Introduction to Computer Systems
15-213/18-243, spring 2009
11\textsuperscript{th} Lecture, Feb. 17\textsuperscript{th}

Instructors:
Gregory Kesden and Markus Püschel
Last Time

- Memory layout

- Buffer overflow, worms, and viruses

```plaintext
FF 00
Stack

80
Heap

Data

Text

08

B

foo stack frame

pad

exploit code

bar stack frame

data written by gets()
```
Last Time

- Program Optimization
Last Time

- Program optimization
  - Overview
  - Removing unnecessary procedure calls
  - Code motion/precomputation
  - Strength reduction
  - Sharing of common subexpressions
  - Optimization blocker: Procedure calls

```
for (i = 0; i < n; i++) {
    get_vec_element(v, i, &val);
    *res += val;
}
```

```
void lower(char *s)
{
    int i;
    for (i = 0; i < strlen(s); i++)
    {
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
    }
}```
Today

- Program optimization
  - Optimization blocker: Memory aliasing
  - Out of order processing: Instruction level parallelism
  - Understanding branch prediction
Optimization Blocker: Memory Aliasing

/* 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];
    }
}

# sum_rows1 inner loop
.L53:
    addsd (%rcx), %xmm0        # FP add
    addq $8, %rcx
    decq %rax
    movsd %xmm0, (%rsi,%r8,8)  # FP store
    jne .L53

- 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];
    }
}
```

```
double A[9] = 
{ 0,  1,  2, 
  4,  8, 16},
32, 64, 128};

sum_rows1(A, B, 3);
```

Value of B:

- init: [4, 8, 16]
- i = 0: [3, 8, 16]
- i = 1: [3, 22, 16]
- i = 2: [3, 22, 224]
Removing Aliasing

/* 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;
    }
}

# sum_rows2 inner loop
.L66:
    addsd (%rcx), %xmm0  # FP Add
    addq $8, %rcx
    decq %rax
    jne  .L66

■ Scalar replacement:
  ▪ Copy array elements that are reused into temporary variables
  ▪ Assumes no memory aliasing (otherwise possibly incorrect)
Unaliased Version When Aliasing Happens

```c
/* Sum rows is of n X n matrix a and store 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;
    }
}
```

```c
double A[9] =
    { 0, 1, 2,
      4, 8, 16,
      32, 64, 128};


sum_rows1(A, B, 3);
```

- Aliasing still creates interference
- Result different than before

Value of B:

<table>
<thead>
<tr>
<th>init:</th>
<th>[4, 8, 16]</th>
</tr>
</thead>
<tbody>
<tr>
<td>i = 0:</td>
<td>[3, 8, 16]</td>
</tr>
<tr>
<td>i = 1:</td>
<td>[3, 27, 16]</td>
</tr>
<tr>
<td>i = 2:</td>
<td>[3, 27, 224]</td>
</tr>
</tbody>
</table>
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: \( t_1 = a[i] \), \( t_2 = b[i+1] \), ...
    - Compute: \( t_4 = t_1 * t_2 \); ...
    - Store: \( a[i] = t_{12} \), \( b[i+1] = t_7 \), ...
More Difficult Example

Matrix multiplication: \( C = A \times B + C \)

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

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

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+=2)
        for (j = 0; j < n; j+=2)
            for (k = 0; k < n; k+=2)
                for (i1 = i; i1 < i+2; i1++)
                    for (j1 = j; j1 < j+2; j1++)
                        for (k1 = k; k1 < k+2; k1++)
                            c[i1*n+j1] += a[i1*n+k1]*b[k1*n+j1];
}
```

```
c  =  a * b  +  c
   i1
```
Step 2: Unrolling Inner Loops

```c
double *c = 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+=2)
        for (j = 0; j < n; j+=2)
            for (k = 0; k < n; k+=2)
                <body>
                
                c[i*n + j]        = a[i*n + k]*b[k*n + j] + a[i*n + k+1]*b[(k+1)*n + j]  
                                  + c[i*n + j]
                c[(i+1)*n + j]    = a[(i+1)*n + k]*b[k*n + j] + a[(i+1)*n + k+1]*b[(k+1)*n + j]  
                                  + c[(i+1)*n + j]
                c[i*n + (j+1)]    = a[i*n + k]*b[k*n + (j+1)] + a[i*n + k+1]*b[(k+1)*n + (j+1)]  
                                  + c[i*n + (j+1)]
                c[(i+1)*n + (j+1)] = a[(i+1)*n + k]*b[k*n + (j+1)]
                                  + a[(i+1)*n + k+1]*b[(k+1)*n + (j+1)]
                                  + c[(i+1)*n + (j+1)]
            
