Program Optimization

15-213: Introduction to Computer Systems
25\textsuperscript{th} Lecture, Nov. 23, 2010

\textbf{Instructors:}
Randy Bryant and Dave O’Hallaron
Today

- Overview
- Generally Useful Optimizations
  - Code motion/precomputation
  - Strength reduction
  - Sharing of common subexpressions
  - Removing unnecessary procedure calls
- Optimization Blockers
  - Procedure calls
  - Memory aliasing
- Exploiting Instruction-Level Parallelism
- Dealing with Conditionals
Performance Realities

- *There’s more to performance than asymptotic complexity*

- **Constant factors matter too!**
  - Easily see 10:1 performance range depending on how code is written
  - Must optimize at multiple levels:
    - algorithm, data representations, procedures, and loops

- **Must understand system to optimize performance**
  - How programs are compiled and executed
  - How to measure program performance and identify bottlenecks
  - How to improve performance without destroying code modularity and generality
Optimizing Compilers

- Provide efficient mapping of program to machine
  - register allocation
  - code selection and ordering (scheduling)
  - dead code elimination
  - eliminating minor inefficiencies

- Don’t (usually) improve asymptotic efficiency
  - up to programmer to select best overall algorithm
  - big-O savings are (often) more important than constant factors
    - but constant factors also matter

- Have difficulty overcoming “optimization blockers”
  - potential memory aliasing
  - potential procedure side-effects
Limitations of Optimizing Compilers

- **Operate under fundamental constraint**
  - Must not cause any change in program behavior
  - Often prevents it from making optimizations when would only affect behavior under pathological conditions.

- **Behavior that may be obvious to the programmer can be obfuscated by languages and coding styles**
  - e.g., Data ranges may be more limited than variable types suggest

- **Most analysis is performed only within procedures**
  - Whole-program analysis is too expensive in most cases

- **Most analysis is based only on static information**
  - Compiler has difficulty anticipating run-time inputs

- **When in doubt, the compiler must be conservative**
Generally Useful Optimizations

- Optimizations that you or the compiler should do regardless of processor / compiler

- Code Motion
  - Reduce frequency with which computation performed
    - If it will always produce same result
    - Especially moving code out of loop

```c
void set_row(double *a, double *b, long i, long n)
{
    long j;
    for (j = 0; j < n; j++)
        a[n*i+j] = b[j];
}
```

```c
long j;
int ni = n*i;
for (j = 0; j < n; j++)
    a[ni+j] = b[j];
```
Compiler-Generated Code Motion

```c
void set_row(double *a, double *b, long i, long n) {
    long j;
    for (j = 0; j < n; j++)
        a[n*i+j] = b[j];
}
```

```c
long j;
long ni = n*i;
double *rowp = a+ni;
for (j = 0; j < n; j++)
    *rowp++ = b[j];
```

Where are the FP operations?

```
set_row:
    testq %rcx, %rcx
    jle .L4
    movq %rcx, %rax
    imulq %rdx, %rax
    leaq (%rdi,%rax,8), %rdx
    movl $0, %r8d
    .L3:
        movq (%rsi,%r8,8), %rax
        movq %rax, (%rdx)
        addq $1, %r8
        addq $8, %rdx
        cmpq %r8, %rcx
        jg .L3
    .L4:
        rep ; ret
```

Where are the FP operations?
Reduction in Strength

- Replace costly operation with simpler one
- Shift, add instead of multiply or divide
  \[ 16 \times x \quad \rightarrow \quad x \ll 4 \]
  - Utility machine dependent
  - Depends on cost of multiply or divide instruction
    - On Intel Nehalem, integer multiply requires 3 CPU cycles
- Recognize sequence of products

```c
int ni = 0;
for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++)
        a[ni + j] = b[j];
    ni += n;
}
```
Share Common Subexpressions

- Reuse portions of expressions
- Compilers often not very sophisticated in exploiting arithmetic properties

```c
/* Sum neighbors of i,j */
up = val[(i-1)*n + j ];
down = val[(i+1)*n + j ];
left = val[i*n + j-1];
right = val[i*n + j+1];
sum = up + down + left + right;
```

3 multiplications: i*n, (i-1)*n, (i+1)*n

```c
leaq   1(%rsi), %rax  # i+1
leaq   -1(%rsi), %r8  # i-1
imulq  %rcx, %rsi     # i*n
imulq  %rcx, %rax     # (i+1)*n
imulq  %rcx, %r8      # (i-1)*n
addq   %rdx, %rsi     # i*n+j
addq   %rdx, %rax     # (i+1)*n+j
addq   %rdx, %r8      # (i-1)*n+j
```

