

Code Optimization

15-213/15-513/14-513: Introduction to Computer Systems
12th Lecture, February 24, 2022

Today

- **Principles and goals of compiler optimization**
- **Local optimization**
 - Constant folding, strength reduction, dead code elimination, common subexpression elimination
- **Global optimization**
 - Inlining, code motion, loop transformations
- **Obstacles to optimization**
 - Memory aliasing, procedure calls, non-associative arithmetic
- **Quiz**
- **Machine-dependent optimization**
 - Branch predictability, loop unrolling, scheduling, vectorization

*Back in the Good Old Days,
when the term "software" sounded funny
and Real Computers were made out of drums
and vacuum tubes,*

Real Programmers wrote in machine code.

*Not FORTRAN. Not RATFOR. Not, even,
assembly language.*

Machine Code.

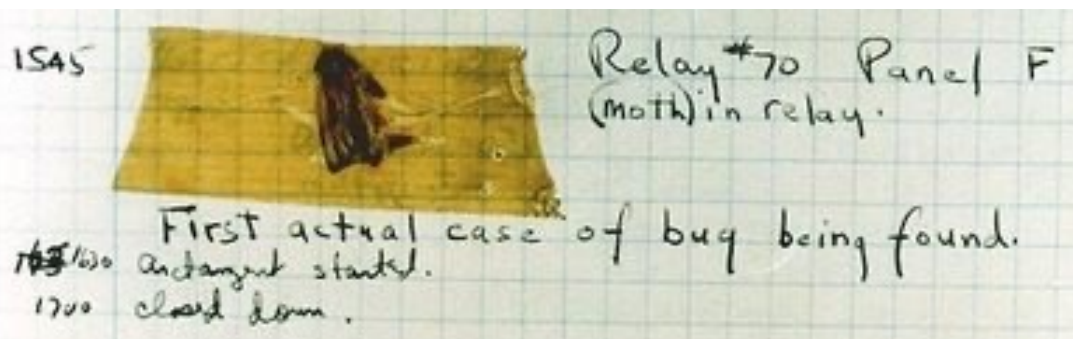
Raw, unadorned, inscrutable hexadecimal numbers. Directly.

— “The Story of Mel, a Real Programmer”

Ed Nather, 1983

Rear Admiral Grace Hopper

- First person to find an actual bug (a moth)
- Invented first compiler in 1951 (precursor to COBOL)
- “I decided data processors ought to be able to write their programs in English, and the computers would translate them into machine code”



John Backus

- Developed FORTRAN in 1957 for the IBM 704
- Oldest machine-independent programming language still in use today
- “Much of my work has come from being lazy. I didn't like writing programs, and so, when I was working on the IBM 701, I started work on a programming system to make it easier to write programs”



Fran Allen

- Pioneer of many optimizing compilation techniques
- Wrote a paper in 1966 that introduced the concept of the control flow graph, which is still central to compiler theory today
- First woman to win the ACM Turing Award



Goals of compiler optimization

■ Minimize number of instructions

- Don't do calculations more than once
- Don't do unnecessary calculations at all
- Avoid slow instructions (multiplication, division)

■ Avoid waiting for memory

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

■ Avoid branching

- Don't make unnecessary decisions at all
- Make it easier for the CPU to predict branch destinations
- “Unroll” loops to spread cost of branches over more instructions

Limits to compiler optimization

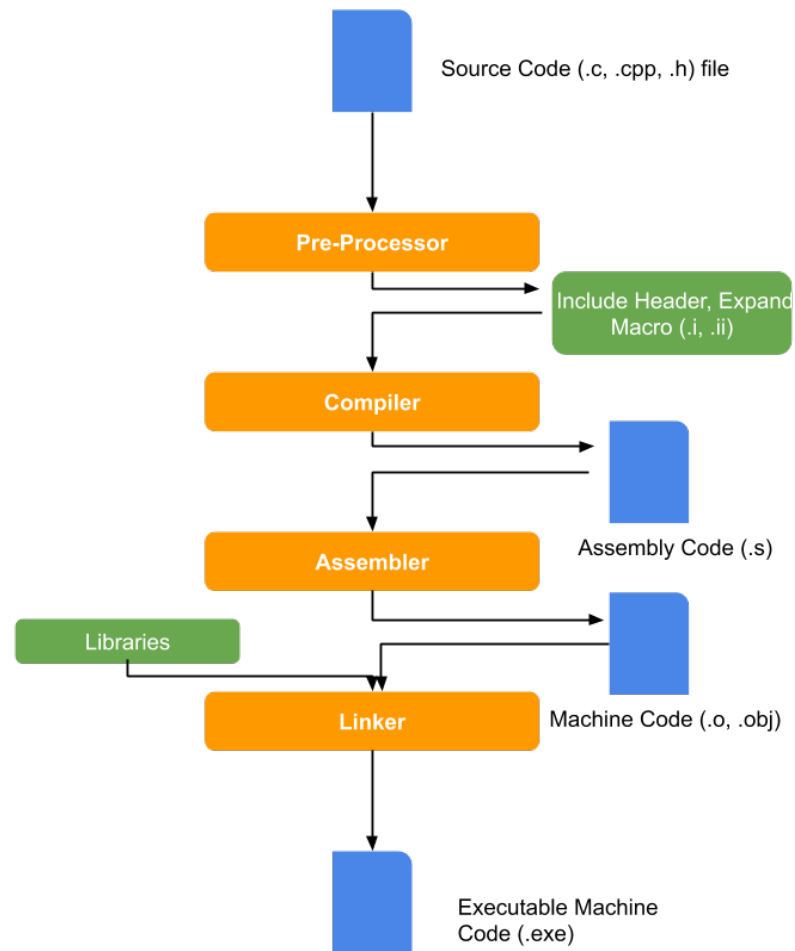
- **Generally cannot improve algorithmic complexity**
 - Only constant factors, but those can be worth 10x or more...
- **Must not cause *any* change in program behavior**
 - Programmer may not care about “edge case” behavior, but compiler does not know that
 - Exception: language may declare some changes acceptable
- **Often only analyze one function at a time**
 - Whole-program analysis (“LTO”) expensive but gaining popularity
 - Exception: *inlining* merges many functions into one
- **Tricky to anticipate run-time inputs**
 - Profile-guided optimization can help with common case, but...
 - “Worst case” performance can be just as important as “normal”
 - Especially for code exposed to *malicious* input (e.g. network servers)

