Lecture 9: Optimization

CS 105

October 1, 2019

Under the Abstraction Barrier

#include<stdio.h>	pushq %rbp	55
	movq %rsp, %rbp	48 89 e5
int main(int argc,	subq $32, %rsp	48 83 ec 20
char ** argv){	leaq L_.str(%rip), %rax	48 8d 05 25 00 00 00
	movl $0, -4(%rbp)	c7 45 fc 00 00 00 00
printf("Hello world!\n");	movl %edi, -8(%rbp)	89 7d f8
return 0;	movq %rsi, -16(%rbp)	48 89 75 f0
}	movq %rax, %rdi	48 89 c7
	movb $0, %al	b0 00
	callq _printf	e8 00 00 00 00
	xorl %ecx, %ecx	31 c9
	movl %eax, -20(%rbp)	89 45 ec
	movl %ecx, %eax	89 c8
	addq $32, %rsp	48 83 c4 20
	popq %rbp	5d
	retq	c3

Techniques for Improving Performance

1. ~~Use better algorithms/data structures~~

2. Compile to efficient byte code

3. Write code that compiles to efficient byte code

4. Parallelize your execution

Optimizing Compilers

- Provide efficient mapping of program to machine
  - register allocation
  - code selection and ordering (scheduling)
  - dead code elimination
  - eliminating minor inefficiencies

- Seldom improve asymptotic efficiency
  - up to programmer to select best overall algorithm
  - big-O savings are (often) more important than constant factors
    - but constant factors also matter

Eliminating Dead Code (-O0)

```
int dead_code(int input){
  if(47 > 0){
    return input;
  } else {
    return -1 *input;
  }
}
```

```
int dead_code(int input){

    return input;



}
```

```
dead_code:
        movl    %edi, %eax
        ret
```

Code Motion (-O1)

- Reduce frequency with which computation is performed
- For example, move code out of a loop

```
void set_row(int *a, int *b,
             int i,  int n) {

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

```
set_row:
        testl   %ecx, %ecx
        jle     .L1
        imull   %ecx, %edx
        addl    %edx, %ecx
.L3:
        movl    (%rsi), %r8d
        movslq  %edx, %rax
        movl    %r8d, (%rdi,%rax,4)
        addl    $1, %edx
        addq    $4, %rsi
        cmpl    %ecx, %edx
        jne     .L3
.L1:
        rep ret
```

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

Factoring out Subexpressions (-O1)

- Share common subexpressions
  - Gcc will do this with –O1

```
/* 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

```
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
```

Loop Elimination (-O1)

```
int loop_while(int a)
{
  int b = 16;
  int i = 0;
  int result = 0;
  while (i < 64) {
    result += a;
    a -= b;
    i += b;
  }
  return result;
}
```

```
loop_while:
        leal    -96(,%rdi,4), %eax
        ret
```

```
int loop_while(int a)
{
  return 4*a-96;
}
```

Reduction in Strength (-O2)

- Replace costly operation with simpler one
- For example, replace multiplication with shift or addition

```
void set_matrix(long *a, long *b,
                long n){

  for (long i = 0;
    long ni = n*i;
    for (long j = 0
      a[ni + j] = b
    }

  }
}
```

```
set_matrix:
         xorl     %
         testq    %
         leaq     0
         jle      .
.L6:     xorl     %
.L3:     movq     (
         movq     %
         addq     $
         cmpq     %rax, %rdx
         jne      .L3
         addq     $1, %r8
         addq     %r9, %rdi
         cmpq     %r8, %rdx
         jne      .L6
.L1:     rep ret
```

```
void set_matrix(long *a, long *b,
                long n){
  int ni = 0;
  for (long i = 0; i < n; i++) {

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

Limitations of Optimizing Compilers

1. Must not cause any change in program behavior
    - Often prevents optimizations that would only affect behavior under pathological conditions.
        - Data ranges may be more limited than variable type suggests
        - Compiler cannot know run-time inputs
    - When in doubt, the compiler must be conservative

Limitations of Optimizing Compilers

```
void mystery(int *xp,
             int *yp){
  *xp = *xp + *yp;
  *yp = *xp - *yp;
  *xp = *xp - *yp;
}
```

?

