[CSED211] Introduction to Computer Software Systems

Lecture 9: Optimizations

Prof. Jisung Park

COMPUTER ARCHITECTURE & OPERATING SYSTEMS LABORATORY

2023.10.30

Lecture Agenda

- Overview
- Generally-Useful Optimizations
  - Code Motion and Precomputation
  - Strength Reduction
  - Sharing of Common Subexpressions
- Optimization Blockers
  - Procedure Calls
  - Memory Aliasing
- Exploiting Instruction-Level Parallelism
- Dealing with Conditionals

Performance Realities

- There is 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 the 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

Goals of Compiler-Level Optimization

- Minimize the number of instructions
  - Avoid calculations more than once
  - Avoid unnecessary calculations at all
  - Avoid slow instructions (multiplication, division)

- Minimize memory-access latency
  - Keep everything in registers whenever possible
  - Access memory in cache-friendly patterns
  - Load data from memory early, and only once

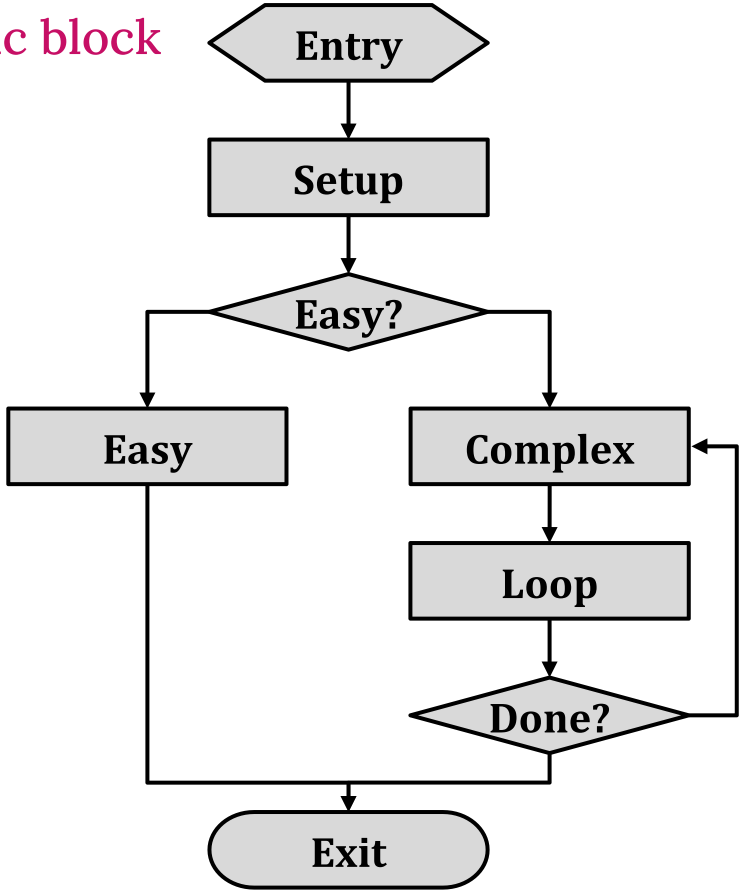
- Minimize branching
  - Avoid unnecessary decisions at all
  - Make it easier for the CPU to predict branch destinations
  - Unroll loops to spread cost of branches over more instructions

Limitations of Compiler-Level Optimization

- Generally cannot improve algorithmic complexity
  - Only constant factors (but their benefits can be 10× or even more)
- Must guarantee no change from the orignial program behavior
  - Programmer may not care about edge-case behavior, but compiler cannot know it
  - Exception: language may declare some changes acceptable
- Often only analyze one function at a time
  - Costly whole-program analysis, e.g., Link-Time Opt. (but gaining popularity)
  - Exception: inlining to merge many functions into one
- Tricky to anticipate run-time inputs
  - Profile-guided optimization can help with common case, but worst-case performance can be important as well
  - Especially for code exposed to malicious input (e.g., network servers)

Two Kinds of Optimizations

- Local optimizations work inside a single basic block
  - Constant folding, strength reduction, dead code elimination, (local) CSE, …
- Global optimizations process the entire control flow graph of a function
  - Loop transformations, code motion, (global) CSE, …



Constant Folding

- Do arithmetic in the compiler

```
long mask = 0xFF << 8;
```

Constant Folding

- Do arithmetic in the compiler

```
~~long mask = 0xFF << 8;~~
long mask = 0xFF00;
```

- Any expression with constant inputs can be folded

- Might even be able to remove library calls

```
size_t namelen = strlen("Harry Bovik");
```

Constant Folding

- Do arithmetic in the compiler

```
~~long mask = 0xFF << 8;~~
long mask = 0xFF00;
```

- Any expression with constant inputs can be folded

- Might even be able to remove library calls

```
~~size_t namelen = strlen("Harry Bovik");~~
size_t namelen = 11;
```

Reduction in Strength

- Replace a costly operation with simpler one
  - e.g., shift and add instead of multiply or divide

    `16 * x` → `x << 4`

  - Machine-dependent effectiveness
    - Depends on the cost of multiply or divide instruction
    - On Intel Nehalem, integer multiply requires three CPU cycles

- Recognize sequence of products

```c
for (i = 0; i < n; i++) {
    int ni = n * i;
    for (j = 0; j < n; j++)
        a[ni + j] = b[j];
}
```

→

```c
int ni = 0;
for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++)
        a[ni + j] = b[j];
    ni += n;
}
```

Dead Code Elimination

- Do not emit code that will never be executed

```
if (0) { puts("Kilroy was here"); }
if (1) { puts("Only bozos on this bus"); }
```

Dead Code Elimination

- Do not emit code that will never be executed

```
if (0) { puts("Kilroy was here"); }
if (1) { puts("Only bozos on this bus"); }
```

- Do not emit code whose result is overwritten

```
x = 23;
x = 42;
```

Dead Code Elimination

- Do not emit code that will never be executed

```
~~if (0) { puts("Kilroy was here"); }~~
~~if (1) {~~ puts("Only bozos on this bus"); ~~}~~
```

- Do not emit code whose result is overwritten

```
~~x = 23;~~
x = 42;
```

- These may look silly, but
  - Can be produced by other optimizations
  - Assignments to **x** might be far apart

Common Subexpression Elimination

- Factor out repeated calculations, only do them once

```
norm[i] = v[i].x * v[i].x + v[i].y * v[i].y;
```

```
elt = &v[i];
x = elt->x;
y = elt->y;
norm[i] = x * x + y * y;
```

Share Common Subexpressions

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

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

3 multiplications: `i*n, (i–1)*n, (i+1)*n`

```
leaq    1(%rsi), %rax   # i+1
leaq    -1(%rsi), %r8   # i-1
imulq   %rcx, %rsi      # i*n
imulq   %rcx, %rax      # (i+1)*n
imulq   %rcx, %r8       # (i-1)*n
addq    %rdx, %rsi      # i*n+j
addq    %rdx, %rax      # (i+1)*n+j
addq    %rdx, %r8       # (i-1)*n+j
```

```c
long inj = i * n + j;
up    = val[inj - n];
down  = val[inj + n];
left  = val[inj - 1];
right = val[inj + 1];
sum = up + down + left + right;
```

1 multiplication: `i*n`

```
imulq     %rcx, %rsi   # i*n
addq      %rdx, %rsi   # i*n+j
movq      %rsi, %rax   # i*n+j
subq      %rcx, %rax   # i*n+j-n
leaq      (%rsi,%rcx), %rcx # i*n+j+n
```

Code Motion

- Move calculations out of a loop
- Valid only if every iteration would produce the same result