            <body>```

- Every array element a[...], b[...], c[...] used twice
- Now scalar replacement can be applied
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 Pentium 4 Nocona, 3.2 GHz
    - Fish Machines
  - Intel Core 2, 2.7 GHz

- **Compiler Versions**
  - GCC 3.4.2 (current on Fish machines)

### Cycles per element (or per mult)

<table>
<thead>
<tr>
<th>Machine</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>

Something changed from Pentium 4 to Core: Details later
Optimization 1: Loop Unrolling

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

```c
int fact_u3a(int n) {
    int i;
    int result = 1;

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

<table>
<thead>
<tr>
<th>Machine</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>
</tbody>
</table>
That seems to help. Can one get even faster?

Explanation: instruction level parallelism – details later
Modern CPU Design

Instruction Control

Instruction Cache

Fetch Control

Instruction Decode

Retirement Unit

Register File

Address

Instructions

Operations

Prediction OK?

Register Updates

Functional Units

Integer/Branch

General Integer

FP Add

FP Mult/Div

Load

Store

Operation Results

Data Cache

Data

Addr.

Addr.

Data
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:**
  - **Integer Multiply**
  - **Latency:** 10 cycles/issue
  - **Cycles/issue:** 1

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

- **Major problem for fast execution:** *Keep pipelines filled*
Hard Bounds

Latency and throughput of instructions

<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>

How many cycles at least if

- Function requires n int mults?
- Function requires n float adds?
- Function requires n float ops (adds and mults)?
Performance in Numerical Computing

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

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

Theoretical scalar (no vector SSE) peak performance on fish machines?
- 3.2 Gflop/s = 3200 Mflop/s. Why?
## Nocona vs. Core 2

**Nocona (3.2 GHz) (Saltwater fish machines)**

<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) (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 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

Goal: Each operation utilizes single functional unit

- Requires: Load, integer arithmetic, store

```plaintext
load 8(%rbx,%rdx,4) ➞ temp1
imulq %rax, temp1 ➞ temp2
store temp2, 8(%rbx,%rdx,4)
```

- Exact form and format of operations is trade secret
Traditional View of Instruction Execution

- **Imperative View**
  - Registers are fixed storage locations
    - Individual instructions read & write them
  - Instructions must be executed in specified sequence to guarantee proper program behavior

```plaintext
addq %rax, %rbx  # I1
andq %rbx, %rdx  # I2
mulq %rcx, %rbx  # I3
xorq %rbx, %rdi  # I4
```
**Dataflow View of Instruction Execution**

- Functional View
  - View each write as creating new instance of value
  - Operations can be performed as soon as operands available
  - No need to execute in original sequence

```
addq %rax, %rbx       # I1
andq %rbx, %rdx       # I2
mulq %rcx, %rbx       # I3
xorq %rbx, %rdi       # I4
```
Example Computation

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
  - + / 0
  - * / 1
Cycles Per Element (CPE)

- Convenient way to express performance of program that operators on vectors or lists
- Length = n
- In our case: CPE = cycles per OP (gives hard lower bound)
- T = CPE*n + Overhead

![Graph showing the relationship between cycles and number of elements with two linear functions: vsum1: Slope = 4.0 and vsum2: Slope = 3.5.](image)
x86-64 Compilation of Combine4

```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;
}
```

Cycles per element (or per OP)

<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>bound</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

### Inner Loop (Case: Integer Multiply)

```
L33:       # Loop:
movl (%eax,%edx,4), %ebx # temp = d[i]
incl %edx    # i++
imull %ebx, %ecx # x *= temp
compl %esi, %edx # i: length
jl L33       # if < goto Loop
```
Combine4 = Serial Computation (OP = *)

- Computation (length=8)
  \[ (((((1 \times d[0]) \times d[1]) \times d[2]) \times d[3]) \times d[4]) \times d[5]) \times d[6]) \times d[7]) \]

- **Sequential dependence**! Hence,
  - Performance: determined by latency of OP!

<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>bound</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>
Loop Unrolling

void unroll2a_combine(vec_ptr v, data_t *dest)
{
    int length = vec_length(v);
    int limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = 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;
}
Effect of Loop Unrolling

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

\[
x = (x \; \text{OP} \; d[i]) \; \text{OP} \; d[i+1];
\]
Loop Unrolling with Reassociation

Can this change the result of the computation?

Yes, for FP. \textit{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
    
    \[ x = x \, \text{OP} \, (d[i] \, \text{OP} \, d[i+1]) ; \]

  - Why is that? (next slide)
Reassociated Computation

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:
    \[ CPE = \frac{D}{2} \]
  - Measured CPE slightly worse for FP
Loop Unrolling with Separate Accumulators

void unroll2a_combine(vec_ptr v, data_t *dest) {
    int length = vec_length(v);
    int limit = length - 1;
    data_t *d = 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

