Program Optimization

CSCI 2021: Machine Architecture and Organization

Pen-Chung Yew
Department Computer Science and Engineering
University of Minnesota

With Slides from Bryant, O’Hallaron and Zhai

Outline

• 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 is Important in Practice

• Constant factors matter too (e.g. \(O(n^2), O(n^3)\))!
• Even exact operation count does not predict performance
  • Easily see 10:1 performance gain depending on how code is written
  • Remember loop interchange in matrix multiplication?
• Must optimize at multiple levels: algorithm, data representations, procedures, and loops
  • Must understand system to optimize performance
    • How programs compiled and executed (optimization levels)
    • 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 — top level of memory hierarchy
  • Code selection and ordering (scheduling) — data/control dependences
  • Dead code elimination — reduce number of instructions executed
  • 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 optimizations even if it only affects pathological cases
    (often need programmers’ intervention)
• Behavior that may be obvious to the programmer can be obfuscated by languages and coding styles to a compiler
  • 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 (time and space) in most cases
• Most analysis is based only on static (i.e. compile time) information
  • Compiler has difficulty anticipating run-time inputs (e.g. pointers’ targets)
• When in doubt, the compiler must be conservative to guarantee correctness

Outline

• 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
Generally Useful Optimizations

- Optimizations that you or the compiler should do regardless of processor/compiler
  * Generally useful optimizations
  * Code motion
    - Reduce frequency with which computation performed
      * If it always produce same result
      * Especially, moving code out of loop

Outline

- 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

Share Common Subexpressions

- Reuse portions of expressions
  * Compilers not very sophisticated in exploiting arithmetic properties
  * Example: reduce 8 instructions to 5 instructions (3/8 = 37.5% reduction)

Optimization Blocker #1: Procedure Calls

- Procedure to Convert String to Lower Case

Compiler-Generated Code Motion

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

Reduction in Strength

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

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

```c
for (i = 0; i < n; i++)
    a[i*n] = b[i];
```
Calling Strlen

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

- Overall performance,
- String of length N
- N calls to strlen
- Require times N
- Overall O(N²) performance

Lower Case Conversion Performance

- Time quadruples when double string length
- Quadratic performance

```
void lower(const char *s) /* original code */
int i;
int len = strlen(s);
for (i = 0; i < strlen(s); i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
        s[i] -= ('A' - 'a');
return len;
```

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

Improving Performance

```
void lower2(char *s) /* "improved" code */
int i;
int len = strlen(s);
for (i = 0; i < strlen(s); i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
        s[i] -= ('A' - 'a');
```

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

Lower Case Conversion Performance

- Time doubles when double string length
- Linear performance of lower2
- Big-O change in performance - O(n²) to O(n)

Optimization Blocker: Procedure Calls

- Why couldn't compiler move strlen out of inner loop?
  - Procedure may have side effects
    - After global state each time called
  - Function may not return same value for given arguments
    - Depend on other parts of global state
  - Procedure lower2 could interact with strlen
- Warning:
  - Compiler treats function call as black box
  - Weak optimizations near them

Another Function Call Example

```
long func1()
```

What happened if f() is implemented as follows?

```
long func2()
```

Compiler has to consider side effect of function calls, hence, tend to be conservative
Function Inlining Comes to Rescue

```
long f1()
{ long counter = 0;
  long f1();
  return counter;
}
```

Inlining
```
long f1()
{ long t = counter++; /* +0 */
  t += counter++; /* +2 */
  t += counter++; /* +3 */
  return t;
}
```

Remedies:
• Inline functions
• GCC does this with -O2
• Do your own code motion

Memory References Matters

```
void sum_rows1(double *a, double *b, long n) {
  for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++)
      b[i] = a[i*n + j];
  }
}
```

Why couldn’t compiler optimize this away? Memory Aliasing

```
for (j = 0; j < n; j++)
  b[i] += a[i*n + j];
```

- Code updates (stores) b[1] in memory on every iteration
- Why couldn’t compiler optimize this away?

```
void sum_rows2(double *a, double *b, long n)
{ long i; 
  for (i = 0; i < n; i++)
    b[2] += a[i*n + j];
}
```

Incorrect “Optimized” Code

```
for (j = 0; j < n; j++)
  b[i] += a[i*n + j];
```

Optimization Blocker: Memory Aliasing

• Aliasing
  • Two different memory references specify same location
• Easy to have happened 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
  • Avoid using pointers if you can!

```
inline
```
long f1()
{ long t = counter++; /* +0 */
  t += counter++; /* +2 */
  t += counter++; /* +3 */
  return t;
}
```

Another Pointer Aliasing Example

```
void twiddle1(long *p, long *q, long *r)
{ long i;
  for (i = 0; i < n; i++)
    *p = *p + 2 * (*q);
}
```

Does twiddle2 work the same way as twiddle1?

