Program Optimization

CSci 2021: Machine Architecture and Organization
Lecture #33-35, April 18th-22nd, 2016

Your instructor: Stephen McCamant

Based on slides originally by:
Randy Bryant, 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 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
  - Except, possibly when program making use of nonstandard language
    features
  - 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

```
void set_row(double *a, double *b, long i, long n)
{
    long j;
    for (j = 0; j < n; j++)
        a[i+j] = b[j];
}
```
Compiler-Generated Code Motion (-O1)

```c
void set_row(double *a, double *b, long i, long n) {
    long j;
    long n1 = n*ni;
    # Test n
    for (j = 0; j < n; j++)
        *ropm = a[n1] = b[j];
}
```

Reduction in Strength

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

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 CMU 213 lab submissions, Fall, 1998

Share Common Subexpressions

- Reuse portions of expressions
- GCC will do this with -O1

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

- 3 multiplications: \(P_{n}\), \(P_{2n}\), \(P_{4n}\)
- 1 multiplication: \(P_{n}\)

Lower Case Conversion Performance

- Time quadruples when double string length
- Quadratic performance

```
```
Calling strlen

/* 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

```c
void lower(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

![Graph showing the performance of lower Case Conversion](image)

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 -O1
  - 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;
}
```

Exercise Break: Weird Pointers

- Can the following function ever return 12, and if so how?
  ```c
  int f(int *p1, int *p2, int *p3) {
    *p1 = 100;
    *p2 = 10;
    *p3 = 1;
    return *p1 + *p2 + *p3;
  }
  ```
- Yes, for instance:
  ```c
  int a, b;
  f(&a, &b, &a);
  ```

- [https://chimein.cla.umn.edu/course/view/2021](https://chimein.cla.umn.edu/course/view/2021)

Memory Matters

- Code updates `b[i]` on every iteration
- Why couldn't compiler optimize this away?

```c
# sum rows of n X n matrix a and store in vector b */
void sum_rows(double *, double *, long n) {
    long i, j;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
}
```
Memory Aliasing

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

```c
/* Sum rows 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];
    }
}
```

```c
double A[9] = { 0, 1, 2, 4, 8, 16, 32, 64, 128};
sum_rows1(A, B, 3);
```

Removing Aliasing
No need to store intermediate results

```c
# sum_rows2 inner loop
.L10:
addsd (%rdi), %xmm0
# FP load + add
addq $8, %rdi
cmpq %rax, %rdi
jne .L10
```

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

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

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

Benchmark Computation
Compute sum or product of vector elements

```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;
    }
}
```

Exploiting Instruction-Level Parallelism

```
void combine2(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;
    }
}
```
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
- \( T = \text{CPE}\cdot n + \text{Overhead} \)
  - CPE is slope of line

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;
    }
}
```

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer Add</th>
<th>Integer Mult</th>
<th>Double FP Add</th>
<th>Double FP Mult</th>
</tr>
</thead>
<tbody>
<tr>
<td>Combine1 unoptimized</td>
<td>22.68</td>
<td>20.02</td>
<td>19.98</td>
<td>20.18</td>
</tr>
<tr>
<td>Combine1 –O1</td>
<td>10.12</td>
<td>10.12</td>
<td>10.17</td>
<td>11.14</td>
</tr>
</tbody>
</table>

Basic Optimizations

```c
void combine4(vec_ptr v, data_t *dest)
{
    long i;
    long length = vec_length(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)
{
    long i;
    long length = vec_length(v);
    data_t t = 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 Add</th>
<th>Integer Mult</th>
<th>Double FP Add</th>
<th>Double FP Mult</th>
</tr>
</thead>
<tbody>
<tr>
<td>Combine1 –O1</td>
<td>10.12</td>
<td>10.12</td>
<td>10.17</td>
<td>11.14</td>
</tr>
<tr>
<td>Combine4</td>
<td>1.27</td>
<td>3.01</td>
<td>3.01</td>
<td>5.01</td>
</tr>
</tbody>
</table>

