15-213
“The course that gives CMU its Zip!”

Code Optimization II:
Machine Dependent Optimizations
Oct. 1, 2002

Topics
- Machine-Dependent Optimizations
  - Pointer code
  - Unrolling
  - Enabling instruction level parallelism
- Understanding Processor Operation
  - Translation of instructions into operations
  - Out-of-order execution of operations
- Branches and Branch Prediction
- Advice

class11.ppt
Previous Best Combining Code

```c
void combine4(vec_ptr v, int *dest)
{
    int i;
    int length = vec_length(v);
    int *data = get_vec_start(v);
    int sum = 0;
    for (i = 0; i < length; i++)
        sum += data[i];
    *dest = sum;
}
```

Problem

- Compute sum of all elements in vector
- Vector represented by C-style abstract data type
- Achieved CPE of 2.00
  - Cycles per element
General Forms of Combining

```c
void abstract_combine4(vec_ptr v, data_t *dest)
{
    int i;
    int length = vec_length(v);
    data_t *data = get_vec_start(v);
    data_t t = IDENT;
    for (i = 0; i < length; i++)
        t = t OP data[i];
    *dest = t;
}
```

Data Types
- Use different declarations for `data_t`
- `int`
- `float`
- `double`

Operations
- Use different definitions of `OP` and `IDENT`
- `+ / 0`
- `* / 1`
## Machine Independent Opt. Results

### Optimizations
- Reduce function calls and memory references within loop

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Floating Point</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Abstract -g</td>
<td>42.06</td>
<td>41.44</td>
</tr>
<tr>
<td>Abstract -O2</td>
<td>31.25</td>
<td>31.25</td>
</tr>
<tr>
<td>Move vec_length</td>
<td>20.66</td>
<td>21.15</td>
</tr>
<tr>
<td>data access</td>
<td>6.00</td>
<td>8.00</td>
</tr>
<tr>
<td>Accum. in temp</td>
<td>2.00</td>
<td>3.00</td>
</tr>
</tbody>
</table>

### Performance Anomaly
- Computing FP product of all elements exceptionally slow.
- Very large speedup when accumulate in temporary
- Caused by quirk of IA32 floating point
  - Memory uses 64-bit format, register use 80
  - Benchmark data caused overflow of 64 bits, but not 80
**Optimizer Code**

```c
void combine4p(vec_ptr v, int *dest) {
    int length = vec_length(v);
    int *data = get_vec_start(v);
    int *dend = data + length;
    int sum = 0;
    while (data < dend) {
        sum += *data;
        data++;
    }
    *dest = sum;
}
```

**Optimization**

- Use pointers rather than array references
- **CPE:** 3.00 (Compiled -O2)
  - **Oops!** We’re not making progress here!

**Warning:** Some compilers do better job optimizing array code
Pointer vs. Array Code Inner Loops

Array Code

```
.L24: # Loop:
    addl (%eax,%edx,4),%ecx # sum += data[i]
    incl %edx # i++
    cmpl %esi,%edx # i:length
    jl .L24 # if < goto Loop
```

Pointer Code

```
.L30: # Loop:
    addl (%eax),%ecx # sum += *data
    addl $4,%eax # data ++
    cmpl %edx,%eax # data:dend
    jb .L30 # if < goto Loop
```

Performance

- **Array Code**: 4 instructions in 2 clock cycles
- **Pointer Code**: Almost same 4 instructions in 3 clock cycles
Modern CPU Design

Instruction Control
- Fetch Control
- Instruction Decode
- Instruction Cache
- Address
- Instrs.

Operations

Prediction OK?
- Register Updates
- Integer/Branch
- General Integer
- FP Add
- FP Mult/Div
- Load
- Store
- Data Cache

Execution

Operation Results
- Addr
- Data

Retirement Unit
- Register File

Functional Units
CPU Capabilities of Pentium III

Multiple Instructions Can Execute in Parallel

- 1 load
- 1 store
- 2 integer (one may be branch)
- 1 FP Addition
- 1 FP Multiplication or Division

Some Instructions Take > 1 Cycle, but Can be Pipelined

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Latency</th>
<th>Cycles/Issue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load / Store</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Integer Multiply</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Integer Divide</td>
<td>36</td>
<td>36</td>
</tr>
<tr>
<td>Double/Single FP Multiply</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>Double/Single FP Add</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Double/Single FP Divide</td>
<td>38</td>
<td>38</td>
</tr>
</tbody>
</table>
Instruction Control

