Program Optimization

Jo, Heeseung

Today

Overview

Generally Useful Optimizations

- Code motion/precomputation
- Strength reduction
- Sharing of common subexpressions
- Removing unnecessary procedure calls

Optimization Blockers

- Procedure calls
- Memory aliasing

Performance Realities

Constant factors matter

- 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 and generality

Optimizing Compilers

Provide efficient mapping of program to machine

- Register allocation
- Code selection and ordering (scheduling)
- Dead code elimination
- Eliminating minor inefficiencies

Up to programmer to select best overall algorithm

- Big-O savings are (often) more important than constant factors
- But constant factors also matter

Have difficulty overcoming "optimization blockers"

- Potential memory aliasing
- Potential procedure side-effects

Limitations of Optimizing Compilers

Operate under fundamental constraint

- Must not cause any change in program behavior

Most analysis is performed only within procedures

- Whole-program analysis is too expensive in most cases

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

1. Code motion

- Reduce frequency with which computation performed
  - If it will always produce same result
  - Especially moving code out of 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];
}
```

int ni = n*i 추가

⇒ 곱하기를 반복문 안에서 하면 비효율적임. → 밖으로 빼냄.

Compiler-Generated Code Motion

```c
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];
}
```

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

```
set_row:
          testq     %rcx, %rcx            # Test n
          jle       .L4                   # If 0, goto done
          movq      %rcx, %rax            # rax = n
          imulq     %rdx, %rax            # rax *= i
          leaq      (%rdi,%rax,8), %rdx   # rowp = a + n*i*8
          movl      $0, %r8d              # j = 0
.L3:                                      # loop:
          movq      (%rsi,%r8,8), %rax    # t = b[j]
          movq      %rax, (%rdx)          # *rowp = t
          addq      $1, %r8               # j++
          addq      $8, %rdx              # rowp++
          cmpq      %r8, %rcx             # Compare n:j
          jg        .L3                   # If >, goto loop
.L4:                                      # done:
          rep ; ret
```

어차피 컴파일러가
알아서 빼내서 해준
=> code motion

Generally Useful Optimizations

2. Reduction in strength

- Replace costly operation with simpler one
- Shift, add instead of multiply or divide
  - 16*x   -->   x << 4
  - Utility machine dependent
  - Depends on cost of multiply or divide instruction
  - On Intel Nehalem, integer multiply requires 3 CPU cycles
- Recognize sequence of products

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

int ni = 0;

ni += n;

Generally Useful Optimizations

3. Share common subexpressions

- Reuse portions of expressions
- Compilers often not very sophisticated in exploiting arithmetic properties

```
/* 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

```
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
```

