Lecture 17
Intro to Instruction Scheduling

Reading: Chapter 10.1 – 10.2

The Goal of Instruction Scheduling

- Assume that the remaining instructions are all essential
  - (otherwise, earlier passes would have eliminated them)
- How can we perform this fixed amount of work in less time?
  - Answer: execute the instructions in parallel

<table>
<thead>
<tr>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = 1 + x; b = 2 + y; c = 3 + z;</td>
</tr>
</tbody>
</table>

Optimization: What’s the Point? (A Quick Review)

Machine-Independent Optimizations:
- e.g., constant propagation & folding, redundancy elimination, dead-code elimination, etc.
- Goal: eliminate work

Machine-Dependent Optimizations:
- register allocation
  - Goal: reduce cost of accessing data
- instruction scheduling
  - Goal: ???
  - ...

Hardware Support for Parallel Execution

- Three forms of parallelism are found in modern machines:
  - Pipelining
  - Superscalar Processing
  - Multiprocessing

Instruction Scheduling
Automatic Parallelization (covered later in class)
Pipelining

Basic idea:
- break instruction into stages that can be overlapped

Example: simple 5-stage pipeline from early RISC machines

1 instruction

IF RF EX ME WB

IF = Instruction Fetch
RF = Decode & Register Fetch
EX = Execute on ALU
ME = Memory Access
WB = Write Back to Register File

Beyond 5-Stage Pipelines: Even More Parallelism

- Should we simply make pipelines deeper and deeper?
  - registers between pipeline stages have fixed overheads
    - hence diminishing returns with more stages (Amdahl’s Law)
  - value of pipe stage unclear if \(<\) time for integer add
- However, many consumers think “performance = clock rate”
- perceived need for higher clock rates \(>\) deeper pipelines
  - e.g., Pentium 4 processor had a 20-stage pipeline

Pipe Register
Pipe Register
Pipe Register

Time

• In a given cycle, each instruction is in a different stage
Beyond Pipelining: “Superscalar” Processing

- Basic Idea:
  - multiple (independent) instructions can proceed simultaneously through the same pipeline stages
  - requires additional hardware
  - example: “Execute” stage

Requires additional hardware – example: “Execute” stage

Hardware for Scalar Pipeline: 1 ALU
Hardware for 2-way Superscalar: 2 ALUs

The Ideal Scheduling Outcome

- What prevents us from achieving this ideal?

Limitations Upon Scheduling

1. Hardware Resources
2. Data Dependences
3. Control Dependences
Limitation #1: Hardware Resources

- Processors have finite resources, and there are often constraints on how these resources can be used.

Examples:
- Finite issue width
- Limited functional units (FUs) per given instruction type
- Limited pipelining within a given functional unit (FU)

Finite Issue Width

- Prior to superscalar processing:
  - processors only “issued” one instruction per cycle
- Even with superscalar processing:
  - limit on total # of instructions issued per cycle

Limited FUs per Instruction Type

- e.g., a 4-way superscalar might only be able to issue up to 2 integer, 1 memory, and 1 floating-point insts per cycle

Limited Pipelining within a Functional Unit

- e.g., only 1 new floating-point division every 2 cycles
Limitations Upon Scheduling

1. Hardware Resources
2. Data Dependences
3. Control Dependences

Limitation #2: Data Dependences

- If we read or write a data location “too early”, the program may behave incorrectly.

(Assume that initially, \( x = 0 \).)

- Read-after-Write ("True" dependence)
- Write-after-Write ("Output" dependence)
- Write-after-Read ("Anti" dependence)

Can potentially fix through renaming.

Why Data Dependences are Challenging

- \( x = a[i] \);
- \(*p = 1;\)
- \( y = *q;\)
- \(*r = z;\)

- which of these instructions can be reordered?

- ambiguous data dependences are very common in practice
  - difficult to resolve, despite fancy pointer analysis

Given Ambiguous Data Dependences, What To Do?