```
void twiddle2(long *p, long *q, long *r)
{ long i;
  for (i = 0; i < n; i++)
    *p = *p + 2 * (*q);
}
```

What happened when a user call twiddle1 (*p, *p)?

```
*p = *p + 2 * (*q);
```

Value of b:

```
'p = 'p + 'p;
'p = 'p + 'p;
'p = 'p + 'p;
```

Compilers have to consider all possible usage of pointers, hence, tend to be conservative
Review: Optimization Blocker - Aliasing

- Aliasing
  - Two different memory references specify same location
  - Easy to have happen in C with pointers
    - 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
  - Avoid using pointers if you can

Outline

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

Exploiting Instruction-Level Parallelism (ILP)

- Need general understanding of modern processor design
  - Hardware can execute multiple instructions in parallel (i.e. instruction-level parallelism, ILP)
- 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

Performance Metrics: Cycles Per Element (CPE)

- Convenient way to express performance of program that operates on vectors or lists
- 2 GHz clock has $2 \times 10^9$ (billions) cycles per second
- In our case: $CPE = \text{cycles per OP}$
- $T = CPE \times n + \text{Overhead}$ (n is the length of vector)
  - $CPE$ is slope of line

Performance of Prefix Sum Function

- $T = CPE \times n + \text{Overhead}$ (n is the length of vector)
- Slope of the lines indicates the $CPE$
- What is the overhead?
**Benchmark Example: Data Type for Vectors**

```c
typedef struct{
    int len;
    double *data;
} vec_t;

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

// Return length of vector /*
long vec_length(vec_ptr v)
{
    return v->len;
}
```

**Reference machine:** Intel i7 Haswell

---

**Benchmark Performance**

```c
void combine1(vec_ptr v, data_t *dest)
{
    long 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>Operation</td>
<td>Add</td>
<td>Mult</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**

```c
#define IDENT 1
#define OP +
#define IDENT 0
#define OP *
```

**Effect of Basic Optimizations**

```c
void combine4(vec_ptr v, data_t *dest)
{
    long int i;
    long length = vec_length(v);
    data_t *d = get_vec_start(v);
    data_t acc = IDENT;
    for (i = 0; i < length; i++)
    {
        acc = acc OP d[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>Combine1 -O1</td>
<td>10.12</td>
<td>10.12</td>
</tr>
<tr>
<td>Combine4</td>
<td>1.27</td>
<td>3.01</td>
</tr>
</tbody>
</table>

* eliminates sources of overhead in loop

---

**Benchmark Computation (Reduction Ops)**

```c
void combine(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;
}
```

**Basic Optimizations**

```c
void combine(vec_ptr v, data_t *dest)
{
    long int i;
    long length = vec_length(v);
    data_t *d = get_vec_start(v);
    data_t acc = IDENT;
    for (i = 0; i < length; i++)
        acc = acc OP d[i];
    *
    
