Lecture 14: Program Optimization

Computer Architecture and Systems Programming
(252-0061-00)

Timothy Roscoe
Herbstsemester 2012

Last time

Making the pipelined processor work!

• Data Hazards
  – Instruction having register R as source follows shortly after instruction having
    register R as destination
  – Common condition, don’t want to slow down pipeline

• Control Hazards
  – Mispredict conditional branch
    • Our design predicts all branches as being taken
    • Naïve pipeline executes two extra instructions
  – Getting return address for ret instruction
    • Naïve pipeline executes three extra instructions

• Making sure it really works
  – What if multiple special cases happen simultaneously? Must analyze carefully
    • Only arises with unusual instruction combination

Today

• Program optimization
  – Optimization blocker: Memory aliasing
  – Out of order processing: Instruction level parallelism
  – Understanding branch prediction

Optimization Blocker: Memory Aliasing

• Code updates b[i] (= memory access) on every iteration
• Why couldn’t compiler optimize this away?

Reason

• If memory is accessed, compiler assumes the possibility of side effects

Example:

```c
/* Sums rows of n x n matrix a 
   and stores in vector b */
void sum_rows1(double *a, double *b, long n) {
  long i, j;
  for (i = 0; i < n; i++) {
    b[i] = 0;
    for (j = 0; j < n; j++)
      b[i] += a[i*n + j];
  }
}
```

Value of B:

<table>
<thead>
<tr>
<th>init: [4, 8, 16]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 = 0: [3, 8, 16]</td>
</tr>
<tr>
<td>1 = 1: [3, 22, 16]</td>
</tr>
<tr>
<td>1 = 2: [3, 22, 224]</td>
</tr>
</tbody>
</table>

Removing Aliasing

```c
/* Sums rows of n x n matrix a 
   and stores in vector b */
void sum_rows2(double *a, double *b, long n) {
  long i, j;
  for (i = 0; i < n; i++) {
    double val = 0;
    for (j = 0; j < n; j++)
      val += a[i*n + j];
    b[i] = val;
  }
}
```

Value of B:

<table>
<thead>
<tr>
<th>init: [4, 8, 16]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 = 0: [3, 8, 16]</td>
</tr>
<tr>
<td>1 = 1: [3, 22, 16]</td>
</tr>
<tr>
<td>1 = 2: [3, 22, 224]</td>
</tr>
</tbody>
</table>

• Scalar replacement:
  – Copy array elements that are reused into temporary variables
  – Assumes no memory aliasing (otherwise possibly incorrect)
Unaliased version when aliasing happens

```
/* Sum rows is of n x n matrix a and store in vector b */
void sumrow(double *a, double *b, double *c, int n) {
    long i, j;
    for (i = 0; i < n; i++) {
        double val = 0;
        for (j = 0; j < n; j++)
            val += a[i*n + j];
        b[i] = val;
    }
}
```

- Allowing still creates interference
- Result different from before

Value of B:

```
Init: [4, 8, 16]
1 = 8: [3, 8, 16]
1 = 1: [3, 27, 16]
1 = 2: [3, 27, 24]
```

Optimization Blocker: Memory Aliasing

- Memory aliasing: Two different memory references write to the same location
- Easy to have happen in C
  - Since allowed to do address arithmetic
  - Direct access to storage structures
- Hard to analyze = compiler cannot figure it out
  - Hence is conservative

- Solution: Scalar replacement in innermost loop
  - Copy memory variables that are reused into local variables
  - Basic scheme:
    - Load: t1 = a[i], t2 = b[j], ...
    - Compute: t4 = t1 * t2, ...
    - Store: a[i] = t2, b[j] = t7, ...

• Matrix multiplication: C = A*B + C

```
c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            for (k = 0; k < n; k++)
                c[i*n+j] += a[i*n+k] * b[k*n+j];
}
```

- Which array elements are reused?
- All of them! But how to take advantage?