Performance Realities

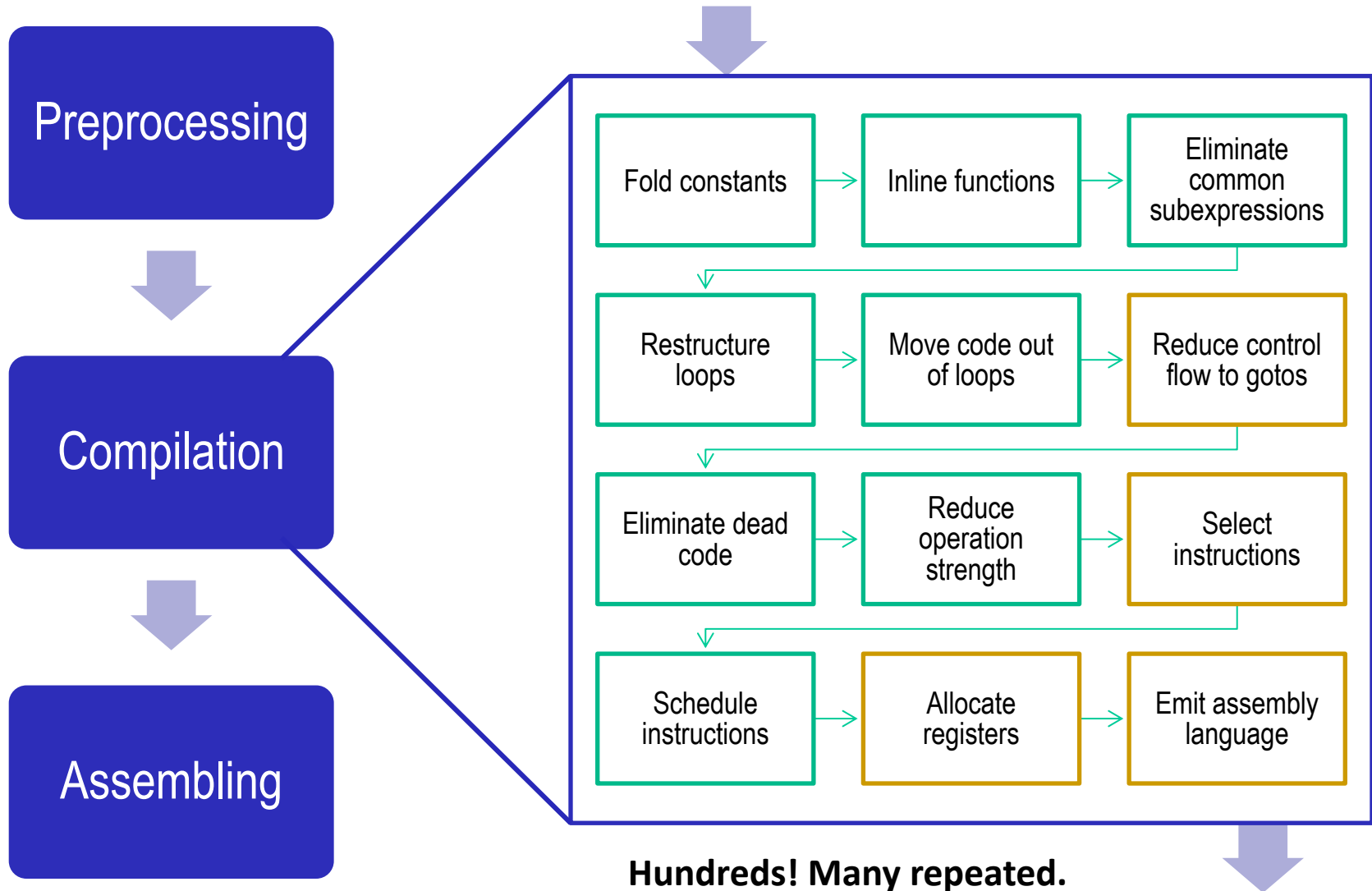
- **There's more to performance than asymptotic complexity**
- **Constant factors matter too!**
 - Easily see 10:1 performance range depending on how code is written
 - Must optimize at multiple levels:
 - algorithm, data representations, procedures, and loops
- **Must understand system to optimize performance**
 - 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

What does it mean to compile code?

