

# Code Optimization - Lab 9

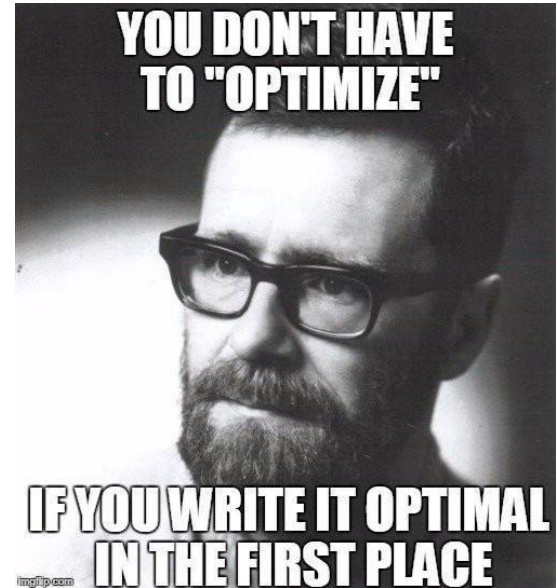
COMP201 Spring 2022



**KOÇ  
UNIVERSITY**

# What is Code Optimization?

- A program transformation technique to:
  - **improve** the intermediate code,
  - make it **consume fewer resources** (CPU, memory),
  - reduce the **size** of the code,
  - **speed up** execution.
- It is **not** optimizing an algorithm.
  - It is beyond our scope for now.



# Types of Code Optimization

- **Machine Independent Optimization**
- **Machine Dependent Optimization**

# Types of Code Optimization

## Machine Independent Optimization

- Improve the intermediate code to get a better target code.
- Does not involve any CPU registers, or absolute memory locations.

```
do{  
    item = 10;  
    value = value + item;  
}  
while (value<100) ;  
  
// this code involves repeated  
// assignment of 'item'. Instead:
```

# Types of Code Optimization

## Machine Independent Optimization

- Improve the intermediate code to get a better target code.
- Does not involve any CPU registers, or absolute memory locations.

```
do{
    item = 10;
    value = value + item;
}
while(value<100);

// this code involves repeated
// assignment of 'item'. Instead:

item = 10;
do{
    value = value + item;
}
while(value<100);
```

# Types of Code Optimization

## Machine Dependent Optimization

- **Goal:** Take maximum advantage of the memory hierarchy.
- After the target code is generated,
  - optimization is done according to the target machine architecture.
- Involves CPU registers,
- May have absolute memory references, rather than relative references

# Compiler Optimizations

- **GCC** supports automatic optimizations.
- Without any optimization option, the compiler's goal is to **reduce the cost of compilation** and to make debugging produce the expected results.
- Turning on optimization flags makes the compiler attempt to **improve the performance** and/or **code size** at the expense of compilation time and possibly the **ability to debug** the program.
- Most optimizations are completely disabled at `-O0` or if an `-O` level is not set on the command line, even if individual optimization flags are specified. Similarly, `-Og` suppresses many optimization passes.

# Compiler Optimizations

- **-O or -O1 option (Optimize).** the compiler tries to reduce code size and execution time, without performing any optimizations that take a great deal of compilation time.
- **-O2 (Optimize even more).** GCC performs nearly all supported optimizations that do not involve a space-speed tradeoff. As compared to -O, this option increases both compilation time and the performance of the generated code.
- **-O3 (Optimize yet more).** -O3 turns on all optimizations specified by -O2 and also more options such as loop unrolling and jamming.
- **-Os (Optimize for size).** -Os enables all -O2 optimizations except those that often increase code size.