1 multiplication: i*n

```c
long inj = i*n + j;
up = val[inj - n];
down = val[inj + n];
left = val[inj - 1];
right = val[inj + 1];
sum = up + down + left + right;
```

```c
imulq  %rcx, %rsi     # i*n
addq   %rdx, %rsi     # i*n+j
movq   %rsi, %rax     # i*n+j
subq   %rcx, %rax     # i*n+j-n
leaq   (%rsi,%rcx), %rcx # i*n+j+n
```
Optimization Blocker #1: Procedure Calls

- Procedure to Convert String to Lower Case

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

- Extracted from 213 lab submissions, Fall, 1998
Lower Case Conversion Performance

- Time quadruples when double string length
- Quadratic performance
Convert Loop To Goto Form

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

- `strlen` executed every iteration
Calling Strlen

```c
/* My version of strlen */
size_t strlen(const char *s)
{
    size_t length = 0;
    while (*s != '\0') {
        s++;
        length++;
    }
    return length;
}
```

- **Strlen performance**
  - Only way to determine length of string is to scan its entire length, looking for null character.

- **Overall performance, string of length N**
  - N calls to strlen
  - Require times N, N-1, N-2, ..., 1
  - Overall O(N^2) performance
Improving Performance

- Move call to `strlen` outside of loop
- Since result does not change from one iteration to another
- Form of code motion
Lower Case Conversion Performance

- Time doubles when double string length
- Linear performance of lower2
Optimization Blocker: Procedure Calls

- **Why couldn’t compiler move strlen out of inner loop?**
  - Procedure may have side effects
    - Alters global state each time called
  - Function may not return same value for given arguments
    - Depends on other parts of global state
    - Procedure `lower` could interact with `strlen`

- **Warning:**
  - Compiler treats procedure call as a black box
  - Weak optimizations near them

- **Remedies:**
  - Use of `inline` functions
    - GCC does this with `-O2`
    - See web aside ASM:OPT
  - Do your own code motion

```c
int lencnt = 0;
size_t strlen(const char *s)
{
    size_t length = 0;
    while (*s != '\0') {
        s++; length++;
    }
    lencnt += length;
    return length;
}
```
Memory Matters

/* Sum rows is of n X n matrix a and store 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] on every iteration
- Why couldn’t compiler optimize this away?
Memory Aliasing

/* Sum rows is of n X n matrix a and store 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);

- Code updates b[i] on every iteration
- Must consider possibility that these updates will affect program behavior
Removing Aliasing

/* 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;
    } 
}

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

- No need to store intermediate results
Optimization Blocker: Memory Aliasing

- **Aliasing**
  - Two different memory references specify single location
  - Easy to have happen in C
    - Since allowed to do address arithmetic
    - Direct access to storage structures
  - Get in habit of introducing local variables
    - Accumulating within loops
    - Your way of telling compiler not to check for aliasing
Exploiting Instruction-Level Parallelism

- Need general understanding of modern processor design
  - Hardware can execute multiple instructions in parallel

- Performance limited by data dependencies

- Simple transformations can have dramatic performance improvement
  - Compilers often cannot make these transformations
  - Lack of associativity and distributivity in floating-point arithmetic
Benchmark Example: Data Type for Vectors

```c
/* data structure for vectors */
typedef struct {
    int len;
    double *data;
} vec;

/* retrieve vector element and store at val */
double get_vec_element(*vec, idx, double *val) {
    if (idx < 0 || idx >= v->len)
        return 0;
    *val = v->data[idx];
    return 1;
}
```
Benchmark Computation

```c
void combine1(vec_ptr v, data_t *dest)
{
    long int i;
    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++) {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest OP val;
    }
}
```

Data Types

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

Operations

- Use different definitions of `OP` and `IDENT`
  - + / 0
  - * / 1

Compute sum or product of vector elements
Cycles Per Element (CPE)

- Convenient way to express performance of program that operates on vectors or lists
- Length = n
- In our case: CPE = cycles per OP
- T = CPE*n + Overhead
  - CPE is slope of line

![Graph showing two lines with slopes of 4.0 and 3.5](image)
Benchmark Performance

```c
void combine1(vec_ptr v, data_t *dest)
{
    long int i;
    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++) {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest OP val;
    }
}
```