Grabs Instruction Bytes From Memory
- Based on current PC + predicted targets for predicted branches
- Hardware dynamically guesses whether branches taken/not taken and (possibly) branch target

Translates Instructions Into Operations
- Primitive steps required to perform instruction
- Typical instruction requires 1–3 operations

Converts Register References Into Tags
- Abstract identifier linking destination of one operation with sources of later operations

Operations
Translation Example

Version of Combine4

- Integer data, multiply operation

```assembly
.L24:                          # Loop:
imull (%eax,%edx,4),%ecx    # t *= data[i]
inl %edx                      # i++
cmpl %esi,%edx               # i: length
jl .L24                      # if < goto Loop
```

Translation of First Iteration

```assembly
.L24:
imull (%eax,%edx,4),%ecx    
inl %edx                      
cmpl %esi,%edx               
jl .L24                      
```

```assembly
load (%eax,%edx.0,4) → t.1
imull t.1, %ecx.0 → %ecx.1
incl %edx.0 → %edx.1
cmpl %esi, %edx.1 → cc.1
jl-taken cc.1
```
Translation Example #1

<table>
<thead>
<tr>
<th>imull (%eax,%edx,4),%ecx</th>
<th>load (%eax,%edx.0,4) ➔ t.1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>imull t.1, %ecx.0 ➔ %ecx.1</td>
</tr>
</tbody>
</table>

- Split into two operations
  - `load` reads from memory to generate temporary result `t.1`
  - Multiply operation just operates on registers

- Operands
  - Registers `%eax` does not change in loop. Values will be retrieved from register file during decoding
  - Register `%ecx` changes on every iteration. Uniquely identify different versions as `%ecx.0, %ecx.1, %ecx.2, ...`
    - Register `renaming`
    - Values passed directly from producer to consumers
Translation Example #2

- Register `%edx` changes on each iteration. Rename as `%edx.0`, `%edx.1`, `%edx.2`, ...
Translation Example #3

- `cmpl %esi, %edx`
- `cmpl %esi, %edx.1 ➜ cc.1`

- Condition codes are treated similar to registers
- Assign tag to define connection between producer and consumer
Translation Example #4

- Instruction control unit determines destination of jump
- Predicts whether will be taken and target
- Starts fetching instruction at predicted destination
- Execution unit simply checks whether or not prediction was OK
- If not, it signals instruction control
  - Instruction control then “invalidates” any operations generated from misfetched instructions
  - Begins fetching and decoding instructions at correct target
Visualizing Operations

Operations
- Vertical position denotes time at which executed
  - Cannot begin operation until operands available
- Height denotes latency

Operands
- Arrows shown only for operands that are passed within execution unit

load (%eax, %edx, 4) → t.1
imull t.1, %ecx.0 → %ecx.1
incl %edx.0 → %edx.1
cmpl %esi, %edx.1 → cc.1
jl-taken cc.1
Visualizing Operations (cont.)

Time

Operations
- Same as before, except that add has latency of 1

load (%eax, %edx, 4) \(\rightarrow\) t.1
iaddl t.1, %ecx.0 \(\rightarrow\) %ecx.1
incl %edx.0 \(\rightarrow\) %edx.1
cmpl %esi, %edx.1 \(\rightarrow\) cc.1
jl-taken cc.1
3 Iterations of Combining Product

Unlimited Resource Analysis

- Assume operation can start as soon as operands available
- Operations for multiple iterations overlap in time

Performance

- Limiting factor becomes latency of integer multiplier
- Gives CPE of 4.0
4 Iterations of Combining Sum

Unlimited Resource Analysis

Performance

- Can begin a new iteration on each clock cycle
- Should give CPE of 1.0
- Would require executing 4 integer operations in parallel
Combining Sum: Resource Constraints

- Only have two integer functional units
- Some operations delayed even though operands available
- Set priority based on program order

Performance
- Sustain CPE of 2.0
void combine5(vec_ptr v, int *dest)
{
    int length = vec_length(v);
    int limit = length-2;
    int *data = get_vec_start(v);
    int sum = 0;
    int i;
    /* Combine 3 elements at a time */
    for (i = 0; i < limit; i+=3) {
        sum += data[i] + data[i+2]
             + data[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        sum += data[i];
    }
    *dest = sum;
}
Visualizing Unrolled Loop