```
x0 = x0 OP d[i];
x1 = x1 OP d[i+1];
```
**Separate Accumulators**

\[
x_0 = x_0 \text{ OP } d[i]; \\
x_1 = x_1 \text{ OP } d[i+1];
\]

- **What changed:**
  - Two independent “streams” of operations

- **Overall Performance**
  - N elements, D cycles latency/op
  - Should be \((N/2+1)\times D\) cycles:
    \[ \text{CPE} = D/2 \]
  - CPE matches prediction!

What Now?
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 FP *

**Case**
- Intel Nocona (Saltwater fish machines)
- 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</td>
</tr>
<tr>
<td>1</td>
<td>7.00</td>
</tr>
<tr>
<td>2</td>
<td>3.50</td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>
Unrolling & Accumulating: Intel FP +

- **Case**
  - Intel Nocona (Saltwater fish machines)
  - FP Addition
  - Theoretical Limit: 2.00

<table>
<thead>
<tr>
<th>FP +</th>
<th>Unrolling Factor L</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>K</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>5.00</td>
</tr>
<tr>
<td>2</td>
<td>2.50</td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>
Unrolling & Accumulating: Intel Int *

**Case**

- Intel Nocona (Saltwater fish machines)
- 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>1</td>
<td>10.00 10.00 10.00 10.01</td>
</tr>
<tr>
<td>2</td>
<td>5.00 5.01 5.00</td>
</tr>
<tr>
<td>3</td>
<td>3.33</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>
Unrolling & Accumulating: Intel Int +

- **Case**
  - Intel Nocona (Saltwater fish machines)
  - Integer addition
  - Theoretical Limit: 1.00 (unrolling enough)

<table>
<thead>
<tr>
<th>Int +</th>
<th>K</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>6</th>
<th>8</th>
<th>10</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2.20</td>
<td>1.50</td>
<td>1.10</td>
<td></td>
<td>1.03</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
<td>1.50</td>
<td>1.10</td>
<td></td>
<td>1.03</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td>1.34</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>1.09</td>
<td>1.03</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.01</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.03</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.04</td>
<td></td>
</tr>
<tr>
<td></td>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.11</td>
</tr>
</tbody>
</table>
### Nocona versus Core 2

#### Machines
- Intel Nocona
  - 3.2 GHz
- Intel Core 2
  - 2.7 GHz

#### Performance
- Core 2 lower latency & fully pipelined (1 cycle/issue)

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

<table>
<thead>
<tr>
<th>FP *</th>
<th>Unrolling Factor L</th>
</tr>
</thead>
<tbody>
<tr>
<td>K</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>4.00</td>
</tr>
<tr>
<td>2</td>
<td>2.00</td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>
Can We Go Faster?

- Yes, SSE!
  - But not in this class😊
  - 18-645
Today

- Program optimization
  - Optimization blocker: Memory aliasing
  - Out of order processing: Instruction level parallelism
  - Understanding branch prediction
What About Branches?

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>What to Do</th>
</tr>
</thead>
<tbody>
<tr>
<td>80489f3</td>
<td>movl</td>
<td>$0x1,%ecx</td>
</tr>
<tr>
<td>80489f8</td>
<td>xorl</td>
<td>%edx,%edx</td>
</tr>
<tr>
<td>80489fa</td>
<td>cmpl</td>
<td>%esi,%edx</td>
</tr>
<tr>
<td>80489fc</td>
<td>jnl</td>
<td>8048a25</td>
</tr>
<tr>
<td>80489fe</td>
<td>movl</td>
<td>%esi,%esi</td>
</tr>
<tr>
<td>8048a00</td>
<td>imull</td>
<td>(%eax,%edx,4),%ecx</td>
</tr>
</tbody>
</table>

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

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

