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!)
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!)

• 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

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 \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 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

- 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 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 $n_1$

**S(n):**

<table>
<thead>
<tr>
<th>Step</th>
<th>Instruction</th>
<th>Source</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>loadAl r0,@w</td>
<td></td>
<td>$\Rightarrow$ r1</td>
</tr>
<tr>
<td>3</td>
<td>add r1,r1</td>
<td>$\Rightarrow$</td>
<td>r1</td>
</tr>
<tr>
<td>4</td>
<td>loadAl r0,@x</td>
<td>$\Rightarrow$</td>
<td>r2</td>
</tr>
<tr>
<td>7</td>
<td>mult r1,r2</td>
<td>$\Rightarrow$</td>
<td>r1</td>
</tr>
<tr>
<td>8</td>
<td>loadAl r0,@y</td>
<td>$\Rightarrow$</td>
<td>r3</td>
</tr>
<tr>
<td>11</td>
<td>mult r1,r3</td>
<td>$\Rightarrow$</td>
<td>r1</td>
</tr>
<tr>
<td>12</td>
<td>loadAl r0,@z</td>
<td>$\Rightarrow$</td>
<td>r2</td>
</tr>
<tr>
<td>15</td>
<td>mult r1,r2</td>
<td>$\Rightarrow$</td>
<td>r1</td>
</tr>
<tr>
<td>17</td>
<td>storeAl r1</td>
<td>$\Rightarrow$</td>
<td>r0,@w</td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**The Code**

- 20 cycles

**The Precedence Graph**

All other dependences (output & anti) are covered, i.e., are satisfied through the dependencies shown.
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
The big picture
1. Build a dependence graph, \( P \)
2. Compute a \textit{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

\[
\text{Cycle} \leftarrow 1 \\
\text{Ready} \leftarrow \text{leaves of } P \\
\text{Active} \leftarrow \emptyset \\
\text{while (Ready } \cup \text{ Active } \neq \emptyset) \\
\quad \text{if (Ready } \neq \emptyset) \text{ then} \\
\quad \quad \text{remove an op from Ready} \\
\quad \quad S(\text{op}) \leftarrow \text{Cycle} \\
\quad \quad \text{Active} \leftarrow \text{Active } \cup \text{ op} \\
\text{Cycle} \leftarrow \text{Cycle } + 1 \\
\text{for each } \text{op} \in \text{Active} \\
\quad \text{if (S(\text{op}) + delay(\text{op}) } \leq \text{ Cycle) then} \\
\quad \quad \text{remove op from Active} \\
\quad \quad \text{for each successor } s \text{ of } \text{op} \text{ in } P \\
\quad \quad \quad \text{if (s is ready) then} \\
\quad \quad \quad \quad \text{Ready} \leftarrow \text{Ready } \cup \text{ s}
\]

Removal in priority order

op has completed execution

If successor’s operands are ready, put it on Ready
### Operation Cycles

<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>

### Scheduling Example

- **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

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

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 cycles

The Dependence Graph
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
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

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

The Code

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

We assume full latency for anti-dependences here
1. Build the dependence graph
2. Determine priorities: longest latency-weighted path
3. Perform list scheduling (forward)

S(n):

0  a: loadAl  r0,@w  ⇒ r1  
1  c: loadAl  r0,@x  ⇒ r2  
2  e: loadAl  r0,@y  ⇒ r3  
3  b: add  r1,r1  ⇒ r1  
4  d: mult  r1,r2  ⇒ r1  
6  g: loadAl  r0,@z  ⇒ r2  
7  f: mult  r1,r3  ⇒ r1  
9  h: mult  r1,r2  ⇒ r1  
11  i: storeAl  r1  ⇒ r0,@w  
14

The Code

We assume full latency for anti-dependences here

The Dependence Graph

cycles 14

true

anti
### More on Scheduling

<table>
<thead>
<tr>
<th>Forward list scheduling</th>
<th>Backward list scheduling</th>
</tr>
</thead>
<tbody>
<tr>
<td>• start with available ops</td>
<td>• start with no successors</td>
</tr>
<tr>
<td>• work forward</td>
<td>• work backward</td>
</tr>
<tr>
<td>• ready ⇒ all operands available</td>
<td>• ready ⇒ latency covers operands</td>
</tr>
</tbody>
</table>

### 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)
- ...

---

cs415, spring 18 Lecture 6
Lexical Analysis

Read EaC: Chapters 2.1 - 2.5;
The purpose of the front end is to deal with the input language

- Perform a membership test: $\text{code} \in \text{source language}$?
- Is the program well-formed (semantically)?
- Build an IR version of the code for the rest of the compiler

The front end is not monolithic
The Front End

Scanner

- Maps stream of characters into words
  - Basic unit of syntax
  - \( x = x + y \); becomes
    \(<\text{id},x><\text{eq},>=<\text{id},x><\text{pl},+><\text{id},y><\text{sc},;>\)