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

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

Compiler-Generated Code Motion (-O1)

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

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

```
set_row:
        testq %rcx, %rcx              # Test n
        jle   .L1                     # If 0, goto done
        imulq %rcx, %rdx              # ni = n * i
        leaq  (%rdi,%rdx,8), %rdx     # rowp = A + ni * 8
        movl  $0, %eax                # j = 0
.L3:                                  # loop:
        movsd (%rsi,%rax,8), %xmm0    # t = b[j]
        movsd %xmm0, (%rdx,%rax,8)    # M[A + ni * 8 + j * 8] = t
        addq  $1, %rax                # j++
        cmpq  %rcx, %rax              # j:n
        jne   .L3                     # if !=, goto loop
.L1:                                  # done:
        rep ; ret
```

Inlining

- Copy a function's body into its caller(s)
    - Can create opportunities for many other optimizations
    - Can make code much bigger and therefore slower (size → i-cache)

```
int pred(int x){
    if(x == 0)
        return 0;
    else
        return x - 1;
}

int func(int y){
    return pred(y)
         + pred(0)
         + pred(y + 1);
}
```

```
int func(int y) {
    int tmp;
    if (y == 0)
        tmp = 0;
    else
        tmp = y - 1;
    if (0 == 0)
        tmp += 0;
    else
        tmp += 0 - 1;
    if (y + 1 == 0)
        tmp += 0;
    else
        tmp += (y + 1) - 1;
    return tmp;
}
```

Inlining

- Copy a function's body into its caller(s)
  - Can create opportunities for many other optimizations
  - Can make code much bigger and therefore slower (size → i-cache)

```c
int pred(int x){
    if(x == 0)
        return 0;
    else
        return x - 1;
}

int func(int y){
    return pred(y)
        + pred(0)
        + pred(y + 1);
}
```

```c
int func(int y) {
    int tmp;
    if (y == 0)
        tmp = 0;
    else
        tmp = y - 1;
    if (0 == 0)     Always true
        tmp += 0;
    else
        tmp += 0 - 1;
    if (y + 1 == 0)
        tmp += 0;
    else
        tmp += (y + 1) - 1;
    return tmp;
}
```

Inlining

- Copy a function's body into its caller(s)
  - Can create opportunities for many other optimizations
  - Can make code much bigger and therefore slower (size → i-cache)

```c
int pred(int x){
    if(x == 0)
        return 0;
    else
        return x - 1;
}

int func(int y){
    return pred(y)
        + pred(0)
        + pred(y + 1);
}
```

```c
int func(int y) {
    int tmp;
    if (y == 0)
        tmp = 0;
    else
        tmp = y - 1;
    if (0 == 0)    Always true
        tmp += 0;
    else
        tmp += 0 - 1;
    if (y + 1 == 0)
        tmp += 0;
    else
        tmp += (y + 1) - 1;
    return tmp;
}
```

Inlining

- Copy a function's body into its caller(s)
  - Can create opportunities for many other optimizations
  - Can make code much bigger and therefore slower (size → i-cache)

```
int pred(int x){
    if(x == 0)
        return 0;
    else
        return x - 1;
}

int func(int y){
    return pred(y)
        + pred(0)
        + pred(y + 1);
}
```

```
int func(int y) {
    int tmp;
    if (y == 0)
        tmp = 0;
    else
        tmp = y - 1;
    if (0 == 0)
        tmp += 0;
    else
        tmp += 0 - 1;
    if (y + 1 == 0)
        tmp += 0;
    else
        tmp += (y + 1) - 1;
    return tmp;
}
```

No impact

Inlining

- Copy a function's body into its caller(s)
  - Can create opportunities for many other optimizations
  - Can make code much bigger and therefore slower (size → i-cache)

```
int pred(int x){
    if(x == 0)
        return 0;
    else
        return x - 1;
}

int func(int y){
    return pred(y)
        + pred(0)
        + pred(y + 1);
}
```

```
int func(int y) {
    int tmp;
    if (y == 0)
        tmp = 0;
    else
        tmp = y - 1;
    if (0 == 0)
        tmp += 0;
    else
        tmp += 0 - 1;
    if (y + 1 == 0)
        tmp += 0;
    else
        tmp += (y + 1) - 1;
    return tmp;
}
```

No impact

Inlining

- Copy a function's body into its caller(s)
  - Can create opportunities for many other optimizations
  - Can make code much bigger and therefore slower (size → i-cache)

```
int pred(int x){
    if(x == 0)
        return 0;
    else
        return x - 1;
}

int func(int y){
    return pred(y)
        + pred(0)
        + pred(y + 1);
}
```

```
int func(int y) {
    int tmp;
    if (y == 0)
        tmp = 0;
    else
        tmp = y - 1;
    if (0 == 0)
        tmp += 0;
    else
        tmp += 0 - 1;
    if (y + 1 == 0)
        tmp += 0;
    else
        tmp += (y + 1) - 1;
    return tmp;
}
```

Constant Folding

Inlining

- Copy a function's body into its caller(s)
  - Can create opportunities for many other optimizations
  - Can make code much bigger and therefore slower (size → i-cache)

```
int pred(int x){
    if(x == 0)
        return 0;
    else
        return x - 1;
}

int func(int y){
    return pred(y)
        + pred(0)
        + pred(y + 1);
}
```

```
int func(int y) {
    int tmp;
    if (y == 0)
        tmp = 0;
    else
        tmp = y - 1;
    if (0 == 0)
        tmp += 0;
    else
        tmp -= 1;
    if (y == -1)
        tmp += 0;
    else
        tmp += y;
    return tmp;
}
```

Constant Folding

Inlining

- Copy a function's body into its caller(s)
  - Can create opportunities for many other optimizations
  - Can make code much bigger and therefore slower (size → i-cache)

```c
int pred(int x){
    if(x == 0)
        return 0;
    else
        return x - 1;
}

int func(int y){
    return pred(y)
        + pred(0)
        + pred(y + 1);
}
```

```c
int func(int y) {
    int tmp;
    if (y == 0)
        tmp = 0;
    else
        tmp = y - 1;
    if (0 == 0)
        tmp += 0;
    else
        tmp -= 1;
    if (y == -1)
        tmp += 0;
    else
        tmp += y;
    return tmp;
}
```

→

```c
int func(int y) {
    int tmp = 0;
    if (y != 0)
        tmp = y - 1;
    if (y != -1)
        tmp += y;
    return tmp;
}
```

Optimization Example: Bubble Sort

- Bubble sort program that sorts an array **A** allocated in static storage
  - An element of **A** requires four bytes of a byte-addressible machine
  - Elements of **A** are numbered 1 through n (n is a variable)
  - **A[j]** is in location **&A + 4 * (j - 1)**

```
for (i = n - 1; i >= 1; i--)
    for (j = 1; j <= i; j++)
        if (A[j] > A[j+1]) {
            temp = A[j];
            A[j] = A[j+1];
            A[j+1] = temp;
        }
```

Translated (Pseudo) Code

```
for (i = n - 1; i >= 1; i--)
    for (j = 1; j <= i; j++)
        if (A[j] > A[j+1]) {
            temp = A[j];
            A[j] = A[j+1];
            A[j+1] = temp;
        }
```

