Many of the following slides are taken with permission from

Complete Powerpoint Lecture Notes for
Computer Systems: A Programmer's Perspective (CS:APP)

Randal E. Bryant and David R. O'Hallaron

http://csapp.cs.cmu.edu/public/lectures.html

The book is used explicitly in CS 2505 and CS 3214 and as a reference in CS 2506.

Many other slides were based on notes written by Dr Godmar Back for CS 3214.

Asymptotic Complexity

“Big-O” O(…), Θ(…)

- Describes asymptotic behavior of time or space cost function of an algorithm as input size grows
- Subject of complexity analysis (CS 3114)
- Determine if a problem is tractable or not

Example:

- Quicksort – O(n log n) average case
- Bubble Sort – O(n^2) average case
  - Actual cost may be C_1 * N^2 + C_2 * N + C_3

These constants can matter and optimization can reduce them

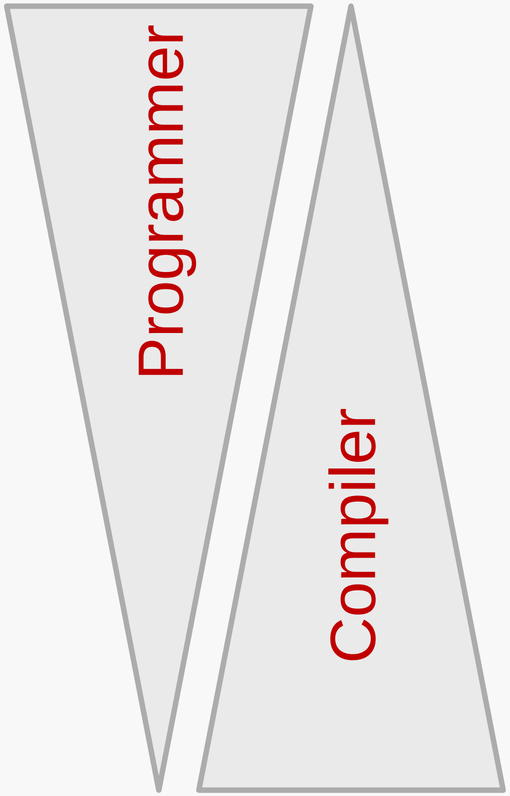
- Determine how big of a problem a tractable algorithm can handle in a concrete implementation on a given machine

Roles of Programmer vs Compiler

Programmer:

- Choice of algorithm, Big-O
- Manual application of some optimizations
- Choice of program structure that's amenable to optimization
- Avoidance of "optimization blockers"

High-Level



Low-Level

Optimizing Compiler

- Applies transformations that preserve semantics, but reduce amount of, or time spent in computations
- Provides efficient mapping of code to machine:
  - Selects and orders code
  - Performs register allocation
- Usually consists of multiple stages

Controlling Optimization with gcc

-O0 (“O zero”)

- This is the default: minimal optimizations

-O1

- Apply optimizations that can be done quickly

-O2

- Apply more expensive optimizations. That’s a reasonable default for running production code. Typical ratio between –O2 and –O0 is 5-20.

-O3

- Apply even more expensive optimizations

-Os

- Optimize for code size

See ‘info gcc’ for list which optimizations are enabled when; note that –f switches may enable additional optimizations that are not included in –O

Note: ability to debug code symbolically under gdb decreases with optimization level; usually use –O0 –g or –O1 –g or –ggdb3

Limitations of Optimizing Compilers

Fundamentally, must emit code that implements specified semantics under *all* conditions

- Can’t apply optimizations even if they would only change behavior in corner case a programmer may not think of
- Due to memory aliasing
- Due to unseen procedure side-effects

Do not see beyond current compilation unit

Intraprocedural analysis typically more extensive (since cheaper) than interprocedural analysis