Modern CPU Design

- **Instruction Control**
  - Fetch Control
  - Address Decode
  - Instruction Cache
- **Execution**
  - Functional Units
  - Branch, Arith, Load, Store, Functional Units
  - Operation Results
  - Branch, Arith, Load, Store, Functional Units
  - 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 modern CPUs are superscalar.
  - Intel: since Pentium (1993)
Pipelined Functional Units

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

- 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

### Haswell CPU
- 8 Total 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>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>Integer / Long Divide</td>
<td>3-30</td>
<td>3-30</td>
</tr>
<tr>
<td>Single / Double FP Multiply</td>
<td>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>3-15</td>
<td>3-15</td>
</tr>
</tbody>
</table>

### x86-64 Compilation of Combine4
- **Inner Loop (Case: Integer Multiply)**

```c
/* Loop: lsl1: (t = t + d[i]) * d[i+1];
addl $0, t, t;
mpq t, d[i], %reg
shl $1, t, t;*/
```

<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>

### Loop Unrolling (2x1)

```c
void unroll2a_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;
    /* Loop:*/
    for (i = 0; i < limit; i+=2) {
        x = x OP d[i] OP d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < limit; i++) {
        x = x OP d[i];
    }
    *dest = x;
}
```

- Perform 2x more useful work per iteration

### Effect of Loop Unrolling
- Helps integer add
  - Achieves latency bound
  - Others don’t improve. Why?
  - Still sequential dependency
Loop Unrolling with Reassociation (2x1a)

```
void unroll2a_combine(vec_ptr t, data_t *dest) {
    long length = vec_length(t);
    long limit = length-1;
    data_t *d = get_vec_start(t);
    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 < limit; i++) {
        x = x OP d[i];
    }
    *dest = x;
}
```

**Can this change the result of the computation?**

**Yes, for FP. Why?**

Reasssociated 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
  - \((N/2+1)\)D cycles:
    - CPE = D/2

---

Effect of Reassociation

<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>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>

- Nearly 2x speedup for Int *, FP +, FP *
  - Reason: Breaks sequential dependency

- Why is that? (next slide)

Loop Unrolling with Separate Accumulators (2x2)

```
void unroll2a_combine(vec_ptr t, data_t *dest) {
    long length = vec_length(t);
    long limit = length-1;
    data_t *d = get_vec_start(t);
    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 < limit; i++) {
        x0 = x0 OP d[i];
        x1 = x1 OP d[i+1];
    }
    *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>Operation</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>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>

- Int + makes use of two load units

  - \(x0 = x0 \ OP d[i];\)
  - \(x1 = x1 \ OP d[i+1];\)

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

---

Separate Accumulators

```
x0 = x0 \ OP d[i];
x1 = x1 \ OP d[i+1];
```

- **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!

- 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: Double *

Case
- Intel Haswell
- Double FP Multiplication
- Latency bound: 5.00. Throughput bound: 0.50

<table>
<thead>
<tr>
<th>FP ( \times )</th>
<th>Unrolling Factor L</th>
</tr>
</thead>
<tbody>
<tr>
<td>( K )</td>
<td>1</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>1.67</td>
</tr>
<tr>
<td>4</td>
<td>1.25</td>
</tr>
<tr>
<td>6</td>
<td>0.84</td>
</tr>
<tr>
<td>8</td>
<td>0.63</td>
</tr>
<tr>
<td>10</td>
<td>0.51</td>
</tr>
<tr>
<td>12</td>
<td>0.52</td>
</tr>
</tbody>
</table>

Accumulators

Achievable Performance

<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>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>

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

Programming with AVX2

YMM Registers
- 16 total, each 32 bytes
- 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

```
%ymm0
%ymm1
%ymm2
%ymm3
%ymm4
%ymm5
%ymm6
%ymm7
%ymm8
%ymm9
%ymm10
%ymm11
%ymm12
%ymm13
%ymm14
%ymm15
```

SIMD Operations: Double Precision

