Performance Optimizations

Han, Hwansoo
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 modern processors + memory systems operate
  ▪ 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 that 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
  - Newer versions of GCC do interprocedural analysis within individual files
    - But, not between code in different files

- **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];
```
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];
}

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

❖ Replace costly operation with simpler one
  ▪ Shift, add instead of multiply or divide
    16*x  -->  x << 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
for (i = 0; i < n; i++) {
    int ni = n*i;
    for (j = 0; j < n; j++)
        a[ni + j] = b[j];
}
```

```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
  ▪ GCC will do this with –O1

/* 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

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

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;
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
Procedure to Convert String to Lower Case

```c
void lower1(char *s)
{
    size_t i;
    for (i = 0; i < strlen(s); i++)
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
}
```
Lower Case Conversion Performance

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

- `strlen` executed every iteration

```c
void lower1_(char *s)
{
    size_t 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:
}
```
Calling Strlen

- **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

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

❖ **Move call to strlen outside of loop**
  - Since result does not change from one iteration to another
  - Form of code motion

```c
void lower2(char *s)
{
    size_t i;
    size_t len = strlen(s);
    for (i = 0; i < len; i++)
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
}
```
Lower Case Conversion Performance

- Time doubles when double string length
  - Linear performance of lower2
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 the 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 `-O1`
    - Within single file
- Do your own code motion

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

- **Code updates** $b[i]$ on every iteration (inner loop)
  - Why couldn’t compiler optimize this away?

```c
/* 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
.L4:
    movsd (%rsi,%rax,8), %xmm0    # FP load
    addsd (%rdi), %xmm0           # FP add
    movsd %xmm0, (%rsi,%rax,8)    # FP store
    addq $8, %rdi
    cmpq %rcx, %rdi
    jne .L4
```
Memory Aliasing

- **Code updates** $b[i]$ **on every iteration (inner loop)**
  - Must consider possibility that these updates will affect program behavior

```c
/* 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};

double *B = A+3;

```c
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

- No need to store intermediate results

```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;
    }
}
```

```assembly
# sum_rows2 inner loop
.L10:
    addsd (%rdi), %xmm0    # FP load + add
    addq $8, %rdi
    cmpq %rax, %rdi
    jne .L10
```
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 yield dramatic performance improvement**
  - Compilers often cannot make these transformations
  - Lack of associativity and distributivity in floating-point arithmetic
Benchmark Example: Data Type for Vectors

❖ Data Types

▪ Use different declarations for data_t
  ▪ int
  ▪ long
  ▪ float
  ▪ double

/* data structure for vectors */
typedef struct {
  size_t len;
  data_t *data;
} vec;

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

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

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

```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
Cycles Per Element (CPE)

- Convenient way to express performance of program that operates on vectors or lists

\[ T = \text{CPE} \times n + \text{Overhead} \]

- CPE is slope of line
- \( n \) is Length of vector

![Graph showing Cycles vs. Elements](image-url)
Benchmark Performance

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;
    }
}

<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>Combine1 unoptimized</td>
<td>22.68</td>
<td>20.02</td>
</tr>
<tr>
<td>Combine1 –O1</td>
<td>10.12</td>
<td>10.12</td>
</tr>
</tbody>
</table>

Compute sum (or product) of vector elements
Basic Optimizations

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

```c
void combine4(vec_ptr v, data_t *dest) {
    long i;
    long 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;
}
```
Effect of Basic Optimizations

- Eliminates sources of overhead in loop

```c
void combine4(vec_ptr v, data_t *dest) {
    long i;
    long 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>Operation</th>
<th>Integer</th>
<th></th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Add</td>
<td>Mult</td>
<td></td>
<td>Add</td>
</tr>
<tr>
<td>Combine1</td>
<td>–O1</td>
<td>10.12</td>
<td>10.12</td>
<td>10.17</td>
</tr>
<tr>
<td>Combine4</td>
<td></td>
<td>1.27</td>
<td>3.01</td>
<td>3.01</td>
</tr>
</tbody>
</table>
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 modern CPUs are superscalar

Intel: since Pentium (1993)
Pipelined Functional Units

- **Divide computation into stages**
  - Pass partial computations from stage to stage
  - Stage i can start on new computation once values passed to i+1
    - E.g., complete 3 multiplications in 7 cycles, even though each requires 3 cycles