Usually base decisions on static information

Types of Optimizations

Copy Propagation

Code Motion

Strength Reduction

Common Subexpression Elimination

Eliminating Memory Accesses

- Through use of registers

Inlining

Copy Propagation

```
int arith1(int x, int y, int z)
{
    int x_plus_y = x + y;
    int x_minus_y = x - y;
    int x_plus_z = x + z;
    int x_minus_z = x - z;
    int y_plus_z = y + z;
    int y_minus_z = y - z;
    int xy_prod = x_plus_y * x_minus_y;
    int xz_prod = x_plus_z * x_minus_z;
    int yz_prod = y_plus_z * y_minus_z;

    return xy_prod + xz_prod + yz_prod;
}
```

```
int arith2(int x, int y, int z)
{
    return (x + y) * (x - y)
         + (x + z) * (x - z)
         + (y + z) * (y - z);
}
```

Which produces faster code?

Copy Propagation

```
arith1:
    leal    (%rdx,%rdi), %ecx
    movl    %edi, %eax
    subl    %edx, %eax
    imull   %ecx, %eax
    movl    %esi, %ecx
    subl    %edx, %ecx
    addl    %esi, %edx
    imull   %edx, %ecx
    movl    %edi, %edx
    subl    %esi, %edx
    addl    %edi, %esi
    imull   %esi, %edx
    addl    %ecx, %eax
    addl    %edx, %eax
    ret
```

```
arith2:
    leal    (%rdx,%rdi), %ecx
    movl    %edi, %eax
    subl    %edx, %eax
    imull   %ecx, %eax
    movl    %esi, %ecx
    subl    %edx, %ecx
    addl    %esi, %edx
    imull   %edx, %ecx
    movl    %edi, %edx
    subl    %esi, %edx
    addl    %edi, %esi
    imull   %esi, %edx
    addl    %ecx, %eax
    addl    %edx, %eax
    ret
```

Constant Propagation

```
#include <stdio.h>

int sum(int a[], int n)
{
    int i, s = 0;
    for (i = 0; i < n; i++)
        s += a[i];
    return s;
}

int main()
{
    int v[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    int s = sum(v, 10);

    printf("Sum is %d\n", s);
}
```

```
.LC0:
    .string "Sum is %d\n"
main:
     …
    movl    $55, 4(%esp)
    movl    $.LC0, (%esp)
    call    printf
```

Code Motion

Do not repeat computations if result is known

Usually out of loops (“code hoisting”)

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

→

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

Strength Reduction

Substitute lower cost operation for more expensive one

- E.g., replace 48*x with (x << 6) – (x << 4)
- Often machine dependent

Common Subexpression Elimination

Reuse already computed expressions

```
/* 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

```
int 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

Eliminating Memory Accesses, Take 1

Register accesses are faster than memory accesses

```
int sp1(int *px, int *py)
{
   int sum =   *px * *px + *py * *py;
   int diff =  *px * *px - *py * *py;
   return sum * diff;
}
```

8 pointer dereferences

```
sp1:
   movl    (%rdi), %ecx          # eax = *px
   movl    (%rsi), %edx          # edx = *py
   imull   %ecx, %ecx
   imull   %edx, %edx
   leal    (%rcx,%rdx), %eax   # no access
   subl    %edx, %ecx
   imull   %ecx, %eax
   ret
```

2 memory accesses

Number of memory accesses at runtime not determined by how often pointer dereferences occur in source code

Eliminating Memory Accesses, Take 2

```
void sp2(int *px, int *py,
         int *psum, int *pprod) {

   *psum =  *px + *py;
   *pprod = *px * *py;
}
```

6 pointer dereferences

```
sp2:
   movl    (%rdi), %eax    # eax = *px
   addl    (%rsi), %eax    # eax = eax + *py
   movl    %eax, (%rdx)    # *psum = eax
   movl    (%rdi), %eax    # eax = *px
   imull   (%rsi), %eax    # eax = eax * *py
   movl    %eax, (%rcx)    # *pprod = eax
   ret
```