    *dest = acc;
}
```

**Move vec_length out of loop – code motion**

**Avoid bounds check on each cycle in get_vec_element**

* Use get_vec_start instead of get_vec_element

* Move it outside of loop (code motion)

* Accumulate in temporary acc – reduce # of memory ops

---

**Data Types**

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

**Operations**

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

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

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 scheduled/run dynamically.

- **Benefit:**
  - Without programming effort, superscalar processor can take advantage of the instruction level parallelism (ILP) that most programs have
  - Most CPUs since about 1998 are superscalar.
  - Intel: since Pentium Pro

Modern CPU Design

Intel Nehalem CPU

- Multiple instructions can execute in parallel
- 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 Add</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 Add</td>
<td>10-23</td>
<td>10-23</td>
</tr>
</tbody>
</table>

Note: the textbook uses Intel Core i7 Haswell as reference machine

<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>Combine1</td>
<td>10.12</td>
<td>9.02</td>
</tr>
<tr>
<td>Combine4</td>
<td>1.27</td>
<td>3.01</td>
</tr>
</tbody>
</table>

Review: Effect of Basic Optimizations

- Eliminates sources of overhead in loop

Latency, Issue Time and Capacity

- Intel i7 Haswell processor

<table>
<thead>
<tr>
<th>Operation</th>
<th>Integer</th>
<th>Floating Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>Latency</td>
<td>Issue</td>
<td>Capacity</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Addition</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Multiplication</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Division</td>
<td>3-30</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>3-30</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>3-15</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>3-15</td>
<td>1</td>
</tr>
</tbody>
</table>

Latency – time to perform an operation
Issue – time between two independent operations
Capacity – number of functional units
**x86-64 Compilation of Combine4**

- **Throughput Bound:**
  - `combine4` for `Throughput Bound`
  - `Latency Bound` *Overhead*

- **Latency Bound:**
  - The time when a series of operations must be performed in strict sequence due to data dependence

- **Throughput Bound:**
  - The maximum rate functional units can produce results

- **Measurement shows combine4 for add:**
  - 

<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>3.0</td>
</tr>
<tr>
<td>Latency Bound</td>
<td>1.0</td>
<td>3.0</td>
</tr>
<tr>
<td>Throughput Bound</td>
<td>0.5</td>
<td>1.0</td>
</tr>
</tbody>
</table>

**More on Data-Flow Graphs**

- **Two blue boxes are in dependence chains:**
  1. `acc` in `xmm0`
  2. `loop index i` in `ldrd`

**Data-Flow Graph for Multiple Iterations**

- **Critical Path** is formed by dependence chain
  - Floating multiplication: 5 cycles
  - Integer add: 1 cycle
  - Floating point multipliers are limiting resources, i.e. Latency bound
  - We want to change program to make it throughput bound

<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>3.0</td>
</tr>
<tr>
<td>Latency Bound</td>
<td>1.0</td>
<td>3.0</td>
</tr>
<tr>
<td>Throughput Bound</td>
<td>0.5</td>
<td>1.0</td>
</tr>
</tbody>
</table>

**2 x 1 Loop Unrolling**

- **Effect of 2 x 1 Loop Unrolling**
  - 10 x 1 unrolling won’t help. Why?
  - Stall sequential dependency

<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>3.0</td>
</tr>
<tr>
<td>(Int) unrolling</td>
<td>1.01</td>
<td>3.01</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>
Why 2 x 1 Loop Unrolling is Insufficient

Data-Flow Graph: 2 x 1 Unrolling, n Iterations

Enhancing Parallelism: 2 x 2 Loop Unrolling

Separate Accumulators

What Really Happens in 2 x 2 Unrolling? (I)
What Really Happens in 2 x 2 Unrolling? (II)

We eliminate the data dependency between the two multiplication operations!!!
Re-associated Computation

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

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

- What changed:
  - Ops in the next iteration can be started early (no dependency)
  - Cut down the critical path in half

- Overall Performance
  - \( N \) elements, \( D \) cycles latency/op
  - Should be \((N/2+1)*D\) cycles:
    \[ \text{CPE} = D/2 \]

Another Trick: Using Vector Instructions

* Make use of Intel SSE (Streaming SIMD Extension) Instructions
* Parallel operations on multiple data elements

The more we know about the machine, the higher the performance we could get for our programs!!

Data-Flow Graph for 2 x 1a Loop Unrolling (I)

Data-Flow Graph for 2 x 1a Loop Unrolling (II)

Data-Flow Graph for 2 x 1a Loop Unrolling (III)

Data-Flow Graph for 2 x 1a Loop Unrolling (IV)

<table>
<thead>
<tr>
<th>Method</th>
<th>Operation</th>
<th>Integer</th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Add</td>
<td>Mult</td>
<td>Add</td>
</tr>
<tr>
<td>Combine4</td>
<td>1.27</td>
<td>3.01</td>
<td>3.01</td>
</tr>
<tr>
<td>Unroll 2x1</td>
<td>1.01</td>
<td>3.01</td>
<td>3.01</td>
</tr>
<tr>
<td>Unroll 2x2</td>
<td>0.81</td>
<td>1.51</td>
<td>1.51</td>
</tr>
<tr>
<td>Unroll 10x10</td>
<td>0.55</td>
<td>1.00</td>
<td>1.01</td>
</tr>
<tr>
<td>Unroll 2x1a</td>
<td>1.01</td>
<td>1.51</td>
<td>1.51</td>
</tr>
<tr>
<td>Latency Bound</td>
<td>1.00</td>
<td>3.00</td>
<td>3.00</td>
</tr>
<tr>
<td>Throughput Bound</td>
<td>0.5</td>
<td>1.0</td>
<td>1.0</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