- \( x = a[i] \);
- \(*p = 1;\)
- \( y = *q;\)
- \(*r = z;\)

- Conservative approach: don’t reorder instructions
  - ensures correct execution
  - but may suffer poor performance

- Aggressive approach?
  - is there a way to safely reorder instructions?
Hardware Limitations: Multi-cycle Execution Latencies

- Simple instructions often "execute" in one cycle
  - (as observed by other instructions in the pipeline)
  - e.g., integer addition
- More complex instructions may require multiple cycles
  - e.g., integer division, square-root
  - cache misses!
- These latencies, when combined with data dependencies, can result in non-trivial critical path lengths through code

Limitations Upon Scheduling

1. Hardware Resources
2. Data Dependences
   - 3. Control Dependences

Scheduling Constraints: Summary

- Hardware Resources
  - finite set of FUs with instruction type, bandwidth, and latency constraints
  - cache hierarchy also has many constraints
- Data Dependences
  - can’t consume a result before it is produced
  - ambiguous dependences create many challenges
- Control Dependences
  - impractical to schedule for all possible paths
  - choosing an “expected” path may be difficult
    - recovery costs can be non-trivial if you are wrong
**Hardware- vs. Compiler-Based Scheduling**

- The hardware can also attempt to reschedule instructions (on-the-fly) to improve performance.
- What advantages/disadvantages would hardware have (vs. the compiler) when trying to reason about:
  - Hardware Resources
  - Data Dependences
  - Control Dependences
- Which is better:
  - doing more of the scheduling work in the compiler?
  - doing more of the scheduling work in the hardware?

**Spectrum of Hardware Support for Scheduling**

<table>
<thead>
<tr>
<th>Compiler-Centric</th>
<th>Hardware-Centric</th>
</tr>
</thead>
<tbody>
<tr>
<td>VLIW (Very Long Instruction Word)</td>
<td>In-Order Superscalar</td>
</tr>
<tr>
<td></td>
<td>Out-of-Order Superscalar</td>
</tr>
<tr>
<td>e.g.: Itanium</td>
<td>e.g.: Original Pentium</td>
</tr>
<tr>
<td></td>
<td>e.g.: Pentium 4</td>
</tr>
</tbody>
</table>

**VLIW Processors**

- **Motivation:**
  - if the hardware spends zero (or almost zero) time thinking about scheduling, it can run faster
- **Philosophy:**
  - give full control over scheduling to the compiler
- **Implementation:**
  - expose control over all FUs directly to software via a "very long instruction word"

**Compiling for VLIW**

**Predicting Execution Latencies:**
- easy for most functional units (latency is fixed)
- but what about memory references?

**Data Dependences:**
- in "pure" VLIW, the hardware does not check for them
  - the compiler takes them into account to produce safe code

```plaintext
Example #1
a = b + 1;
c = a - d;
e = c / 3;
f = g - e;

Example #2
while (p != NULL) {
    if (test(p->val))
        q->next = p->left;
    p = p->next;
}
```

**Example #1**
```
a = b + 1;
c = a - d;
e = c / 3;
f = g - e;
```

**Example #2**
```
while (p != NULL) {
    if (test(p->val))
        q->next = p->left;
    p = p->next;
}
```
“VLIW” Today

- Hardware checks for data dependences through memory
- Compiler can do a good job with register dependences

- Runtime software dynamically generates VLIW code

In-Order Superscalar Processors

In contrast with VLIW:
- Hardware does full data dependence checking
- Hence, no need to encode NOPs for empty slots

- Once an instruction cannot be issued, no instructions after it will be issued.

Bottom Line:
- Hardware matches code to available resources; recompilation is not necessary for correctness
- Compiler’s role is still important
- For performance, not correctness!
Out-of-Order Superscalar Processors

Motivation:
- when an instruction is stuck, perhaps there are subsequent instructions that can be executed

```plaintext
x = *p;
y = x + 1;
z = a + 2;
b = c / 3;
```

- suffers expensive cache miss
- stuck waiting on true dependence
- these do not need to wait

Sounds great! But how does this complicate the hardware?

---

Compiler- vs. Hardware-Centric Scheduling: Bottom Line

**Compiler-Centric**
- VLIW

**Hardware-Centric**
- In-Order Superscalar
- Out-of-Order Superscalar

- High-end processors will probably remain out-of-order
  - moving instructions small distances is probably useless
  - BUT, moving instructions large distances may still help
- Cheap, power-efficient processors may be in-order/VLIW
  - instruction scheduling may have a large impact

---

Scheduling Roadmap

**List Scheduling:**
- within a basic block

**Global Scheduling:**
- across basic blocks

**Software Pipelining:**
- across loop iterations