CS415 Compilers

Instruction Scheduling

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

• First project and second homework will come out on tonight or tomorrow morning

• Homeworks: No late submissions; however, you may ask for an extension in class

• Paper on register allocation on web site (8th International Conference on Compiler Construction, CC’99, March 1999)
Instruction Scheduling

Motivation

• Instruction latency (pipelining)
  several cycles to complete instructions; instructions can be issued every cycle
• Instruction-level parallelism (VLIW, superscalar)
  execute multiple instructions per cycle

Issues

• Reorder instructions to reduce execution time
• Static schedule - insert NOPs to preserve correctness
• Dynamic schedule - hardware pipeline stalls
• Preserve correctness, improve performance
• Interactions with other optimizations (register allocation!)
Motivation

- Instruction latency (pipelining)
  several cycles to complete instructions; instructions can be issued every cycle
- Instruction-level parallelism (VLIW, superscalar)
  execute multiple instructions per cycle

Issues

- Reorder instructions to reduce execution time
- Static schedule - insert NOPs to preserve correctness
- Dynamic schedule - hardware pipeline stalls
- Preserve correctness, improve performance
- Interactions with other optimizations (register allocation!)

Note: After register allocation, code shape contains real, not virtual registers ==> register may be redefined
The Problem

Given a code fragment for some target machine and the latencies for each individual operation, reorder the operations to minimize execution time.

The Concept

Machine description

slow code

Scheduler

fast code

The task

- Produce correct code
- Minimize wasted (idle) cycles
- Scheduler operates efficiently
Dependences ⇒ defined on memory locations / registers

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

- **true** of flow dependence
  - $a$ writes a location/register that $b$ later reads (RAW conflict)

- **anti** dependence
  - $a$ reads a location/register that $b$ later writes (WAR conflict)

- **output** dependence
  - $a$ writes a location/register that $b$ later writes (WAW conflict)

Dependences define ORDER CONSTRAINTS that need to be respected in order to generate correct code.

<table>
<thead>
<tr>
<th>true</th>
<th>anti</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a = a$</td>
<td>$a = a$</td>
<td>$a = a$</td>
</tr>
</tbody>
</table>
To capture properties of the code, build a precedence graph $G$

- Nodes $n \in G$ are operations with $\text{type}(n)$ and $\text{delay}(n)$
- An edge $e = (n_1, n_2) \in G$ if $n_2$ depends on $n_1$

The Code

<table>
<thead>
<tr>
<th>Code</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>loadAl r0,@w =&gt; r1</td>
</tr>
<tr>
<td>b</td>
<td>add r1,r1 =&gt; r1</td>
</tr>
<tr>
<td>c</td>
<td>loadAl r0,@x =&gt; r2</td>
</tr>
<tr>
<td>d</td>
<td>mult r1,r2 =&gt; r1</td>
</tr>
<tr>
<td>e</td>
<td>loadAl r0,@y =&gt; r3</td>
</tr>
<tr>
<td>f</td>
<td>mult r1,r3 =&gt; r1</td>
</tr>
<tr>
<td>g</td>
<td>loadAl r0,@z =&gt; r2</td>
</tr>
<tr>
<td>h</td>
<td>mult r1,r2 =&gt; r1</td>
</tr>
<tr>
<td>i</td>
<td>storeAl r1 =&gt; r0,@w</td>
</tr>
</tbody>
</table>

The Precedence Graph

All other dependences (output & anti) are covered, i.e., are satisfied through the dependencies shown.
### Example latencies

<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>loadAl</td>
<td>3</td>
</tr>
<tr>
<td>store</td>
<td>3</td>
</tr>
<tr>
<td>storeAl</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>branch</td>
<td>0 to 8</td>
</tr>
</tbody>
</table>
To capture properties of the code, build a precedence graph $G$

- Nodes $n \in G$ are operations with $\text{delay}(n)$
- An edge $e = (n_1, n_2) \in G$ if $n_2$ depends on $n_1$

