CS415 Compilers
Instruction Scheduling
Part 2

These slides are based on slides copyrighted by Keith Cooper, Ken Kennedy & Linda Torczon at Rice University
Announcements

• First project and second homework have been posted

• First homework deadline extension: Tuesday Feb. 9 @11:59pm; sample solution will be posted by tomorrow (sakai/Resources)

• First quiz will be posted over the weekend. Topic: register allocation; you will have a window of a few days to complete the quiz (30 - 45 minutes)

• Please attend recitations!
• register-register model
  → Values that may safely reside in registers are assigned to a unique virtual register (alias analysis; unambiguous values); there are different “flavors”

• memory-memory model
  → All program-named values reside in memory, and are only kept in registers as briefly as possible (load operands from memory, perform computation, store result back into memory)

```
a := 1
b := 2
c := a + b + 3
```

```
loadI 1 ⇒ r1
loadI 2 ⇒ r2
add r1, r2 ⇒ r3
loadI 3 ⇒ r4
add r3, r4 ⇒ r5
loadI 1 ⇒ r1
loadI 2 ⇒ r2
loadI r0, @a
loadI r0, @b
loadI r3
loadI r4
add r3, r4 ⇒ r5
loadI 3 ⇒ r7
add r5, r7 ⇒ r8
storeAI r8 ⇒ r0, @c
```

```
loadI 1 ⇒ r1
storeAI r1 ⇒ r0, @a
loadI 2 ⇒ r2
storeAI r2 ⇒ r0, @b
loadAI r0, @a ⇒ r3
loadAI r0, @b ⇒ r4
add r3, r4 ⇒ r5
loadI 3 ⇒ r7
add r5, r7 ⇒ r8
storeAI r8 ⇒ r0, @c
```

preserve memory view

all in registers
loadI 1 ⇒ r1
loadI 2 ⇒ r2
add r1, r2 ⇒ r3
loadI 3 ⇒ r4
add r3, r4 ⇒ r5
storeAI r5 ⇒ r0, @c

register-register
Instruction Scheduling

EaC Chapter 12
12.1 - 12.3

Part of the compiler’s back end

Instruction Selection

Register Allocation

Instruction Scheduling

Machine code

IR → m register → IR

IR → k register → IR

IR → Machine code
A **correct schedule** $S$ maps each $n \in N$ into a non-negative integer representing its **cycle number** such that

1. $S(n) \geq 0$, for all $n \in N$, **obviously**
2. If $(n_1, n_2) \in E$, $S(n_1) + \text{delay}(n_1) \leq S(n_2)$

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

$$L(S) = \max_{n \in N} (S(n) + \text{delay}(n))$$

The goal is to find the shortest possible correct schedule. $S$ is **time-optimal** if $L(S) \leq L(S_1)$, for all other schedules $S_1$

**Note**: we are trying to minimize execution time here.
Critical Points
• All operands must be available
• Multiple operations can be ready
• Instructions can have multiple predecessors
Together, these issues make scheduling hard (NP-Complete)

Idealized execution model:
→ Issue single instruction per cycle
→ Multiple instructions “in-flight” (active)
→ Independent instructions can execute out-of-order

Local scheduling is the simple case
• Restricted to straight-line code (single basic block)
• Consistent and predictable latencies
The big picture
1. Build a dependence graph, \( P \)
2. Compute a *priority function* over the nodes in \( P \)
3. Use list scheduling to construct a schedule, one cycle at a time
   (can only issue/schedule at most one instructions per cycle)
   a. Use a set of operations that are ready
   b. At each cycle
      I. Choose a ready operation and schedule it
      II. Increment cycle
      III. Update the ready set

Local list scheduling
• The dominant algorithm for many years
• A greedy, heuristic, local technique
Scheduling Example

<table>
<thead>
<tr>
<th>Operation</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>load</td>
<td>3</td>
</tr>
<tr>
<td>loadl</td>
<td>1</td>
</tr>
<tr>
<td>loadAI</td>
<td>3</td>
</tr>
<tr>
<td>store</td>
<td>3</td>
</tr>
<tr>
<td>storeAI</td>
<td>3</td>
</tr>
<tr>
<td>add</td>
<td>1</td>
</tr>
<tr>
<td>mult</td>
<td>2</td>
</tr>
<tr>
<td>fadd</td>
<td>1</td>
</tr>
<tr>
<td>fmult</td>
<td>2</td>
</tr>
<tr>
<td>shift</td>
<td>1</td>
</tr>
<tr>
<td>output</td>
<td>1</td>
</tr>
<tr>
<td>outputAI</td>
<td>1</td>
</tr>
</tbody>
</table>

