CS 105

"Tour of the Black Holes of Computing"

Code Optimization and Performance

Great Reality

There's more to performance than

asymptotic complexity

Constant factors matter too!

- Easily see 10:1 performance range depending on how code is written
- Must optimize at multiple levels:
  - Algorithm, data representations, procedures, and loops

Must understand system to optimize performance

- How programs are compiled and executed
- How to measure program performance and identify bottlenecks
- How to improve performance without destroying code modularity, generality, readability

Optimizing Compilers

Provide efficient mapping of program to machine

- Register allocation
- Code selection and ordering
- Eliminating minor inefficiencies

Don't (usually) 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
  - E.g., $O(N^2)$ sort is faster for 7 or fewer items

Have difficulty overcoming "optimization blockers"

- Potential memory aliasing
- Potential procedure side effects

Limitations of Optimizing Compilers

Compilers operate under fundamental constraint

- Must not cause any change in program behavior under *any possible* condition
- Often prevents optimizations that would only affect behavior in pathological situations

Behavior obvious to the programmer can be obfuscated by languages and coding styles

- E.g., data ranges may be more limited than variable types suggest

Most analysis is performed only within procedures

- Whole-program analysis is too expensive in most cases
- (gcc does lots of interprocedural analysis—but not across files)

Most analysis is based only on *static* information

- Compiler has difficulty anticipating run-time inputs

When in doubt, the compiler must be conservative

Generally Useful Optimizations

- Optimizations you should do regardless of processor / compiler

Code Motion

- Reduce frequency with which computation performed
  - If it will always produce same result
  - Especially moving code out of loop
  - Gcc often does this for you (so check assembly)

```
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];
}
```


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];
}
```

```
    long j;
    long ni = n*i;
    double *rowp = a + ni;
    for (j = 0; j < n; j++)
        rowp[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
```


Strength Reduction

- Replace costly operation with simpler one
- Shift, add instead of multiply or divide
  `16*x  ⇒  x << 4`
  - Utility is machine-dependent
  - Depends on cost of multiply or divide instruction
  - On Intel Nehalem, integer multiply requires only 3 CPU cycles
- Recognize sequence of products
- Again, gcc often does it

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

→

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


Share Common Subexpressions

- Reuse portions of expressions
- Gcc will do this with –O1 and up

```
/* 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;
```

```
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;
```

3 multiplications: i*n, (i–1)*n, (i+1)*n

1 multiplication: i*n

```
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
```

```
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
```

Optimization Blocker #1: Procedure Calls

Procedure to Convert String to Lower Case

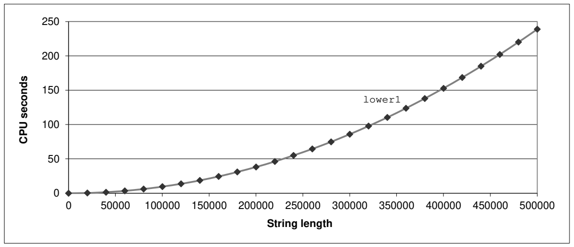
```
#include <ctype.h>

void lower(char *s)
{
  size_t i;
  for (i = 0; i < strlen(s); i++)
    if (isupper(s[i]))
      s[i] = tolower(s[i]);
}
```

- Extracted from *many* student programs

Lower-Case Conversion Performance

- Time quadruples when double string length
- Quadratic performance



Convert Loop To Goto Form

```
void lower(char *s)
{
   size_t i = 0;
   if (i >= strlen(s))
     goto done;
 loop:
   if (isupper(s[i]))
       s[i] = tolower(s[i]);
   i++;
   if (i < strlen(s))
     goto loop;
 done:
}
```

- `strlen` executed every iteration

Calling Strlen

```
/* My version of strlen */
size_t strlen(const char *s)
{
    size_t length = 0;
    while (*s != '\0') {
       s++;
       length++;
    }
    return length;
}
```

Strlen performance

- Only way to determine length of string is to scan its entirety, looking for NUL character.

Overall performance, string of length N

- N calls to strlen, each takes O(N) time
- Overall $O(N^2)$ performance

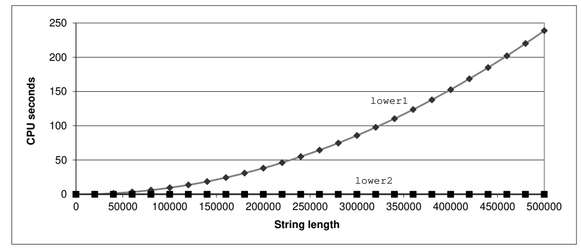
Improving Performance