```
%ymm0
%ymm1
%ymm2
%ymm3
%ymm4
%ymm5
%ymm6
%ymm7
%ymm8
%ymm9
%ymm10
%ymm11
%ymm12
%ymm13
```
Using Vector Instructions

<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>Scalar Best</td>
<td>0.54</td>
<td>1.01</td>
</tr>
<tr>
<td>Vector Best</td>
<td>0.06</td>
<td>0.24</td>
</tr>
<tr>
<td>Latency Bound</td>
<td>0.50</td>
<td>3.00</td>
</tr>
<tr>
<td>Throughput Bound</td>
<td>0.50</td>
<td>1.00</td>
</tr>
<tr>
<td>Vec Throughput Bound</td>
<td>0.06</td>
<td>0.12</td>
</tr>
</tbody>
</table>

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

What About Larger Programs?

- If your program has just one loop, it’s obvious where to change to make it go faster
- In more complex programs, what to optimize is a key question
- When you first write a non-trivial program, it often has a single major algorithm performance problem
  - Textbook’s example: insertion sort
  - Last program I wrote: missed opportunity for dynamic programming
  - Fixing this problem is way more important than any other changes

Amdahl’s Law

- If you speed up one part of a system, the total benefit is limited by how much time that part took to start with
- Speedup $S$ is:
  $$ S = \frac{1}{1 - \alpha} + \frac{\alpha}{k} $$
  where the acceleration factor is $k$ and the original time fraction is $\alpha$.
- Limiting case: even if $k$ is effectively infinite, the upper limit on speedup is
  $$ S_{\infty} = \frac{1}{1 - \alpha} $$

Knowing What’s Slow: Profiling

- Profiling makes a version of a program that records how long it spends on different tasks
  - Use to find bottlenecks, at least in typical operation
- Common Linux tools:
  - gprof: GCC flag plus a tool to interpret output of the profiled program
    - Counts functions and randomly samples for time
    - Discussed in textbook’s 5.14.1
  - Valgrind callgrind/cachegrind
    - Counts everything, precise but slow
  - OProfile
    - Uses hardware performance counters, can be whole-system

What About Branches?

- Challenge
  - Instruction Control Unit must work well ahead of Execution Unit to generate enough operations to keep EU busy
  - When encounters conditional branch, cannot reliably determine where to continue fetching

Modern CPU Design

*Note: Diagram continues on the next page.*
**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

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

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

**Branch Misprediction**

- **Invalidation**
  - Assume vector length = 100
  - Execute Predict Taken (OK)
  - Fetch Predict Taken (Oops)
  - Read Invalid location
  - Invalidate

```assembly
401029: vmsl %edx, %xmm0, %xmm0
40102d: add $0x8, %rdx
401031: cmp %rax, %rdx
401034: jne 401029
401029: vmsl %edx, %xmm0, %xmm0
40102d: add $0x8, %rdx
401031: cmp %rax, %rdx
401034: jne 401029
401029: vmsl %edx, %xmm0, %xmm0
40102d: add $0x8, %rdx
```

**Branch Misprediction Recovery**

- Definitely not taken
- Reload Pipeline

```assembly
401029: vmsl %edx, %xmm0, %xmm0
40102d: add $0x8, %rdx
401031: cmp %rax, %rdx
401034: jne 401029
```

**Performance Cost**

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

**Branch Prediction Through Loop**

Assume vector length = 100

```
401029: vmsl %edx, %xmm0, %xmm0
40102d: add $0x8, %rdx
401031: cmp %rax, %rdx
401034: jne 401029
```

**Effect of Branch Prediction: Good News**

- **Loops**
  - Typically, only miss when hit loop end
- **Checking code**
  - Reliably predicts that error won't occur

```c
def combine4b(data_t *dest, int i)
{
    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;
}
```

**Method**

<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>Combine4b</td>
<td>4.0</td>
<td>4.0</td>
</tr>
</tbody>
</table>
Branch Prediction: Bad News

- Some program branches are inherently unpredictable
  - E.g., if based on input data, binary search tree, etc.
  - Indirect jumps are also often hard to predict
- These can be a major performance bottleneck
  - Misprediction penalty is typically 10-20 cycles
- Partial solution: write code to be compiled to conditional moves
  - For GCC: use math and ?: instead of if
  - Textbook gives min/max and mergesort examples

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)