**The Code**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a: loadAI r0,@w $\Rightarrow$ r1</td>
</tr>
<tr>
<td>3</td>
<td>b: add r1,r1 $\Rightarrow$ r1</td>
</tr>
<tr>
<td>4</td>
<td>c: loadAI r0,@x $\Rightarrow$ r2</td>
</tr>
<tr>
<td>7</td>
<td>d: mult r1,r2 $\Rightarrow$ r1</td>
</tr>
<tr>
<td>8</td>
<td>e: loadAI r0,@y $\Rightarrow$ r3</td>
</tr>
<tr>
<td>11</td>
<td>f: mult r1,r3 $\Rightarrow$ r1</td>
</tr>
<tr>
<td>12</td>
<td>g: loadAI r0,@z $\Rightarrow$ r2</td>
</tr>
<tr>
<td>15</td>
<td>h: mult r1,r2 $\Rightarrow$ r1</td>
</tr>
<tr>
<td>17</td>
<td>i: storeAI r1 $\Rightarrow$ r0,@w</td>
</tr>
</tbody>
</table>

**The Precedence Graph**

All other dependences (output & anti) are covered, i.e., are satisfied through the dependencies shown.

$S(n)$:

- 0
- 3
- 4
- 7
- 8
- 11
- 12
- 15
- 17

$\Rightarrow$ **20 cycles**
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 $(n_1, n_2) \in E$, $S(n_1) + \text{delay}(n_1) \leq S(n_2)$
3. For each type $t$, there are no more operations of type $t$ in any cycle than the target machine can issue;
   (Note: we only use a single type here - single pipeline)

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
• Operands can have multiple predecessors
Together, these issues make scheduling hard (NP-Complete)

Local scheduling is the simple case

• Restricted to straight-line code (single basic block)
• Consistent and predictable latencies
Instruction Scheduling

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 queue of operations that are ready
   b. At each cycle
      I. Choose a ready operation and schedule it
      II. Update the ready queue

Local list scheduling
• The dominant algorithm for many years
• A greedy, heuristic, local technique
Local (Forward) List Scheduling

Cycle $\leftarrow 1$
Ready $\leftarrow$ leaves of $P$
Active $\leftarrow \emptyset$

while (Ready $\cup$ Active $\neq \emptyset$)
  if (Ready $\neq \emptyset$) then
    remove an op from Ready
    $S(op) \leftarrow$ Cycle
    Active $\leftarrow$ Active $\cup$ op
  
Cycle $\leftarrow$ Cycle + 1

for each op $\in$ Active
  if ($S(op) +$ delay($op) \leq$ Cycle) then
    remove op from Active
    for each successor s of op in $P$
      if (s is ready) then
        Ready $\leftarrow$ Ready $\cup$ s

Removal in priority order
op has completed execution
If successor’s operands are ready, put it on Ready
### Scheduling Example

<table>
<thead>
<tr>
<th>Operation</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>load</td>
<td>3</td>
</tr>
<tr>
<td>loadI</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>branch</td>
<td>0 to 8</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

The Code

The Dependence Graph

S(n):

0  a: loadAl  r0,@w  \(\Rightarrow\) r1
3  b: add  r1,r1  \(\Rightarrow\) r1
4  c: loadAl  r0,@x  \(\Rightarrow\) r2
7  d: mult  r1,r2  \(\Rightarrow\) r1
8  e: loadAl  r0,@y  \(\Rightarrow\) r3
11  f: mult  r1,r3  \(\Rightarrow\) r1
12  g: loadAl  r0,@z  \(\Rightarrow\) r2
15  h: mult  r1,r2  \(\Rightarrow\) r1
17  i: storeAl  r1  \(\Rightarrow\) r0,@w
20

\(\Rightarrow\) 20 cycles

\(\Rightarrow\) 20 cycles
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
1. Build the dependence graph
2. Determine priorities: longest latency-weighted path

