Instruction Scheduling
Adapted from Lectures by Profs. Alex Aiken and George Necula (UCB)

Lecture Outline
- Instruction-Level Parallelism
- Instruction Scheduling
- List Scheduling
  - A simple and effective heuristic for instruction scheduling
  - An extended example

Instruction-Level Parallelism (ILP)
- Modern CPUs can execute multiple instructions concurrently
- Two sources of parallelism are exploited
  - Some machines issue multiple instructions in one cycle ⇒ superscalar machine
  - Some machines overlap various execution phases of different instructions ⇒ pipelining
  - Most modern machines do both
- ILP can be improved by reordering instructions ⇒ instruction scheduling

Instruction Dependencies
- Any two instructions cannot be reordered
- Resource dependencies
  - Two instructions that must use the same functional unit cannot execute at once
- Data dependencies: A must finish before B if
  - B reads a register written by A
  - Read-after-written dependency
  - B writes a register also written by A
  - Write-after-write dependency
  - B writes a register that A reads (write-after-read)

Two Kinds of Scheduling Techniques
- Dynamic-scheduling
  - The processor decides the order at run-time
  - Also called out-of-order execution
- Static-scheduling
  - The compiler decides the order at compiler time
  - We will look at this technique.

The MIPS
- On the MIPS most operations execute in 1 cycle
- Conditional branches require 2 cycles to complete
  - `addiu $t1 $t1 1`
  - `addiu $t2 $t2 1`
  - `beq $t2 $t3 label`
- This code requires 4 cycles to complete
  (1 for each add and 2 for the branch)
MIPS Branch Delay Slots

- We can insert a nop to make the second branch cycle explicit:
  ```
  addiu $t1 $t1 1
  addiu $t2 $t2 1
  beq $t2 $t3 label
  *nop
  ```
  - The * means that the nop executes in the branch’s second cycle.
  - This cycle is called branch delay slot.

MIPS Branch Delay Slots (Cont.)

- The code can be improved by scheduling something useful in the delay slot:
  ```
  addiu $t2 $t2 1
  beq $t2 $t3 label
  *addiu $t1 $t1 1
  ```
  - This code is equivalent to the original.
  - The final state of the machine is the same.
  - But executes 25% faster (3 cycles).

Beyond MIPS

- On the MIPS the only scheduling problem is filling the delay slot.
- On newer architectures the scheduling problem is both:
  - More complex, and
  - More critical to good performance.

- We will show this on a “made-up” architecture.

Our Architecture

- Based on Motorola 88xxx but simplified.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>j + j</td>
<td>Integer add</td>
<td>1</td>
</tr>
<tr>
<td>j + k</td>
<td>Floating-point add</td>
<td>2</td>
</tr>
<tr>
<td>j * k</td>
<td>Multiply (integer or float)</td>
<td>3</td>
</tr>
<tr>
<td>j(k)</td>
<td>Memory load from j + k</td>
<td>2</td>
</tr>
<tr>
<td>j + j gets L</td>
<td>Branch</td>
<td>4</td>
</tr>
</tbody>
</table>
An Example

• We’ll use the following loop as a running example:
  \[
  \text{innerprod} := 0; \\
  \text{for}(i = 1; i \leq n; i++) \{ \\
    \text{innerprod} := A[i] \times B[i] + \text{innerprod}; \\
  \}
  \]
  - Where A and B are two floating-point arrays

The Code for the Example

• The generated code might be like this:
  \[
  \text{innerprod} := 0 \\
  i := 1 \\
  \text{top: } t_1 := 4 \times i \\
  t_2 := t_1(A) \\
  t_3 := t_1(B) \\
  t_4 := t_2 \times t_3 \\
  \text{innerprod} := t_4 + \text{innerprod} \\
  i := i + 1 \\
  \text{if } i \leq n \text{ goto top}
  \]

The Optimized Code for the Example

• Note that \( t_1 \) and \( t_3 \) are common subexpr.
• After local optimization:
  \[
  \text{innerprod} := 0 \\
  i := 1 \\
  \text{top: } t_1 := 4 \times i \\
  t_2 := t_1(A) \\
  t_3 := t_1(B) \\
  t_4 := t_2 \times t_3 \\
  \text{innerprod} := t_4 + \text{innerprod} \\
  i := i + 1 \\
  \text{if } i \leq n \text{ goto top}
  \]

The Example’s Performance

• We insert \texttt{nop} where the processor stalls to wait for a value
  \[
  \text{innerprod} := 0 \\
  i := 1 \\
  \text{top: } t_1 := 4 \times i \\
  t_2 := t_1(A) \\
  t_3 := t_1(B) \\
  \texttt{nop} \\
  t_4 := t_2 \times t_3 \\
  \text{innerprod} := t_4 + \text{innerprod} \\
  i := i + 1 \\
  \text{if } i \leq n \text{ goto top} \\
  \]
  - Loop body (7 instr.) takes 15 cycles to execute
  - less than 50% of the potential performance

The Example’s Performance (Cont.)

• There is some parallelism between instructions
  - E.g., \texttt{innerprod} := \( t_4 \) + \texttt{innerprod} and \( i := i + 1 \)
• But there are many “bubbles” in the pipeline where the processor is stalled waiting for a previous instruction to complete
• We try to reorder instructions to reduce the number of stalls

An Instruction Scheduling Method

• We will look only at \texttt{basic-block scheduling}
• An optimal schedule is too expensive to compute
  - We will use heuristics
  - \textbf{Step 1:}
    - calculate dependencies between instructions
  - \textbf{Step 2:}
    - pick instructions whose dependencies are satisfied
The Dependence Graph

- Dependencies between instructions can be shown using a directed graph:
  - Each instruction is a node.
  - If B reads the output of A and A completes in k cycles:
    - Draw an edge from A to B with weight k.
  - If B writes the input of A:
    - Draw an edge from A to B with weight 1.
  - If B writes the output of A:
    - Draw an edge from A to B with weight 1.