```
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');
}
```

- Programmer moves call to `strlen` outside of loop
  - Since result does not change from one iteration to another
- Form of code motion

Lower-Case Conversion Performance

- Time doubles when double string length
- Linear performance of lower2



Optimization Blocker: Procedure Calls

Why couldn't compiler move `strlen` out of inner loop?

- Procedure may have side effects
  - Might alter global state each time called
- Function may not return same value for given arguments
  - Depends on other parts of global state
  - Procedure `lower` could interact with `strlen`

```
size_t lencnt = 0;
size_t strlen(const char *s)
{
    size_t length = 0;
    while (*s != '\0') {
        s++;
        length++;
    }
    lencnt += length;
    return length;
}
```

Warning:

- Compiler treats procedure calls as a black box
- Weak optimizations near them

Remedies:

- Use inline functions
  - GCC does this with –O1
    - » But only within single file
- Do your own code motion

Memory Matters

```
/* Sum rows is of n X n matrix a
   and 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
```

- Code updates `b[i]` on every iteration
- Why couldn't compiler optimize this away?

Memory Aliasing

```
/* Sum rows is of n X n matrix a
   and 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];
    }
}
```

```
double A[9] =
  { 0,   1,   2,
    4,   8,  16,
   32,  64, 128};

double* B = A+3;

sum_rows1(A, B, 3);
```

Value of B:

```
init:   [4, 8, 16]
i = 0: [3, 8, 16]
i = 1: [3, 22, 16]
i = 2: [3, 22, 224]
```

- Code updates `b[i]` on every iteration
- Must consider possibility that these updates will affect program behavior

Removing Aliasing

```
/* Sum rows is of n X n matrix a
   and 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

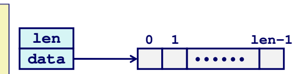
Optimization Blocker: Memory Aliasing

Aliasing

- Two different memory references specify single location
- Easy to have happen in C
  - Since allowed to do address arithmetic
  - Language allows irect access to storage structures
- Get in habit of introducing local variables
  - E.g., accumulating within loops
  - Your way of telling compiler not to check for aliasing

Benchmark Example: Data Type for Vectors

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




```
/* 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;
}
```

Data Types

- Use different declarations for `data_t`
- `int`
- `long`
- `float`
- `double`

Benchmark Computation

```
void combine1(vec_ptr v, data_t *dest)
{
    long int i;
    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++) {
       data_t val;
       get_vec_element(v, i, &val);
       *dest = *dest OP val;
    }
}
```

Compute sum or product of vector elements

Data Types

- Use different declarations for `data_t`
- `int`
- `long`
- `float`
- `double`

Operations

- Use different definitions of `OP` and `IDENT`
- `+ and 0`
- `* and 1`

Cycles Per Element (CPE)

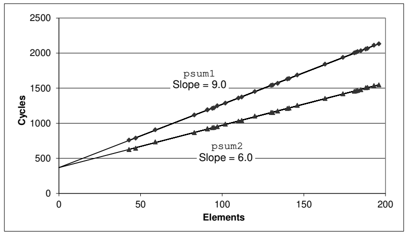
Convenient way to express performance of program that operates on vectors or lists

Length = n

In our case: CPE = cycles per OP

T = CPE*n + Overhead

- CPE is slope of line



Benchmark Performance

```
void combine1(vec_ptr v, data_t *dest)
{
    long int i;
    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++) {
       data_t val;
       get_vec_element(v, i, &val);
       *dest = *dest OP val;
    }
}
```

Compute sum or product of vector elements

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

Basic Optimizations

```
void combine4(vec_ptr v, data_t *dest)
{
  long i;
  long length = vec_length(v);
  data_t *d = get_vec_start(v);
  data_t t = IDENT;
  for (i = 0; i < length; i++)
    t = t OP d[i];
  *dest = t;
}
```

Move vec_length out of loop

Avoid bounds check on each cycle

Accumulate in temporary

Effect of Basic Optimizations

```
void combine4(vec_ptr v, data_t *dest)
{
  long i;
  long length = vec_length(v);
  data_t *d = get_vec_start(v);
  data_t t = IDENT;
  for (i = 0; i < length; i++)
    t = t OP d[i];
  *dest = t;
}
```

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine1 –O1	10.12	10.12	10.17	11.14
Combine4	1.27	3.01	3.01	5.01

Eliminates sources of overhead in loop

Exploiting Instruction-Level Parallelism

We can go farther!

But need general understanding of modern processor design

- Hardware can execute multiple instructions in parallel

Performance limited by data dependencies

Simple transformations can yield dramatic performance improvement

- Compilers often cannot make these transformations
- Lack of associativity and distributivity in floating-point arithmetic

We'll talk about that next time