- The CPU only understands *machine code* directly
- All other languages must be either
 - *interpreted*: executed by software
 - *compiled*: translated to machine code by software

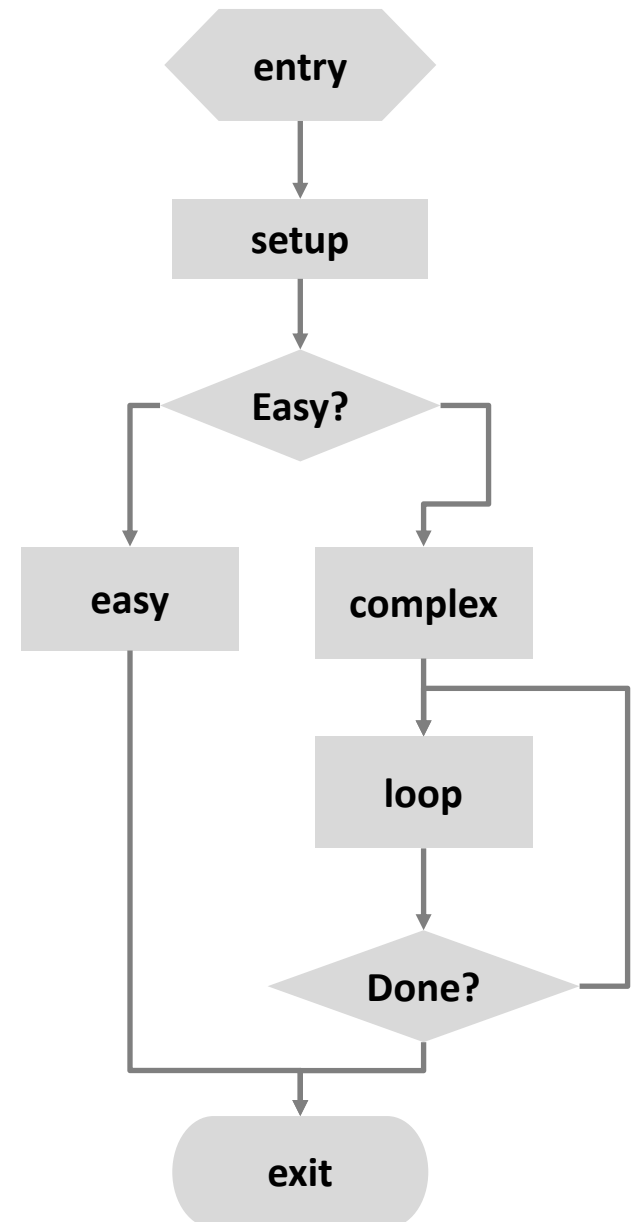


Compilation is a pipeline



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, ...



Today

- Principles and goals of compiler optimization
- **Local optimization**
 - Constant folding, strength reduction, dead code elimination, common subexpression elimination
- **Global optimization**
 - Inlining, code motion, loop transformations
- **Obstacles to optimization**
 - Memory aliasing, procedure calls, non-associative arithmetic
- **Quiz**
- **Machine-dependent optimization**
 - Branch predictability, loop unrolling, scheduling, vectorization

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;
```

Strength reduction

- Replace expensive operations with cheaper ones

```
long a = b * 5;    →  
long a = (b << 2) + b;
```

- Multiplication and division are the usual targets
- Multiplication is often hiding in memory access expressions

Dead code elimination

- Don't emit code that will never be executed

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

- Don't 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;
```

Today

- **Principles and goals of compiler optimization**
- **Local optimization**
 - Constant folding, strength reduction, dead code elimination, common subexpression elimination
- **Global optimization**
 - Inlining, code motion, loop transformations
- **Obstacles to optimization**
 - Memory aliasing, procedure calls, non-associative arithmetic
- **Quiz**
- **Machine-dependent optimization**
 - Branch predictability, loop unrolling, scheduling, vectorization

Inlining

- **Copy body of a function 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 body of a function into its caller(s)**
 - Can create opportunities for many other optimizations
 - Can make code much bigger and therefore slower

```
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;
}
```

Always true

Does nothing

Can constant fold

Inlining

- **Copy body of a function into its caller(s)**
 - Can create opportunities for many other optimizations
 - Can make code much bigger and therefore slower

```
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;  
}
```

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

Code Motion

- Move calculations out of a loop
- Only valid if every iteration would produce 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];
```

Loop Transformations

Rearrange entire loop nests for maximum efficiency

```
/* Two stages of some calculation */
void compute(double *a, double *b, long n) {
    for (long i = 0; i < n; i++)
        for (long j = 0, j < n; j++)
            a[j*n + i] = atan2(i, j);

    for (long i = 0; i < n; i++)
        for (long j = 0, j < n; j++)
            b[i*n + j] = a[i*n + j]
                + (i >= 1 && j >= 1)
                ? a[(i-1)*n + (j-1)]
                : 0;
}
```

Loop Transformations

Loop interchange: do iterations in cache-friendly order