The Code

a: loadAI    r0,@w    \(\rightarrow\) r1
b: add       r1,r1    \(\rightarrow\) r1
c: loadAI    r0,@x    \(\rightarrow\) r2
d: mult      r1,r2    \(\rightarrow\) r1
e: loadAI    r0,@y    \(\rightarrow\) r3
f: mult      r1,r3    \(\rightarrow\) r1
g: loadAI    r0,@z    \(\rightarrow\) r2
h: mult      r1,r2    \(\rightarrow\) r1
i: storeAI   r1        \(\rightarrow\) r0,@w

The Dependence Graph

Note: Here we assume that an operation has to finish to satisfy an anti dependence. Our ILOC simulator takes only one cycle to satisfy an anti dependence since read-stage is executed before write stage (EaC).
Scheduling Example

1. Build the dependence graph
2. Determine priorities: longest latency-weighted path

<table>
<thead>
<tr>
<th>Operation</th>
<th>Source</th>
<th>Register</th>
<th>Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>a: loadAI</td>
<td>r0, @w</td>
<td>r1</td>
<td></td>
</tr>
<tr>
<td>b: add</td>
<td>r1, r1</td>
<td>r1</td>
<td></td>
</tr>
<tr>
<td>c: loadAI</td>
<td>r0, @x</td>
<td>r2</td>
<td></td>
</tr>
<tr>
<td>d: mult</td>
<td>r1, r2</td>
<td>r1</td>
<td></td>
</tr>
<tr>
<td>e: loadAI</td>
<td>r0, @y</td>
<td>r3</td>
<td></td>
</tr>
<tr>
<td>f: mult</td>
<td>r1, r3</td>
<td>r1</td>
<td></td>
</tr>
<tr>
<td>g: loadAI</td>
<td>r0, @z</td>
<td>r2</td>
<td></td>
</tr>
<tr>
<td>h: mult</td>
<td>r1, r2</td>
<td>r1</td>
<td></td>
</tr>
<tr>
<td>i: storeAI</td>
<td>r1</td>
<td>r0, @w</td>
<td></td>
</tr>
</tbody>
</table>

The Code

The Dependence Graph

Note: Here we assume that an operation has to finish to satisfy an anti dependence. Our ILOC simulator takes only one cycle to satisfy an anti dependence since read-stage is executed before write stage (EaC).
1. Build the dependence graph
2. Determine priorities: longest latency-weighted path
3. Perform list scheduling (forward)

The Code

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

The Dependence Graph

We assume full latency for anti-dependences here
Scheduling Example

1. Build the dependence graph
2. Determine priorities: longest latency-weighted path
3. Perform list scheduling (forward)

The Code

\[
\begin{align*}
S(n): \\
0 & a: \text{loadAl} \quad r0,@w \implies r1 \\
1 & c: \text{loadAl} \quad r0,@x \implies r2 \\
2 & e: \text{loadAl} \quad r0,@y \implies r3 \\
3 & b: \text{add} \quad r1,r1 \implies r1 \\
4 & d: \text{mult} \quad r1,r2 \implies r1 \\
6 & g: \text{loadAl} \quad r0,@z \implies r2 \\
7 & f: \text{mult} \quad r1,r3 \implies r1 \\
9 & h: \text{mult} \quad r1,r2 \implies r1 \\
11 & i: \text{storeAl} \quad r1 \implies r0,@w \\
14 & \end{align*}
\]

The Dependence Graph

We assume full latency for anti-dependences here

cycles
More on Scheduling

Forward list scheduling
- start with available ops
- work forward
- ready ⇒ all operands available

Backward list scheduling
- start with no successors
- work backward
- ready ⇒ latency covers operands

Different heuristics (forward) based on Dependence Graph
1. Longest latency weighted path to root (⇒ critical path)
2. Highest latency instructions (⇒ more overlap)
3. Most immediate successors (⇒ create more candidates)
4. Most descendents (⇒ create more candidates)
5. ... (homework)

Interactions with register allocation (Note: we are not doing this)
- perform dynamic register renaming (⇒ may require spill code)
- move life ranges around (⇒ may remove or require spill code)
- ...
Lexical Analysis

Read EaC: Chapters 2.1 – 2.5;