Step 1: Blocking (here: 2 x 2)

- Blocking, also called tiling = partial unrolling + loop exchange
  - Assumes associativity (= compiler will never do it)

```
c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            for (k = 0; k < n; k++)
                c[i*n+j] += a[i*n+k] * b[k*n+j];
}
```

Step 2: Unrolling inner loops

- Every array element a[...], b[...], c[...] used twice
- Now scalar replacement can be applied

```
c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            for (k = 0; k < n; k++)
                c[i*n+j] += a[i*n+k] * b[k*n+j];
}
```

Today

- Program optimization
  - Optimization blocker: Memory aliasing
  - Out of order processing: Instruction level parallelism
  - Understanding branch prediction
Example: Compute factorials

```c
int rfact(int n) {
    if (n <= 1) return 1;
    return n * rfact(n-1);
}

int fact(int n) {
    int i;
    int result = 1;
    for (i = n; i > 0; i--) result = result * i;
    return result;
}
```

- Machines
  - Intel P. 4 Nocona, 3.2 GHz
  - Intel Core 2, 2.7 GHz
- Compiler versions
  - GCC 3.4.2

Optimization 1: Loop unrolling

```c
int fact_u3a(int n) {
    int i;
    int result = 1;
    for (i = n; i >= 3; i--) {
        result *= i;
        result *= (i-1);
        result *= (i-2);
    }
    for (; i > 0; i--) result *= i;
    return result;
}
```

Cycles per element (or per mul)

<table>
<thead>
<tr>
<th></th>
<th>Nocona</th>
<th>Core 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>rfact</td>
<td>15.5</td>
<td>6.0</td>
</tr>
<tr>
<td>fact</td>
<td>10.0</td>
<td>3.0</td>
</tr>
</tbody>
</table>

- Compute more values per iteration
- Does not help here
- Why? Branch prediction – details later

Optimization 2: Multiple accumulators

```c
int fact_u3b(int n) {
    int i;
    int result0 = 1;
    int result1 = 1;
    int result2 = 1;
    for (i = n; i >= 3; i--) {
        result0 *= i;
        result1 *= (i-1);
        result2 *= (i-2);
    }
    for (; i > 0; i--) result0 *= i;
    return result0 * result1 * result2;
}
```

Cycles per element (or per mul)

<table>
<thead>
<tr>
<th></th>
<th>Nocona</th>
<th>Core 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>rfact</td>
<td>15.5</td>
<td>6.0</td>
</tr>
<tr>
<td>fact</td>
<td>10.0</td>
<td>3.0</td>
</tr>
<tr>
<td>fact_u3a</td>
<td>10.0</td>
<td>3.0</td>
</tr>
<tr>
<td>fact_u3b</td>
<td>3.3</td>
<td>1.0</td>
</tr>
</tbody>
</table>

- That seems to help. Can one get even faster?
- Explanation: instruction level parallelism – details later

Superscalar processor

- Definition: A superscalar processor can issue and execute **multiple instructions in one cycle**. The instructions are retrieved from a sequential instruction stream and are usually scheduled dynamically.
- Benefit: without programming effort, superscalar processor can take advantage of the **instruction level parallelism** that most programs have
- Most CPUs since about 1998 are superscalar.
- Intel: since Pentium Pro

Pentium 4 Nocona CPU

- Multiple instructions can execute in parallel
  - 1 load, with address computation
  - 1 store, with address computation
  - 2 simple integer (one may be branch)
  - 1 complex integer (multiply/divide)
  - 1 FP/SSE3 unit
  - 1 FP move (does all conversions)
- 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>5</td>
<td>1</td>
</tr>
<tr>
<td>Integer Multiply</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>Integer/Long Divide</td>
<td>36/106</td>
<td>36/106</td>
</tr>
<tr>
<td>Single/Double FP Multiply</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>Single/Double FP Add</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>Single/Double FP Divide</td>
<td>32/46</td>
<td>32/46</td>
</tr>
</tbody>
</table>
Latency versus Throughput

- Last slide: \[ \text{latency} \times \text{cycles/issue} \]
  - Integer Multiply: 10 cycles/issue
  - Step 1: 1 cycle
  - Step 2: 1 cycle
  - Step 10: 1 cycle

- Consequence:
  - How fast can 10 independent int mults be executed?
    \[ t_1 = t_2^7 + t_3; \quad t_4 = t_5^7 + t_6; \ldots \]
  - How fast can 10 sequentially dependent int mults be executed?
    \[ t_1 = t_2^7 + t_3; \quad t_4 = t_5^7 + t_1; \quad t_6 = t_7^7 + t_4; \ldots \]

- Major problem for fast execution: \textit{Keep pipelines filled}

Hard Bounds

- How many cycles at least if
  - Function requires \( n \) int mults?
  - Function requires \( n \) float adds?
  - Function requires \( n \) float ops (adds and mults)?