# Machine Independent Techniques

- Techniques are various and vast
- Be careful about time you spend on optimization
- With practice, you can write your codes optimized in the first place and optimize it further after having a base code
- Profiling is an invaluable tool
- In practice you must re-use already existing optimized codes
- Compilers offer various optimizations

# Inlining

- C functions can be recoded as macros
  - to obtain similar speedup on compilers with no inlining capability.
- This should be done after the code is completely debugged.
- No function call
  - Fewer instructions!

```
int foo(a, b)
{
    a = a - b;
    b++;
    a = a * b;
    return a;
}
```

Can be replaced by:

```
#define foo(a, b) (((a)-(b)) * ((b)+1))
```

# Avoid Pointer Dereference in Loop

- Pointer dereferencing creates lots of trouble in memory. So better assign it to some temporary variable and then use that temporary variable in the loop.

```
int a = 0;
int* iptr = &a;

for (int i = 1; i < 11; ++i) {
    *iptr = *iptr + i;
}
```

```
int a = 0;
int* iptr = &a;
// Dereferencing pointer outside loop & use temp var
int temp = *iptr;
for (int i = 1; i < 11; ++i) {
    temp = temp + i;
}
// Updating pointer using final value of temp
*iptr = temp;
```

# Loop Unrolling

- `gcc -funroll-loops` will do this.  
But if you know that yours doesn't, you can change your source code a bit to get the same effect.
- This way the test for `i < 100` and the branch back up to the top of the loop only get executed 21 times rather than 101.

```
for (i = 0; i < 100; i++){  
    do_stuff(i);  
}
```

Can be replaced by:

```
for (i = 0; i < 100; ){  
    do_stuff(i); i++;  
    do_stuff(i); i++;  
    do_stuff(i); i++;  
    do_stuff(i); i++;  
    do_stuff(i); i++;  
}
```

# Loop Unrolling Caveat

- An unrolled loop is larger than the "rolled" version.
  - So, **may no longer fit** into the instruction cache (on machines which have them).
  - This will make the unrolled version slower.
- Also, in this example, the **call** to `do_stuff()` **overshadows** the **cost of the loop**.
  - So any savings from loop unrolling are insignificant in comparison to what you'd **achieve from inlining** in this case.

```
for (i = 0; i < 100; i++){  
    do_stuff(i);  
}
```

Can be replaced by:

```
for (i = 0; i < 100; ){  
    do_stuff(i); i++;  
    do_stuff(i); i++;  
    do_stuff(i); i++;  
    do_stuff(i); i++;  
    do_stuff(i); i++;  
}
```

# Code Motion

- Reduce frequency with which computation performed if it will always produce same result.
- Especially moving code out of loop

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

Can be replaced by:

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

# Share Common Subexpressions

```
/* 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



```
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
```

- Especially problematic if this function is getting called inside a loop

M <sub>0,0</sub>	M <sub>1,0</sub>	M <sub>2,0</sub>	M <sub>3,0</sub>
M <sub>0,1</sub>	M <sub>1,1</sub>	M <sub>2,1</sub>	M <sub>3,1</sub>
M <sub>0,2</sub>	M <sub>1,2</sub>	M <sub>2,2</sub>	M <sub>3,2</sub>
M <sub>0,3</sub>	M <sub>1,3</sub>	M <sub>2,3</sub>	M <sub>3,3</sub>

M



M <sub>0,0</sub>	M <sub>1,0</sub>	M <sub>2,0</sub>	M <sub>3,0</sub>	M <sub>0,1</sub>	M <sub>1,1</sub>	M <sub>2,1</sub>	M <sub>3,1</sub>	M <sub>0,2</sub>	M <sub>1,2</sub>	M <sub>2,2</sub>	M <sub>3,2</sub>	M <sub>0,3</sub>	M <sub>1,3</sub>	M <sub>2,3</sub>	M <sub>3,3</sub>
------------------	------------------	------------------	------------------	------------------	------------------	------------------	------------------	------------------	------------------	------------------	------------------	------------------	------------------	------------------	------------------

# Share Common Subexpressions

```
/* Sum neighbors of i,j */
```

```
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



```
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
```

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

M <sub>0,0</sub>	M <sub>1,0</sub>	M <sub>2,0</sub>	M <sub>3,0</sub>
M <sub>0,1</sub>	M <sub>1,1</sub>	M <sub>2,1</sub>	M <sub>3,1</sub>
M <sub>0,2</sub>	M <sub>1,2</sub>	M <sub>2,2</sub>	M <sub>3,2</sub>
M <sub>0,3</sub>	M <sub>1,3</sub>	M <sub>2,3</sub>	M <sub>3,3</sub>