```c
long mult_eg(long a, long b, long c) {
    long p1 = a*b;
    long p2 = a*c;
    long p3 = p1 * p2;
    return p3;
}
```

<table>
<thead>
<tr>
<th>Time</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stage 1</td>
<td>[a*b]₁</td>
<td>[a*c]₁</td>
<td></td>
<td>[p1*p2]₁</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stage 2</td>
<td>[a*b]₂</td>
<td>[a*c]₂</td>
<td></td>
<td>[p1*p2]₂</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stage 3</td>
<td>[a*b]₃</td>
<td>[a*c]₃</td>
<td></td>
<td></td>
<td>[p1*p2]₃</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Haswell CPU (Jun. 2013)

- **8 Ports to Functional Units**
  - Multiple instructions can execute in parallel
    - 2 load, with address computation
    - 1 store, with address computation
    - 4 integer
    - 2 FP multiply
    - 1 FP add
    - 1 FP divide
  - Some instructions take > 1 cycle, but can be pipelined

<table>
<thead>
<tr>
<th>Operation</th>
<th>Integer</th>
<th>FP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Latency</td>
<td>Issue</td>
</tr>
<tr>
<td>Add</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Multiply</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Divide</td>
<td>3-30</td>
<td>3-30</td>
</tr>
</tbody>
</table>

Latency: total number of cycles required to perform the actual operations
Issue time: the minimum number of cycles between two independent operations
Capacity: the maximum number of operations that can be issued simultaneously
Inner Loop (Case: Integer Multiply)

.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>1.27</td>
<td>3.01</td>
</tr>
<tr>
<td>Latency Bound</td>
<td>1.00</td>
<td>3.00</td>
</tr>
</tbody>
</table>
Combine4 = Serial Computation \((OP = \ast)\)

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

- **Sequential dependence**
  - Performance: determined by latency of OP
Loop Unrolling (2x1)

- Perform 2x more useful work per iteration

```c
void unroll2x1_combine(vec_ptr v, data_t *dest) {
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    long 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 add**
  ▪ Achieves latency bound

❖ **Others don’t improve. *Why?***
  ▪ Still sequential dependency

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

<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>1.27</td>
<td>3.01</td>
</tr>
<tr>
<td>Unroll 2x1</td>
<td>1.01</td>
<td>3.01</td>
</tr>
<tr>
<td>Latency Bound</td>
<td>1.00</td>
<td>3.00</td>
</tr>
</tbody>
</table>
Loop Unrolling with Reassociation (2x1a)

❖ Can this change the result of the computation?
❖ Yes, for FP.  \textit{Why?}

```c
void unroll2x1a_combine(vec_ptr v, data_t *dest) {
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    long 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;
}
```

Compare to before

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

- Nearly 2x speedup for Int *, FP +, FP *
  - Reason: Breaks sequential dependency
  
  \[ x = x \text{ OP } (d[i] \text{ OP } d[i+1]) \]

- Why? (next slide)
Reassociated Computation

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

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

❖ **Overall Performance**
  - N elements, D cycles latency/op
  - \((N/2+1)\times D\) cycles: \(\text{CPE} = D/2\)
Loop Unrolling: Separate Accumulators (2x2)

- Different form of reassociation

```c
void unroll2x2_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length - 1;
    data_t *d = get_vec_start(v);
    data_t x0 = IDENT;
    data_t x1 = IDENT;
    long 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;
}
```
Effect of Separate Accumulators

- **Int +** makes use of two load units
  
  \[
  x_0 = x_0 \text{ OP } d[i]; \\
  x_1 = x_1 \text{ OP } d[i+1];
  \]

- **2x speedup (over unroll2x1) for Int *, FP +, FP ***

<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>Combine4</td>
<td>1.27</td>
<td>3.01</td>
</tr>
<tr>
<td>Unroll 2x1</td>
<td>1.01</td>
<td>3.01</td>
</tr>
<tr>
<td>Unroll 2x1a</td>
<td>1.01</td>
<td>1.51</td>
</tr>
<tr>
<td>Unroll 2x2</td>
<td>0.81</td>
<td>1.51</td>
</tr>
<tr>
<td>Latency Bound</td>
<td>1.00</td>
<td>3.00</td>
</tr>
<tr>
<td>Throughput Bound</td>
<td>0.50</td>
<td>1.00</td>
</tr>
</tbody>
</table>
Separate Accumulators

What changed:
- Two independent “streams” of operations

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

What Now?
Unrolling & Accumulating

❖ **Idea**
  - Can unroll to any degree $L$
  - Can accumulate $K$ results in parallel

❖ **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 Haswell
- Double FP Multiplication
- Latency bound: 5.00  Throughput bound: 0.50

<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>5.01</td>
</tr>
<tr>
<td>2</td>
<td>2.51</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: Int +