The Dependence Graph. Example.

- The DG for the loop body in our example:

Dependence Graphs Describe Legal Orderings

- If A \rightarrow_k B in the DG then:
  - A must appear before B, and
  - At least k-1 instructions must separate A and B.
- Thus, \( t_1 := 4 * i \) must be the first instruction.
- Followed by either:
  - \( t_2 := t_1(A) \) after 2 instructions
  - \( t_4 := t_1(B) \) after 2 instructions
  - \( i := i + 1 \) after 0 instructions
- We'll pick \( i := i + 1 \) next.

The Instruction Scheduling Algorithm (I)

- Prioritize instructions according to how early in the computation they should be executed:
  - Assign to each instruction A a priority \( <L, D> \):
    - L is the weight of the longest path in the DG from A to the end of the block.
    - D is the number of instructions depending on A.

The Instruction Scheduling Algorithm (II)

- Build a schedule cycle by cycle:
  1. Pick an eligible instruction A such that:
     a) It is a root in the current DG, and
     b) Its inputs are available in this cycle, and
     c) If A is a branch then all unscheduled instructions can complete in the delay slots.
     - Among eligible instructions pick that with largest L.
     - Break ties in favor of instructions with larger D.
  2. Insert a nop if no eligible instruction.
  3. Remove A from the DG and repeat from 1.
Intuition

• The most important instructions are those on the **critical path** (the longest chain of dependencies)

• Delaying instructions on the critical path is likely to result in a longer schedule

• Picking instructions with more dependents will make more instructions eligible later

Constructing the Schedule. Example (1)

```
\begin{align*}
\text{top: } t_1 &= 4 \times i + 0, 4 \\
\text{if } i &\leq n \text{ goto top} \\
\text{end of block } &<0, 0>
\end{align*}
```

Constructing the Schedule. Example (2)

```
\begin{align*}
\text{top: } t_1 &= 4 \times i + 0, 4 \\
\text{if } i &\leq n \text{ goto top} \\
\text{end of block } &<0, 0>
\end{align*}
```

Constructing the Schedule. Example (3)

```
\begin{align*}
\text{top: } t_1 &= 4 \times i + 0, 4 \\
\text{if } i &\leq n \text{ goto top} \\
\text{end of block } &<0, 0>
\end{align*}
```

Constructing the Schedule. Example (4)

```
\begin{align*}
\text{top: } t_1 &= 4 \times i + 0, 4 \\
\text{if } i &\leq n \text{ goto top} \\
\text{end of block } &<0, 0>
\end{align*}
```

Constructing the Schedule. Example (5)

```
\begin{align*}
\text{top: } t_1 &= 4 \times i + 0, 4 \\
\text{if } i &\leq n \text{ goto top} \\
\text{end of block } &<0, 0>
\end{align*}
```
Constructing the Schedule. Example (6)

```
if i <= n gototop

innerprod := t_5 + F innerprod
```

Constructing the Schedule. Example (7)

```
if i <= n gototop

innerprod := t_5 + F innerprod
```

Constructing the Schedule. Example (8)

```
if i <= n gototop

innerprod := t_5 + F innerprod
```

Constructing the Schedule. Example (9)

```
if i <= n goto top

innerprod := t_5 + F innerprod
```

Constructing the Schedule. Example (10)

```
if i <= n goto top

innerprod := t_5 + F innerprod
```

Constructing the Schedule. Example (11)

```
if i <= n goto top

innerprod := t_5 + F innerprod
```
Notes

• Now one iteration takes 11 cycles
  - 26% faster than before
  - 64% utilization of the machine (7 busy cycles)
  - It seems that this is almost the best we can do since \( t_1 = 4 * 1 \) has a critical path of length 10
• However we can much better than that...

Loop Unrolling

• The problem with the example is that the basic block is too small
• The scheduler does not have enough instructions to fill the bubbles
• We get a bigger block by unrolling the loop
• Loop unrolling duplicates the loop body and combines two iterations in one

Loop Unrolling. Example.

• The unrolled loop for computing the inner product (assuming \( n \) is even):
  ```
  innerprod := 0;
  for(i:=1; i<=n; i += 2) {
    innerprod := A[i+1] * B[i+1] + innerprod;
  }
  ```

Loop Unrolling (Cont.)

• In general, to unroll \( k \) times a loop with body \( E \)
  - Create \( k \) copies of \( E \)
  - In the \( j \)th copy replace iteration variable \( i \) by \( i+j-1 \)
  - Make the iteration variable step by \( k \)
  - Adjust loop termination tests
• Two advantages:
  - It gives the scheduler more instructions
  - Eliminates conditional tests between iterations
• Disadvantage: code size growth

The Example Unrolled

- The subexpression \( 4 * (i+1) \) is stored in \( s_1 \)
- In this form one iteration takes 2*15-4 cycles
- as before, less a branch

A New Schedule

- This loop takes 15 cycles
- Twice as fast as before
- 50% faster than the single iteration schedule
- 86% machine utilization

innerprod := 0

- innerprod := \( t_1 + \) innerprod
- if \( i < n \) goto top

innerprod := \( s_1 + \) innerprod
- nop
Conclusions

• Instruction scheduling is useful for modern pipelined and superscalar architectures
• Typical programs have small basic blocks
  - Limits the effectiveness of instruction scheduler
  - There are global scheduling algorithms, whose cost and complexity is high
• Loop unrolling exposes more opportunities for instruction scheduling