         code2  code4
                    ---->
code3  code5         code3  code5
                    ---->
code6                code6

```

### Result

- better instruction schedule along trace
- less efficient schedule off trace
- increase in code size (from repair code)

## Loops Unrolling

### Approach

- create multiple copies of loop
- more candidates for scheduling

### Example

```

// do i=1,N      // do i=1,N,3
1   load         load
2                               load
3   add          add    load
4   store        store  add
5                               store  add
6                               store

```

### Problems

- choosing degree of unrolling  $k$
- pipeline hiccup every  $k$  iterations
- increased compilation time
- instruction cache overflow

## Trace Scheduling Repair Code

### Code moved below split

- create basic block  $\mathcal{B}$  at branch target
- replicate code  $\mathcal{C}$  moved below split
- insert  $\mathcal{C}$  in  $\mathcal{B}$  in original order

```

code1      if (...)
if (...)   code1    code1
           code2    code3
code2      code3

```

### Code moved above split

- only move code which is dead off trace
- may perform unnecessary work

```

code1      code1
if (...)   code2
           if (...)
code2      code3          // code2
                           // is dead
                           code3

```

## Software Pipelining

### Approach

- overlap iterations of loop
- select schedule for loop body, with constant initiation interval
- initiate iteration after every  $k$  cycles, before previous iterations complete

### Example

```

T   i=1  i=2  i=3  i=4
1   load
2           load
3   add          load
4 L: store add          load (cjmp L)
5           store add
6           store add
7           store

```

### Properties

- prolog, epilog code for pipeline
- steady state within body of loop
- hiccups in pipeline at entry, exit

## Branch Prediction

---

Will a conditional branch be taken?

- affects instruction scheduling
- execution penalty for incorrect guess

Prediction approaches

- hardware history (branch bit)
- history plus branch correlation
- profiling (feedback to compiler)
- compile-time heuristics

Static branch prediction

- no run-time information
- single prediction for all executions
- perfect prediction = 50–100% correct

Simple (target) heuristic

- predict conditional branch taken
- catches loop back edges

## Predication

---

Predicated instructions

- method to deal with conditional branches
- designed for small number of instructions

Mechanism

- instructions are executed *conditionally*
- predicate allowed for each instruction
- instruction only executed if predicate is true
- example [P1] MOV R1, 0  $\Rightarrow$  if (P1 == true) R1 = 0

Issues

- branch prediction, instruction scheduling still useful
- false predicate  $\rightarrow$  wasted NOP instruction
- can use dedicated predicate registers (64 1-bit registers in IA64)
- more complexity for the compiler...

## Branch Prediction

---

Loop branch heuristic

- find forward, back, exit edges
- predict back edge, non-exit edge

Op code heuristic

- predict greater than zero (error conditions)
- predict floating point values differ

Loop heuristic

- predict branch leading to loop header

Call heuristic

- predict branch not leading to call

Return heuristic

- predict branch not leading to return

Guard heuristic

- predict branch leading to guarded variable

etc...

## Predication example

---

```
Source code      if (a == b)
                  a = 0;
                  else
                  a = b;
```

Branching code

```
                CMP R1, R2      // check (a == b)
                JNE else        // if no, goto else
                MOV R1, 0       // R1 = 0
                JMP end         // skip past else
else MOV R1, R2                // R1 = R2
end   ...                      // rest of program
```

Predicated code

```
                CMPEQ R1, R2, P1/P2 // check (a == b) sets P1, P2
[P1] MOV R1, 0      // if (P1 == true) R1 = 0
[P2] MOV R1, R2    // if (P2 == true) R2 = R1
```