Code Optimization

15-213/15-513/14-513: Introduction to Computer Systems
11th Lecture, June 14, 2022

Today

- Eberly Center early course feedback

- Principles and goals of compiler optimization
- Examples of optimizations
- Obstacles to optimization
- Tangent: branch prediction
- Troubleshooting the optimizer

Early Course Feedback

- Researchers from the Eberly Center are here to interview you about what's working well and not so well in this course.

- First half hour
- Faculty and TAs will leave the room
- We only get a summary afterward
- No obligation to participate

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.

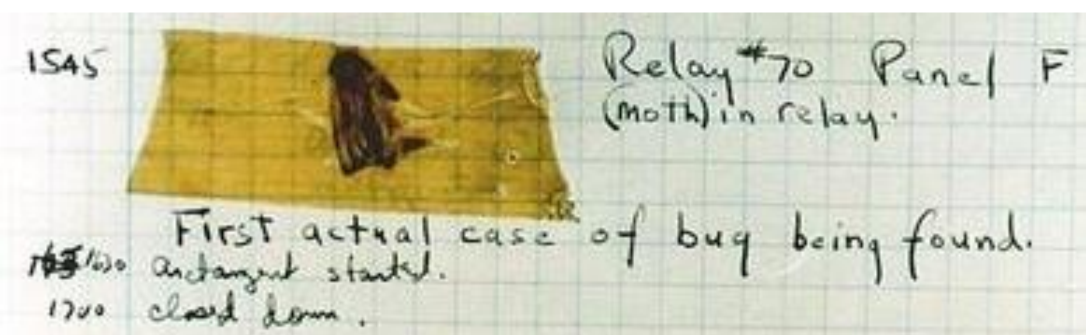
Machine Code.

Raw, unadorned, inscrutable hexadecimal numbers. Directly.

