Lecture 10: Machine-Independent Optimization

CS 105

February 25, 2019

Under the Abstraction Barrier

```
#include<stdio.h>

int main(int argc,
         char ** argv){

  printf("Hello world!\n");
  return 0;
}
```

```
pushq   %rbp
movq    %rsp, %rbp
subq  $32, %rsp
leaq  L_.str(%rip), %rax
movl  $0, -4(%rbp)
movl  %edi, -8(%rbp)
movq  %rsi, -16(%rbp)
movq  %rax, %rdi
movb  $0, %al
callq _printf
xorl  %ecx, %ecx
movl  %eax, -20(%rbp)
movl  %ecx, %eax
addq  $32, %rsp
popq  %rbp
retq
```

```
55
48 89 e5
48 83 ec 20
48 8d 05 25 00 00 00
c7 45 fc 00 00 00 00
89 7d f8
48 89 75 f0
48 89 c7
b0 00
e8 00 00 00 00
31 c9
89 45 ec
89 c8
48 83 c4 20
5d
c3
```

Techniques for Improving Performance

1. Use better algorithms/data structures

2. Write code that compiles to efficient byte code

3. 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

Code Motion

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

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

  long j;

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

```
void set_row(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];
  }
}
```

Compiler-Generated Code Motion (-O1)

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

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

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

```
set_row:
        testq   %rcx, %rcx              # Test n
        jle     .L1                     # If 0, goto done
        imulq   %rcx, %rdx              # ni = n*i
        leaq    (%rdi,%rdx,8), %rdx     # rowp = A + ni*8
        movl    $0, %eax                # j = 0
.L3:                                    # loop:
        movsd   (%rsi,%rax,8), %xmm0    # t = b[j]
        movsd   %xmm0, (%rdx,%rax,8)    # M[A+ni*8 + j*8] = t
        addq    $1, %rax                # j++
        cmpq    %rcx, %rax              # j:n
        jne     .L3                     # if !=, goto loop
.L1:                                    # done:
        rep ; ret
```

Reduction in Strength

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

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

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

  }
}
```

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

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

Factoring out Subexpressions

- 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

```
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
```

```
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
```

Eliminating Dead Code

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

```
int dead_code(int input){

    return input;



}
```

```
_dead_code:                                    ## @dead_code
        pushq   %rbp
        movq    %rsp, %rbp
        movl    %edi, %eax
        popq    %rbp
        retq
                                               ## -- End function
```

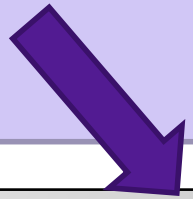
Loop Unrolling

```
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];
  }
}
```

Loop Elimination

```
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;
}
```



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

```
_loop_while:
        pushq   %rbp
        movq    %rsp, %rbp
        leal    -96(,%rdi,4), %eax
        popq    %rbp
        retq
```

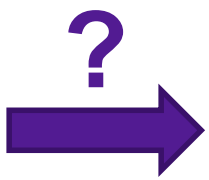
Limitations of Optimizing Compilers

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

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

  }
}
```

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

Case Study 1: Summing Matrix Rows

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

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

No need to store intermediate results

Optimization Blocker 1

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

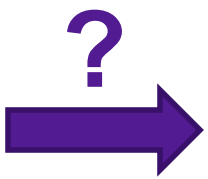
- Develop habit of introducing local variables
  - To accumulate within loops, for example
  - Your way of telling the compiler not to check for aliasing

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

Example

```
long f1();

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



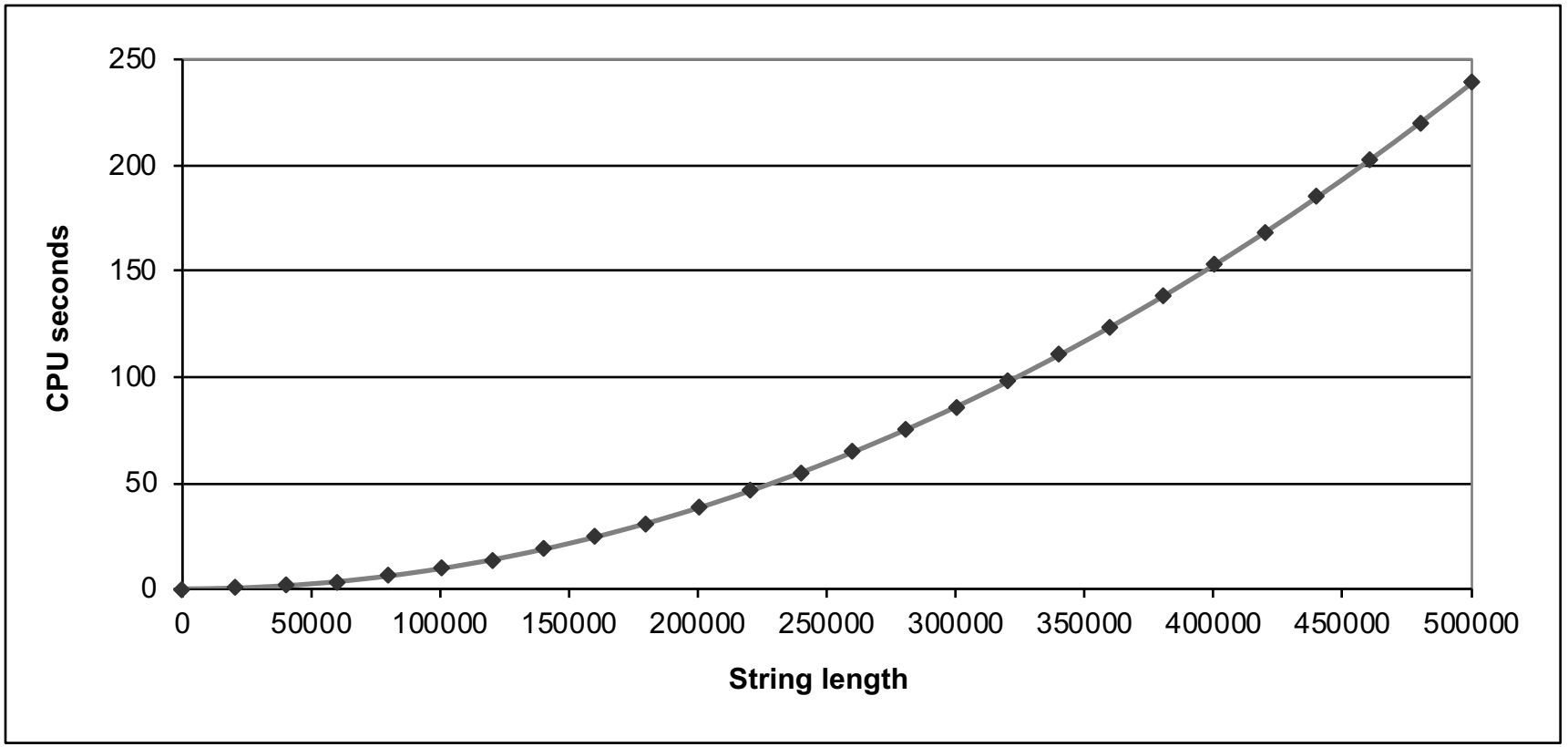

