COMP2310/COMP6310
Systems, Networks, & Concurrency

Convener: Shoaib Akram

Code Optimization – 1

Acknowledgement of material: With changes suited to ANU needs, the slides are obtained from Carnegie Mellon University: https://www.cs.cmu.edu/~213/

Today

- Principles and goals of compiler optimization
- Examples of optimizations

Goals of compiler optimization

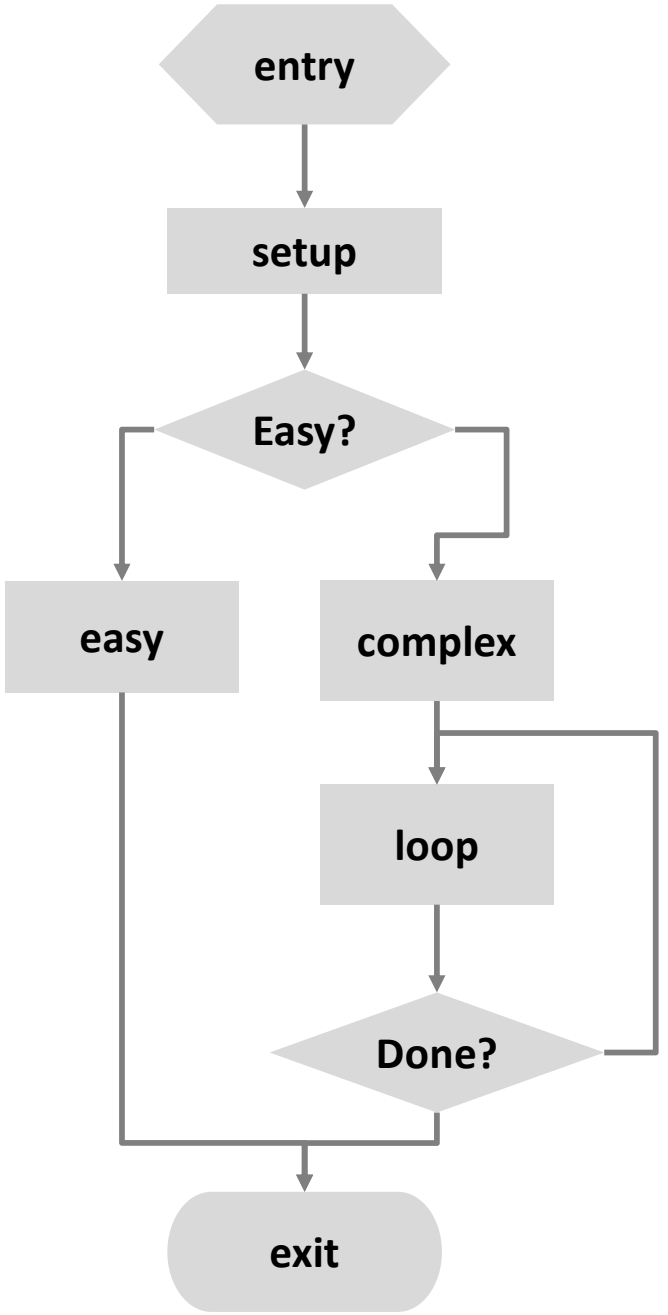
- **Minimize number of instructions**
  - Don’t do calculations more than once
  - Don’t do unnecessary calculations at all
  - Avoid slow instructions (multiplication, division)
- **Avoid waiting for memory**
  - Keep everything in registers whenever possible
  - Access memory in cache-friendly patterns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

- Principles and goals of compiler optimization
- **Examples of optimizations**

Try it yourself

- https://godbolt.org/z/Es5s8qsvj

- **Go to Godbolt (the compiler explorer) to play around with C and the resulting assembly generated under different compiler optimizations (change the flag from –O3 to –Og, etc. to see more or less aggressive optimization).**

- **Read descriptions of optimization levels**
  - https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

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;
}
```

More on Optimization

- **We will have another lecture on optimization after understanding memory and caches**