**Instruction Scheduling**

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

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

---

**Data Dependencies**

Dependency ⇒ 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 that $b$ 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 = a =   a
```

---

**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 <- 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 ⇒ all ops available 
- **Backward list scheduling**
  - start with no successors 
  - work backward 
  - ready ⇒ latency covers uses

---

**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 $i_1$, $S(i_1) + \text{delay}(i_1) \leq S(i_2)$
- the length of schedule $S$, denoted $L(S)$, is 
  $L(S) = \max_i S(i) + \text{delay}(i)$
- $S$ is optimal if $L(S) \leq L(T)$, for legal schedule $T$

**Scope**
- basic blocks 
  - (list scheduling)
- branches 
  - (trace scheduling, peeling)
- loops 
  - (unrolling, software pipelining)

---

**Precedence Graph**

**Construction**
- instructions ⇒ no ops 
- dependancies ⇒ edges

**Example**

```
<op> <dst, st1, st2>
  1 bad r1, X
  2 bad r2, Y
  3 mult r3, r1
  4 bad r4, A
  5 mult r2, r1
  6 add r5, r2, r4
  7 mult r4, r2, r3
  8 bad r6, B
  9 add r7, r7, r1, r3
```

**Renaming can eliminate anti & output dependences**

```
a =   a =   a =
- a   - a - a =
```

**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 dependants (create more candidates)

**Approach**
- uses multiple heuristics (to help break ties) 
- try multiple schedules (then take best result)
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
- cycle latency for load
- cycle latency otherwise

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

Trace Scheduling

Overview
- 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

<table>
<thead>
<tr>
<th>code1</th>
<th>code1</th>
</tr>
</thead>
<tbody>
<tr>
<td>code2</td>
<td>code4</td>
</tr>
<tr>
<td>code3</td>
<td>code5</td>
</tr>
<tr>
<td>code6</td>
<td></td>
</tr>
</tbody>
</table>

Code moved below split

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

Code moved above split

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

```
code2 code3
// is dead
```

Trace Scheduling 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 add       load
3 add       add
4 store     store
5 store     add
6 store
```

Problems
- choosing degree of unrolling k
- pipeline hazard every k iterations
- increased compilation time
- instruction cache overflow

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 store add load (cjmp L)
5 store add
6 store
7 store
```

Properties
- probing, epitaxial code for pipeline
- steady state within body of loop
- hazards 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 hit)
- 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 ⇒ if (P1 === true) R1 = 0

Issues
- branch prediction, instruction scheduling still useful
- false predicate → wasted NOP instruction
- can use dedicated predicate registers (64-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
- predict floating point values differ
(error conditions)

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