# of Instructions
- 29 in outer loop
- 25 in inner loop

```
    i := n - 1
L5: if i < 1 goto L1
    j := 1
L4: if j > i goto L2
    t1 := j - 1
    t2 := 4 * t1
    t3 := A[t2]         # A[j]
    t4 := j + 1
    t5 := t4 - 1        t6 := 4 * j
    t6 := 4 * t5
    t7 := A[t6]         # A[j+1]
```

```
    if t3 <= t7 goto L3
    t8 := j - 1
    t9 := 4 * t8
    temp := A[t9]       # temp := A[j]
    t10 := j + 1
    t11 := t10 - 1      t12 := 4 * j
    t12 := 4 * t11
    t13 := A[t12]       # A[j + 1]
    t14 := j - 1
    t15 := 4 * t14      A[t9] := t13
    A[t15] := t13       # A[j] := A[j + 1]
    t16 := j + 1
    t17 := t16 - 1      A[t12] := temp
    t18 := 4 * t17
    A[t18] := temp      # A[j + 1] := temp
L3: j := j + 1
    goto L4
L2: i := i - 1
    goto L5
L1:
```

Translated (Pseudo) Code

```
for (i = n - 1; i >= 1; i--)
    for (j = 1; j <= i; j++)
        if (A[j] > A[j+1]) {
            temp = A[j];
            A[j] = A[j+1];
            A[j+1] = temp;
        }
```

# of Instructions

- ~~29~~ 20 in outer loop
- ~~25~~ 16 in inner loop

```
    i := n - 1
L5: if i < 1 goto L1
    j := 1
L4: if j > i goto L2
    t1 := j - 1
    t2 := 4 * t1
    t3 := A[t2]     # A[j]
    t6 := 4 * j
    t7 := A[t6]     # A[j+1]
    if t3 <= t7 goto L3
```

```
    t8 := j - 1
    t9 := 4 * t8
    temp := A[t9]     # temp := A[j]
    t12 := 4 * j
    t13 := A[t12]     # A[j + 1]
    A[t9] := t13      # A[j] := A[j + 1]
    A[t12] := temp    # A[j + 1] := temp
L3: j := j + 1
    goto L4
L2: i := i - 1
    goto L5
L1:
```

Translated (Pseudo) Code

```
for (i = n - 1; i >= 1; i--)
    for (j = 1; j <= i; j++)
        if (A[j] > A[j+1]) {
            temp = A[j];
            A[j] = A[j+1];
            A[j+1] = temp;
        }
```

# of Instructions
- ~~29~~ 20 in outer loop
- ~~25~~ 16 in inner loop

```
    i := n - 1
L5: if i < 1 goto L1
    j := 1
L4: if j > i goto L2
    t1 := j - 1
    t2 := 4 * t1
    t3 := A[t2]      # Old A[j]
    t6 := 4 * j
    t7 := A[t6]      # A[j+1]
    if t3 <= t7 goto L3
```

```
    t8  := j - 1
    t9  := 4 * t8
    temp  := A[t9]     # temp := Old A[j]
    t12  := 4 * j
    t13  := A[t12]     # A[j + 1]
    A[t9]  := t13      # A[j] := A[j + 1]
    A[t12]  := temp    # A[j + 1] := temp
L3: j := j + 1         A[t2] := t7
    goto L4            A[t6] := t3
L2: i := i - 1
    goto L5
L1:
```

Translated (Pseudo) Code

```
for (i = n - 1; i >= 1; i--)
    for (j = 1; j <= i; j++)
        if (A[j] > A[j+1]) {
            temp = A[j];
            A[j] = A[j+1];
            A[j+1] = temp;
        }
```

# of Instructions

- ~~29~~ ~~20~~ 15 in outer loop
- ~~25~~ ~~16~~ 11 in inner loop

```
    i := n - 1
L5: if i < 1 goto L1
    j := 1
L4: if j > i goto L2
    t1 := j - 1
    t2 := 4 * t1
    t3 := A[t2]    # Old A[j]
    t6 := 4 * j
    t7 := A[t6]    # A[j+1]
    if t3 <= t7 goto L3
```

```
    A[t2] := t7   # temp := A[j]
    A[t6] := t3   # A[j + 1] := Old A[j]
L3: j := j + 1
    goto L4
L2: i := i - 1
    goto L5
L1:
```

Translated (Pseudo) Code

```
    i := n - 1
L5: if i < 1 goto L1
    j := 1
L4: if j > i goto L2
    t1 := j - 1
    t2 := 4 * t1
    t3 := A[t2]    # Old A[j]
    t6 := 4 * j
    t7 := A[t6]    # A[j+1]
    if t3 <= t7 goto L3
    A[t2] := t7    # temp := A[j]
    A[t6] := t3    # A[j + 1] := Old A[j]
L3: j := j + 1
    goto L4
L2: i := i - 1
    goto L5
L1:
```

```
    i := n - 1
L5: if i < 1 goto L1
    t2 := 0
    t6 := 4
    t19 := 4 * t1
L4: if t6 > t19 goto L2
    t3 := A[t2]    # Old A[j]
    t7 := A[t6]    # A[j+1]
    if t3 <= t7 goto L3
    A[t2] := t7    # temp := A[j]
    A[t6] := t3    # A[j + 1] := Old A[j]
L3: t2 := t2 + 4
    t6 := t6 + 4
    goto L4
L2: i := i - 1
    goto L5
L1:
```

Final (Pseudo) Code

- # of instructions before optimizations
  - 29 in outer loop
  - 25 in inner loop
- # of instructions after optimizations
  - 15 in outer loop
  - 9 in inner loop
- These were machine-independent optimizations

```
     i := n - 1
L5:  if i < 1 goto L1
     t2  := 0
     t6  := 4
     t19 := i << 2
L4:  if t6 > t19 goto L2
     t3  := A[t2]    # Old A[j]
     t7  := A[t6]    # A[j+1]
     if t3 <= t7 goto L3
     A[t2] := t7     # temp := A[j]
     A[t6] := t3     # A[j + 1] := Old A[j]
L3:  t2  := t2 + 4
     t6  := t6 + 4
     goto L4
L2:  i  := i - 1
     goto L5
L1:
```

Lecture Agenda

- Overview
- Generally-Useful Optimizations
  - Code Motion and Precomputation
  - Strength Reduction
  - Sharing of Common Subexpressions
- Optimization Blockers
  - Procedure Calls
  - Memory Aliasing
- Exploiting Instruction-Level Parallelism
- Dealing with Conditionals

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.
- Obvious behavior to the programmer can be obfuscated to the machine
  - 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