- **Case**
  - Intel Haswell
  - Integer addition
  - Latency bound: 1.00  Throughput bound: 0.50

<table>
<thead>
<tr>
<th>Int +</th>
<th>Unrolling Factor L</th>
</tr>
</thead>
<tbody>
<tr>
<td>K</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1.27</td>
</tr>
<tr>
<td>2</td>
<td>0.81</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>
Achievable Performance

- Limited only by throughput of functional units
- Up to 42X improvement over original, unoptimized code

<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>Best</td>
<td>0.54</td>
<td>1.01</td>
</tr>
<tr>
<td>Latency Bound</td>
<td>1.00</td>
<td>3.00</td>
</tr>
<tr>
<td>Throughput Bound</td>
<td>0.50</td>
<td>1.00</td>
</tr>
</tbody>
</table>
**Programming with AVX2**

- **YMM Registers: 16 total, each 32 bytes (256 bits)**
  - 32 single-byte integers
  - 16 16-bit integers
  - 8 32-bit integers
  - 8 single-precision floats
  - 4 double-precision floats
  - 1 single-precision float
  - 1 double-precision float
SIMD Operations

- SIMD Operations: Single Precision
  \[ \text{vaddsd} \ %\text{ymm0}, \ %\text{ymm1}, \ %\text{ymm1} \]

- SIMD Operations: Double Precision
  \[ \text{vaddpd} \ %\text{ymm0}, \ %\text{ymm1}, \ %\text{ymm1} \]
Using Vector Instructions

- Make use of AVX Instructions
  - Parallel operations on multiple data elements
  - See Web Aside OPT:SIMD on CS:APP web page

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th></th>
<th></th>
<th>Double FP</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Operation</td>
<td>Add</td>
<td>Mult</td>
<td></td>
<td>Add</td>
<td>Mult</td>
</tr>
<tr>
<td>Scalar Best</td>
<td>0.54</td>
<td>1.01</td>
<td></td>
<td>1.01</td>
<td>0.52</td>
</tr>
<tr>
<td>Vector Best</td>
<td>0.06</td>
<td>0.24</td>
<td></td>
<td>0.25</td>
<td>0.16</td>
</tr>
<tr>
<td>Latency Bound</td>
<td>0.50</td>
<td>3.00</td>
<td></td>
<td>3.00</td>
<td>5.00</td>
</tr>
<tr>
<td>Throughput Bound</td>
<td>0.50</td>
<td>1.00</td>
<td></td>
<td>1.00</td>
<td>0.50</td>
</tr>
<tr>
<td>Vec Throughput Bound</td>
<td>0.06</td>
<td>0.12</td>
<td></td>
<td>0.25</td>
<td>0.12</td>
</tr>
</tbody>
</table>
What About Branches?

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

```assembly
404663:  mov   $0x0,%eax
404668:  cmp   (%rdi),%rsi
40466b:  jge   404685
40466d:  mov   0x8(%rdi),%rax
        ...  
404685:  repz retq
```

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

Instruction Control

- Instruction Cache
  - Instructions
  - Address

- Fetch Control
  - Operations
  - Instructions

- Instruction Decode
  - Operations

- Retirement Unit
  - Register File
  - Prediction OK?

- Branch
- Arith
- Arith
- Arith
- Load
- Store

Functional Units

Operation Results

- Addr.
- Data

Execution

Data Cache

Register Updates
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

```
404663:  mov     $0x0,%eax
404668:  cmp     (%rdi),%rsi
40466b:  jge     404685
40466d:  mov     0x8(%rdi),%rax

...  
404685:  repz retq
```
Branch Prediction

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

```
404663:  mov     $0x0,%eax
404668:  cmp     (%rdi),%rsi
40466b:  jge     404685
40466d:  mov     0x8(%rdi),%rax

... 

404685:  repz    retq
```
Branch Prediction Through Loop

Assume
vector length = 100

401029: vmulsd (%rdx),%xmm0,%xmm0
40102d: add $0x8,%rdx
401031: cmp %rax,%rdx
401034: jne 401029  

i = 98

Predict Taken (OK)

401029: vmulsd (%rdx),%xmm0,%xmm0
40102d: add $0x8,%rdx
401031: cmp %rax,%rdx
401034: jne 401029  

i = 99

Predict Taken (Oops)