- Loads can pipeline, since don’t have dependencies
- Only one set of loop control operations

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>load (%eax, %edx.0, 4)</td>
<td>t.1a</td>
</tr>
<tr>
<td>iaddl t.1a, %ecx.0c</td>
<td>%ecx.1a</td>
</tr>
<tr>
<td>load 4(%eax, %edx.0, 4)</td>
<td>t.1b</td>
</tr>
<tr>
<td>iaddl t.1b, %ecx.1a</td>
<td>%ecx.1b</td>
</tr>
<tr>
<td>load 8(%eax, %edx.0, 4)</td>
<td>t.1c</td>
</tr>
<tr>
<td>iaddl t.1c, %ecx.1b</td>
<td>%ecx.1c</td>
</tr>
<tr>
<td>iaddl $3,%edx.0</td>
<td>%edx.1</td>
</tr>
<tr>
<td>cmpl %esi, %edx.1</td>
<td>cc.1</td>
</tr>
<tr>
<td>jl-taken cc.1</td>
<td></td>
</tr>
</tbody>
</table>
### Executing with Loop Unrolling

- **Predicted Performance**
  - Can complete iteration in 3 cycles
  - Should give CPE of 1.0

- **Measured Performance**
  - CPE of 1.33
  - One iteration every 4 cycles
## Effect of Unrolling

<table>
<thead>
<tr>
<th>Unrolling Degree</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer Sum</td>
<td>2.00</td>
<td>1.50</td>
<td>1.33</td>
<td>1.50</td>
<td>1.25</td>
<td>1.06</td>
</tr>
<tr>
<td>Integer Product</td>
<td></td>
<td></td>
<td></td>
<td>4.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP Sum</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3.00</td>
<td></td>
</tr>
<tr>
<td>FP Product</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5.00</td>
</tr>
</tbody>
</table>

- Only helps integer sum for our examples
  - Other cases constrained by functional unit latencies
- Effect is nonlinear with degree of unrolling
  - Many subtle effects determine exact scheduling of operations
Serial Computation

Computation

\[ ((((((1 \times x_0) \times x_1) \times x_2) \times x_3)\]
\[ \times x_4) \times x_5) \times x_6) \times x_7) \times x_8) \times x_9)\]
\[ \times x_{10}) \times x_{11})\]

Performance

- N elements, D cycles/operation
- N*D cycles
Parallel Loop Unrolling

```c
void combine6(vec_ptr v, int *dest)
{
    int length = vec_length(v);
    int limit = length-1;
    int *data = get_vec_start(v);
    int x0 = 1;
    int x1 = 1;
    int i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x0 *= data[i];
        x1 *= data[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x0 *= data[i];
    }
    *dest = x0 * x1;
}
```

**Code Version**
- Integer product

**Optimization**
- Accumulate in two different products
  - Can be performed simultaneously
- Combine at end

**Performance**
- CPE = 2.0
- 2X performance
Dual Product Computation

Computation

\[
(((1 \times x_0) \times x_2) \times x_4) \times x_6 \times x_8 \times x_{10}) \times ((((1 \times x_1) \times x_3) \times x_5 \times x_7) \times x_9 \times x_{11})
\]

Performance

- N elements, D cycles/operation
- \((N/2+1)\)*D cycles
- ~2X performance improvement
Requirements for Parallel Computation

Mathematical

- Combining operation must be associative & commutative
  - OK for integer multiplication
  - Not strictly true for floating point
    » OK for most applications

Hardware

- Pipelined functional units
- Ability to dynamically extract parallelism from code
Visualizing Parallel Loop

- Two multiplies within loop no longer have data dependency
- Allows them to pipeline

load (%eax,%edx.0,4) \(\rightarrow\) t.1a
imull t.1a, %ecx.0 \(\rightarrow\) %ecx.1
load 4(%eax,%edx.0,4) \(\rightarrow\) t.1b
imull t.1b, %ebx.0 \(\rightarrow\) %ebx.1
iaddl $2,%edx.0 \(\rightarrow\) %edx.1
cmpl %esi, %edx.1 \(\rightarrow\) cc.1
jl-taken cc.1
Executing with Parallel Loop

- Predicted Performance
  - Can keep 4-cycle multiplier busy performing two simultaneous multiplications
  - Gives CPE of 2.0
## Optimization Results for Combining