— "The Story of Mel, a Real Programmer"
Ed Nather, 1983

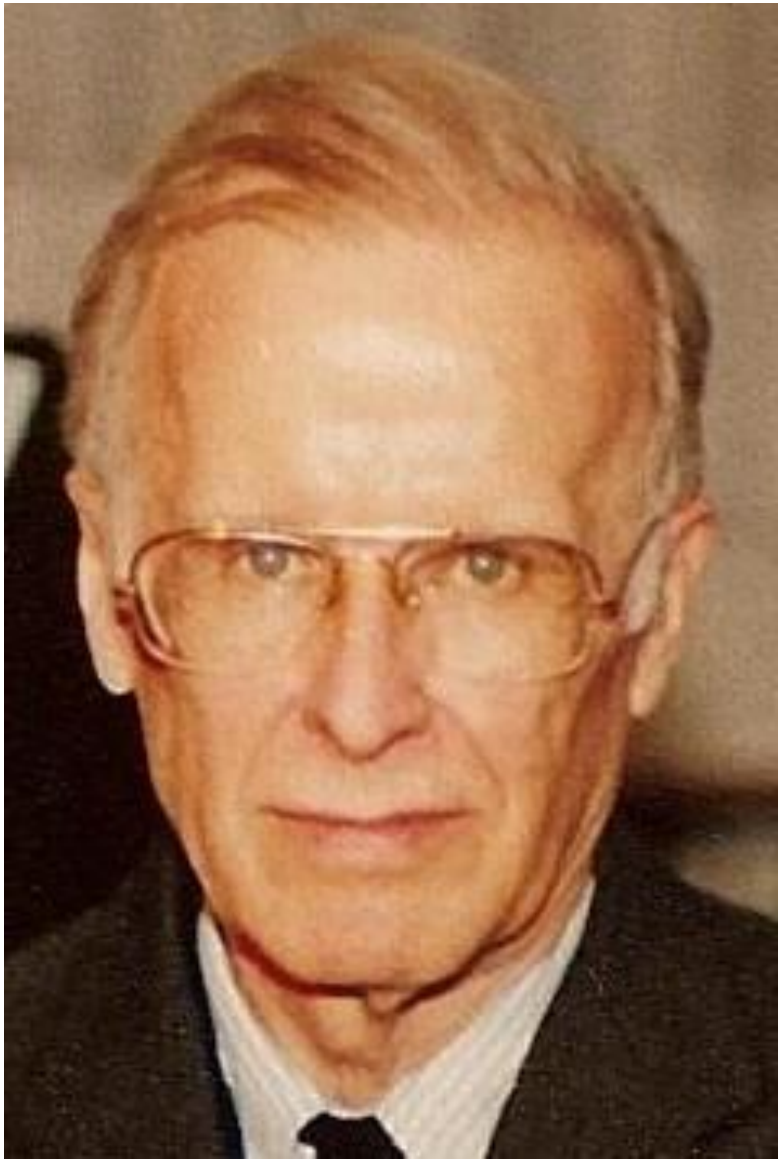
Rear Admiral Grace Hopper

- 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

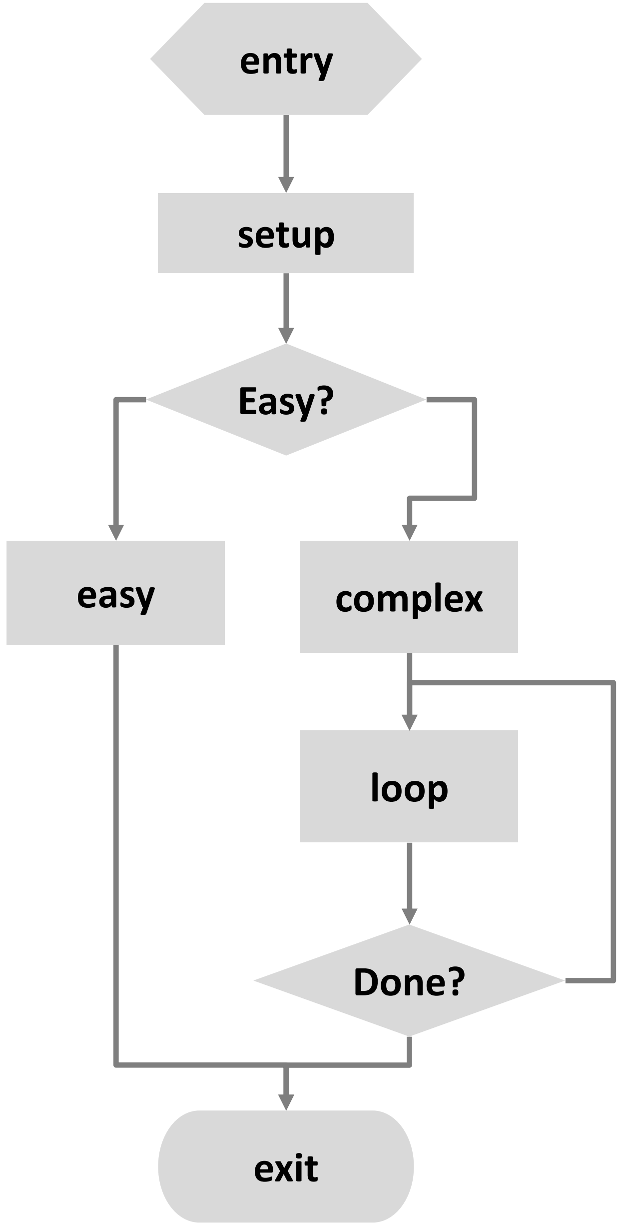
- **Minimize number of instructions**
  - Don't do calculations more than once
  - Don't do unnecessary calculations at all
  - Avoid slow instructions (multiplication, division)
- **Avoid waiting for memory**
  - Keep everything in registers whenever possible
  - Access memory in cache-friendly patterns
  - 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)

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

- Eberly Center early course feedback

- Principles and goals of compiler optimization
- Examples of optimizations
- Obstacles to optimization
- Tangent: branch prediction
- Troubleshooting the optimizer

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;
```

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;
```

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];
```

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;
}
```

Today

- Eberly Center early course feedback

- Principles and goals of compiler optimization
- Examples of optimizations
- Obstacles to optimization
- Tangent: branch prediction
- Troubleshooting the optimizer

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];
    }
}
```

```
        movq    $0, (%rsi)
        pxor    %xmm0, %xmm0
.L4:
        addsd   (%rdi), %xmm0
        movsd   %xmm0, (%rsi)
        addq    $8, %rdi
        cmpq    %rcx, %rdi
        jne     .L4
```

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

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;
    }
}
```

```
        pxor    %xmm0, %xmm0
.L4:
        addsd   (%rdi), %xmm0
        addq    $8, %rdi
        cmpq    %rax, %rdi
        jne     .L4
        movsd   %xmm0, (%rsi)
```