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Latency</th>
<th>Cycles/Issue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load / Store</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>Integer Multiply</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>Integer/Long Divide</td>
<td>36/106</td>
<td>36/106</td>
</tr>
<tr>
<td>Single/Double FP Multiply</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>Single/Double FP Add</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>Single/Double FP Divide</td>
<td>32/46</td>
<td>32/46</td>
</tr>
</tbody>
</table>

Performance in Numerical computing

- Numerical computing = computing dominated by floating point operations
- Example: Matrix multiplication

- Performance measure:
  - Floating point operations per second \( \text{(flop/s)} \)
    - Counting only floating point adds and mults
    - Higher is better
    - Like inverse runtime

- Theoretical scalar (no vector SSE) peak performance on Nocona?
  - 3.2 Gflop/s = 3200 Mflop/s. \textit{Why?}

Nocona vs. Core 2

Nocona (3.2 GHz)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Latency</th>
<th>Cycles/Issue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load / Store</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>Integer Multiply</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>Integer/Long Divide</td>
<td>36/106</td>
<td>36/106</td>
</tr>
<tr>
<td>Single/Double FP Multiply</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>Single/Double FP Add</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>Single/Double FP Divide</td>
<td>32/46</td>
<td>32/46</td>
</tr>
</tbody>
</table>

Core 2 (2.7 GHz) (More recent Intel microprocessors)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Latency</th>
<th>Cycles/Issue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load / Store</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>Integer Multiply</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Integer/Long Divide</td>
<td>18/50</td>
<td>18/50</td>
</tr>
<tr>
<td>Single/Double FP Multiply</td>
<td>4/5</td>
<td>1</td>
</tr>
<tr>
<td>Single/Double FP Add</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Single/Double FP Divide</td>
<td>18/32</td>
<td>18/32</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 \textit{micro-operations} (for CISC style CPUs)
  - Micro-op = primitive step 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

Translating into micro-operations

- \texttt{imulq %rax, 8(%rbx,%rdx,4)}
- \texttt{load 8(%rbx,%rdx,4) → templ}
  \texttt{imulq %rax, templ}
  \texttt{store templ, 8(%rbx,%rdx,4)}
- Exact form and format of operations is trade secret
Example Computation

```c
void combine4(vec_ptr v, data_t *dest)
{
    int i;
    int length = vec_length(v);
    data_t *d = get_vec_start(v);
    data_t t = IDENT;
    for (i = 0; i < length; i++)
    {
        t = t OP d[i];
        *dest = t;
    }
}
```

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

- **Operations**
  - Use different definitions of `OP` and `IDENT`
  - `+ / *`
**Loop Unrolling**

```c
void unroll2a_combine(vec_ptr v, data_t *dest) {
    int length = vec_length(v);
    int limit = length-1;
    data_t x = get_vec_start(v);
    data_t d = IDENT;
    int i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x = x OP d[i] OP d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x = x OP d[i];
    }
    *dest = x;
}
```

- Perform 2x more useful work per iteration

**Effect of loop unrolling**

<table>
<thead>
<tr>
<th>Method</th>
<th>Int (add/mult)</th>
<th>Float (add/mult)</th>
</tr>
</thead>
<tbody>
<tr>
<td>combine4</td>
<td>2.2</td>
<td>10.0</td>
</tr>
<tr>
<td>unroll2</td>
<td>1.5</td>
<td>10.0</td>
</tr>
<tr>
<td>bound</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

- Helps integer sum
- Others don’t sum. *Why?*
  - Still sequential dependency

```c
x = (x OP d[i]) OP d[i+1];
```