```
/* Two stages of some calculation */
void compute(double *a, double *b, long n) {
    for (long i = 0; i < n; i++)
        for (long j = 0, j < n; j++)
            a[i*n + j] = atan2(j, i);

    for (long i = 0; i < n; i++)
        for (long j = 0, j < n; j++)
            b[i*n + j] = a[i*n + j]
                + (i >= 1 && j >= 1)
                ? a[(i-1)*n + (j-1)]
                : 0;
}
```


Loop Transformations

Loop fusion: combine adjacent loops with the same limits

```

/* Two stages of some calculation */
void compute(double *a, double *b, long n) {
  for (long i = 0; i < n; i++) {
    for (long j = 0, j < n; j++) {
      a[i*n + j] = atan2(j, i);

for (long i = 0; i < n; i++)
for (long j = 0, j < n; j++)
      b[i*n + j] = a[i*n + j]
        + (i >= 1 && j >= 1)
          ? a[(i-1)*n + (j-1)]
          : 0;
    }
  }
}

```

Loop Transformations

Induction variable elimination: replace loop indices with algebra

```
/* Two stages of some calculation */
void compute(double *a, double *b, long n) {
  for (long i = 0; i < n*n; i++) {
    for (long j = 0; j < n; j++) {
      a[i] = atan2(i%n, i/n);

```

```

      b[i] = a[i]
          + (i >= n && i%n >= 1)
            ? a[i - n - 1]
            : 0;
    }
  }
}
```

Today

- **Principles and goals of compiler optimization**
- **Local optimization**
 - Constant folding, strength reduction, dead code elimination, common subexpression elimination
- **Global optimization**
 - Inlining, code motion, loop transformations
- **Obstacles to optimization**
 - Memory aliasing, procedure calls, non-associative arithmetic
- **Quiz**
- **Machine-dependent optimization**
 - Branch predictability, loop unrolling, scheduling, vectorization

Memory Aliasing

```

/* 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];
    }
}

```

```

# 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

```

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

Memory Aliasing

```

/* 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];
    }
}

```

```

double A[9] =
    { 0, 1, 2,
      4, 8, 16},
    { 32, 64, 128};

double B[3] = A+3;

sum_rows1(A, B, 3);

```

```

double A[9] =
    { 0, 1, 2,
      3, 22, 224},
    { 32, 64, 128};

```

Value of B:

```
init: [4, 8, 16]
```

```
i = 0: [3, 8, 16]
```

```
i = 1: [3, 22, 16]
```

```
i = 2: [3, 22, 224]
```

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

Avoiding Aliasing Penalties

```
/* 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++) {  
        double val = 0;  
        for (j = 0; j < n; j++)  
            val += a[i*n + j];  
        b[i] = val;  
    }  
}
```

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

- Use a local variable for intermediate results

Avoiding Aliasing Penalties

```

/* 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++) {
        double val = 0;
        for (j = 0; j < n; j++)
            val += a[i*n + j];
        b[i] = val;
    }
}

```

```

double A[9] =
    { 0, 1, 2,
      4, 8, 16},
    { 32, 64, 128};

double B[3] = A+3;

sum_rows1(A, B, 3);

```

```

double A[9] =
    { 0, 1, 2,
      3, 27, 224},
    { 32, 64, 128};

```

Value of B:

```
init: [4, 8, 16]
```

```
i = 0: [3, 8, 16]
```

```
i = 1: [3, 27, 16]
```

```
i = 2: [3, 27, 224]
```

- Still changes A in the middle of the operation
- Different results

Avoiding Aliasing Penalties

```
/* Sum rows of n X n matrix a and store in vector b. */  
void sum_rows3(double *restrict a, double *restrict 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_rows3 inner loop  
.Loop:  
    addsd    (%rdi), %xmm0        # FP load + add  
    addq    $8, %rdi  
    cmpq    %rax, %rdi  
    jne     .Loop
```

- Use `restrict` qualifier to tell compiler that `a` and `b` cannot alias
- Less reliable than using local variables

Avoiding Aliasing Penalties

```

subroutine sum_rows4(a, b, n)
  implicit none
  integer, parameter :: dp = kind(1.d0)
  real(kind=dp), dimension(:), intent(in) :: a
  real(kind=dp), dimension(:), intent(out) :: b
  integer, intent(in) :: n
  integer :: i, j
  do i = 1,n
    b(i) = 0
    do j = 1,n
      b(i) = b(i) + a(i*n + j)
    end
  end
end

```