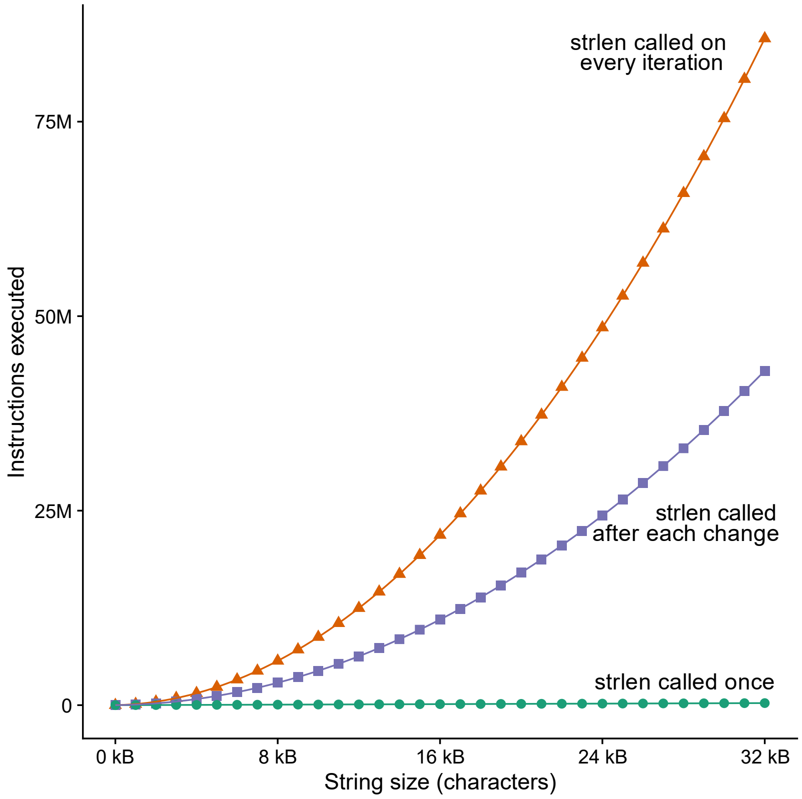
- Use a local variable for intermediate results

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';
}
```



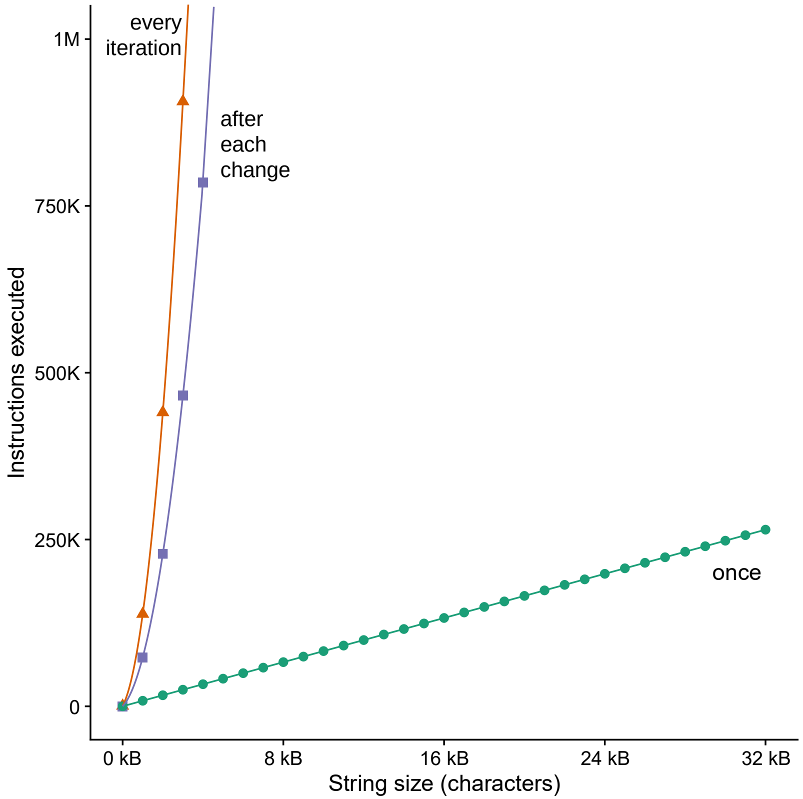
Lots more examples of this kind of bug: accidentallyquadratic.tumblr.com

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';
}
```

