COMP 201

Fall 2025

KOÇ UNIVERSITY

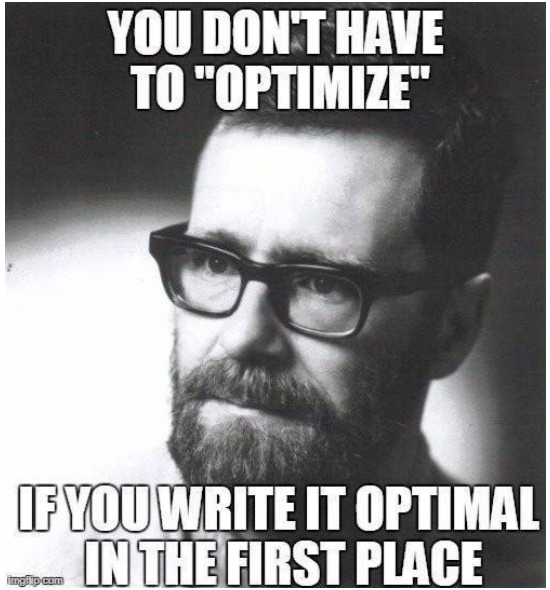
Lab 9

Code Optimization

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:** Exploit hardware features.
- After the target code is generated,
  - optimization is done according to the target machine architecture.
- Involves use of special instructions,
- Allocating CPU registers,
- May have absolute memory references, rather than relative references

  etc.

Compiler Optimizations

- **GCC** applies both machine independent and dependent 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 decrease the **code size** at the expense of compilation time and possibly the **ability to debug** the program.
- Most optimizations are completely disabled at `-O0` (or equivalently when `-O` level is not set), 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 should reuse already existing optimized codes

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. It's better to 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;
```

Avoid Pointer Dereference in Loop

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

```
struct Warrior{
      double damage; double HP;
      ...
};
void main(){
      struct Warrior* hero1;
      struct Warrior* enemies[n];
      ...
      for (int i = 0; i < n; ++i) {
           enemies[i]->HP -= hero1->damage;
      }
}
```

```
void main(){
      struct Warrior* hero1;
      struct Warrior* enemies[n];
      ...
      double damage = hero1->damage;
      for (int i = 0; i < n; ++i) {
           enemies[i]->HP -= damage;
      }
}
```

KOÇ UNIVERSITY

Loop Unrolling

- `gcc -funroll-loops` will do this. But if you know that yours doesn't, you can modify your code to get the same effect.

- This way, the loop condition (`i < 100`) and the backward branch are executed only 21 times instead of 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 the machines which have them).
  - This will make the unrolled version slower.

- In general, **function call** overhead is much **larger** than **loop-control** overhead.
  - 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

- Avoid recomputing expressions that produce the same result each iteration.
- In particular, move loop-invariant code outside the 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];

}
```

KOÇ UNIVERSITY

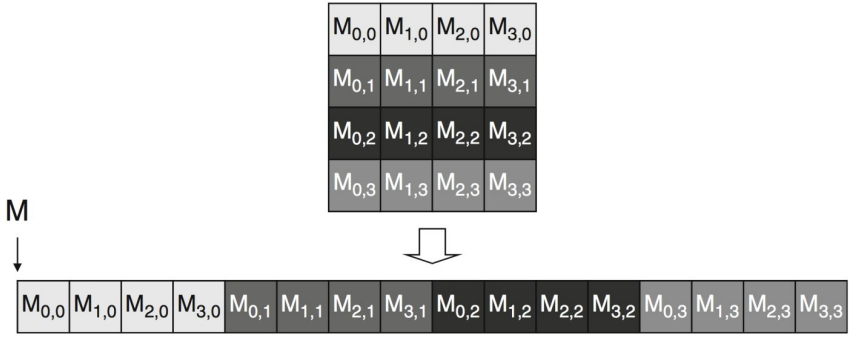
Share Common Subexpressions

```
/* 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



Share Common Subexpressions

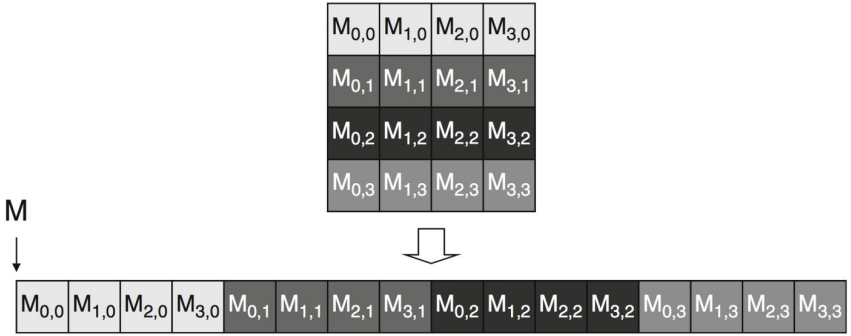
```
/* Sum neighbors of i,j */
long inj = i*n + j;
up = val[inj - n];
down = val[inj + n];
left = val[inj - 1];
right = val[inj + 1];
sum = up + down + left +
right;
```

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



KOÇ UNIVERSITY

Reduction in Strength

- Replace costly operations with simpler ones
- E.g. replace Shift/add with multiply/divide
    - E.g `x << 4; // is equivalent to x * 16;`

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
- Provided the second loop does not access elements from future iterations
  - 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 architectures provide an efficient *decrement-and-branch-if-not-zero* instruction.
- Counting down to zero can reduce instruction count and branches
- Assuming the loop is insensitive to direction

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

Can be replaced by:

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

Table Lookup

- Use a lookup table when the input domain is **small/bounded** and values are **reused** often
  - e.g. small factorial range
- Replaces computation with an array index (time ↔ memory tradeoff)
- Tables can be **static (compile-time)** or **initialized at startup** (runtime precompute)

```
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];
}
```

KOÇ UNIVERSITY

Stack Usage

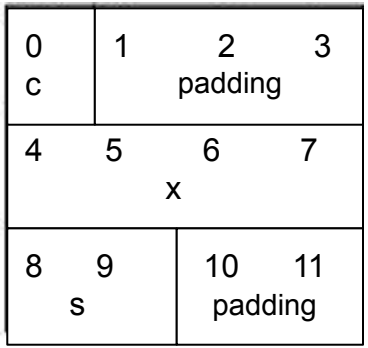
- Each function call consumes stack space for locals and parameters
- Large local arrays can easily exhaust stack space
- Large structs used as locals or passed by value have the same issue
- Prefer static, global, or heap allocation for large data
- Pass large objects by pointer instead of by value

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;
};
```



Reduce Padding

- Field ordering affects struct size due to alignment and padding
- Group fields by alignment, placing the most strictly aligned types first
- Reordering fields can significantly reduce memory footprint
- **`char`** and **`short`** are commonly used for flags or mode values
- Multiple flags can be packed using bit-fields, at the cost of 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