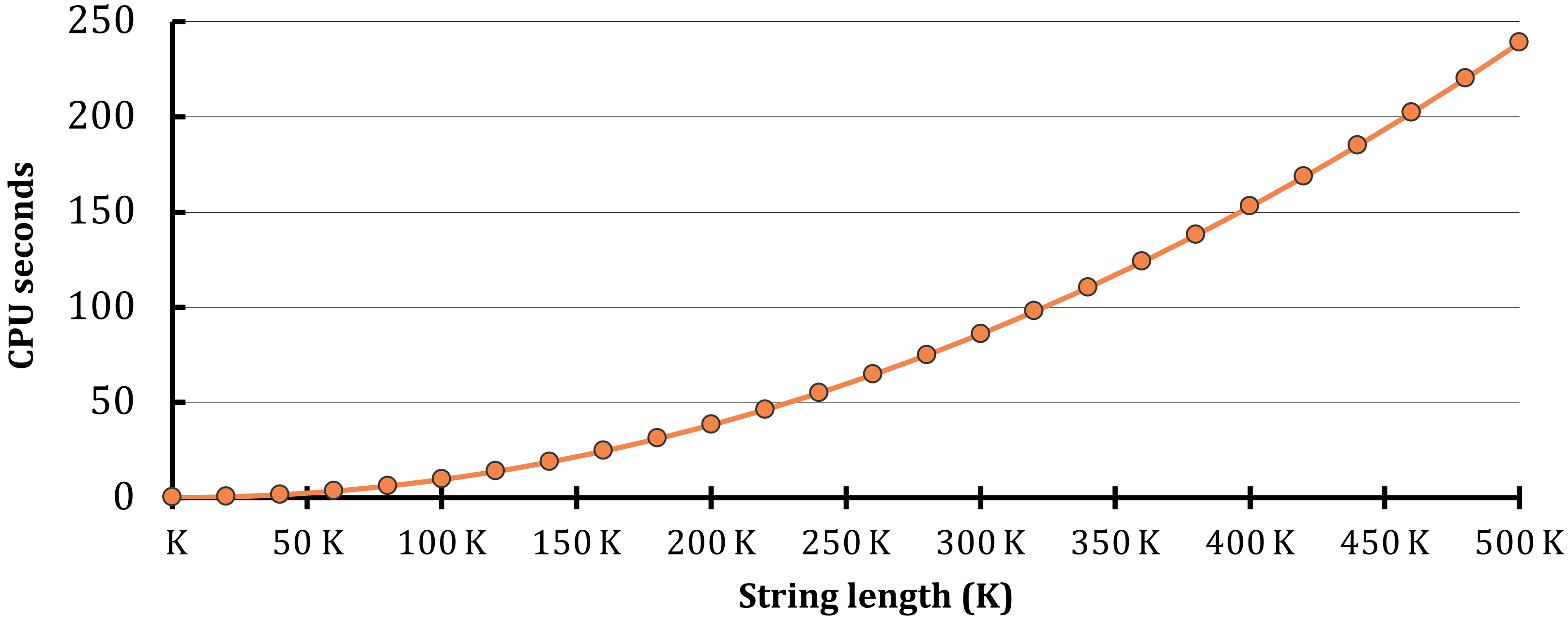
Optimization Blocker: Procedure Calls

- Procedure to convert a string to lower case

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

Lower Case Conversion Performance

- Time quadruples when double string length
- Quadratic performance



Convert Loop to Goto Form

- **`strlen`** is executed every iteration

```
void lower1_goto(char *s) {
    size_t i = 0;
    if (i >= strlen(s))
      goto DONE;
 LOOP:
    if (s[i] >= 'A' && s[i] <= 'Z')
        s[i] -= ('A' - 'a');
    i++;
    if (i < strlen(s))
      goto LOOP;
 DONE:
}
```

Calling `strlen`

- `strlen` performance
  - Only way to determine a string' length: scan the entire string, looking for the null character

- Overall performance for a 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;
}
```

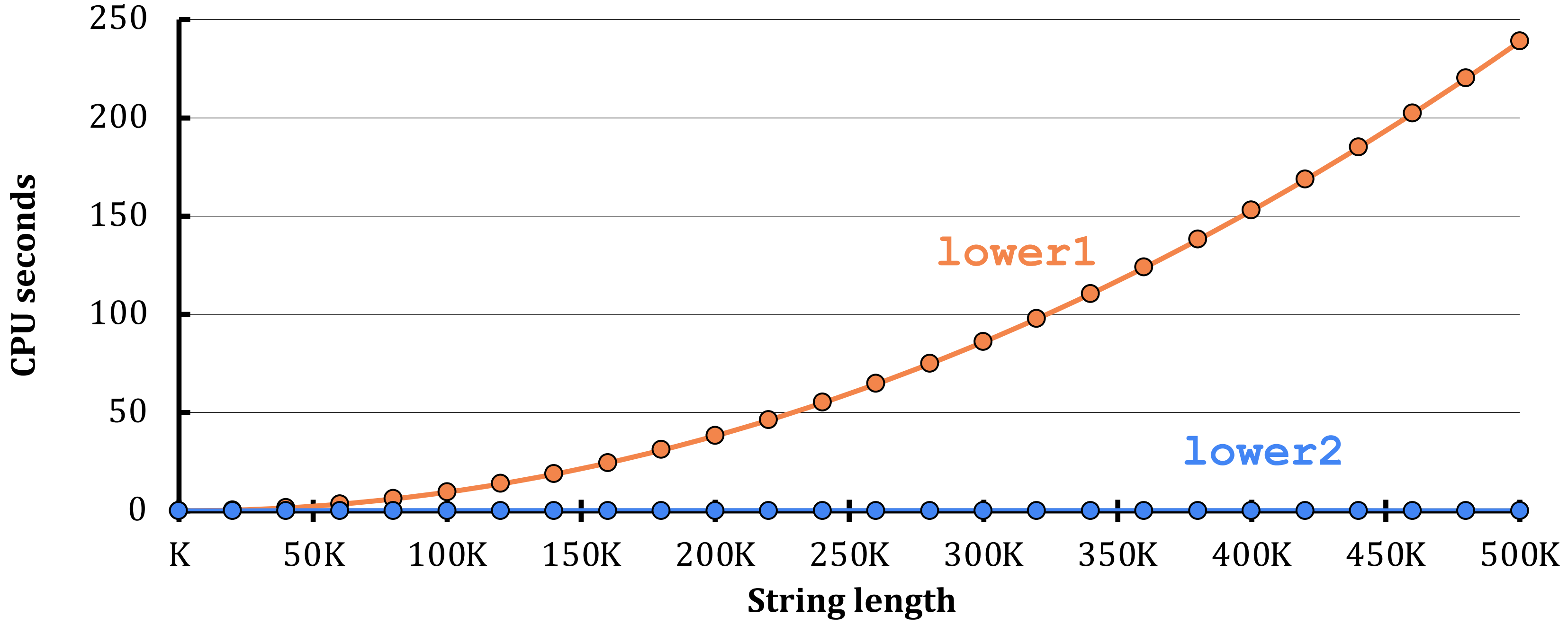
Improving Performance

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

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

Lower Case Conversion Performance

- Time doubles when double string length
- Linear performance of **`lower2`**



Optimization Blocker: Procedure Calls

- Why could compiler not move **`strlen`** out of inner loop?
  - Procedure may have side effects
    - Alters global state each time called
  - Function may not return the same value for given arguments
    - Depends on other parts of global state
    - Procedure **`lower`** could interact with **`strlen`**

- Compiler treats procedure call as a black box
  - Weak optimizations near them

- Remedies
  - Use of inline functions
    - GCC does this with **`-O1`**
    - Within a single file
  - Do your own code motion

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

Optimization Blocker: Memory Aliasing

- Aliasing: two different memory references specify single location

- Easy to have happen in C
  - Since it is 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

Memory Aliasing

- Code updates `b[i]` on every iteration
  - Why could compiler not optimize this away?