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Floating Point</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>+</td>
<td>*</td>
</tr>
<tr>
<td>Abstract -g</td>
<td>42.06</td>
<td>41.86</td>
</tr>
<tr>
<td>Abstract -O2</td>
<td>31.25</td>
<td>33.25</td>
</tr>
<tr>
<td>Move vec_length</td>
<td>20.66</td>
<td>21.25</td>
</tr>
<tr>
<td>data access</td>
<td>6.00</td>
<td>9.00</td>
</tr>
<tr>
<td>Accum. in temp</td>
<td>2.00</td>
<td>4.00</td>
</tr>
<tr>
<td>Pointer</td>
<td>3.00</td>
<td>4.00</td>
</tr>
<tr>
<td>Unroll 4</td>
<td>1.50</td>
<td>4.00</td>
</tr>
<tr>
<td>Unroll 16</td>
<td>1.06</td>
<td>4.00</td>
</tr>
<tr>
<td>2 X 2</td>
<td>1.50</td>
<td>2.00</td>
</tr>
<tr>
<td>4 X 4</td>
<td>1.50</td>
<td>2.00</td>
</tr>
<tr>
<td>8 X 4</td>
<td>1.25</td>
<td>1.25</td>
</tr>
<tr>
<td>Theoretical Opt.</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td><strong>Worst : Best</strong></td>
<td><strong>39.7</strong></td>
<td><strong>33.5</strong></td>
</tr>
</tbody>
</table>
Parallel Unrolling: Method #2

Code Version
- Integer product

Optimization
- Multiply pairs of elements together
- And then update product
- “Tree height reduction”

Performance
- CPE = 2.5

void combine6aa(vec_ptr v, int *dest)
{
    int length = vec_length(v);
    int limit = length-1;
    int *data = get_vec_start(v);
    int x = 1;
    int i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x *= (data[i] * data[i+1]);
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x *= data[i];
    }
    *dest = x;
}
Method #2 Computation

Computation

\[ (((((1 \times (x_0 \times x_1)) \times (x_2 \times x_3)) \times (x_4 \times x_5)) \times (x_6 \times x_7)) \times (x_8 \times x_9)) \times (x_{10} \times x_{11})) \]

Performance

- N elements, D cycles/operation
- Should be \((N/2+1)\times D\) cycles
  - CPE = 2.0
- Measured CPE worse

<table>
<thead>
<tr>
<th>Unrolling</th>
<th>CPE (measured)</th>
<th>CPE (theoretical)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2.50</td>
<td>2.00</td>
</tr>
<tr>
<td>3</td>
<td>1.67</td>
<td>1.33</td>
</tr>
<tr>
<td>4</td>
<td>1.50</td>
<td>1.00</td>
</tr>
<tr>
<td>6</td>
<td>1.78</td>
<td>1.00</td>
</tr>
</tbody>
</table>
Understanding Parallelism

/* Combine 2 elements at a time */
for (i = 0; i < limit; i+=2) {
    x = (x * data[i]) * data[i+1];
}

- CPE = 4.00
- All multiplies performed in sequence

/* Combine 2 elements at a time */
for (i = 0; i < limit; i+=2) {
    x = x * (data[i] * data[i+1]);
}

- CPE = 2.50
- Multiplies overlap
Limitations of Parallel Execution

Need Lots of Registers

■ To hold sums/products
■ Only 6 usable integer registers
  ● Also needed for pointers, loop conditions
■ 8 FP registers
■ When not enough registers, must spill temporaries onto stack
  ● Wipes out any performance gains
■ Not helped by renaming
  ● Cannot reference more operands than instruction set allows
  ● Major drawback of IA32 instruction set
Register Spilling Example

Example

- 8 X 8 integer product
- 7 local variables share 1 register
- See that are storing locals on stack
- E.g., at -8 (%ebp)

```
.L165:
imull (%eax),%ecx
movl -4(%ebp),%edi
imull 4(%eax),%edi
movl %edi,-4(%ebp)
movl -8(%ebp),%edi
imull 8(%eax),%edi
movl %edi,-8(%ebp)
movl -12(%ebp),%edi
imull 12(%eax),%edi
movl %edi,-12(%ebp)
movl -16(%ebp),%edi
imull 16(%eax),%edi
movl %edi,-16(%ebp)
...
addl $32,%eax
addl $8,%edx
cmpl -32(%ebp),%edx
jl .L165
```
## Summary: Results for Pentium III