Compute sum or product of vector elements

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Add</td>
<td>Mult</td>
</tr>
<tr>
<td></td>
<td>Add</td>
<td>Mult</td>
</tr>
<tr>
<td>Operation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Combine1 unoptimized</td>
<td>29.0</td>
<td>29.2</td>
</tr>
<tr>
<td></td>
<td>27.4</td>
<td>27.9</td>
</tr>
<tr>
<td>Combine1 –O1</td>
<td>12.0</td>
<td>12.0</td>
</tr>
<tr>
<td></td>
<td>12.0</td>
<td>13.0</td>
</tr>
</tbody>
</table>
Basic Optimizations

```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;
}
```

- Move `vec_length` out of loop
- Avoid bounds check on each cycle
- Accumulate in temporary
Effect of Basic Optimizations

```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;
}
```

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer Operation</th>
<th>Double FP Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Combine1 –O1</td>
<td>12.0 Add</td>
<td>12.0 Add</td>
</tr>
<tr>
<td></td>
<td>12.0 Mult</td>
<td>12.0 Add</td>
</tr>
<tr>
<td></td>
<td></td>
<td>13.0 Mult</td>
</tr>
<tr>
<td>Combine4</td>
<td>2.0 Add</td>
<td>3.0 Add</td>
</tr>
<tr>
<td></td>
<td>3.0 Mult</td>
<td>3.0 Add</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5.0 Mult</td>
</tr>
</tbody>
</table>

- Eliminates sources of overhead in loop
Modern CPU Design

**Instruction Control**
- **Retirement Unit**
  - **Register File**
  - **Fetch Control**
  - **Instruction Decode**
- **Instruction Cache**
- **Address**
- **Instructions**
- **Operations**

**Execution**
- **Prediction OK?**
- **Register Updates**
- **Integer/Branch**
  - **General Integer**
  - **FP Add**
  - **FP Mul/Div**
  - **Load**
  - **Store**
  - **Functional Units**
  - **Operation Results**
  - **Addr.**
  - **Data**
- **Data Cache**
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
Nehalem 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 Multiply
  - 1 FP Add

- **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>4</td>
<td>1</td>
</tr>
<tr>
<td>Integer Multiply</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td><strong>Integer/Long Divide</strong></td>
<td>11--21</td>
<td>11--21</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><strong>Single/Double FP Divide</strong></td>
<td>10--23</td>
<td>10--23</td>
</tr>
</tbody>
</table>
x86-64 Compilation of Combine4

- Inner Loop (Case: Integer Multiply)

```assembly
.L519:
imull (%rax,%rdx,4), %ecx # t = t * d[i]
addq $1, %rdx # i++
cmpq %rdx, %rbp # Compare length:i
jg .L519 # If >, goto Loop
```

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td>Add</td>
<td>Mult</td>
</tr>
<tr>
<td>Combine4</td>
<td>2.0</td>
<td>3.0</td>
</tr>
<tr>
<td>Latency Bound</td>
<td>1.0</td>
<td>3.0</td>
</tr>
</tbody>
</table>
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**
  - Performance: determined by latency of OP
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;
}

- Perform 2x more useful work per iteration
Effect of Loop Unrolling

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Operation</strong></td>
<td><strong>Add</strong></td>
<td><strong>Mult</strong></td>
</tr>
<tr>
<td>Combine4</td>
<td>2.0</td>
<td>3.0</td>
</tr>
<tr>
<td>Unroll 2x</td>
<td>2.0</td>
<td>1.5</td>
</tr>
<tr>
<td>Latency Bound</td>
<td>1.0</td>
<td>3.0</td>
</tr>
</tbody>
</table>

- Helps integer multiply
  - below latency bound
  - Compiler does clever optimization

- Others don’t improve. Why?
  - Still sequential dependency

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

```c
void unroll2aa_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;
}
```

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

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th></th>
<th>Double FP</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Add</td>
<td>Mult</td>
<td>Add</td>
<td>Mult</td>
</tr>
<tr>
<td>Operation</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Combine4</td>
<td>2.0</td>
<td>3.0</td>
<td>3.0</td>
<td>5.0</td>
</tr>
<tr>
<td>Unroll 2x</td>
<td>2.0</td>
<td>1.5</td>
<td>3.0</td>
<td>5.0</td>
</tr>
<tr>
<td>Unroll 2x, reassociate</td>
<td>2.0</td>
<td>1.5</td>
<td>1.5</td>
<td>3.0</td>
</tr>
<tr>
<td>Latency Bound</td>
<td>1.0</td>
<td>3.0</td>
<td>3.0</td>
<td>5.0</td>
</tr>
<tr>
<td>Throughput Bound</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