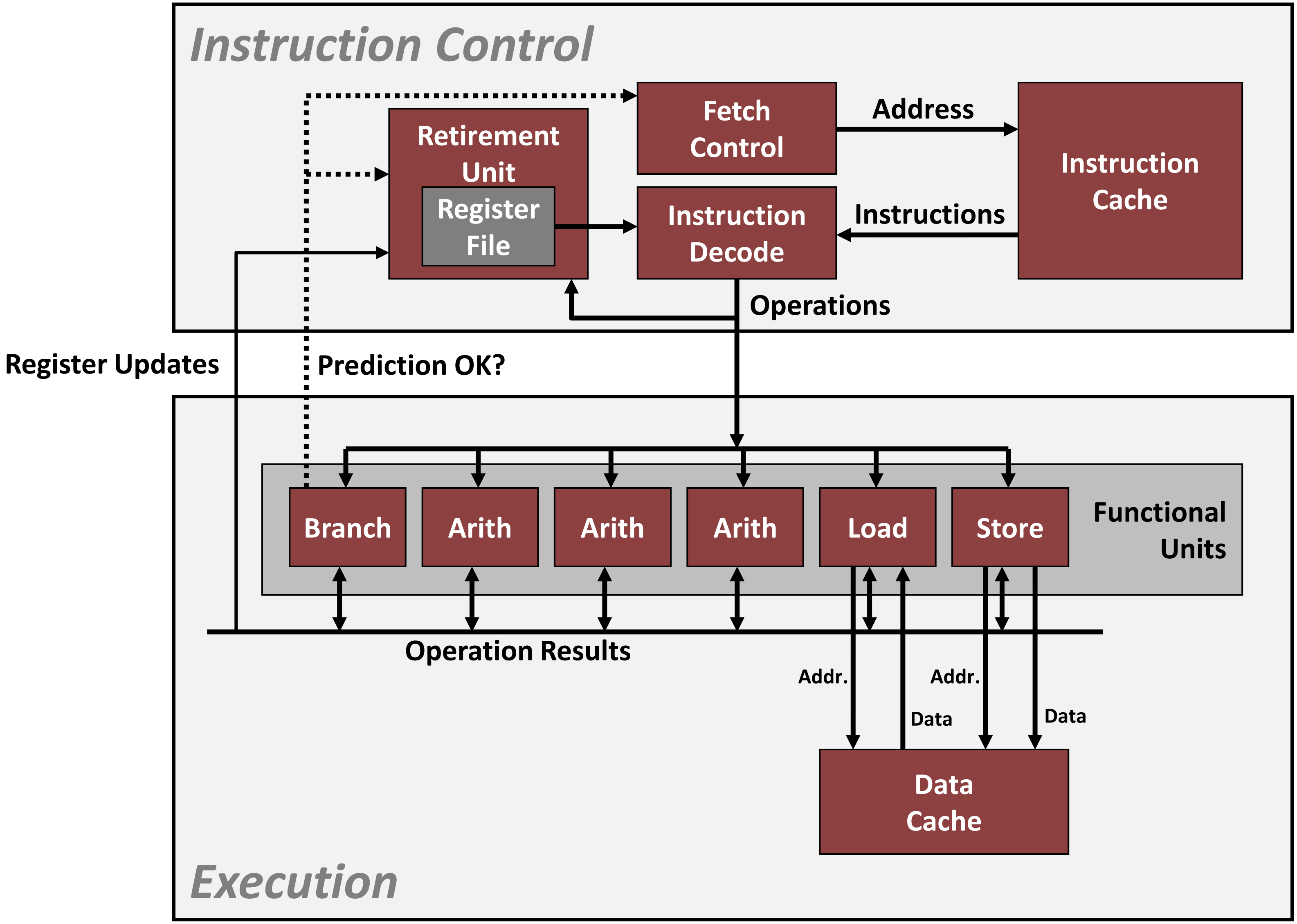


Today

- Eberly Center early course feedback

- Principles and goals of compiler optimization
- Examples of optimizations
- Obstacles to optimization
- Tangent: branch prediction
- Troubleshooting the optimizer

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

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

 . . .

404685:  repz retq
```

Executing

Need to know which way to branch …

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

Executed

```
401029:   mulsd   (%rdx),%xmm0,%xmm0
40102d:   add     $0x8,%rdx
401031:   cmp     %rax,%rdx
401034:   jne     401029
```
i = 101

Fetched

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

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

i = 101

Invalidate

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!

Today

- Eberly Center early course feedback

- Principles and goals of compiler optimization
- Examples of optimizations
- Obstacles to optimization
- Tangent: branch prediction
- **Troubleshooting the optimizer**