```c
/* Sum rows is of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a,
               double *b,
               long n){
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i * n + j];
    }
}
```

```
# sum_rows1 inner loop
.L4:
    movsd   (%rsi,%rax,8), %xmm0  # FP load
    addsd   (%rdi), %xmm0         # FP add
    movsd   %xmm0, (%rsi,%rax,8)  # FP store
    addq    $8, %rdi
    cmpq    %rcx, %rdi
    jne     .L4
```

Memory Aliasing

- Code updates `b[i]` on every iteration
  - Why could compiler not optimize this away?

    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];
    }
}
```

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

Value of **B**:

```
init : [4, 8, 16]
i = 0: [3, 8, 16]
i = 1: [3, 22, 16]
i = 2: [3, 22, 224]
```

Removing Aliasing

- No need to store intermediate results

```c
/* Sum rows is of n X n matrix a
   and store in vector b  */
void sum_rows2(double *a,
               double *b,
               long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        double val = 0;
        for (j = 0; j < n; j++)
            val += a[i * n + j];
        b[i] = val;
    }
}
```

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

Lecture Agenda

- Overview
- Generally-Useful Optimizations
  - Code Motion and Precomputation
  - Strength Reduction
  - Sharing of Common Subexpressions
- Optimization Blockers
  - Procedure Calls
  - Memory Aliasing
- Exploiting Instruction-Level Parallelism
- Dealing with Conditionals

Exploiting Instruction-Level Parallelism

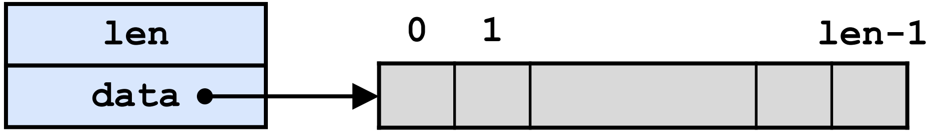
- Needs general understanding of modern processor design
  - Hardware can execute multiple instructions in parallel

- Performance limited by data dependencies

- Simple transformations can yield dramatic performance improvement
  - Compilers often cannot make these transformations
  - Lack of associativity and distributivity in floating-point arithmetic

Benchmark Example: Data Type for Vectors

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




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

Supports multiple data types: different declarations for **data_t**

```
#define data_t int
#define data_t long
#define data_t float
#define data_t double
```

Benchmark Computation

```
// Compute sum or product of vector elements
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;
    }
}
```

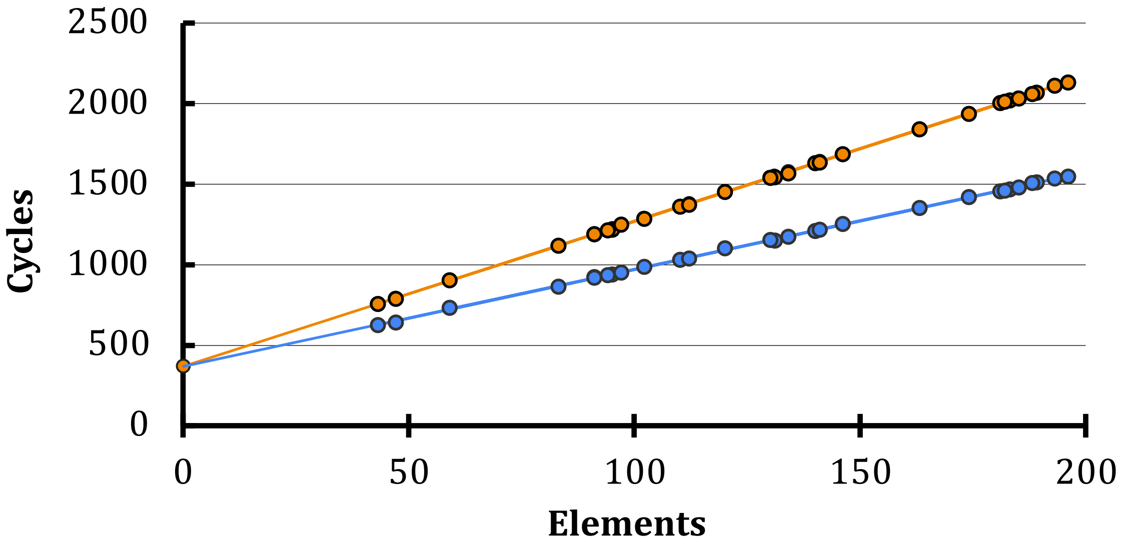
- Supports different
    - Data types: use different declaration for `data_t`
        - `int`, `long`, `flot`, and `double`
    - Operations: use different definitions of `OP` and `IDENT`
        - `+/0` and `*/1`

Cycles Per Element (CPE)

- Useful to express the performance of program operating on vectors or lists

- In the example, CPE = # of cycles per operation (`OP`)
  - T = CPE × n + α
  - α: overhead
  - n: vector length

- CPE is slope of line



Benchmark Performance

```
// Compute sum or product of vector elements
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;
    }
}
```

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine1 unoptimized	22.68	20.02	19.98	20.18
Combine1 –O1	10.12	10.12	10.17	11.14

Basic Optimizations

```
// Compute sum or product of vector elements
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;
    }
}
```

```
// Compute sum or product of vector elements
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;
}
```

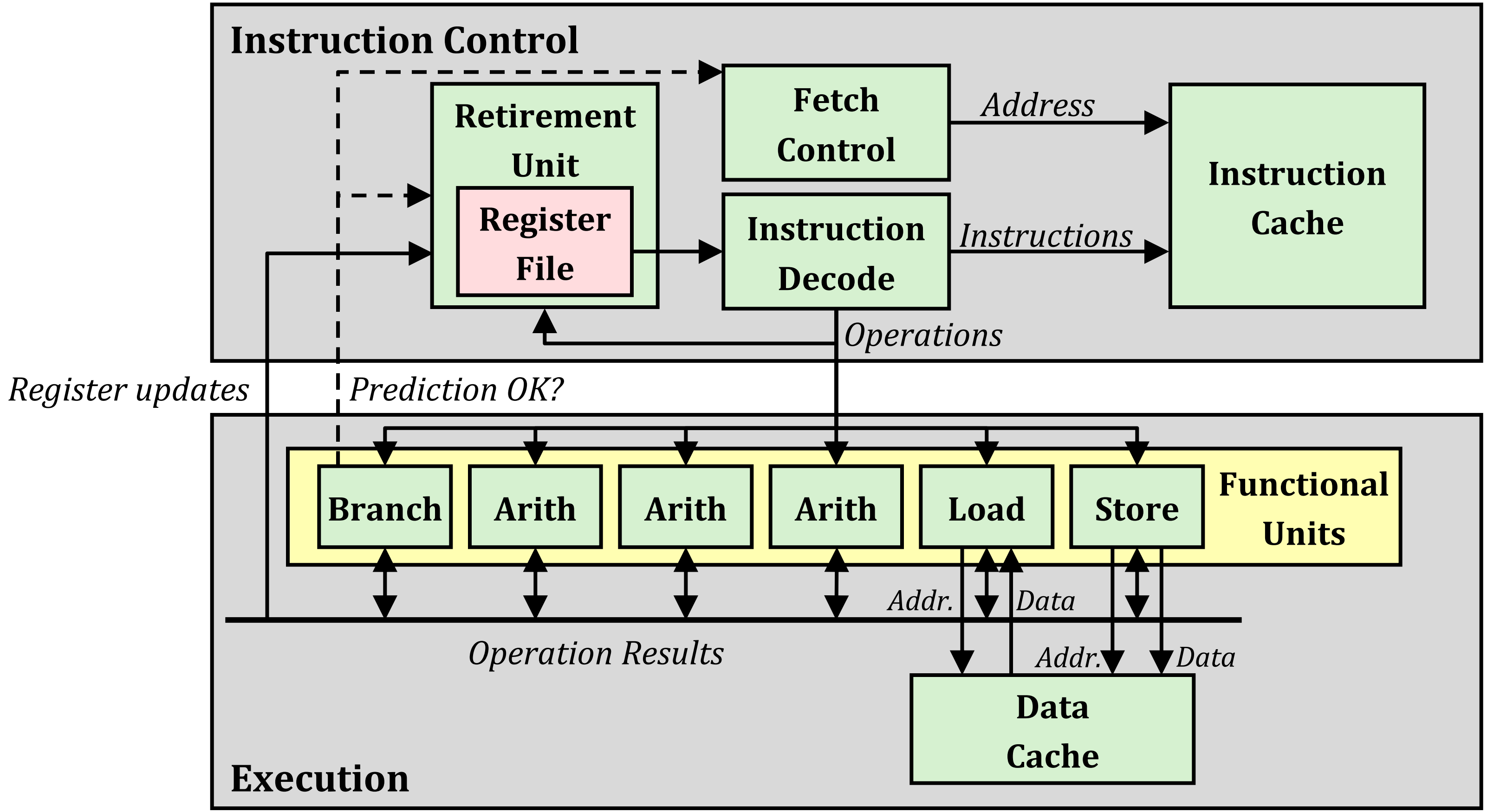
- Move **`vec_length`** out of loop
- Bounds check only once
- Accumulate in temporary