```
void mystery(int *xp,
             int *yp){
  int temp = *xp;
  *xp = *yp;
  *yp = temp;
}
```

- Potential problem: xp and yp might be different aliases for the same value
  - i.e., xp == yp

Case Study 1: Summing Matrix Rows

```
/* Sum rows of nxn matrix a, store
   in vector b                      */
void sum_rows1(double *a, double *b,
               long n) {
for (long i = 0; i < n; i++) {
    b[i] = 0;
    for (long j = 0; j < n; j++){
      b[i] += a[i*n + j];
    }

  }
}
```

```
/* Sum rows of nxn matrix a, store
   in vector b                      */
void sum_rows2(double *a, double *b,
               long n) {
  for (long i = 0; i < n; i++) {
    double val = 0;
    for (long j = 0; j < n; j++){
      val  += a[i*n + j];
    }
    b[i] = val;
  }
}
```

```
# sum_rows1 inner loop
.L4:
        movsd   (%rsi,%rax,8), %xmm0
        addsd   (%rdi), %xmm0
        movsd   %xmm0, (%rsi,%rax,8)
        addq    $8, %rdi
        cmpq    %rcx, %rdi
        jne     .L4
```

```
# sum_rows2 inner loop
.L10:
        addsd   (%rdi), %xmm0
        addq    $8, %rdi
        cmpq    %rax, %rdi
        jne     .L10
```

Optimization Blocker 1

- Aliasing: Two different references to a single location
  - Easy to happen in C

- Develop habit of introducing local variables
  - To accumulate within loops, for example
  - Your way of telling the compiler not to check for aliasing

Limitations of Optimizing Compilers

1. Must not cause any change in program behavior
   - Often prevents optimizations that would only affect behavior under pathological conditions.
     - Data ranges may be more limited than variable type suggests
     - Compiler cannot know run-time inputs
   - When in doubt, the compiler must be conservative

2. Most analysis is performed only within procedures
   - Whole-program analysis is too expensive in most cases
   - Newer versions of `gcc` do interprocedural analysis within files

Example

```
long f1();

long f2(){
  return f1() + f1();
}
```

?

```
long f1();

long f2(){
  return 2*f1();
}
```

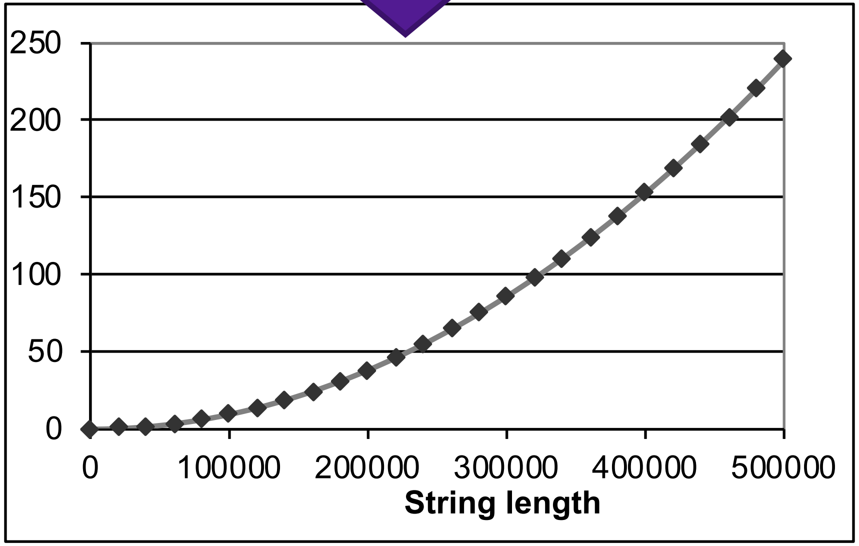
- problem: f1 might have side-effects
  - update global variables
  - write to file/network
  - UI feature

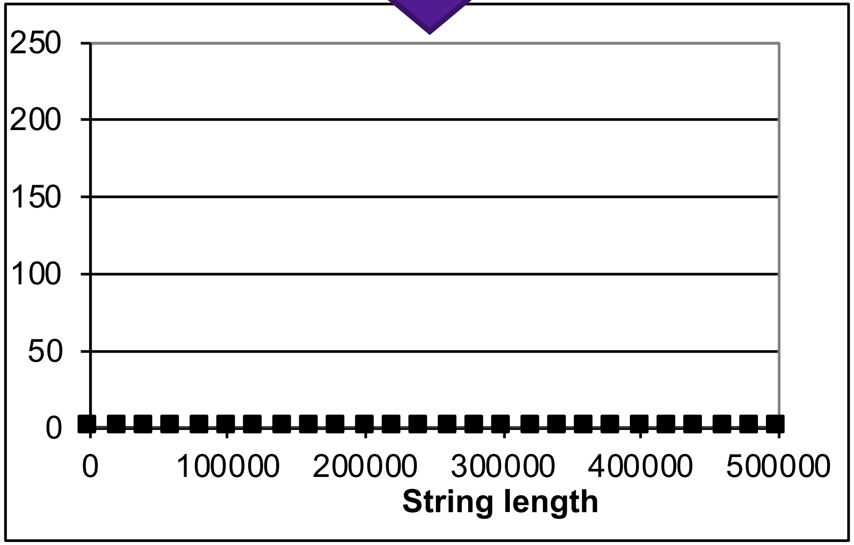
Case Study 2: Lowering Case