1 multiplication: i*n

```
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

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

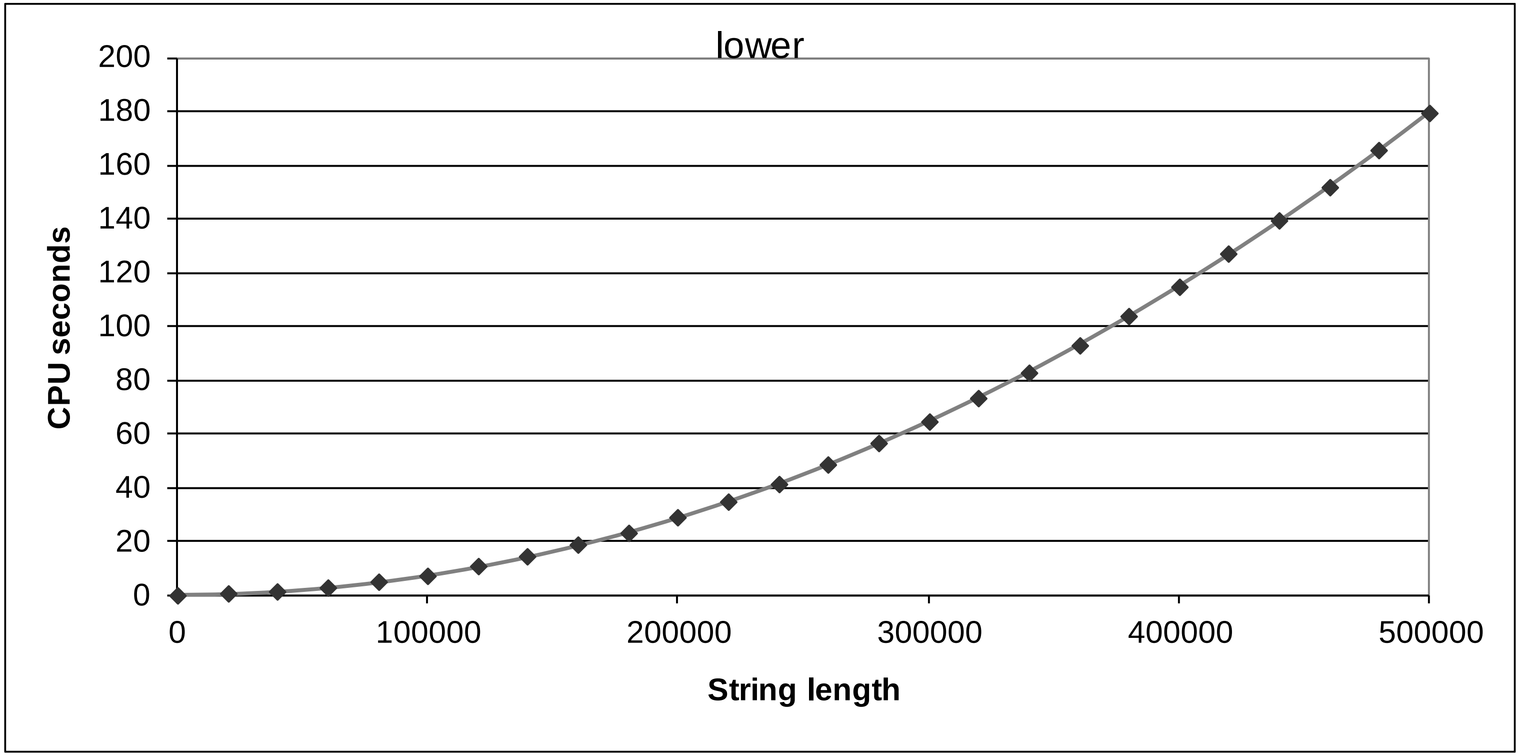
What's wrong?

Nothing but wrong

Lower Case Conversion Performance

Time quadruples when double string length

- Quadratic performance -> Why??



Convert Loop To Goto Form

strlen executed every iteration

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

```
// goto version of for statement

void lower(char *s)
{
   int i = 0;
   if (i >= strlen(s))
     goto done;
 loop:
   if (s[i] >= 'A' && s[i] <= 'Z')
       s[i] -= ('A' - 'a');
   i++;
   if (i < strlen(s))
     goto loop;
 done:
}
```

Calling strlen

strlen() performance

- Only way to determine length of string is to scan its entire length, looking for null character

Overall performance, string of length N

- N calls to strlen
- Require times N
- Overall $O(N^2)$ performance

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

Improving Performance

Move call to strlen() outside of loop

- Since result does not change from one iteration to another
- Form of code motion

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

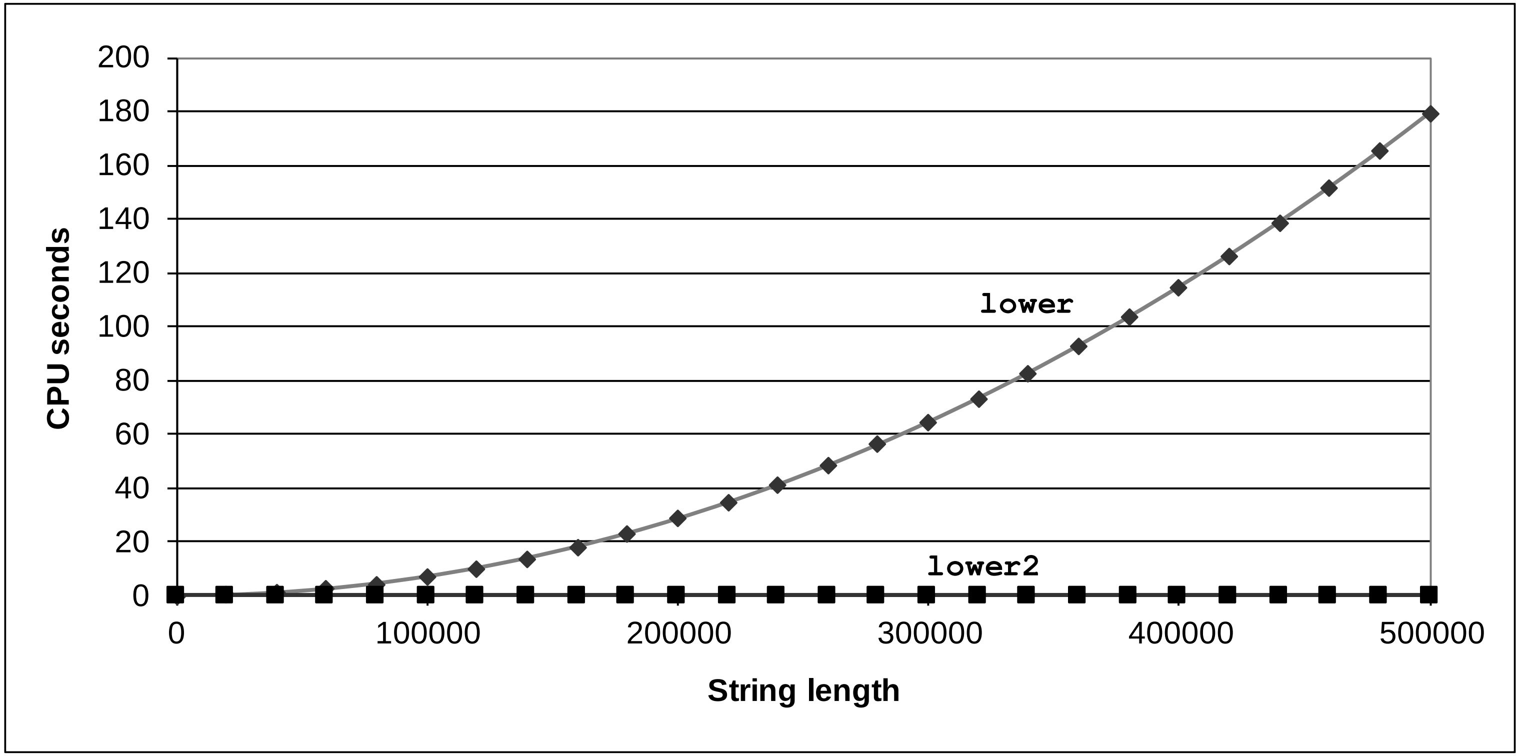
이게 더 좋음.