```

# sum_rows4 inner loop
.Loop:
    addsd    (%rdi), %xmm0          # FP load + add
    addq    $8, %rdi
    cmpq    %rax, %rdi
    jne     .Loop

```

- Use Fortran
- Array parameters in Fortran are assumed not to alias

Function calls are opaque

- **Compiler examines one function at a time**
 - Some exceptions for code in a single file
- **Must assume a function call could do anything**
- **Cannot usually**
 - move function calls
 - change number of times a function is called
 - cache data from memory in registers across function calls

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

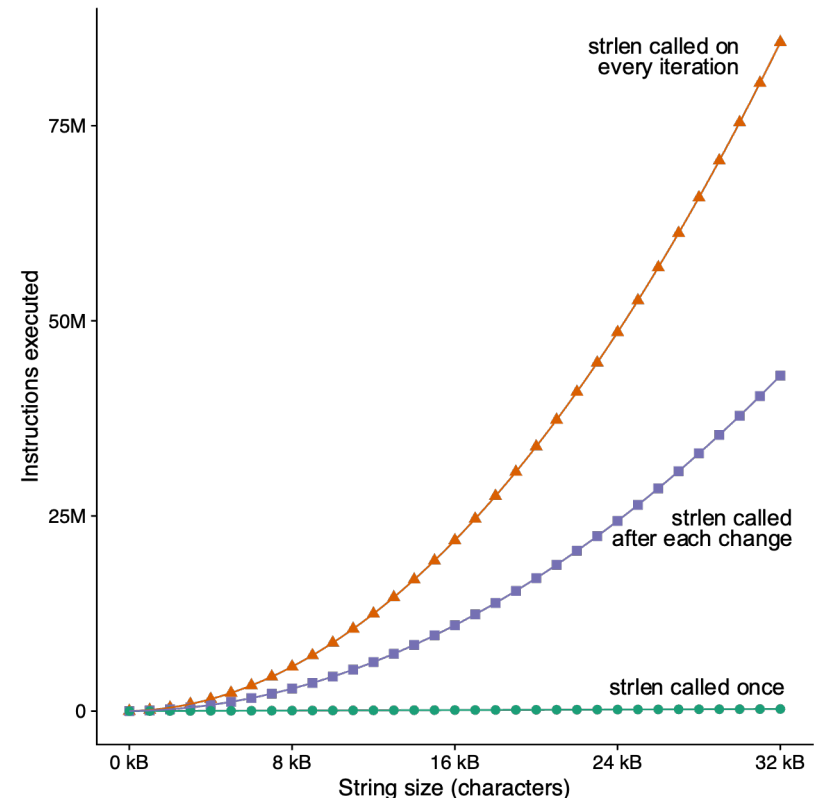
- **O(n) execution time**
- **Return value depends on:**
 - value of *s*
 - contents of memory at address *s*
 - Only cares about whether individual bytes are zero
 - Does not modify memory
- **Compiler might know *some* of that (but probably not)**

Can't move function calls out of loops

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

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

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



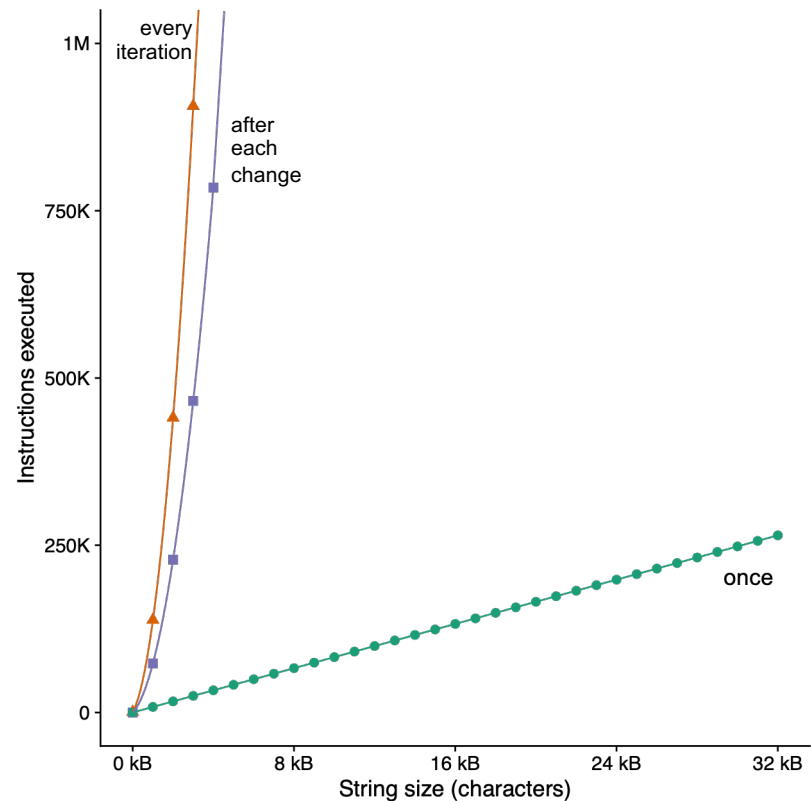
Lots more examples of this kind of bug:
accidentallyquadratic.tumblr.com

Can't move function calls out of loops

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

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

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



Non-associative arithmetic

- When is $(a \odot b) \odot c$ not equal to $a \odot (b \odot c)$?
 - Octonions
 - Vector cross product
 - *Floating-point numbers*
- **Example:** $a = 1.0$, $b = 1.5 \times 10^{38}$, $c = -1.5 \times 10^{38}$ (single precision IEEE fp)

$$a + b = 1.5 \times 10^{38} \quad (a + b) + c = 0$$

$$b + c = 0 \quad a + (b + c) = 1$$

- **Blocks any optimization that changes order of operations**

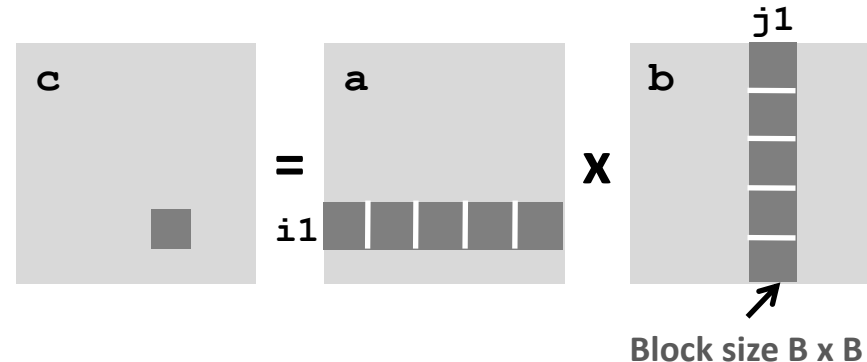
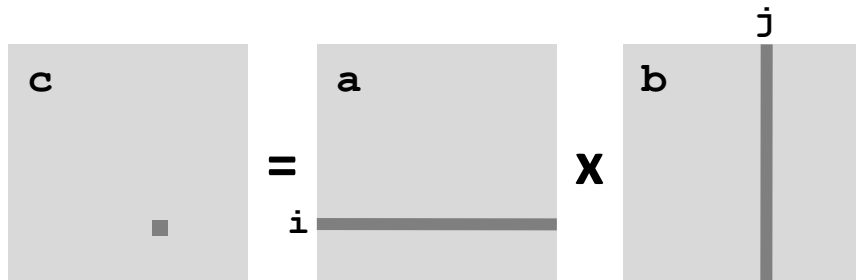
Non-associative arithmetic