```
long f1();

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');
    }
  }
}
```

Case Study 2: Lowering Case

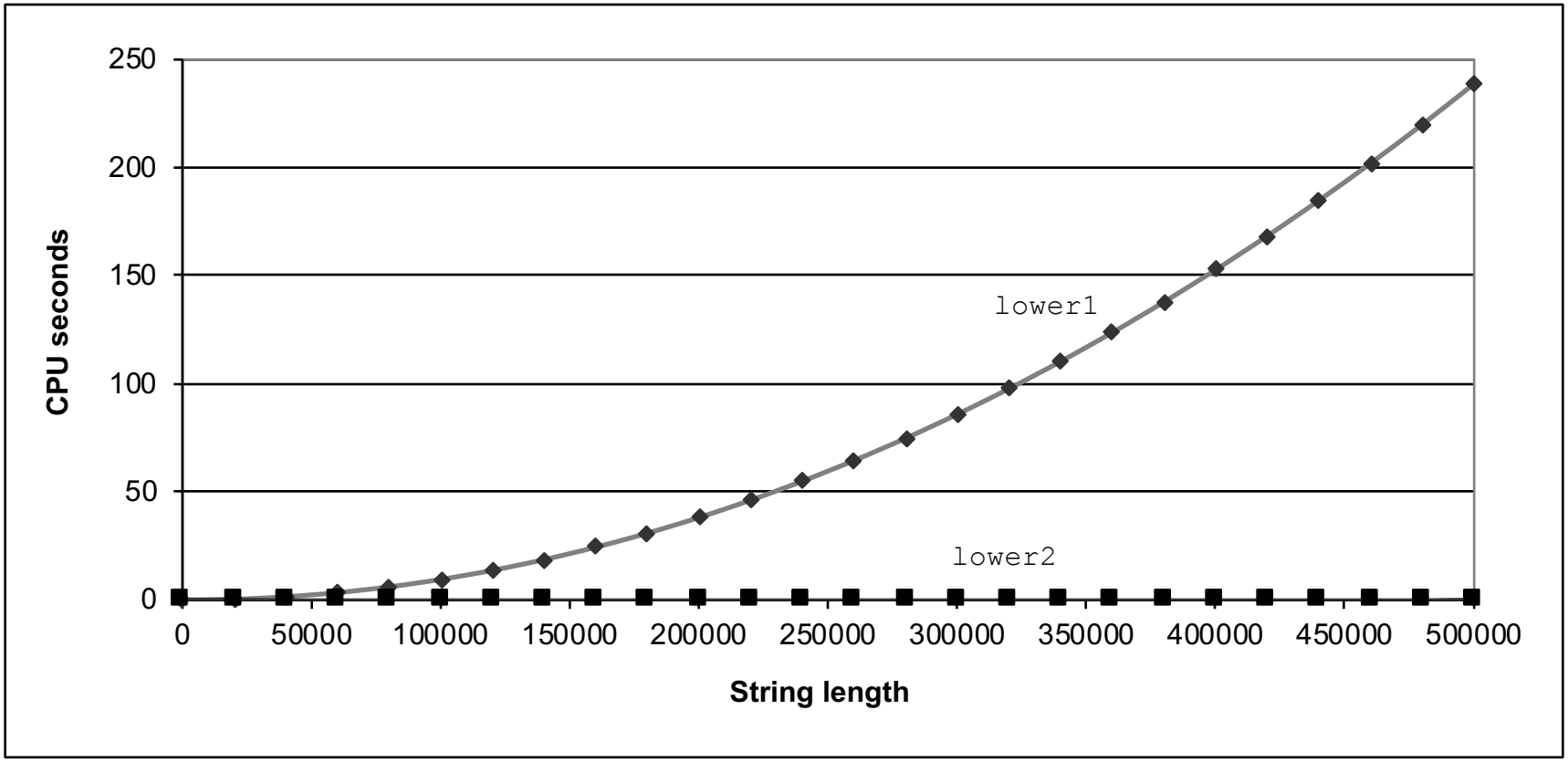


Case Study 2: Lowering Case

```
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');
    }
  }
}
```

- Move call to **strlen** outside of loop
- Since result does not change from one iteration to another

Case Study 2: Lowering Case



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 bounds check on each cycle

Effect of 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