```
void lower(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(char *s){
  size_t i;
  size_t len = strlen(s);
  for (i = 0; i < len; i++){
    if (s[i] >= 'A' && s[i] <= 'Z'){
      s[i] -= ('A' - 'a');
    }
  }
}
```





Optimization Blocker 2

- Compiler treats procedure calls as black boxes
  - Unknown side-effects
  - **`strlen`** may not always return the same value

- Alternatives:
  - Do your own code motion (necessary here)
  - Use inline functions
    - **`gcc`** will optimize within a single file with **`–O1`**

Machine Independent Optimization

- Compilers optimize assembly code
  - Code motion
  - Strength reduction
  - Common subexpressions
  - Dead code elimination
  - Loop unwinding/elimination

- Optimization blockers:
  - Procedure calls
    - Move them yourself
  - Aliasing
    - Use local variables

Case Study 3: Vector Data Type

```
/* data structure for vectors */
typedef struct{
    size_t len;
    data_t *data;
} vec;
```

`data_t` will vary by example

- `int`
- `long`
- `float`
- `double`

```
/* retrieve vector element and store at val */

int get_vec_element
  (vec *v, size_t idx, data_t *val) {

    if (idx >= v->len)
        return 0;
    *val = v->data[idx];
    return 1;
}
```

Benchmark Computation

```
void combine1(vec_ptr v, data_t *dest)
{
  long i;
  *dest = IDENT;

  for (i = 0; i < vec_length(v); i++) {
    data_t val;
    get_vec_element(v, i, &val);
    *dest = *dest OP val;
  }
}
```

Sum or product of vector elements

IDENT/OP may be 0/+ or 1/*

Metric: CPE, cycles per element

Time = CPE * n + Overhead

Benchmark Performance

```
void combine1(vec_ptr v, data_t *dest)
{
  long i;
  *dest = IDENT;

  for (i = 0; i < vec_length(v); i++) {
    data_t val;
    get_vec_element(v, i, &val);
    *dest = *dest OP val;
  }
}
```

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine1 –O0	22.68	20.02	19.98	20.18
Combine1 –O1	10.12	10.12	10.17	11.14

Exercise: how could you optimize this code to get even better performance?

Code-Level Optimizations

```
void combine2(vec_ptr v, data_t *dest)
{
  long i;
  data_t t = IDENT;
  long length = vec_length(v);
  data_t *d = get_vec_start(v);
  for (i = 0; i < length; i++){
    t = t OP d[i];
  }
  *dest = t;
}
```

- Accumulate in temporary variable
- Move vec_length out of loop
- Avoid extra bounds check on each cycle

Code-Level Optimizations

```
void combine2(vec_ptr v, data_t *dest)
{
  long i;
  data_t t = IDENT;
  long length = vec_length(v);
  data_t *d = get_vec_start(v);
  for (i = 0; i < length; i++){
    t = t OP d[i];
  }
  *dest = t;
}
```

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine1 –O0	22.68	20.02	19.98	20.18
Combine1 –O1	10.12	10.12	10.17	11.14
Combine2	1.27	3.01	3.01	5.01

Loop Unrolling

```
int psum1(float a[],float p[],long n){(){
  long i;
  p[0] = a[0];
  for(i = 1; i < n; i++){
    p[i] = p[i-1] + a[i];
  }
}
```

```
int psum2(float a[],float p[],long n){(){
  long i;
  p[0] = a[0];
  for(i = 1; i < n-1; i+=2){
    p[i]   = p[i-1] + a[i];
    p[i+1] = p[i] + a[i+1];
  }
  if (i < n){ //handle even n
    p[i] = p[i-1] + a[i];
  }
}
```

Combine with Unrolling

```
void unroll2_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
       x = (x OP d[i]) OP d[i+1];
```

Method	Integer		Double FP	
Operation	**Add**	**Mult**	**Add**	**Mult**
Combine1 –O0	22.68	20.02	19.98	20.18
Combine1 –O1	10.12	10.12	10.17	11.14
Combine2	1.27	3.01	3.01	5.01
Unroll 2	1.01	3.01	3.01	5.01
Latency Bound	1.00	3.00	3.00	5.00

Machine-Dependent Parallelization

Method	Integer		Double FP	
Operation	**Add**	**Mult**	**Add**	**Mult**
Combine1 –O0	22.68	20.02	19.98	20.18
Combine1 –O1	10.12	10.12	10.17	11.14
Combine2	1.27	3.01	3.01	5.01
Unroll 2	1.01	3.01	3.01	5.01
Optimal Unrolling	0.54	1.01	1.01	0.52
Latency Bound	1.00	3.00	3.00	5.00
Throughput Bound	0.50	1.00	1.00	0.50

- Limited only by throughput of functional units
- Up to 42X improvement over original, unoptimized code
  - Combines machine-independent and machine-dependent factors