```
void lower2(char *s)
{
    int i;
    int len = strlen(s);
    for (i = 0; i < len; i++)
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
}
```

strlen에 글로벌 변수가 값을 주고 있고,
~ ~ 컴파일 시간이 오래걸릴 수 있음.

Lower Case Conversion Performance

Linear performance of lower2



Optimization Blocker: Procedure Calls

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

- Procedure may have side effects
  - Alters 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

strtok()

```
int 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 call as a black box
- Weak optimizations near them

Remedies:

- Use of inline functions
  - GCC does this with -O2
- Do your own code motion

Memory Matters

Code updates b[i] on every iteration

- Why couldn't compiler optimize this?

```
/* 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 = {0, 0, 0};
sum_rows1(A, B, 3);
```

Value of B:

```
init:  [0, 0, 0]
```

```
i = 0: [3, 0, 0]
```

```
i = 1: [3, 28, 0]
```

```
i = 2: [3, 28, 224]
```

Memory Aliasing

Must consider possibility that these updates will affect program behavior

```
/* 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:

3 0 16

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

왜 22가 나오는가?

Memory Aliasing

Must consider possibility that these updates will affect program behavior

```
/* 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);
```

init:	4	8	16
i = 0 ;	0	8	16
i = 0 :	3	8	16
i = 1 :	3	0	16
i = 1:	3	3	16
i = 1:	3	6	16
i = 1:	3	22	16
i = 2:	3	22	224

Removing Aliasing

No need to store intermediate results

```
/* 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;
    }
}
```

```
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, 27, 16]
```

```
i = 2: [3, 27, 224]
```

변수 추가해서 최적화

Optimization Blocker: Memory Aliasing

Aliasing

- Two different memory references specify single location
- Easy to have happen in C
  - Since allowed to do address arithmetic
  - Direct access to storage structures
- Get in habit of introducing local variables
  - Accumulating within loops
  - Your way of telling compiler not to check for aliasing

Summary

Must optimize at multiple levels:

- algorithm, data representations, procedures, and loops

Must understand system to optimize performance

Generally Useful Optimizations

- Code motion/precomputation
- Strength reduction
- Sharing of common subexpressions
- Removing unnecessary procedure calls

Optimization Blockers

- Procedure calls
- Memory aliasing