• Loads & stores may or may not block
  > Non-blocking ⇒ fill those issue slots

• Branches typically have delay slots
  > Fill slots with operations unrelated to branch condition evaluation
  > Percolates branch upward

• Branch Prediction may hide branch latencies (hardware feature)

Build a simple local scheduler (basic block)
- non-blocking loads & stores
- different latencies load/store vs. arith. etc. operations
- different heuristics
- forward / backward scheduling
1. Build the dependence graph

S(n):

0  a:  loadAI  r0,@w  ⇒ r1
3  b:  add    r1,r1  ⇒ r1
4  c:  loadAI r0,@x  ⇒ r2
7  d:  mult   r1,r2  ⇒ r1
8  e:  loadAI r0,@y  ⇒ r3
11  f:  mult   r1,r3  ⇒ r1
12  g:  loadAI r0,@z  ⇒ r2
15  h:  mult   r1,r2  ⇒ r1
17  i:  storeAI r1     ⇒ r0,@w
20

The Code

⇒ 20

cycles

The Dependence Graph
1. Build the dependence graph
2. Determine priorities: **longest latency-weighted path**

The Code

```
a: loadAl r0,@w  ⇒ r1
b: add  r1,r1  ⇒ r1
c: loadAl r0,@x  ⇒ r2
d: mult  r1,r2  ⇒ r1
e: loadAl r0,@y  ⇒ r3
f: mult  r1,r3  ⇒ r1
g: loadAl r0,@z  ⇒ r2
h: mult  r1,r2  ⇒ r1
i: storeAl  r1  ⇒ r0,@w
```

The Dependence Graph

true

anti
List Scheduling Example

The Code

\[
S(n) = 
\]

The Generated Code

\[
a: \text{loadAI.} \ r0, @w \Rightarrow r1 \\
b: \text{add} \ r1, r1 \Rightarrow r1 \\
c: \text{loadAI} \ r0, @x \Rightarrow r2 \\
d: \text{mult} \ r1, r2 \Rightarrow r1 \\
e: \text{loadAI} \ r0, @y \Rightarrow r3 \\
f: \text{mult} \ r1, r3 \Rightarrow r1 \\
g: \text{loadAI} \ r0, @z \Rightarrow r2 \\
h: \text{mult} \ r1, r2 \Rightarrow r1 \\
i: \text{storeAI} \ r1 \Rightarrow r0, @w
\]

The Dependence Graph

(The longest latency-weighted)
List Scheduling Example

The Code

```
0: a: loadAI. r0, @w => r1
1: c: loadAI r0, @x => r2
2: e: loadAI r0, @y => r3
3: b: add r1, r1 => r1
4: d: mult r1, r2 => r1
5: g: loadAI r0, @z => r2
6: f: mult r1, r3 => r1
7: h: mult r1, r2 => r1
8: i: storeAI r1 => r0, @w
9: 10: 11: 12: 13: 14:

The Generated Code

S(n) =

READY - SET

ACTIVE - SET

CYCLE = 14
```

The Dependence Graph

(longest latency-weighted)
Local (Forward) List Scheduling

\begin{align*}
\text{Cycle} & \leftarrow 0 \\
\text{Ready} & \leftarrow \text{leaves of } P \\
\text{Active} & \leftarrow \emptyset \\
\end{align*}

while (Ready \cup \text{Active} \neq \emptyset)
\begin{align*}
\text{if } (\text{Ready} \neq \emptyset) \text{ then} \\
\quad \text{remove an op from Ready} \\
\quad \text{S}(\text{op}) \leftarrow \text{Cycle} \\
\quad \text{Active} \leftarrow \text{Active} \cup \text{op} \\
\end{align*}
\text{Cycle} \leftarrow \text{Cycle} + 1

\text{for each } \text{op} \in \text{Active}
\begin{align*}
\text{if } (\text{S}(\text{op}) + \text{delay}(\text{op}) \leq \text{Cycle}) \text{ then} \\
\quad \text{remove op from Active} \\
\quad \text{for each successor s of op in P} \\
\qquad \text{if } (\text{s is ready}) \text{ then} \\
\qquad \quad \text{Ready} \leftarrow \text{Ready} \cup \text{s} \\
\end{align*}
Wrap-up Instruction Scheduling

Lexical Analysis

Read EaC: Chapters 2.1 - 2.5;