Program Optimization

CSci 2021: Machine Architecture and Organization
Lecture #22-23, March 13th-16th, 2015
Your instructor: Stephen McCamant

Based on slides originally by:
Randy Bryant, Dave O’Hallaron, Antonia Zhai

These Slides

Overview

Generally Useful Optimizations
- Code motion/precomputation
- Strength reduction
- Sharing of common subexpressions
- Removing unnecessary procedure calls

Optimization Blockers
- Procedure calls
- Memory aliasing
- Optimizing In Larger Programs: Profiling
- 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
- (Loop Invariant) 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];
}
```
Compiler-Generated Code Motion

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

Where are the FP operations?

```
/* 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;
```

Share Common Subexpressions

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

```
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
```

Optimization Blocker #1: Procedure Calls

- Procedure to Convert String to Lower Case

```
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:
}
```

- Extracted from 213 lab submissions, Fall, 1998

Lower Case Conversion Performance

- Time quadruples when double string length
- Quadratic performance

```
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:
}
```

- strlcn executed every iteration
Calling strlen

- 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

```
/* 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

```
void lower(char *s) {
    int i;
    int 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

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

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

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

Exercise Break: Weird Pointers

- Can the following function ever return 12, and if so how?

```
int f(int *p1, int *p2, int *p3) {
    *p1 = 100;
    *p2 = 10;
    *p3 = 1;
    return *p1 + *p2 + *p3;
}
```

- Yes, for instance:

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

```
# sum_row is of n X n matrix a
# and store in vector b */
void sum_row1(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 rows is of n X n matrix a
and store in vector b */
# sum_row is of n X n matrix a
# and store in vector b */
void sum_row1(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];
    }
}
```

Memory Matters

- Code updates `b[1]` on every iteration
- Why couldn’t compiler optimize this away?
Memory Aliasing

- Code updates $b[i]$ on every iteration
- 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];
    }
}
```

Value of B:

<table>
<thead>
<tr>
<th>i</th>
<th>B[i]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>32</td>
</tr>
<tr>
<td>2</td>
<td>64</td>
</tr>
</tbody>
</table>

```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 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;
    }
}
```

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

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
- 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
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 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;
    }
}
```

Benchmark 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>Mul</td>
</tr>
<tr>
<td>Combine1 unoptimized</td>
<td>29.0</td>
<td>29.2</td>
</tr>
<tr>
<td>Combine1 -O1</td>
<td>12.0</td>
<td>12.0</td>
</tr>
</tbody>
</table>

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 */
int get_vec_element(*vec, idx, double *val)
{
    if (idx < 0 || idx >= v->len)
        return 0;
    *val = v->data[idx];
    return 1;
}
```

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

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td>Add</td>
<td>Mul</td>
</tr>
<tr>
<td>Combine1 unoptimized</td>
<td>29.0</td>
<td>29.2</td>
</tr>
<tr>
<td>Combine1 -O1</td>
<td>12.0</td>
<td>12.0</td>
</tr>
</tbody>
</table>

Basic Optimizations
- 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;
}
```

- Eliminates sources of overhead in 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>
</tbody>
</table>

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

x86-64 Compilation of Combine4

- **Inner Loop (Case: Integer Multiply)**

```assembly
.L519:
    # Loop:
    cmpq $0, %rdx
    jge .L519
    imull (%rax,%rdx,4), %ecx
    addq $1, %rdx
    cmpq %rdx, %rbp
    jg .L519
```

- **Computation (length=8)**

```
((((((1 * d[0]) * d[1]) * d[2]) * d[3])
  * d[4]) * d[5]) * d[6]) * d[7])
```

- **Sequential dependence**

Modern CPU Design

<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>
</tbody>
</table>

<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>
</tbody>
</table>

```

x86-64 Compilation of Combine4

- **Inner Loop (Case: Integer Multiply)**

```assembly
.L519:
    # Loop:
    cmpq $0, %rdx
    jge .L519
    imull (%rax,%rdx,4), %ecx  # t = t * d[i]
    addq $1, %rdx
    cmpq %rdx, %rbp
    jg .L519
```

- **Computation (length=8)**

```
((((((1 * d[0]) * d[1]) * d[2]) * d[3])
  * d[4]) * d[5]) * d[6]) * d[7])
```

- **Sequential dependence**

Nehalem CPU

- Multiple instructions can execute in parallel
  - 1 load, 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>Integer/Long Divide</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>Single/Double FP Divide</td>
<td>10–23</td>
<td>10–23</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])
```

- Performance: determined by latency of OP
Loop Unrolling

```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 *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>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>
</tbody>
</table>

- Helps integer multiply below latency bound
- Compiler does clever optimization
- Others don’t improve. Why?
  - Still sequential dependency
  ```c
  x = (x OP d[i]) OP d[i+1];
  ```

Loop Unrolling with Reassociation

```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 *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>Double FP</th>
</tr>
</thead>
<tbody>
<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>
</tbody>
</table>

- Nearly 2x speedup for Int *, FP +, FP *
- Reason: Breaks sequential dependency
  ```c
  x = x OP (d[i] OP d[i+1])
  ```
  - Why is that? [next slide]

Reassociated Computation

- 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>Operation</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

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!

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 Nehalem
- Double FP Multiplication
- Latency bound: 5.00. Throughput bound: 1.00

Achievable Performance

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

<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>Operation</td>
<td>Add</td>
<td>Mult</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>
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>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Vector Best</td>
<td>0.25</td>
<td>0.53</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>
<tr>
<td>Vec Throughput Bound</td>
<td>0.25</td>
<td>0.50</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

```c
048f8b3: movl $0x0, %ecx
048f8b8: xorl %edx, %edx
048f8be: cmpl %esi, %edx
048f8c0: jnl 048f2a5
048f8c4: movl %esi, %esi
048f8ca: imull (%eax, %edx, 4), %ecx
```

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

Modern CPU Design

- Instruction Control Unit
- Fetch Control
- Instruction Decode
- Instruction Cache
- Execution
- Data Cache
- Address Instructions
- Operations
- Instruction Results
- Fetch Control
- Register Cache
- Register Decode
- Address Instructions
- Functional Units
- Operation Results
- Data Cache
- Branch Prediction

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

```c
048f8b3: movl $0x0, %ecx
048f8b8: xorl %edx, %edx
048f8be: cmpl %esi, %edx
048f8c0: jnl 048f2a5
```

- Branch预测
  - Predict Taken (OK)
  - Predict Taken (Oops)

```c
048f8b1: movl (%eax, %edx, 4), %eax
048f8b4: addl %eax, (%edi)
048f8b6: incl %edx
048f8b7: cmpl %esi, %edx
```

- Branch Prediction Through Loop

```c
048f8b1: movl (%eax, %edx, 4), %eax
048f8b4: addl %eax, (%edi)
048f8b6: incl %edx
048f8b7: cmpl %esi, %edx
```

- Branch Not-Taken: Continue with next instruction in sequence
- Cannot resolve until outcome determined by branch/integer unit

```c
048f8b5: cmpl %edi, %edx
048f8b7: jl 048f2a0
048f8b9: movl 0xc(%ebp), %eax
048f8bf: leal 0xffffffe8(%ebp), %esp
048f8c2: movl %ecx, (%eax)
```

Branch Taken
### 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)**

```
i = 98
```

**Predict Taken (Oops)**

```
i = 99
```

**Invalidate**

```
i = 100
```

### 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 0xffffffe8(%ebp),%esp
80488be: popl %ebx
80488bf: popl %esi
80488c0: popl %edi
```

**Definitely not taken**

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

### 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>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

### Summary: 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)