M

M <sub>0,0</sub>	M <sub>1,0</sub>	M <sub>2,0</sub>	M <sub>3,0</sub>	M <sub>0,1</sub>	M <sub>1,1</sub>	M <sub>2,1</sub>	M <sub>3,1</sub>	M <sub>0,2</sub>	M <sub>1,2</sub>	M <sub>2,2</sub>	M <sub>3,2</sub>	M <sub>0,3</sub>	M <sub>1,3</sub>	M <sub>2,3</sub>	M <sub>3,3</sub>
------------------	------------------	------------------	------------------	------------------	------------------	------------------	------------------	------------------	------------------	------------------	------------------	------------------	------------------	------------------	------------------



# Reduction in Strength

- Replace costly operations with simpler ones
- E.g. replace Shift/add with multiply/divide
  - E.g.  $16 * x \rightarrow x \ll 4$

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

Can be replaced by:

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

# Loop Jamming

- Combine adjacent loops which loop over the same range of the same variable.
- Incrementing and testing of *i* is done only half as often
- Assuming nothing in the second loop indexes forward
  - e.g `array[i+3]`.

```
for (i = 0; i < MAX; i++)      /* initialize 2d array to 0's */
    for (j = 0; j < MAX; j++)
        a[i][j] = 0.0;

for (i = 0; i < MAX; i++)      /* put 1's along the diagonal */
    a[i][i] = 1.0;
```

Can be replaced by:

```
for (i = 0; i < MAX; i++){
    for (j = 0; j < MAX; j++)
        a[i][j] = 0.0;      /* initialize 2d array to 0's */
    a[i][i] = 1.0;          /* put 1's along the diagonal */
}
```

# Loop Inversion

- Some machines have a special instruction for decrement and compare with 0.
- Assuming the loop is insensitive to direction
- **Positive** values interprets as **True** while **negatives** interpret as **False**

```
for (i = 1; i < MAX; i++){  
    ...  
}
```

Can be replaced by:

```
for (i = MAX; i--; ){  
    ...  
}
```

# Table Lookup

- Consider using lookup tables especially if a computation is **iterative or recursive**.
  - e.g. convergent series or factorial.
- If the table is too large to type, you can have some initialization code compute all the values on startup

```
long factorial(int i){  
    if (i == 0)  
        return 1;  
    else  
        return i * factorial(i - 1);  
}
```

Can be replaced by:

```
static long factorial_table[] =  
    {1, 1, 2, 6, 24, 120, 720 /* etc */};  
long factorial(int i){  
    return factorial_table[i];  
}
```

# Stack Usage

- A typical cause of stack-related problems is having large arrays as local variables.
- In that case the solution is to rewrite the code so it can use a static or global array, or perhaps allocate it from the heap.
- Similar solution applies to functions which have large structs as locals or parameters.

# Recap: Struct Padding

- Generally when a struct is stored in RAM, it is padded to correspond to the word-size of the architecture of the CPU.
- Additional padding is provided for arrays to make the first bytes of each item in the array a multiple of the item size.

```
/* Assume 32 bit Architecture  
   Sizeof foo = 12 bytes */
```

```
struct foo{  
    char c;  
    int x;  
    short s;  
};
```

0 c	1	2	3 padding
4	5	6	7 x
8	9 s	10	11 padding

# Reduce Padding

- You can save a tiny amount of space by arranging similarly-typed fields together in a structure
  - with the most restrictively aligned types first.
- A typical use of `char` or `short` variables is to hold a flag or mode bit.
- You can combine several of these flags into one byte using bit-fields at the cost of data portability.

```
/* sizeof = 64 bytes
*/
```

```
struct foo {
    float    a;
    double   b;
    float    c;
    double   d;
    short    e;
    long     f;
    short    g;
    long     h;
    char     i;
    int      j;
    char     k;
    int      l;
};
```

```
/* sizeof = 48 bytes
*/
```

```
struct foo {
    double   b;
    double   d;
    long     f;
    long     h;
    float    a;
    float    c;
    int      j;
    int      l;
    short    e;
    short    g;
    char     i;
    char     k;
};
```

# References

<https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html>

<http://www.cs.cmu.edu/afs/cs/academic/class/15213-s19/www/schedule.html>