Basic Optimizations: Effect

```
// Compute sum or product of vector elements
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;
    }
}
```

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine1 –O1	10.12	10.12	10.17	11.14
Combine4	1.27	3.01	3.01	5.01

Modern CPU Design



Superscalar Processor

- A processor that can issue and execute multiple instructions in one cycle

- The instructions are
    - Retrieved from a sequential instruction stream
    - Usually scheduled dynamically

- Benefit: can take advantage of the instruction-level parallelism (ILP) that most programs have, without programming effort

- Most modern CPUs are superscalar
    - Since Pentium (1993) in Intel

Pipelined Functional Units

- What is pipelining?
  - A key approach in computer science to improve application/system performance
  - By executing multiple tasks in parallel
  - Maximally using hardware components that can operate independently

Task: Vote

Stage#1	Stage#2	Stage#3	Stage#4	[Stage#5]
ID Check	Get Paper	Marking	Submission	[Interview]

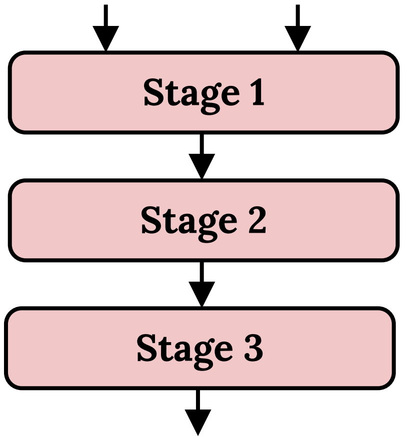
- General methodology
  - Divide computation into multiple stages that operate in parallel
  - Pass partial computations from stage to stage
  - Stage i can start on new computation once values passed to stage i+1

Pipelined Functional Units

- Complete 3 multiplications in 7 cycles, even though each requires 3 cycles

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

Time							
Stage	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Stage#1	a*b	a*c			p1*p2		
Stage#2		a*b	a*c			p1*p2	
Stage#3			a*b	a*c			p1*p2



Intel Haswell CPU

- Multiple instructions can execute in parallel
  - 2 loads and 1 store, with address computation
  - 4 integer computations
  - 2 FP multiplications
  - 1 FP division and addition

- Some instructions take > 1 cycles, but can be pipelined

Instruction	Latency	Cycles/Issue
Load / Store	4	1
Integer Multiplication	3	1
Integer/Long Division	**3-30**	**3-30**
Single/Double FP Multiplication	5	1
Single/Double FP Addition	3	1
Single/Double FP Division	**3-15**	**3-15**

x86-64 Compilation of Combine4

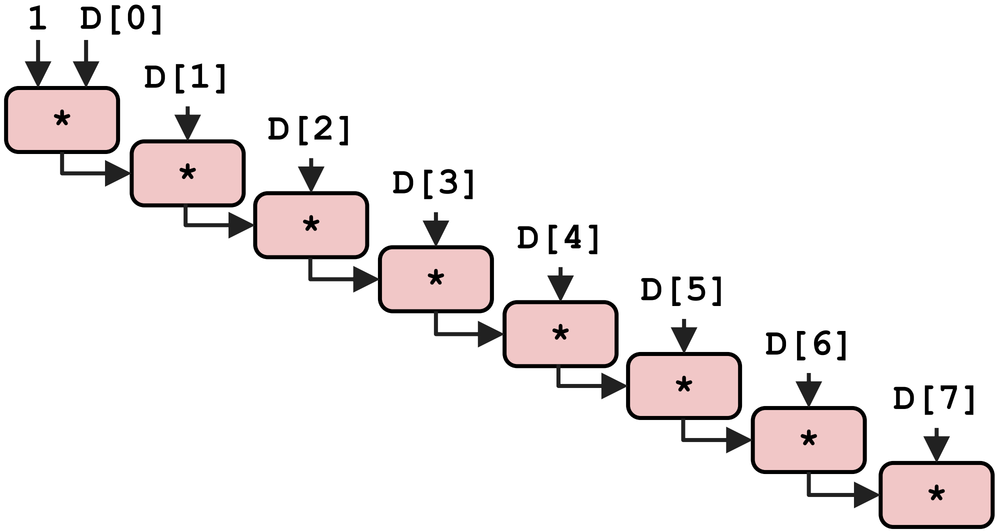
- Inner loop (case: integer multiplication)

```
.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
```

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Latency Bound	1.00	3.00	3.00	5.00

Combine4 = Serial Computation (`OP = *`)

- Computation of eight operations:
  `((((((((1*d[0])*d[1])*d[2])*d[3])*d[4])*d[5])*d[6])*d[7])`
- Sequential dependence
  - Performance dictated by latency of OP



Loop Unrolling (2×1)

- Perform 2× more useful work per iteration

```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;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i += 2) {
        x = (x OP d[i]) OP d[i + 1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x = x OP d[i];
    }
    *dest = x;
}
```

Effect of Loop Unrolling

- Helps integer addition
  - Achieves latency bound
- Others do not improve. Why?
  - Still sequential dependency

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

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Unroll 2×1	1.01	3.01	3.01	5.01
Latency Bound	1.00	3.00	3.00	5.00

Loop Unrolling with Reassociation (2×1a)

- Can this change the result of the computation?
  - Yes, for FP. Why?

```
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;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i += 2) {
        x = x OP (d[i] OP d[i + 1]);
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x = x OP d[i];
    }
    *dest = x;
}
```

Compare to before

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

Effect of Reassociation

- Nearly 2× speedup for integer mult., FP add., and FP mult.
    - Reason: breaks sequential dependency

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

Why?

4 func. units for int +
2 func. units for load

2 func. units for FP *
2 func. units for load

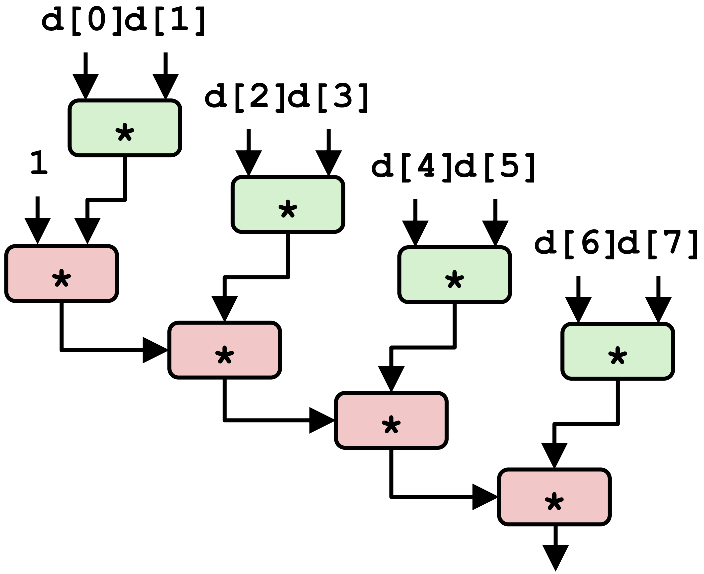
Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Unroll 2×1	1.01	3.01	3.01	5.01
Unroll 2×1a	1.01	1.51	1.51	2.51
Latency Bound	1.00	3.00	3.00	5.00
Throughput Bound	0.50	1.00	1.00	0.50

