Program Optimization

CSci 2021: Machine Architecture and Organization
November 12th-21st, 2018

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

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*n+j] = b[j];
}
```
Compiler-Generated Code Motion (-O1)

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

Share Common Subexpressions

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

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

Lower Case Conversion Performance

- Time quadruples when double string length
- Quadratic performance

![Graph showing lower case conversion performance]

Reduction in Strength

- Replace costly operation with simpler one
- Shift, add instead of multiply or divide
  ```c
  16*x  -->  x << 4
  ```
  - Utility machine dependent
  - Depends on cost of multiply or divide instruction
  - On Intel Nehalem, integer multiply requires 3 CPU cycles
- Most valuable when it can be done within a loop
  - "Induction variable" has value linear in loop execution count

```c
for (i = 0; i < n; i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
        s[i] = ('a' - 'A' + s[i]);
```

Optimization Blocker #1: Procedure Calls

- Procedure to Convert String to Lower Case
  ```c
  void lower(char *s)
  {
      size_t i;
      for (i = 0; i < strlen(s); i++)
          if (s[i] >= 'A' && s[i] <= 'Z')
              s[i] = ('a' - 'A' + s[i]);
  }
  ```

- Extracted from CMU 213 lab submissions, Fall, 1998
- Similar pattern seen in UMN HA1

Convert Loop To Goto Form

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

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');
}

- 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 -O1
    - Within single file
    - But doesn’t help here
    - Do your own code motion

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
  - A program I wrote recently: 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:
  \[ S = \frac{1}{(1 - \alpha) + \alpha/k} \]
  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

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;
}
```

```c
int a, b;
f(&a, &b, &a);
```

- Yes, for instance:
  - `https://chimein.cla.umn.edu/course/view/2021`

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 aliasing is impossible

Memory Matters

```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];
}
```

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

Removing Aliasing

```c
// Sum rows 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++)
        b[i] = 0;
    for (j = 0; j < n; j++)
        b[i] += a[i*n + j];
}
```

- No need to store intermediate results
Announcements break: HA4, etc.

- **HA4** is due tonight. Forum posts today give hints about:
  - Using your check function for tracking down problems
  - Can I just submit mm-implicit? (Short answer: yes)
  - Improving the throughput of realloc
  - Making sure to follow directions when submitting

- **Midterm 2** seemed hard
  - I’ll have more specifics after it has been graded, probably Wednesday
  - May be some adjustment accounting for both midterms

---

**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;

/* 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;
}
```

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

---

**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

**Benchmark Performance**

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Mult</th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td>Add</td>
<td>Add</td>
<td>Add</td>
</tr>
<tr>
<td>Combine1 unoptimized</td>
<td>22.68</td>
<td>20.02</td>
<td>19.98</td>
</tr>
<tr>
<td>Combine1 -O1</td>
<td>10.12</td>
<td>10.12</td>
<td>10.17</td>
</tr>
</tbody>
</table>
Basic Optimizations

```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;
}
```

- 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 *d = get_vec_start(v);
    data_t t = IDENT;
    for (i = 0; i < length; i++)
        t = t OP d[i];
    *dest = t;
}
```

- Eliminates sources of overhead in loop

Modern CPU Design

### Instruction Control

- Fetch Control
- Instruction Decode
- Instruction Cache
- Register Files
- Operation Results
- Instruction Start
- Address
- Data Cache

### Execution

- Functional Units
- Load
- Store
- Branch
- Arith
- Load
- Store

### Instruction Cache

- Prediction

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_add(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 1 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
- Load / Store
- Integer Multiply
- Integer/Long Divide
- Single/Double FPMultiply
- Single/Double FP Add
- Single/Double FP Divide

<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)

```plaintext
.L519:
  # Loop:
  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>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 = *)

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

- Sequential dependence
  - Performance: determined by latency of OP

Effect of Loop Unrolling

- Helps integer add
  - Achieves latency bound
- Others don’t improve. Why?
  - Still sequential dependency

Loop Unrolling (2x1)

```plaintext
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;
  /* 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

Announcements break: midterm stats

Midterm 2 (50-100 shown)

- Adjusted combined midterms (50-100 shown)
  - Adjustment is +6 to M2 or +3 to average

