CS415 Compilers
Instruction Scheduling
Part 1

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

• First project and homework have been posted

  Due dates: homework Monday Feb. 8 @11:59pm
  project code: Tuesday Feb. 23 @11:59pm
  project report: Friday Feb. 26 @11:59pm

• First quiz next week. Topic: register allocation

• Please attend recitations!
Instruction Scheduling

EaC  Chapter 12
12.1 - 12.3

Part of the compiler’s back end

Diagram showing the flow from Instruction Selection, Register Allocation, to Instruction Scheduling, leading to Machine code.
**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 \(\Rightarrow\) 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
- Scheduler operates efficiently
Data Dependences (stmt./instr. level)

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

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

### 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 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 (latency/delay)</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>output</td>
<td>1</td>
</tr>
<tr>
<td>outputAI</td>
<td>1</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 $n_1$

$$
\begin{align*}
S(n): \\
0 & \quad \text{a: load}AI \quad r0,\!@w \quad \Rightarrow \quad r1 \\
3 & \quad \text{b: add} \quad r1,\!r1 \quad \Rightarrow \quad r1 \\
4 & \quad \text{c: load}AI \quad r0,\!@x \quad \Rightarrow \quad r2 \\
7 & \quad \text{d: mult} \quad r1,\!r2 \quad \Rightarrow \quad r1 \\
8 & \quad \text{e: load}AI \quad r0,\!@y \quad \Rightarrow \quad r3 \\
11 & \quad \text{f: mult} \quad r1,\!r3 \quad \Rightarrow \quad r1 \\
12 & \quad \text{g: load}AI \quad r0,\!@z \quad \Rightarrow \quad r2 \\
15 & \quad \text{h: mult} \quad r1,\!r2 \quad \Rightarrow \quad r1 \\
17 & \quad \text{i: store}AI \quad r1 \quad \Rightarrow \quad r0,\!@w \\
\end{align*}
$$

$\Rightarrow 20 \quad \text{The Code} \quad \Rightarrow 20 \quad \text{cycles}$

**The Precedence/Dependence Graph**

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

   *(Note: we only use a single type here - single pipeline)*

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

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

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 $S_1$

**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
More on List Scheduling

Lexical Analysis

Read EaC: Chapters 2.1 – 2.5;