```
void mmm(double *a, double *b,
         double *c, int n) {
    memset(c, 0, n*n*sizeof(double));

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

```
void mmm(double *a, double *b,
         double *c, int n) {
    memset(c, 0, n*n*sizeof(double));

    int i, j, k, i1, j1, k1;
    for (i = 0; i < n; i+=B)
        for (j = 0; j < n; j+=B)
            for (k = 0; k < n; k+=B)
                for (i1 = i; i1 < i+B; i1++)
                    for (j1 = j; j1 < j+B; j1++)
                        for (k1 = k; k1 < k+B; k1++)
                            c[i1*n+j1] += a[i1*n + k1]
                                * b[k1*n + j1];
}
```



Compiler cannot do this transformation automatically

Quiz Time!

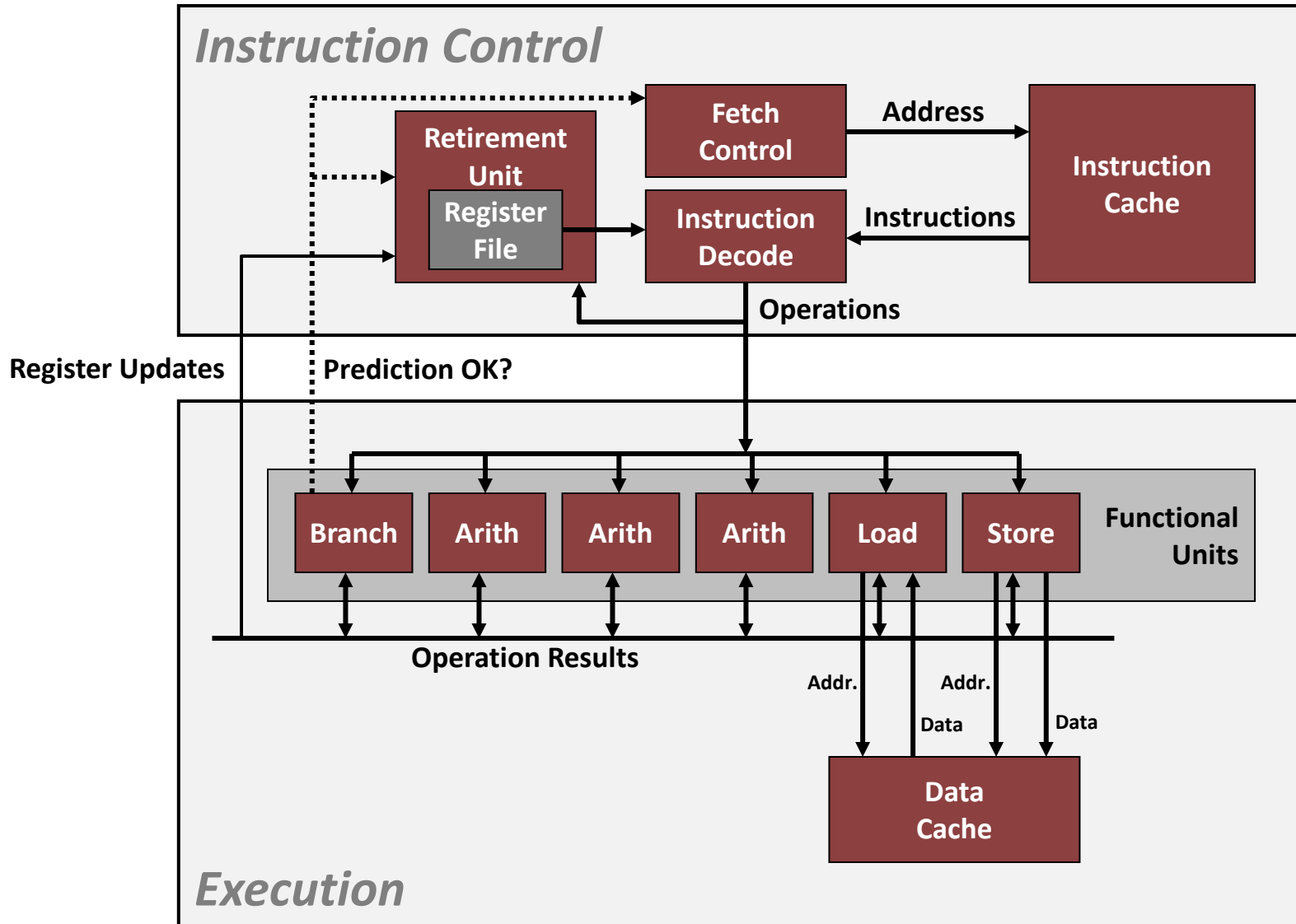
Check out:

<https://canvas.cmu.edu/courses/24383/quizzes/67220>

Today

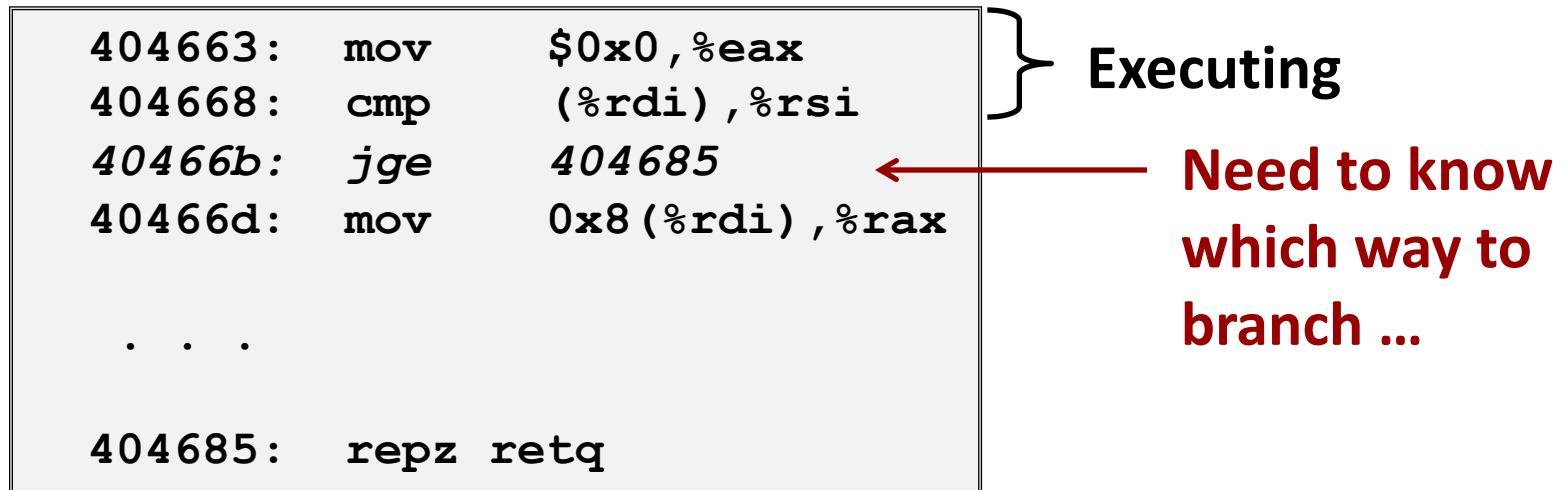
- **Principles and goals of compiler optimization**
- **Local optimization**
 - Constant folding, strength reduction, dead code elimination, common subexpression elimination
- **Global optimization**
 - Inlining, code motion, loop transformations
- **Obstacles to optimization**
 - Memory aliasing, procedure calls, non-associative arithmetic
- **Quiz**
- **Machine-dependent optimization**
 - Branch predictability, loop unrolling, scheduling, vectorization

Modern CPU Design



Branches Are A Challenge

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



If the CPU has to wait for the result of the `cmp` before continuing to fetch instructions, may waste tens of cycles doing nothing!