**Loop unrolling with Reassociation**

```c
void unroll2ra_combine(vec_ptr v, data_t *dest) {
    int length = vec_length(v);
    int limit = length-1;
    data_t x = get_vec_start(v);
    data_t x1 = IDENT;
    int i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x = x OP d[i] OP d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x = x OP d[i];
    }
    *dest = x;
}
```

- Can this change the result of the computation?
- Yes, for FP. *Why?*

**Effect of Reassociation**

<table>
<thead>
<tr>
<th>Method</th>
<th>Int (add/mult)</th>
<th>Float (add/mult)</th>
</tr>
</thead>
<tbody>
<tr>
<td>combine4</td>
<td>2.2</td>
<td>10.0</td>
</tr>
<tr>
<td>unroll2</td>
<td>1.5</td>
<td>10.0</td>
</tr>
<tr>
<td>unroll2-ra</td>
<td>1.56</td>
<td>5.0</td>
</tr>
<tr>
<td>bound</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

- Nearly 2x speedup for Int *, FP +, FP *
  - Reason: Breaks sequential dependency
  - Why is that? (next slide)

**Reassociated Computation**

```c
x = x OP (d[i] OP d[i+1]);
```

- What changed:
  - Ops in the next iteration can be started early (no dependency)

- Overall performance
  - N elements, D cycles latency/op
  - Should be (N/2+1)*D cycles: \( \text{CPE} = \frac{D}{2} \)
  - Measured CPE slightly worse for FP

**Loop unrolling with separate accumulators**

```c
void unroll2sa_combine(vec_ptr v, data_t *dest) {
    int length = vec_length(v);
    int limit = length-1;
    data_t x = get_vec_start(v);
    data_t x0 = IDENT;
    data_t x1 = IDENT;
    int i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x0 = x0 OP d[i];
        x1 = x1 OP d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x0 = x0 OP d[i];
    }
    *dest = x0 OP x1;
}
```

- Different form of reassociation
Effect of separate accumulators

<table>
<thead>
<tr>
<th>Method</th>
<th>Int (add/mult)</th>
<th>Float (add/mult)</th>
</tr>
</thead>
<tbody>
<tr>
<td>combine4</td>
<td>2.2</td>
<td>10.0</td>
</tr>
<tr>
<td>unroll2</td>
<td>1.5</td>
<td>10.0</td>
</tr>
<tr>
<td>unroll2-ra</td>
<td>1.56</td>
<td>5.0</td>
</tr>
<tr>
<td>unroll2-sa</td>
<td>1.50</td>
<td>5.0</td>
</tr>
<tr>
<td>bound</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Almost exact 2x speedup (over unroll2) for Int *, FP +, FP *

- Breaks sequential dependency in a "cleaner," more obvious way
  \[
  x_0 = x_0 \text{ OP } d[i]; \\
  x_1 = x_1 \text{ OP } d[i+1];
  \]

Separate accumulators

- What changed:
  - Two independent "streams" of operations
- Overall performance
  - N elements
  - D cycles latency/op
  - Should be \((N/2+1)*D\) cycles:
    - CPE=D/2
  - CPE matches prediction!

Unrolling & Accumulating

- Idea
  - Can unroll to any degree \(L\)
  - Can accumulate \(K\) results in parallel
  - \(L\) must be multiple of \(K\)

- Limitations
  - Diminishing returns
    - Cannot go beyond throughput limitations of execution units
  - Large overhead for short lengths
    - Finish off iterations sequentially

Unrolling & Accumulating: Intel floating point multiply

- Case
  - Intel Nocona
  - FP Multiplication
  - Theoretical Limit: 2.00

<table>
<thead>
<tr>
<th>FP *</th>
<th>Unrolling Factor L</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>K 1  2  3  4  6  8  10 12</td>
</tr>
<tr>
<td>------</td>
<td>---------------------</td>
</tr>
<tr>
<td>1</td>
<td>7.00 7.00 7.01 7.00</td>
</tr>
<tr>
<td>2</td>
<td>3.50 3.50 3.50 3.50</td>
</tr>
<tr>
<td>3</td>
<td>2.34</td>
</tr>
<tr>
<td>4</td>
<td>2.01 2.00 2.01</td>
</tr>
<tr>
<td>6</td>
<td>2.00 2.00 2.00</td>
</tr>
<tr>
<td>8</td>
<td>2.01</td>
</tr>
<tr>
<td>10</td>
<td>2.00</td>
</tr>
<tr>
<td>12</td>
<td>2.00</td>
</tr>
</tbody>
</table>