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Floating Point</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>+</td>
<td>*</td>
</tr>
<tr>
<td>Abstract -g</td>
<td>42.06</td>
<td>41.86</td>
</tr>
<tr>
<td>Abstract -O2</td>
<td>31.25</td>
<td>33.25</td>
</tr>
<tr>
<td>Move vec_length</td>
<td>20.66</td>
<td>21.25</td>
</tr>
<tr>
<td>data access</td>
<td>6.00</td>
<td>9.00</td>
</tr>
<tr>
<td>Accum. in temp</td>
<td>2.00</td>
<td>4.00</td>
</tr>
<tr>
<td>Unroll 4</td>
<td>1.50</td>
<td>4.00</td>
</tr>
<tr>
<td>Unroll 16</td>
<td>1.06</td>
<td>4.00</td>
</tr>
<tr>
<td>4 X 2</td>
<td>1.50</td>
<td>2.00</td>
</tr>
<tr>
<td>8 X 4</td>
<td>1.25</td>
<td>1.25</td>
</tr>
<tr>
<td>8 X 8</td>
<td>1.88</td>
<td>1.88</td>
</tr>
<tr>
<td><strong>Worst : Best</strong></td>
<td><strong>39.7</strong></td>
<td><strong>33.5</strong></td>
</tr>
</tbody>
</table>

- Biggest gain doing basic optimizations
- But, last little bit helps
## Results for Alpha Processor

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Floating Point</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>+</td>
<td>*</td>
</tr>
<tr>
<td>Abstract -g</td>
<td>40.14</td>
<td>47.14</td>
</tr>
<tr>
<td>Abstract -O2</td>
<td>25.08</td>
<td>36.05</td>
</tr>
<tr>
<td>Move vec_length</td>
<td>19.19</td>
<td>32.18</td>
</tr>
<tr>
<td>data access</td>
<td>6.26</td>
<td>12.52</td>
</tr>
<tr>
<td>Accum. in temp</td>
<td>1.76</td>
<td>9.01</td>
</tr>
<tr>
<td>Unroll 4</td>
<td>1.51</td>
<td>9.01</td>
</tr>
<tr>
<td>Unroll 16</td>
<td>1.25</td>
<td>9.01</td>
</tr>
<tr>
<td>4 X 2</td>
<td>1.19</td>
<td>4.69</td>
</tr>
<tr>
<td>8 X 4</td>
<td>1.15</td>
<td>4.12</td>
</tr>
<tr>
<td>8 X 8</td>
<td>1.11</td>
<td>4.24</td>
</tr>
</tbody>
</table>

**Worst : Best**

- Overall trends very similar to those for Pentium III.
- Even though very different architecture and compiler
## Results for Pentium 4

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th></th>
<th>Floating Point</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>+</td>
<td>*</td>
<td>+</td>
<td>*</td>
</tr>
<tr>
<td>Abstract -g</td>
<td>35.25</td>
<td>35.34</td>
<td>35.85</td>
<td>38.00</td>
</tr>
<tr>
<td>Abstract -O2</td>
<td>26.52</td>
<td>30.26</td>
<td>31.55</td>
<td>32.00</td>
</tr>
<tr>
<td>Move vec_length</td>
<td>18.00</td>
<td>25.71</td>
<td>23.36</td>
<td>24.25</td>
</tr>
<tr>
<td>data access</td>
<td>3.39</td>
<td>31.56</td>
<td>27.50</td>
<td>28.35</td>
</tr>
<tr>
<td>Accum. in temp</td>
<td>2.00</td>
<td>14.00</td>
<td>5.00</td>
<td>7.00</td>
</tr>
<tr>
<td>Unroll 4</td>
<td>1.01</td>
<td>14.00</td>
<td>5.00</td>
<td>7.00</td>
</tr>
<tr>
<td>Unroll 16</td>
<td>1.00</td>
<td>14.00</td>
<td>5.00</td>
<td>7.00</td>
</tr>
<tr>
<td>4 X 2</td>
<td>1.02</td>
<td>7.00</td>
<td>2.63</td>
<td>3.50</td>
</tr>
<tr>
<td>8 X 4</td>
<td>1.01</td>
<td>3.98</td>
<td>1.82</td>
<td>2.00</td>
</tr>
<tr>
<td>8 X 8</td>
<td>1.63</td>
<td>4.50</td>
<td>2.42</td>
<td>2.31</td>
</tr>
<tr>
<td><strong>Worst : Best</strong></td>
<td><strong>35.2</strong></td>
<td><strong>8.9</strong></td>
<td><strong>19.7</strong></td>
<td><strong>19.0</strong></td>
</tr>
</tbody>
</table>

- Higher latencies (int * = 14, fp + = 5.0, fp * = 7.0)
  - Clock runs at 2.0 GHz
  - Not an improvement over 1.0 GHz P3 for integer *
- Avoids FP multiplication anomaly
What About Branches?

