CS415 Compilers

Instruction Scheduling

and

Introduction to Lexical Analysis

These slides are based on slides copyrighted by Keith Cooper, Ken Kennedy & Linda Torczon at Rice University
Review: The Back End

Responsibilities

- Translate IR into target machine code
- Choose instructions to implement each IR operation
- Decide which value to keep in registers
- Ensure conformance with system interfaces

Automation has been less successful in the back end
Readings: EaC 12.1-12.3

- **Local**: within single basic block
- **Global**: across entire procedure
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 (or power requirements)
- Static schedule - insert NOPs to preserve correctness
- Dynamic schedule - hardware pipeline stalls
- Preserve correctness
- 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 (or power requirements)
- Static schedule - insert NOPs to preserve correctness
- Dynamic schedule - hardware pipeline stalls
- Preserve correctness
- Interactions with other optimizations (register allocation!)
- **Note:** code shape contains real, not virtual registers
  - $\Rightarrow$ 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

Scheduler

slow code → fast code

The task

- Produce correct code
- Minimize wasted (idle) cycles
- Operate efficiently
Dependences ⇒ defined on memory locations / registers and not values

Statement/instruction \( b \) depends on statement/instruction \( a \) if there exists:

- **true** of flow dependence
  - \( a \) writes a location/register that \( b \) later reads \hspace{1cm} (RAW conflict)

- **anti** dependence
  - \( a \) reads a location/register that \( b \) later writes \hspace{1cm} (WAR conflict)

- **output** dependence
  - \( a \) writes a location/register that \( b \) later writes \hspace{1cm} (WAW conflict)

Dependences define ORDER CONSTRAINTS that need to be respected in order to generate correct code.
To capture properties of the code, build a **dependence graph** $G$

- **Nodes** $n \in G$ are operations with $\text{type}(n)$ and $\text{delay}(n)$
- An edge $e = (n_1, n_2) \in G$ iff $n_2$ uses the result of $n_1$

The Code

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

The Dependence Graph

(all output dependences are covered, i.e., are satisfied through other dependences)
A correct schedule $S$ maps each $n \in \mathbb{N}$ into a non-negative integer representing its cycle number such that

1. $S(n) \geq 0$, for all $n \in \mathbb{N}$, 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

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

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

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 $S_1$

A schedule might also be optimal in terms of registers, power, or space....
Critical Points

- All operands must be available
- Multiple operations can be *ready*
- Moving operations can lengthen register lifetimes
- Placing uses near definitions can shorten register lifetimes
- Operands can have multiple predecessors

Together, these issues make scheduling *hard* (NP-Complete)

Local scheduling is the simpler 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 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 twenty years
• A greedy, heuristic, local technique
Local (Forward) List Scheduling

\[
\begin{align*}
\text{Cycle} & \leftarrow 1 \\
\text{Ready} & \leftarrow \text{leaves of } P \\
\text{Active} & \leftarrow \emptyset
\end{align*}
\]

while (Ready $\cup$ Active $\neq \emptyset$)

\[
\begin{align*}
\text{if (Ready $\neq \emptyset$) then} & \\
& \text{remove an } op \text{ from Ready} \\
& S(op) \leftarrow \text{Cycle} \\
& \text{Active} \leftarrow \text{Active } \cup \text{ op}
\end{align*}
\]

\[
\begin{align*}
\text{Cycle} & \leftarrow \text{Cycle } + 1 \\
\text{for each } op \in \text{Active} & \\
\text{if } (S(op) + \text{delay}(op) \leq \text{Cycle}) & \text{ then} \\
& \text{remove } op \text{ from Active} \\
\text{for each successor } s \text{ of } op \text{ in } P & \\
\text{if } (s \text{ is ready}) & \text{ then} \\
& \text{Ready} \leftarrow \text{Ready } \cup \text{ s}
\end{align*}
\]
### 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>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>

- **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
Scheduling Example

1. Build the dependence graph

<table>
<thead>
<tr>
<th></th>
<th>Instruction</th>
<th>Operands</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>loadAI</td>
<td>r0, @w</td>
<td>r1</td>
</tr>
<tr>
<td>4</td>
<td>add</td>
<td>r1, r1</td>
<td>r1</td>
</tr>
<tr>
<td>5</td>
<td>loadAI</td>
<td>r0, @x</td>
<td>r2</td>
</tr>
<tr>
<td>8</td>
<td>mult</td>
<td>r1, r2</td>
<td>r1</td>
</tr>
<tr>
<td>9</td>
<td>loadAI</td>
<td>r0, @y</td>
<td>r3</td>
</tr>
<tr>
<td>12</td>
<td>mult</td>
<td>r1, r3</td>
<td>r1</td>
</tr>
<tr>
<td>13</td>
<td>loadAI</td>
<td>r0, @z</td>
<td>r2</td>
</tr>
<tr>
<td>16</td>
<td>mult</td>
<td>r1, r2</td>
<td>r1</td>
</tr>
<tr>
<td>18</td>
<td>storeAI</td>
<td>r1</td>
<td>r0, @w</td>
</tr>
</tbody>
</table>

The Code

⇒ 20 cycles

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

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
Scheduling Example

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

The Code

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

The Dependence Graph

Note: Here we assume that 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.
1. Build the dependence graph

2. Determine priorities: longest latency-weighted path

The Code

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

The Dependence Graph

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

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
```

⇒ 14 cycles

The Dependence Graph

Our ILOC simulator takes only one cycle to satisfy an anti dependence
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 Precedence/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. ...

Interactions with register allocation
- 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