**Nearly 2x speedup for Int *, FP +, FP ***

- Reason: Breaks sequential dependency

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

- Why is that? (next slide)
Reassociated Computation

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

- **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:
    - \(CPE = D/2\)
  - Measured CPE slightly worse for FP mult
Loop Unrolling with Separate Accumulators

```c
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>Integer</th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Operation</strong></td>
<td>Add</td>
<td>Mult</td>
</tr>
<tr>
<td>Combine4</td>
<td>2.0</td>
<td>3.0</td>
</tr>
<tr>
<td>Unroll 2x</td>
<td>2.0</td>
<td>1.5</td>
</tr>
<tr>
<td>Unroll 2x, reassociate</td>
<td>2.0</td>
<td>1.5</td>
</tr>
<tr>
<td>Unroll 2x Parallel 2x</td>
<td>1.5</td>
<td>1.5</td>
</tr>
<tr>
<td>Latency Bound</td>
<td>1.0</td>
<td>3.0</td>
</tr>
<tr>
<td>Throughput Bound</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

- **2x speedup (over unroll2) for Int *, FP +, FP ***
  - Breaks sequential dependency in a “cleaner,” more obvious way

\[
\begin{align*}
  x0 = x0 \text{ OP } d[i]; \\
  x1 = x1 \text{ OP } d[i+1];
\end{align*}
\]
**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!

---

\[
\begin{align*}
x_0 &= x_0 \text{ OP } d[i]; \\
x_1 &= x_1 \text{ OP } d[i+1];
\end{align*}
\]
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: Double *

## Case

- Intel Nehelam (Shark machines)
- Double FP Multiplication
- Latency bound: 5.00. Throughput bound: 1.00

<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>5.00</td>
</tr>
<tr>
<td>2</td>
<td>2.50</td>
</tr>
<tr>
<td>3</td>
<td>1.67</td>
</tr>
<tr>
<td>4</td>
<td>1.25</td>
</tr>
<tr>
<td>6</td>
<td>1.00</td>
</tr>
<tr>
<td>8</td>
<td>1.02</td>
</tr>
<tr>
<td>10</td>
<td>1.01</td>
</tr>
<tr>
<td>12</td>
<td>1.00</td>
</tr>
</tbody>
</table>
Unrolling & Accumulating: Int +

- **Case**
  - Intel Nehalem (Shark machines)
  - Integer addition
  - Latency bound: 1.00. Throughput bound: 1.00

<table>
<thead>
<tr>
<th>FP *</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>K</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2.00</td>
<td>2.00</td>
<td>1.00</td>
<td>1.01</td>
<td>1.02</td>
<td>1.03</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1.50</td>
<td>1.26</td>
<td>1.03</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1.00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1.00</td>
<td>1.24</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1.00</td>
<td>1.02</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1.03</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1.01</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>1.09</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Achievable Performance

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Add</td>
<td>Mult</td>
</tr>
<tr>
<td>Operation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scalar Optimum</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Latency Bound</td>
<td>1.00</td>
<td>3.00</td>
</tr>
<tr>
<td>Throughput Bound</td>
<td>1.00</td>
<td>1.00</td>
</tr>
</tbody>
</table>

- Limited only by throughput of functional units
- Up to 29X improvement over original, unoptimized code
Using Vector Instructions

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th></th>
<th>Double FP</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Add</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Operation</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scalar Optimum</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Vector Optimum</td>
<td>0.25</td>
<td>0.53</td>
<td>0.53</td>
<td>0.57</td>
</tr>
<tr>
<td>Latency Bound</td>
<td>1.00</td>
<td>3.00</td>
<td>3.00</td>
<td>5.00</td>
</tr>
<tr>
<td>Throughput Bound</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Vec Throughput</td>
<td>0.25</td>
<td>0.50</td>
<td>0.50</td>
<td>0.50</td>
</tr>
<tr>
<td>Bound</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Make use of SSE Instructions**
  - Parallel operations on multiple data elements
  - See Web Aside OPT:SIMD on CS:APP web page
What About Branches?

**Challenge**

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

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

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