Challenge

- Instruction Control Unit must work well ahead of Exec. Unit
  - To generate enough operations to keep EU busy

```assembly
80489f3:  movl  $0x1,%ecx
80489f8:  xorl  %edx,%edx
80489fa:  cmpl  %esi,%edx
80489fc:  jnl   8048a25
80489fe:  movl  %esi,%esi
8048a00:  imull (%eax,%edx,4),%ecx
```

- When encounters conditional branch, cannot reliably determine where to continue fetching
Branch Outcomes

- When encounter conditional branch, cannot determine where to continue fetching
  - Branch Taken: Transfer control to branch target
  - Branch Not-Taken: Continue with next instruction in sequence

- Cannot resolve until outcome determined by branch/integer unit

```
80489f3:    movl   $0x1,%ecx
80489f8:    xorl   %edx,%edx
80489fa:    cmpl   %esi,%edx
80489fc:    jnl    8048a25
80489fe:    movl   %esi,%esi
8048a00:    imull  (%eax,%edx,4),%ecx
```

- Branch Not-Taken

```
8048a25:    cmpl   %edi,%edx
8048a27:    jl     8048a20
8048a29:    movl   0xc(%ebp),%eax
8048a2c:    leal   0xfffffffffe8(%ebp),%esp
8048a2f:    movl   %ecx,(%eax)
```

- Branch Taken
Branch Prediction

Idea

- Guess which way branch will go
- Begin executing instructions at predicted position
  - But don’t actually modify register or memory data

```
80489f3:  movl  $0x1,%ecx
80489f8:  xorl  %edx,%edx
80489fa:  cmpl  %esi,%edx
80489fc:  jnl   8048a25
...   
```

Predict Taken

```
8048a25:  cmpl  %edi,%edx
8048a27:  jl    8048a20
8048a29:  movl  0xc(%ebp),%eax
8048a2c:  leal  0xfffffffffe8(%ebp),%esp
8048a2f:  movl  %ecx,(%eax)
```

Execute
Branch Prediction Through Loop

Assume vector length = 100

Predict Taken (OK)

Predict Taken (Oops)

Read invalid location

Executed

Fetched

Location

Executed

Fetched

- 42 -
## Branch Misprediction Invalidation

**Assume vector length = 100**

<table>
<thead>
<tr>
<th>Address</th>
<th>Code</th>
<th>Instruction</th>
<th>Predicate</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>80488b1:</td>
<td>movl</td>
<td>(%ecx, %edx, 4), %eax</td>
<td></td>
<td>98</td>
</tr>
<tr>
<td>80488b4:</td>
<td>addl</td>
<td>%eax, (%edi)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>80488b6:</td>
<td>incl</td>
<td>%edx</td>
<td></td>
<td></td>
</tr>
<tr>
<td>80488b7:</td>
<td>cmpq</td>
<td>%esi, %edx</td>
<td></td>
<td></td>
</tr>
<tr>
<td>80488b9:</td>
<td>jle</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Predict Taken (OK)**

<table>
<thead>
<tr>
<th>Address</th>
<th>Code</th>
<th>Instruction</th>
<th>Predicate</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>80488b1:</td>
<td>movl</td>
<td>(%ecx, %edx, 4), %eax</td>
<td></td>
<td>99</td>
</tr>
<tr>
<td>80488b4:</td>
<td>addl</td>
<td>%eax, (%edi)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>80488b6:</td>
<td>incl</td>
<td>%edx</td>
<td></td>
<td></td>
</tr>
<tr>
<td>80488b7:</td>
<td>cmpq</td>
<td>%esi, %edx</td>
<td></td>
<td></td>
</tr>
<tr>
<td>80488b9:</td>
<td>jle</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Predict Taken (Oops)**