Midterm 2 solutions are now on the web site

Exercise Set 4 on caches is posted

- Due in class Wednesday 11/28 a week from today

More announcements

- Midterm 2 solutions are now on the web site
- Exercise Set 4 on caches is posted
  - Due in class Wednesday 11/28 a week from today
Loop Unrolling with Reassociation (2x1a)

```c
void unroll2aa_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;
}
```

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

- **Overall Performance**
  - N elements, D cycles latency/operation
  - \((N/2+1)*D\) cycles

- **Effect of Separate Accumulators**

<table>
<thead>
<tr>
<th>Method</th>
<th>Operation</th>
<th>Add</th>
<th>Mult</th>
<th>Add</th>
<th>Mult</th>
</tr>
</thead>
<tbody>
<tr>
<td>Combine4</td>
<td></td>
<td>1.27</td>
<td>3.01</td>
<td>3.01</td>
<td>5.01</td>
</tr>
<tr>
<td>Unroll 2x1</td>
<td></td>
<td>1.01</td>
<td>3.01</td>
<td>3.01</td>
<td>5.01</td>
</tr>
<tr>
<td>Unroll 2x1a</td>
<td></td>
<td>1.01</td>
<td>1.51</td>
<td>1.51</td>
<td>2.51</td>
</tr>
<tr>
<td>Unroll 2x2</td>
<td></td>
<td>0.81</td>
<td>1.51</td>
<td>1.51</td>
<td>2.51</td>
</tr>
<tr>
<td>Latency Bound</td>
<td></td>
<td>1.00</td>
<td>3.00</td>
<td>3.00</td>
<td>5.00</td>
</tr>
<tr>
<td>Throughput Bound</td>
<td></td>
<td>0.50</td>
<td>1.00</td>
<td>1.00</td>
<td>0.50</td>
</tr>
</tbody>
</table>

- **Int** + makes use of two load units
- **2x speedup (over unroll2)** for Int, FP, +, FP *

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 is that?** (next slide)

Separate Accumulators

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

- **Overall Performance**
  - N elements, D cycles latency/operation
  - 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>$K$</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>Unrolling Factor</td>
<td>5.01</td>
<td>5.01</td>
<td>5.01</td>
<td>5.01</td>
<td>5.01</td>
<td>5.01</td>
<td>5.01</td>
<td>5.01</td>
</tr>
<tr>
<td>Accumulators</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2.51</td>
<td>2.51</td>
<td>2.51</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1.67</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1.25</td>
<td>1.26</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0.84</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>0.63</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></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>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**

```
vaddsd %ymm0, %ymm1, %ymm1
```

```
%ymm0
%ymm1
```

**SIMD Operations: Double Precision**

```
vaddpd %ymm0, %ymm1, %ymm1
```

```
%ymm0
%ymm1
```
Using Vector Instructions

<table>
<thead>
<tr>
<th></th>
<th>Integer</th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method</td>
<td></td>
<td></td>
</tr>
<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>
</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 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

![Diagram of CPU design featuring Instruction Control, Fetch Control, Address, Instruction Cache, Register Updates, Prediction OK?, and Execution Views.]

- **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

```
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
```

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

- **Branch Prediction Through Loop**
  - Assume vector length = 100
  - Problem: Need to fetch valid elements
  - Solution: Use Branch Prediction

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

- **Branch Taken**
  - Predict Taken (OK)

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

- **Branch Not-Taken**
  - Predict Taken (OOPS)

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

- **Executed**
  - Read invalid location

- **Fetched**
  - i = 98

- **Predict Taken**
  - i = 99

- **Predict Taken**
  - i = 100

- **Predict Taken**
  - i = 101
Effect of Branch Prediction: Good News

- **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>Add</td>
<td>Add</td>
</tr>
<tr>
<td></td>
<td>Mult</td>
<td>Mult</td>
</tr>
<tr>
<td>Combine4</td>
<td>2.0</td>
<td>3.0</td>
</tr>
<tr>
<td></td>
<td>3.0</td>
<td>3.0</td>
</tr>
<tr>
<td>Combine4b</td>
<td>4.0</td>
<td>4.0</td>
</tr>
<tr>
<td></td>
<td>4.0</td>
<td>5.0</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)