Unrolling & Accumulating: Intel integer multiplication

- Case
  - Intel Nocona
  - Integer Multiplication
  - Theoretical Limit: 1.00

<table>
<thead>
<tr>
<th>Int *</th>
<th>Unrolling Factor L</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>K 1  2  3  4  6  8  10 12</td>
</tr>
<tr>
<td>-------</td>
<td>---------------------</td>
</tr>
<tr>
<td>1</td>
<td>10.00 10.00 10.00 10.01</td>
</tr>
<tr>
<td>2</td>
<td>5.00 5.01 5.00 5.00</td>
</tr>
<tr>
<td>3</td>
<td>3.33 3.33</td>
</tr>
<tr>
<td>4</td>
<td>2.50 2.51 2.51</td>
</tr>
<tr>
<td>6</td>
<td>1.67 1.67 1.67</td>
</tr>
<tr>
<td>8</td>
<td>1.25 1.25</td>
</tr>
<tr>
<td>10</td>
<td>1.09</td>
</tr>
<tr>
<td>12</td>
<td>1.14</td>
</tr>
</tbody>
</table>
Unrolling & Accumulating: Intel integer addition

• Case
  – Intel Nocona
  – Integer addition
  – Theoretical Limit: 1.00 (unrolling enough)

Nocona vs. Core 2, Int *

• Performance
  – Newer version of GCC does reassociation
  – Why for int’s and not for float’s?

Intel vs. AMD Int *

• Performance
  – AMD multiplier much lower latency
  – Can get high performance with less work
  – Doesn’t achieve as good an optimum

Nocona vs. Core 2, Int +

• Performance
  – AMD gets below 1.0
  – Even just with unrolling

Intel vs. AMD Int +

• Performance
  – AMD gets below 1.0
  – Even just with unrolling

Explanation
  – Both Intel & AMD can "double pump" integer units
  – Only AMD can load two elements / cycle
Can We Go Faster?

- Yes, using SSE!
  - But not in this class 😊

What about branches?

- Challenge
  - Instruction Control Unit must work well ahead of Execution Unit to generate enough operations to keep EU busy

```
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
8048a04: movl %esi,%esi
8048a06: cmpl %esi,%edx
8048a08: xorl %edx,%edx
8048a0d: movl $0x1,%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

```
8048a20: %edi,%edx
8048a25: cmpl %esi,%esi
8048a27: jnl 8048a20
8048a2f: movl %esi,%esi
8048a30: imull (%eax,%edx,4),%ecx
8048a34: movl %esi,%esi
8048a36: cmpl %esi,%edx
8048a38: xorl %edx,%edx
8048a3d: movl $0x1,%ecx
```

Branch prediction

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

```
8048a25: cmpl %edi,%edx
8048a27: jl 8048a25
8048a29: movl %esi,%eax
8048a2c: leal 0xffffffe8(%ebp),%esp
8048a2f: movl %ecx,(%eax)
```

Branch prediction through loop

Assume vector length = 100