  ```
  8048f3: movl $0x1,%ecx
  8048f8: movl %edx,%edx
  8048f9: cmpl %esi,%edx
  8048fc: jnl 8048a25
  8048a00: imull (%eax,%edx,4),%ecx
  ```

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

Modern CPU Design – Branch Predictors

- **Branch Outcomes**
  - When encounter conditional branch, cannot determine where to continue fetching
  - **Branch Not-Taken**: Continue with next instruction in sequence
  - **Branch Taken**: Transfer control to branch target
  - Cannot resolve until outcome determined by branch/integer unit

  ```
  8048a25: cmpl %edi,%edx
  8048a27: jl 8048a20
  8048a29: movl %ecx(%ebp),%eax
  8048a2c: leal 0xffffffe8(%ebp),%esp
  8048a2f: movl %ecx,%eax
  ```

Branch Prediction

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

  ```
  8048f3: movl $0x1,%ecx
  8048f8: movl %edx,%edx
  8048f9: cmpl %esi,%edx
  8048fc: jnl 8048a25
  ```

Outline

- **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

Branch Prediction Through Loop

- **Assume vector length = 100**
- **Predict Taken (OK)**
- **Predict Taken (Oops)**
- **Read invalid location**
- **Executed**
- **Fetched**
Branch Misprediction Invalidation

Assume vector length = 100

Predict Taken (OK)

i = 98

Predict Taken (Oops)

i = 100

Invalidate

Effect of Branch Prediction – Bound Check

• Loops
  • Typically, only miss when loop hit end

• Checking code
  • Array bound check - reliably predicts that error won’t occur

```
void combine4b(vvec_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;
}
```

Method | Integer | Double FP
--- | --- | ---
Operation | Add | Mult | Add | Mult
Combine4 | 1.27 | 3.01 | 3.01 | 5.01
Combine4b | 2.02 | 3.01 | 3.01 | 5.01

Branch Misprediction Recovery

```
80488b1: movl (tекс, heda, 4), текс
80488b4: addl текс, (теди)  
80488b6: incl теди
80488b7: cmpl теди, heda  
80488b9: j1 80488b1  
80488b1: movl (tекс, heda, 4), текс
80488b4: addl текс, (теди)  
80488b6: incl теди
80488b7: cmpl теди, heda  
80488b9: j1 80488b1  
80488b1: movl (tекс, heda, 4), текс
80488b4: addl текс, (теди)  
80488b6: incl теди
80488b7: cmpl теди, heda  
80488b9: j1 80488b1  
80488b1: movl (tекс, heda, 4), текс
```

Definitely not taken

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

Avoid Branches – Use Conditional Moves

```
void combine4b(vvec_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;
}
```

```
void minmax(long a[], long b[], long n) {
    long i;
    for (i = 0; i < n; i++) {
        if (a[i] > b[i]) {
            a[i] = b[i];
            b[i] = t;
        }
    }
}
```

Method | Integer | Double FP
--- | --- | ---
Operation | Add | Mult | Add | Mult
Combine4 | 1.27 | 3.01 | 3.01 | 5.01
Combine4b | 2.02 | 3.01 | 3.01 | 5.01

Memory Performance

Intel Core i7 Haswell has

• 2 Load Units, each has 72 entries that keep 72 pending load requests
• 1 Store Unit that has 42 entries to keep 42 pending store requests
• L1 cache hit time is 4 cycles

```
Load unit

Store unit

Data cache
```

4/25/16
Load Performance

typedef struct ELE {
    struct ELE *next;
    long data;
} list_ele, *list_ptr;

long list_len(list_ptr ls) {
    long len = 0;
    while (ls) {
        len++;
        ls = ls->next;
    }
    return len;
}

Store Performance

/* Set elements of array to 0 */
void clear_array(long *dest, long n) {
    long i;
    for (i = 0; i < n; i++)
        dest[i] = 0;
}

/* Write to dest, read from src */
void write_read(long *src, long *dest, long n) {
    long cnt = n;
    long val = 0;
    while (cnt) {
        *dest = val;
        val = (*src) + 1;
        cnt--;
    }
}

Getting High Performance

- Good compiler and appropriate flags
  - Usually not easy to find all appropriate 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
  - May need to profile the program execution to collect runtime information for performance tuning