6 memory accesses

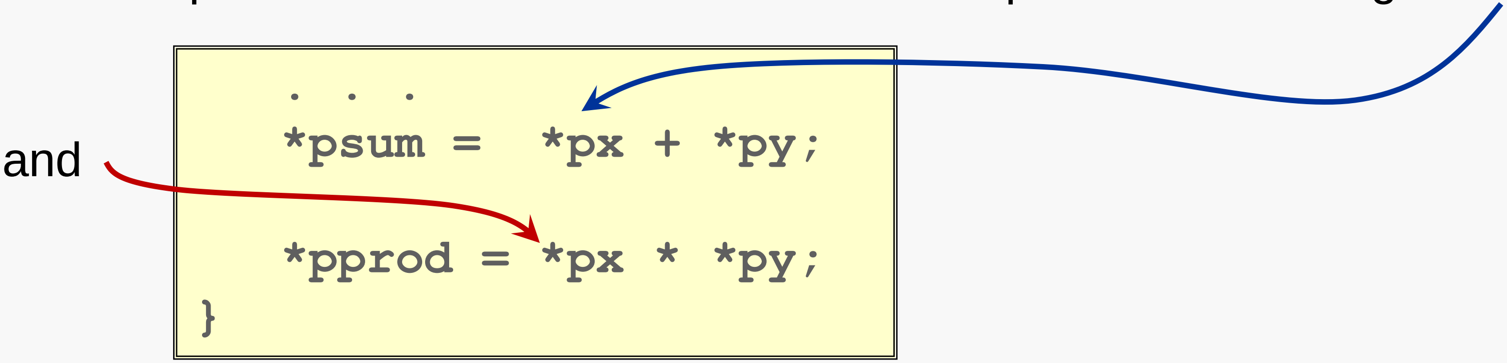
… why were *px and *py each loaded twice?

Optimization Blocker: Pointer Aliasing

```
void sp2(int *px, int *py,
         int *psum, int *pprod) {

   *psum =  *px + *py;
   *pprod = *px * *py;
}
```

The compiler cannot assume that the value of *px does not change between



and

```
    . . .
    *psum =  *px + *py;

    *pprod = *px * *py;
}
```

How could that happen?

What if px and psum pointed to the same variable in the caller's code?

Optimization Blocker: Pointer Aliasing

```
void sp2(int *px, int *py,
         int *psum, int *pprod) {

   *psum =  *px + *py;
   *pprod = *px * *py;
}
```

6 pointer dereferences

Suppose the compiler tried to eliminate memory accesses by doing this:

```
sp2:
   movl   (%rdi), %edi       # edi = *px
   movl   (%rsi), %eax       # eax = *py
   leal   (%rdi,%rax), %esi  # esi = *px + *py
   imull  %edi, %eax         # edi = *px * *py
   movl   %esi, (%rdx)       # *psum = esi
   movl   %eax, (%rcx)
   ret
```

4 memory accesses

Erroneous Optimization

The compiler translation shown on the previous slide is equivalent to this:

```
void sp2(int *px, int *py, int *psum, int *pprod) {

   int xtmp = *px;
   int ytmp = *py;

   *psum =  *px + *py;
   *pprod = *px * *py;
}
```

Suppose the caller wrote this:

```
   . . .
   int X = 10;
   int Y = 20;

   sp2(&X, &Y, &X, &Y);
   . . .
}
```

Code above:
  X = 10 and Y = 20

Original translation:
  X = 30 and Y = 300

Which is correct? Which was intended?

Limitations of Optimizing Compilers

Fundamentally, must emit code that implements specified semantics under *all* conditions

- Can’t apply optimizations even if they would only change behavior in corner case a programmer may not think of
- Due to memory aliasing
- Due to unseen procedure side-effects

Do not see beyond current compilation unit

Intraprocedural analysis typically more extensive (since cheaper) than interprocedural analysis

Usually base decisions on static information

Erroneous Optimization

This code:

```
void sp2(int *px, int *py, int *psum, int *pprod) {

   int xtmp = *px;
   int ytmp = *py;

   *psum =  *px + *py;
   *pprod = *px * *py;
}
```

… is NOT logically equivalent to this code:

```
void sp2(int *px, int *py, int *psum, int *pprod) {

   *psum =  *px + *py;
   *pprod = *px * *py;
}
```