Reassociated Computation

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

- Overall performance
  - N elements, *D* cycles per operation
  - (N/2 + 1) × *D* cycles → CPE = *D*/2

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



Loop Unrolling with Separate Accumulators (2×2)

- Different form of reassociation

```
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 x0 = IDENT;
    data_t x1 = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i += 2) {
       x0 = x0 OP d[i];
       x1 = x1 OP d[i + 1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x0 = x0 OP d[i];
    }
    *dest = x0 OP x1;
}
```

Effect of Separate Accumulators

- Integer add. makes use of two load units

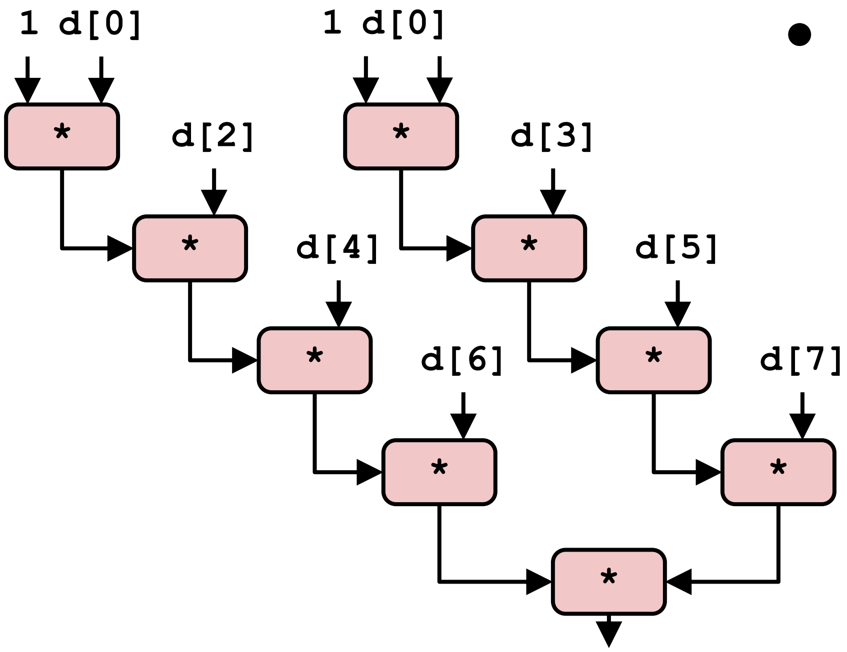
```
x0 = x0 OP d[i];
x1 = x1 OP d[i + 1];
```

- 2× speedup (over unroll2) for inteter mult., FP add., FP mult.

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Unroll 2x1	1.01	3.01	3.01	5.01
Unroll 2x1a	1.01	1.51	1.51	2.51
Unroll 2x2	0.81	1.51	1.51	2.51
Latency Bound	1.00	3.00	3.00	5.00
Throughput Bound	0.50	1.00	1.00	0.50

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 per operation
  - Should be $(N/2 + 1) \times D$ cycles
    → CPE = $D/2$
  - CPE matches the 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 Multiplication

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

K	Unrolling Factor *L*							
	1	2	3	4	6	8	10	12
1	5.01	5.01	5.01	5.01	5.01	5.01	5.01	
2		2.51		2.51		2.51		
3			1.67					
4				1.25		1.26		
6					0.84			0.88
8						0.63		
10							0.51	
12								0.52

Accumulators

Unrolling & Accumulating: Integer Addition

- Case
  - Intel Haswell
  - Integer addition
  - Latency bound: 1.00, Throughput bound: 1.00

K	Unrolling Factor *L*							
	1	**2**	**3**	**4**	**6**	**8**	**10**	**12**
1	1.27	1.01	1.01	1.01	1.01	1.01	1.01	
2		0.81		0.69		0.54		
3			0.74					
4				0.69		1.24		
6					0.56			0.56
8						0.54		
10							0.54	
12								0.56

Accumulators

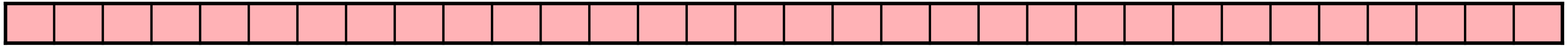
Achievable Performance

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

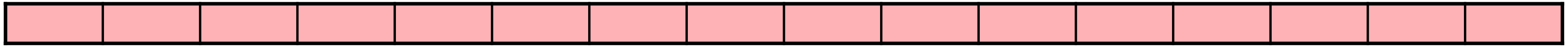
Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Best	0.54	1.01	1.01	0.52
Latency Bound	1.00	3.00	3.00	5.00
Throughput Bound	0.50	1.00	1.00	0.50

Programming with AVX2

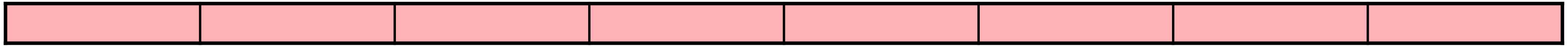
- YMM Registers: 16 total, each 32 bytes
  - 32 single-byte integers