- Characters that form a word are its \textit{lexeme}

- Its \textit{part of speech} (or \textit{syntactic category}) is called its \textit{token type}

- Scanner discards white space & (often) comments

Speed is an issue in scanning
\Rightarrow use a specialized recognizer

Source code \rightarrow \text{Scanner} \rightarrow \text{tokens} \rightarrow \text{Parser} \rightarrow \text{IR}

\text{Errors}
The Front End

**Parser**

- Checks stream of classified words (*parts of speech*) for grammatical correctness
- Determines if code is syntactically well-formed
- Guides checking at deeper levels than syntax
- Builds an IR representation of the code

_We’ll get to parsing in the next lectures_
The Big Picture

- Language syntax is specified with parts of speech, not words
- Syntax checking matches parts of speech against a grammar
- Here is an example context free grammar (CFG) $G$:

1. $goal \rightarrow expr$
2. $expr \rightarrow expr \ op \ term$
3. $\mid \ term$
4. $term \rightarrow number$
5. $\mid \ id$
6. $op \rightarrow +$
7. $\mid -$ 

$G$ in BNF form

$G = (S, T, N, P)$

$S = goal$

$T = \{ \text{number, id, +, -} \}$

$N = \{ goal, expr, term, op \}$

$P = \{ 1, 2, 3, 4, 5, 6, 7 \}$
Why study lexical analysis?

- We want to avoid writing scanners by hand

Goals:

→ To simplify specification & implementation of scanners
→ To understand the underlying techniques and technologies
Lexical patterns form a regular language

*** any finite language is regular ***

Regular expressions (REs) describe regular languages.

Regular Expression (over alphabet \( \Sigma \))

- \( \varepsilon \) is a RE denoting the set \( \{ \varepsilon \} \)
- If “a” is in \( \Sigma \), then \( a \) is a RE denoting \( \{ a \} \)
- If \( x \) and \( y \) are REs denoting \( L(x) \) and \( L(y) \) then
  - \( x | y \) is an RE denoting \( L(x) \cup L(y) \)
  - \( xy \) is an RE denoting \( L(x)L(y) \)
  - \( x^* \) is an RE denoting \( L(x)^* \)

Precedence is closure, then concatenation, then alternation

Ever type “rm *.o a.out” ?
### Set Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Union of $L$ and $M$</td>
<td>$L \cup M = { s \mid s \in L \text{ or } s \in M }$</td>
</tr>
<tr>
<td>Written $L \cup M$</td>
<td></td>
</tr>
<tr>
<td>Concatenation of $L$ and $M$</td>
<td>$LM = { st \mid s \in L \text{ and } t \in M }$</td>
</tr>
<tr>
<td>Written $LM$</td>
<td></td>
</tr>
<tr>
<td>Kleene closure of $L$</td>
<td>$L^* = \bigcup_{0 \leq i \leq \infty} L^i$</td>
</tr>
<tr>
<td>Written $L^*$</td>
<td></td>
</tr>
<tr>
<td>Positive Closure of $L$</td>
<td>$L^+ = \bigcup_{1 \leq i \leq \infty} L^i$</td>
</tr>
<tr>
<td>Written $L^+$</td>
<td></td>
</tr>
</tbody>
</table>

These definitions should be well known
Examples of Regular Expressions

Identifiers:

\[ \text{Letter} \rightarrow (a | b | c \ldots | z | A | B | C \ldots | Z) \]
\[ \text{Digit} \rightarrow (0 | 1 | 2 | \ldots | 9) \]
\[ \text{Identifier} \rightarrow \text{Letter} \ (\text{Letter} \mid \text{Digit})^* \]

Numbers:

\[ \text{Integer} \rightarrow (\pm | - | \varepsilon) (0 | (1 | 2 | 3 \ldots | 9)(\text{Digit}^*)) \]
\[ \text{Decimal} \rightarrow \text{Integer} \ (\text{Digit}^*) \]
\[ \text{Real} \rightarrow (\text{Integer} \mid \text{Decimal}) E (\pm | - | \varepsilon) \ (\text{Digit}^*) \]
\[ \text{Complex} \rightarrow (\text{Real} \mid \text{Real}) \]

Numbers can get much more complicated!
Regular expressions can be used to specify the words to be translated to parts of speech by a lexical analyzer.

Using results from automata theory and theory of algorithms, we can automatically build recognizers from regular expressions.

⇒ We study REs and associated theory to automate scanner construction!
Consider the problem of recognizing ILOC register names

\[ \text{Register} \rightarrow r (0|1|2| \ldots | 9) (0|1|2| \ldots | 9)^* \]

- Allows registers of arbitrary number
- Requires at least one digit

RE corresponds to a recognizer (or DFA)

Transitions on other inputs go to an error state, \( s_e \)