<table>
<thead>
<tr>
<th>Address</th>
<th>Code</th>
<th>Instruction</th>
<th>Predicate</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>80488b1:</td>
<td>movl</td>
<td>(%ecx, %edx, 4), %eax</td>
<td></td>
<td>100</td>
</tr>
<tr>
<td>80488b4:</td>
<td>addl</td>
<td>%eax, (%edi)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>80488b6:</td>
<td>incl</td>
<td>%edx</td>
<td></td>
<td></td>
</tr>
<tr>
<td>80488b7:</td>
<td>cmpq</td>
<td>%esi, %edx</td>
<td></td>
<td></td>
</tr>
<tr>
<td>80488b9:</td>
<td>jle</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Invalidate**
Branch Misprediction Recovery

Assume vector length = 100

Predict Taken (OK)

Definitely not taken

Performance Cost

- Misprediction on Pentium III wastes ~14 clock cycles
- That’s a lot of time on a high performance processor
Avoiding Branches

On Modern Processor, Branches Very Expensive
- Unless prediction can be reliable
- When possible, best to avoid altogether

Example
- Compute maximum of two values
  - 14 cycles when prediction correct
  - 29 cycles when incorrect

```c
int max(int x, int y)
{
    return (x < y) ? y : x;
}
```

```assembly
movl 12(%ebp),%edx    # Get y
movl 8(%ebp),%eax     # rval=x
cmpl %edx,%eax        # rval:y
jge L11                # skip when >=
movl %edx,%eax        # rval=y
L11:
```
Avoiding Branches with Bit Tricks

- In style of Lab #1
- Use masking rather than conditionals

```c
int bmax(int x, int y)
{
    int mask = -(x>y);
    return (mask & x) | (~mask & y);
}
```

- Compiler still uses conditional
  - 16 cycles when predict correctly
  - 32 cycles when mispredict

```
xorl %edx, %edx      # mask = 0
movl 8(%ebp), %eax
movl 12(%ebp), %ecx
cmpl %ecx, %eax
jle L13              # skip if x\leq y
movl $-1, %edx       # mask = -1
L13:
```
Avoiding Branches with Bit Tricks

- Force compiler to generate desired code

```c
int bvmax(int x, int y)
{
    volatile int t = (x>y);
    int mask = ~t;
    return (mask & x) | (~mask & y);
}
```

```asm
movl 8(%ebp),%ecx       # Get x
movl 12(%ebp),%edx      # Get y
cmpl %edx,%ecx          # x:y
setg %al                # (x>y)
movzbl %al,%eax         # Zero extend
movl %eax,-4(%ebp)      # Save as t
movl -4(%ebp),%eax      # Retrieve t
```

- `volatile` declaration forces value to be written to memory
  - Compiler must therefore generate code to compute `t`
  - Simplest way is `setg/movzbl` combination

- Not very elegant!
  - A hack to get control over compiler

- 22 clock cycles on all data
  - Better than misprediction
Conditional Move

- Added with P6 microarchitecture (PentiumPro onward)
- `cmovXX1 %edx, %eax`
  - If condition `XX` holds, copy `%edx` to `%eax`
  - Doesn’t involve any branching
  - Handled as operation within Execution Unit

```
movl 8(%ebp),%edx  # Get x
movl 12(%ebp),%eax  # rval=y
cmpl %edx, %eax    # rval: x
cmovll %edx,%eax   # if <, rval=x
```

- Current version of GCC won’t use this instruction
  - Thinks it’s compiling for a 386

- Performance
  - 14 cycles on all data
Machine-Dependent Opt. Summary

Pointer Code
- Look carefully at generated code to see whether helpful

Loop Unrolling
- Some compilers do this automatically
- Generally not as clever as what can achieve by hand

Exposing Instruction-Level Parallelism
- Very machine dependent

Warning:
- Benefits depend heavily on particular machine
- Best if performed by compiler
  - But GCC on IA32/Linux is not very good
- Do only for performance-critical parts of code
Role of Programmer

How should I write my programs, given that I have a good, optimizing compiler?

Don’t: Smash Code into Oblivion
- Hard to read, maintain, & assure correctness

Do:
- Select best algorithm
- Write code that’s readable & maintainable
  - Procedures, recursion, without built-in constant limits
  - Even though these factors can slow down code
- Eliminate optimization blockers
  - Allows compiler to do its job

Focus on Inner Loops
- Do detailed optimizations where code will be executed repeatedly
- Will get most performance gain here