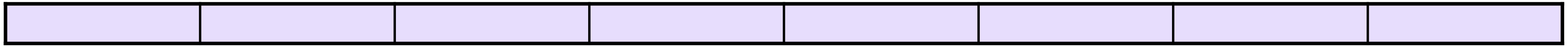
  - 16 16-bit integers



  - 8 32 –bit integers



  - 8 single-precision floats



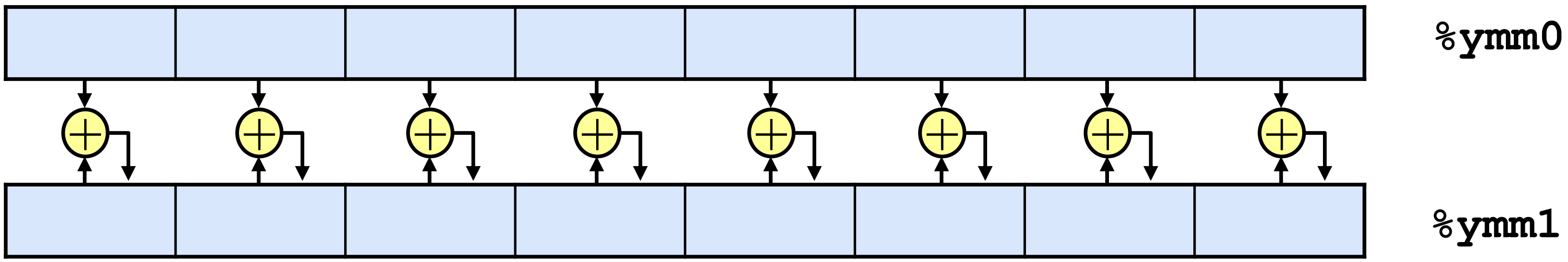
  - 4 double-precision floats

  - 1 single-precision float

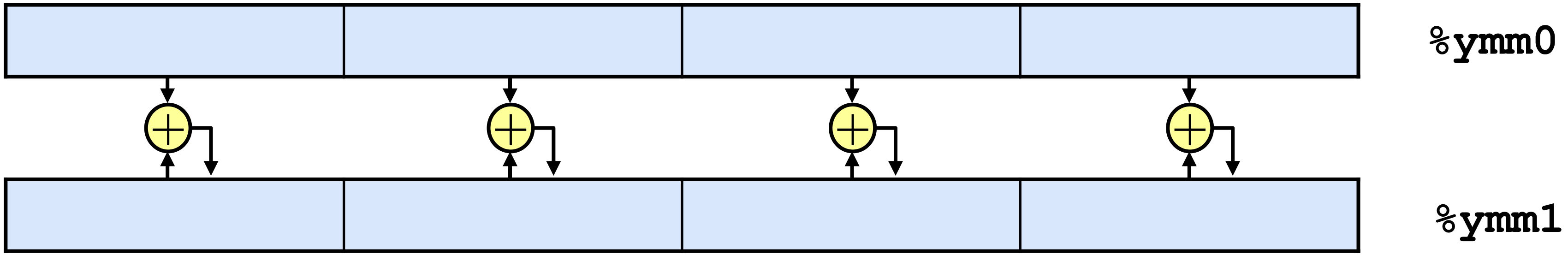
  - 1 doble-precision float

SIMD Operations

- Single Precision: **`vaddsd %ymm0, %ymm1, %ymm1`**



- Double Precision: **`vaddpd %ymm0, %ymm1, %ymm1`**



Using Vector Instructions

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

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Scalar Best	0.54	1.01	1.01	0.52
Vector Best	0.06	0.24	0.25	0.16
Latency Bound	0.50	3.00	3.00	5.00
Throughput Bound	0.50	1.00	1.00	0.50
Vec. Throughput Bound	0.06	0.12	0.25	0.12

What About Branches?

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

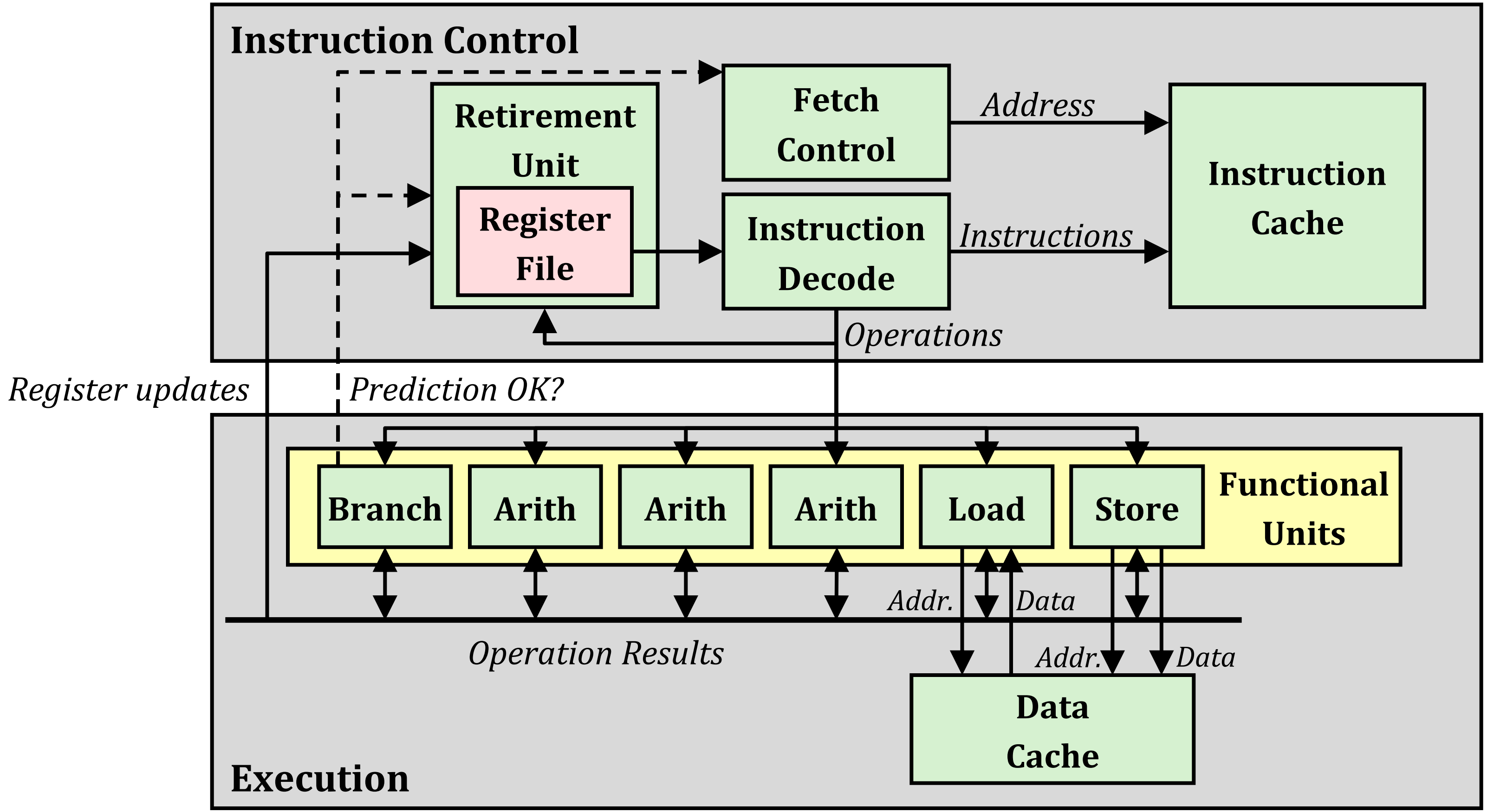
```
404663:  mov    $0x0,%eax
404668:  cmp    (%rdi),%rsi
40466b:  jge    404685
40466d:  mov    0x8(%rdi),%rax

 . . .

404685:  repz   retq
```

Exectuting

How to continue?

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

Modern CPU Design



Branch Outcomes

- Conditional branch: cannot determine where to continue fetching
  - Branch taken: transfer control to branch target
  - Branch not-taken: continue with next instruction in sequence

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

 . . .

404685:  repz   retq
```

Branch Not-Taken

Branch Taken

- Cannot resolve until outcome determined by branch/integer unit

Branch Prediction

- Idea: guess which way branch will go
  - Begin executing instructions at the predicted position
  - While not actually modifying register or memory data

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

 . . .

404685:  repz   retq
```

Prediction: Branch Taken

Begin Execution

Branch Prediction Through Loop

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

i = 98

Assume vector length = 100

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

i = 99

Predict Taken (OK)

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

i = 100

Predict Taken (Oops)

Read invalid location

Executed

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

i = 101

Fetched

Branch Prediction Through Loop

Assume vector length = 100

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

i = 98

Predict Taken (OK)

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

i = 99

Predict Taken (Oops)

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

i = 100

Executed

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

i = 101

Fetched

Invalidate

Branch Misprediction Recovery

```
401029:   vmulsd (%rdx),%xmm0,%xmm0
40102d:   add    $0x8,%rdx
401031:   cmp    %rax,%rdx
401034:   jne    401029
401036:   jmp    401040
  ...
401040:   vmovsd %xmm0,(%r12)
```

i = 99

Definitely not taken

Reload Pipeline

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

Getting High Performance

- Good compiler and flags

- Avoid 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)

[CSED211] Introduction to Computer Software Systems

Lecture 9: Optimizations

Prof. Jisung Park

COMPUTER ARCHITECTURE & OPERATING SYSTEMS LABORATORY

2023.10.30