401029: vmulsd (%rdx),%xmm0,%xmm0
40102d: add $0x8,%rdx
401031: cmp %rax,%rdx
401034: jne 401029  

i = 100

Read invalid location

401029: vmulsd (%rdx),%xmm0,%xmm0
40102d: add $0x8,%rdx
401031: cmp %rax,%rdx
401034: jne 401029  

i = 101

Executed

Fetched
**Branch Misprediction Invalidation**

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>401029</td>
<td><code>vmulsd (%rdx),%xmm0,%xmm0</code></td>
<td></td>
</tr>
<tr>
<td>40102d</td>
<td><code>add $0x8,%rdx</code></td>
<td></td>
</tr>
<tr>
<td>401031</td>
<td><code>cmp %rax,%rdx</code></td>
<td></td>
</tr>
<tr>
<td>401034</td>
<td><code>jne 401029</code></td>
<td></td>
</tr>
</tbody>
</table>

Assume

**vector length = 100**

**Predict Taken (OK)**

- \(i = 98\)
- \(i = 99\)

**Predict Taken (Oops)**

- \(i = 100\)

**Invalidate**

- \(i = 101\)
**Branch Misprediction Recovery**

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>401029:</td>
<td>vmulsd (%rdx),%xmm0,%xmm0</td>
<td></td>
</tr>
<tr>
<td>40102d:</td>
<td>add $0x8,%rdx</td>
<td></td>
</tr>
<tr>
<td>401031:</td>
<td>cmp %rax,%rdx</td>
<td>$i = 99 Definitely not taken</td>
</tr>
<tr>
<td>401034:</td>
<td>jne 401029</td>
<td></td>
</tr>
<tr>
<td>401036:</td>
<td>jmp 401040</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>401040:</td>
<td>vmovsd %xmm0,(%r12)</td>
<td></td>
</tr>
</tbody>
</table>

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

reload Pipeline
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)
Important Tools

❖ **Measurement**
- Accurately compute time taken by code
  - Most modern machines have built in cycle counters
  - Using them to get reliable measurements is tricky
  - Assembly code to access cycle counter, `time` command, `get_time_of_day()`
- Profile procedure calling frequencies
  - Unix tool gprof (need to compile with gcc `–pg`)
  - Intel Vtune, AMD Code Analyst

❖ **Observation**
- Generating assembly code
  - Can see what optimizations compiler can make
  - Understand capabilities/limitations of particular compiler
Code Profiling

❖ **Augment Executable Program with Timing Functions**
  - Computes (approximate) amount of time spent in each function
  - Time computation method
    - Periodically (~ every 10ms) interrupt program
    - Determine what function is currently executing
    - Increment its timer by interval (e.g., 10ms)
  - Also maintains counter for each function indicating number of times called

❖ **Using**
  
  $ gcc -O2 -pg prog.c -o prog
  $ ./prog
    - Executes in normal fashion, but also generates file `gmon.out`
  
  $ gprof prog
    - Generates profile information based on `gmon.out`
Profiling Results

❖ Call Statistics
  - Number of calls and cumulative time for each function

❖ Performance Limiter
  - Using inefficient sorting algorithm
  - Single call uses 87% of CPU time

<table>
<thead>
<tr>
<th>% cumulative</th>
<th>time</th>
<th>cumulative</th>
<th>self</th>
<th>self</th>
<th>total</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>% cumulative</td>
<td>seconds</td>
<td>seconds</td>
<td>calls</td>
<td>ms/call</td>
<td>ms/call</td>
<td>name</td>
</tr>
<tr>
<td>86.60</td>
<td>8.21</td>
<td>8.21</td>
<td>1</td>
<td>8210.00</td>
<td>8210.00</td>
<td>sort_words</td>
</tr>
<tr>
<td>5.80</td>
<td>8.76</td>
<td>0.55</td>
<td>946596</td>
<td>0.00</td>
<td>0.00</td>
<td>lower1</td>
</tr>
<tr>
<td>4.75</td>
<td>9.21</td>
<td>0.45</td>
<td>946596</td>
<td>0.00</td>
<td>0.00</td>
<td>find_ele_rec</td>
</tr>
<tr>
<td>1.27</td>
<td>9.33</td>
<td>0.12</td>
<td>946596</td>
<td>0.00</td>
<td>0.00</td>
<td>h_add</td>
</tr>
</tbody>
</table>
Profiling Observations

❖ **Benefits**
  - Helps identify performance bottlenecks
  - Especially useful when have complex system with many components

❖ **Limitations**
  - Only shows performance for data tested
    - Some optimizations may not show big gain, since the number of input data is too small.
    - Quadratic inefficiency could remain lurking in code
  - Timing mechanism fairly crude
    - Only works for programs that run for > 3 seconds