```
8048a25:
```

Executing

How to continue?
Modern CPU Design

**Instruction Control**

- **Retirement Unit**
  - **Register File**
- **Fetch Control**
- **Instruction Decode**
- **Instruction Cache**

**Execution**

- **Functional Units**
  - **Integer/Branch**
  - **General Integer**
  - **FP Add**
  - **FP Mult/Div**
  - **Load**
  - **Store**

**Operation Results**

- **Addr.**
- **Data**

**Register Updates**

- **Prediction OK?**
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

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>80488b1:</td>
<td>movl (%ecx,%edx,4),%eax</td>
<td>80488b1</td>
<td></td>
</tr>
<tr>
<td>80488b4:</td>
<td>addl %eax,(%edi)</td>
<td>80488b4</td>
<td></td>
</tr>
<tr>
<td>80488b6:</td>
<td>incl %edx</td>
<td>80488b6</td>
<td></td>
</tr>
<tr>
<td>80488b7:</td>
<td>cmpl %esi,%edx</td>
<td>80488b7</td>
<td>i = 98</td>
</tr>
<tr>
<td>80488b9:</td>
<td>jl 80488b1</td>
<td>80488b9</td>
<td></td>
</tr>
</tbody>
</table>

Assume vector length = 100

- **Predict Taken (OK)**
  - i = 98

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>80488b1:</td>
<td>movl (%ecx,%edx,4),%eax</td>
<td>80488b1</td>
<td></td>
</tr>
<tr>
<td>80488b4:</td>
<td>addl %eax,(%edi)</td>
<td>80488b4</td>
<td></td>
</tr>
<tr>
<td>80488b6:</td>
<td>incl %edx</td>
<td>80488b6</td>
<td></td>
</tr>
<tr>
<td>80488b7:</td>
<td>cmpl %esi,%edx</td>
<td>80488b7</td>
<td>i = 99</td>
</tr>
<tr>
<td>80488b9:</td>
<td>jl 80488b1</td>
<td>80488b9</td>
<td></td>
</tr>
</tbody>
</table>

- **Predict Taken (Oops)**
  - i = 99

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>80488b1:</td>
<td>movl (%ecx,%edx,4),%eax</td>
<td>80488b1</td>
<td></td>
</tr>
<tr>
<td>80488b4:</td>
<td>addl %eax,(%edi)</td>
<td>80488b4</td>
<td></td>
</tr>
<tr>
<td>80488b6:</td>
<td>incl %edx</td>
<td>80488b6</td>
<td></td>
</tr>
<tr>
<td>80488b7:</td>
<td>cmpl %esi,%edx</td>
<td>80488b7</td>
<td>i = 100</td>
</tr>
<tr>
<td>80488b9:</td>
<td>jl 80488b1</td>
<td>80488b9</td>
<td></td>
</tr>
</tbody>
</table>

- **Read invalid location**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>80488b1:</td>
<td>movl (%ecx,%edx,4),%eax</td>
<td>80488b1</td>
<td></td>
</tr>
<tr>
<td>80488b4:</td>
<td>addl %eax,(%edi)</td>
<td>80488b4</td>
<td></td>
</tr>
<tr>
<td>80488b6:</td>
<td>incl %edx</td>
<td>80488b6</td>
<td></td>
</tr>
<tr>
<td>80488b7:</td>
<td>cmpl %esi,%edx</td>
<td>80488b7</td>
<td>i = 101</td>
</tr>
<tr>
<td>80488b9:</td>
<td>jl 80488b1</td>
<td>80488b9</td>
<td></td>
</tr>
</tbody>
</table>

- **Executed**

- **Fetched**
Branch Misprediction Invalidation

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

Assume vector length = 100

Predict Taken (OK)

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

Predict Taken (Oops)

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

Invalidate

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

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

i = 98

i = 99

i = 100

i = 101
Branch Misprediction Recovery

```
80488b1: movl (%ecx,%edx,4),%eax
80488b4: addl %eax,(%edi)
80488b6: incl %edx
80488b7: cmpl %esi,%edx
80488b9: jl 80488b1
80488bb: leal 0xfffffffffe8(%ebp),%esp
80488be: popl %ebx
80488bf: popl %esi
80488c0: popl %edi
```

- **Performance Cost**
  - Multiple clock cycles on modern processor
  - Can be a major performance limiter

\[ i = 99 \]

Definitely not taken
Effect of Branch Prediction

- **Loops**
  - Typically, only miss when hit loop end

- **Checking code**
  - Reliably predicts that error won’t occur

```c
void combine4b(vec_ptr v, data_t *dest)
{
    long int i;
    long int length = vec_length(v);
    data_t acc = IDENT;
    for (i = 0; i < length; i++) {
        if (i >= 0 && i < v->len) {
            acc = acc OP v->data[i];
        }
    }
    *dest = acc;
}
```

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add</td>
<td>2.0</td>
<td>3.0</td>
</tr>
<tr>
<td>Mult</td>
<td>3.0</td>
<td>5.0</td>
</tr>
<tr>
<td>Combine4</td>
<td>4.0</td>
<td>4.0</td>
</tr>
<tr>
<td>Combine4b</td>
<td>4.0</td>
<td>5.0</td>
</tr>
</tbody>
</table>
Getting High Performance

- 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
  - 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)