```assembly
80489f3: movl $0x1,%ecx
80489f8: xorl %edx,%edx
80489fa: cmpl %esi,%edx
80489fc: jnl 8048a25
.
.
. 8048a25: cmpl %edi,%edx
8048a27: jl 8048a20
8048a29: movl 0xc(%ebp),%eax
8048a2c: leal 0xfffffffffe8(%ebp),%esp
8048a2f: movl %ecx,(%eax)
```
Branch Prediction Through Loop

Assume vector length = 100

| 80488b1: movl (%ecx,%edx,4),%eax | Predict Taken (OK) |
| 80488b4: addl %eax,(%edi)          |
| 80488b6: incl %edx                |
| 80488b7: cmpl %esi,%edx           |
| 80488b9: jl 80488b1                |

i = 98

| 80488b1: movl (%ecx,%edx,4),%eax | Predict Taken (Oops) |
| 80488b4: addl %eax,(%edi)          |
| 80488b6: incl %edx                |
| 80488b7: cmpl %esi,%edx           |
| 80488b9: jl 80488b1                |

i = 99

| 80488b1: movl (%ecx,%edx,4),%eax | Read invalid location |
| 80488b4: addl %eax,(%edi)          |
| 80488b6: incl %edx                |
| 80488b7: cmpl %esi,%edx           |
| 80488b9: jl 80488b1                |

i = 100

| 80488b1: movl (%ecx,%edx,4),%eax | Executed |
| 80488b4: addl %eax,(%edi)          |
| 80488b6: incl %edx                |
| 80488b7: cmpl %esi,%edx           |
| 80488b9: jl 80488b1                |

i = 101
Branch Misprediction Invalidation

Assume vector length = 100

Predict Taken (OK)

Predict Taken (Oops)

Invalidate
Branch Misprediction Recovery

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>80488b1:</td>
<td>movl (%ecx,%edx,4),%eax</td>
</tr>
<tr>
<td>80488b4:</td>
<td>addl %eax,(%edi)</td>
</tr>
<tr>
<td>80488b6:</td>
<td>incl %edx</td>
</tr>
<tr>
<td>80488b7:</td>
<td>cmpl %esi,%edx</td>
</tr>
<tr>
<td>80488b9:</td>
<td>jl 80488b1</td>
</tr>
<tr>
<td>80488bb:</td>
<td>leal 0xffffffffffe8(%ebp),%esp</td>
</tr>
<tr>
<td>80488be:</td>
<td>popl %ebx</td>
</tr>
<tr>
<td>80488bf:</td>
<td>popl %esi</td>
</tr>
<tr>
<td>80488c0:</td>
<td>popl %edi</td>
</tr>
</tbody>
</table>

- **i = 99**

**Definitely not taken**

**Performance Cost**

- Multiple clock cycles on modern processor
- Can be a major performance limiter
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;
}
```

```assembly
choose_cmov:
    cmp   %esi, %edi         # x:y
    movl  cnt_le(%rip), %eax # r = cnt_le
    cmovg cnt_gt(%rip), %eax # if >= r=cnt_gt
    incl  cnt_all(%rip)      # cnt_all++
    ret                       # return r
```
Forcing Conditional

- Cannot use conditional move when either outcome has side effect

```c
int cnt_gt = 0;
int cnt_le = 0;

int choose_cond(int x, int y) {
    int result;
    if (x > y) {
        result = ++cnt_gt;
    } else {
        result = ++cnt_le;
    }
    return result;
}
```

```assembly
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: \( y = [+1, +1, +1, ..., +1, +1] \)
    - Case −1: \( y = [-1, -1, -1, ..., -1, -1] \)
    - Case A: \( y = [+1, -1, +1, ..., +1, -1] \) (alternate)
    - Case R: \( y = [+1, -1, -1, ..., -1, +1] \) (random)
Testing Outcomes

<table>
<thead>
<tr>
<th>Case</th>
<th>cmov</th>
<th>cond</th>
</tr>
</thead>
<tbody>
<tr>
<td>+1</td>
<td>12.3</td>
<td>18.2</td>
</tr>
<tr>
<td>-1</td>
<td>12.3</td>
<td>12.2</td>
</tr>
<tr>
<td>A</td>
<td>12.3</td>
<td>15.2</td>
</tr>
<tr>
<td>R</td>
<td>12.3</td>
<td>31.2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Case</th>
<th>cmov</th>
<th>cond</th>
</tr>
</thead>
<tbody>
<tr>
<td>+1</td>
<td>8.05</td>
<td>10.1</td>
</tr>
<tr>
<td>-1</td>
<td>8.05</td>
<td>8.1</td>
</tr>
<tr>
<td>A</td>
<td>8.05</td>
<td>9.2</td>
</tr>
<tr>
<td>R</td>
<td>8.05</td>
<td>15.7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Case</th>
<th>cmov</th>
<th>cond</th>
</tr>
</thead>
<tbody>
<tr>
<td>+1</td>
<td>7.17</td>
<td>9.2</td>
</tr>
<tr>
<td>-1</td>
<td>7.17</td>
<td>8.2</td>
</tr>
<tr>
<td>A</td>
<td>7.17</td>
<td>8.7</td>
</tr>
<tr>
<td>R</td>
<td>7.17</td>
<td>17.7</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 * (31.2-15.2) = 32 cycles
  - AMD: 2 * (15.7-9.2) = 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)