

## Instruction Scheduling

---

### Motivation

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

### Issues

- reorder instructions to reduce execution time
- static schedule  $\rightarrow$  insert NOPs
- dynamic schedule  $\rightarrow$  pipeline stalls
- preserve correctness
- operate efficiently
- interactions with optimizations

### Sources of latency (hazards)

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

## 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  $i_1$ ,  $S(i_1) + delay(i_1) \leq S(i_2)$
- there are no more instructions in any cycle than can be issued by the machine
- 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)

## Data Dependences

---

Dependences  $\Rightarrow$  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
```

## Precedence Graph

---

Construction

- instructions  $\Rightarrow$  nodes
- dependences  $\Rightarrow$  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 r5, r4, r3
6 add r6, r2, r3
7 mult r2, r5, r6
8 load r8, B
9 add r9, r8, r2
```

Register renaming eliminates anti & output deps

```
a =          a =
= a          = a
a =          b =
= a          = b
```

## List Scheduling

---

### Algorithm

1. rename to eliminate anti/output deps
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 heuristics

---

### Problems

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

### Heuristics used to prioritize candidates

1. ready to execute (no stalls)
2. highest latency (more overlap)
3. most immediate successors (create candidates)
4. most descendants (create more candidates)
5. longest weighted path to root (critical path)

### Approach

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

## Scheduling Example

---

### Machine model

- 3-cycle latency for load
- 2-cycle latency for mult
- 1-cycle latency for add

### Example

- 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

| Cycle      | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|------------|---|---|---|---|---|---|---|---|
| Candidates |   |   |   |   |   |   |   |   |
| Forw sched |   |   |   |   |   |   |   |   |

|            |  |  |  |  |  |  |  |  |
|------------|--|--|--|--|--|--|--|--|
| Candidates |  |  |  |  |  |  |  |  |
| Back sched |  |  |  |  |  |  |  |  |

|              |  |  |  |  |  |  |  |  |
|--------------|--|--|--|--|--|--|--|--|
| Candidates   |  |  |  |  |  |  |  |  |
| 2-inst sched |  |  |  |  |  |  |  |  |

## 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)

## Trace Scheduling Repair Code

---

### Code moved below split

- create basic block  $B$  at branch target
- replicate code  $C$  moved below split
- insert  $C$  in  $B$  in original order
- example

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

### Code moved above split

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

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

## Loops Unrolling

---

### Approach

- create multiple copies of loop
- more candidates for scheduling

### Example

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

### Problems

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

## Software Pipelining

---

### Approach

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

### Example

|   |          |       |       |          |
|---|----------|-------|-------|----------|
| T | i=1      | i=2   | i=3   | i=4      |
| 1 | load     |       |       |          |
| 2 | mult     | load  |       |          |
| 3 |          | mult  | load  |          |
| 4 | L: store | mult  | load  | (cjmp L) |
| 5 |          | store | mult  |          |
| 6 |          |       | store |          |
| 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

## **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

### **Store heuristic**

- predict branch leading to store of variable

### **Pointer heuristic**

- predict pointer not NULL, pointers differ