```
8048b1: movl (%eax,%edx,4),%eax
8048b4: addl %eax,(%edi)
8048b6: incl %edx
8048b9: jli 8048b1
8048b1: movl (%ecx,%edx,4),%eax
8048b4: addl %eax,(%edi)
8048b6: incl %edx
8048b9: jli 8048b1
```

Predict Taken (OK)
Predict Taken (Oops)
Read invalid location
Executed
Fetched
Branch misprediction invalidation

80488b1: movl (%ecx,%edx,4),%eax  Assume vector length = 100
80488b4: addi %eax,(%edi)  Predict Taken (OK)
80488b7: cmpl %esi,%edx  Predict Taken (Oops)
80488b9: jl 80488b1
\[ i = 98 \]
80488b1: movl (%ecx,%edx,4),%eax  Invalidate
80488b4: addi %eax,(%edi)
80488b7: cmpl %esi,%edx
80488b9: jl 80488b1  \[ i = 99 \]
80488b1: movl (%ecx,%edx,4),%eax
80488b4: addi %eax,(%edi)
80488b7: cmpl %esi,%edx
80488b9: jl 80488b1  \[ i = 100 \]

Branch misprediction recovery

80488b1: movl (%ecx,%edx,4),%eax  \( \text{Definitely not taken} \)
80488b4: addi %eax,(%edi)
80488b7: cmpl %esi,%edx  \( i = 99 \)
80488b9: jl 80488b1
80488b6: incl %edx  \( i = 98 \)
80488b4: addi %eax,(%edi)
80488b7: cmpl %esi,%edx
80488b9: jl 80488b1  \( i = 100 \)
80488b6: incl %edx
80488b4: addi %eax,(%edi)
80488b7: cmpl %esi,%edx
80488b9: jl 80488b1  \( i = 101 \)

Determining misprediction penalty

- GCC/x86-64 tries to minimize use of branches
  - Generates conditional moves when possible/sensible

```c
int cnt_gt = 0;
int cnt_le = 0;
int cnt_all = 0;
int choose_cmov(int x, int y)
{
    int result;
    if (x > y) {
        result = cnt_gt;
    } else {
        result = cnt_le;
    }
    ++cnt_all;
    return result;
}
```

Forcing Conditional

- Cannot use conditional move when either outcome has side effect

```c
choose_cond:  
    cmpl %esi, %edi
    jle .L8
    movl cnt_gt(%rip), %eax  
    incl %eax
    movl %eax, cnt_gt(%rip)
    ret
.L8:
    movl cnt_le(%rip), %eax
    incl %eax
    movl %eax, cnt_le(%rip)
    ret
```

Testing Methodology

- Idea
  - Measure procedure under two different prediction probabilities
    - \( P = 1.0 \): Perfect prediction
    - \( P = 0.5 \): Random data

- Test Data
  - \( x = 0, y = \pm 1 \)
  - Case +1: \( y = [+1, +1, +1, \ldots, +1, +1] \)
  - Case -1: \( y = [-1, -1, -1, \ldots, -1, -1] \)
  - Case A: \( y = [+1, -1, +1, \ldots, +1, -1] \) (alternate)
  - Case R: \( y = [+1, -1, -1, \ldots, -1, +1] \) (random)

Testing Outcomes

<table>
<thead>
<tr>
<th>Case</th>
<th>cmov</th>
<th>cond</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Nocona</td>
<td>+1</td>
<td>12.3</td>
</tr>
<tr>
<td>AMD Opteron</td>
<td>+1</td>
<td>8.05</td>
</tr>
<tr>
<td>Intel Core 2</td>
<td>+1</td>
<td>7.17</td>
</tr>
</tbody>
</table>

- Observations:
  - Conditional move insensitive to data
  - Perfect prediction for regular patterns
  - Else case requires 6 (Nocona), 2 (AMD), or 1 (Core 2) extra cycles
  - Averages to 15.2
  - Branch penalties: (for R, processor will get it right half of the time)
    - Nocona: \( 2^* (15.2 - 3.2) \) = 32 cycles
    - AMD: \( 2^* (15.7 - 8.3) \) = 13 cycles
    - Core 2: \( 2^* (17.7 - 8.7) \) = 18 cycles
Getting high performance so far

• Good compiler and flags

• Don’t do anything stupid
  – Watch out for hidden algorithmic inefficiencies
  – Write compiler-friendly code
    • Watch out for optimization blockers:
      procedure calls & memory references
    • Careful with implemented abstract data types
  – Look carefully at innermost loops (where most work is done)

• Tune code for machine
  – Exploit instruction-level parallelism
  – Avoid unpredictable branches
  – Make code cache friendly (Covered later in course)