Branch Prediction

■ *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
```

Predict Taken

**Continue
Fetching
Here**

Branch Prediction Through Loop

```

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

```

i = 98

Assume
array length = **100**

Predict Taken (OK)

```

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

```

i = 99

Predict Taken
(Oops)

```

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

```

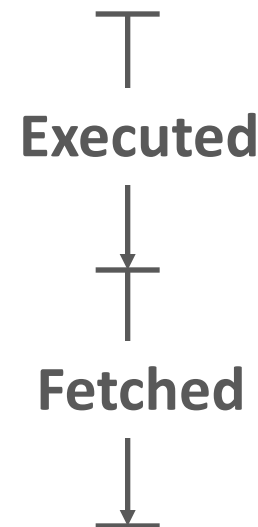
i = 100

Read
invalid
location

```

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

```

i = 101

Branch Misprediction Invalidation

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

i = 98

Assume
array length = 100

Predict Taken (OK)

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

i = 99

Predict Taken
(Oops)

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

i = 100

Invalidate

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

i = 101

Branch Misprediction Recovery

```

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

```

i = 99

Definitely not taken

Reload
Pipeline

■ Performance Cost

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

Branch Prediction Numbers

■ A simple heuristic:

- Backwards branches are often loops, so predict taken
- Forwards branches are often ifs, so predict not taken
- >95% prediction accuracy just with this!

■ Fancier algorithms track behavior of each branch

- Subject of ongoing research
- 2011 record (<https://www.jilp.org/jwac-2/program/JWAC-2-program.htm>): 34.1 mispredictions per 1000 instructions
- Current research focuses on the remaining handful of “impossible to predict” branches (strongly data-dependent, no correlation with history)
 - e.g. https://hps.ece.utexas.edu/pub/PruettPatt_BranchRunahead.pdf

Optimizing for Branch Prediction

■ Reduce # of branches

- Transform loops
- Unroll loops
- Use conditional moves
 - Not always a good idea

■ Make branches predictable

- Sort data
 - <https://stackoverflow.com/questions/11227809>
- Avoid indirect branches
 - function pointers
 - virtual methods

```
.Loop:
    movzbl 0(%rbp,%rbx), %edx
    leal   -65(%rdx), %ecx
    cmpb   $25, %cl
    ja     .Lskip
    addl   $32, %edx
    movb   %dl, 0(%rbp,%rbx)
.Lskip:
    addl   $1, %rbx
    cmpq   %rax, %rbx
    jb     .Loop
```

```
.Loop:
    movzbl 0(%rbp,%rbx), %edx
    movl   %edx, %esi
    leal   -65(%rdx), %ecx
    addl   $32, %edx
    cmpb   $25, %cl
    cmova  %esi, %edx
    movb   %dl, 0(%rbp,%rbx)
    addl   $1, %rbx
    cmpq   %rax, %rbx
    jb     .Loop
```

Memory write
now
unconditional!

Loop Unrolling

- Amortize cost of loop condition by duplicating body
- Creates opportunities for CSE, code motion, scheduling
- Prepares code for vectorization
- Can hurt performance by increasing code size

```
for (size_t i = 0; i < nelts; i++) {  
    A[i] = B[i]*k + C[i];  
}
```

```
for (size_t i = 0; i < nelts - 4; i += 4) {  
    A[i] = B[i]*k + C[i];  
    A[i+1] = B[i+1]*k + C[i+1];  
    A[i+2] = B[i+2]*k + C[i+2];  
    A[i+3] = B[i+3]*k + C[i+3];  
}
```

When would this change be incorrect?

Scheduling

- Rearrange instructions to make it easier for the CPU to keep all functional units busy
- For instance, move all the loads to the top of an unrolled loop
 - Now maybe it's more obvious why we need lots of registers

```
for (size_t i = 0; i < nelts - 4; i += 4) {
  A[i ] = B[i ]*k + C[i ];
  A[i+1] = B[i+1]*k + C[i+1];
  A[i+2] = B[i+2]*k + C[i+2];
  A[i+3] = B[i+3]*k + C[i+3];
}
```

```
for (size_t i = 0; i < nelts - 4; i += 4) {
  B0 = B[i]; B1 = B[i+1]; B2 = B[i+2]; B3 = B[i+3];
  C0 = C[i]; C1 = C[i+1]; C2 = C[i+2]; C3 = C[i+3];
  A[i ] = B0*k + C0;
  A[i+1] = B1*k + C1;
  A[i+2] = B2*k + C2;
  A[i+3] = B3*k + C3;
}
```

When would *this* change be incorrect?

Vectorization

- **Use special instructions that operate on several array elements at once**
 - Often called “SIMD” for “Single Instruction Multiple Data”
 - Invented in 1966 for ILLIAC IV supercomputer
 - Valuable for audio and video processing; has become ubiquitous

```
for (size_t i = 0; i < nelts - 4; i += 4) {
    B0 = B[i]; B1 = B[i+1]; B2 = B[i+2]; B3 = B[i+3];
    C0 = C[i]; C1 = C[i+1]; C2 = C[i+2]; C3 = B[i+3];
    A[i  ] = B0*k + C0;
    A[i+1] = B1*k + C1;
    A[i+2] = B2*k + C2;
    A[i+3] = B3*k + C3;
}
```

```
kkkk = _mm_set_ps1(k);
for (size_t i = 0; i < nelts - 4; i += 4) {
    B0123 = _mm_load_ps(&B[i]);
    C0123 = _mm_load_ps(&C[i]);
    A0123 = _mm_fmadd_ps(B0123, kkkk, C0123);
    _mm_store_ps(&A[i], A0123);
}
```

Summary: Getting High Performance

- **Good compiler and flags**
- **Don't do anything sub-optimal**
 - 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