The Hardware/Software Interface

CSE351 Winter 2011

Module 9: Compiler Optimizations

Today

- **Context for this material**
- **Overview of program optimizations**
  - Removing procedure calls
  - Code motion/precomputation
  - Strength reduction
  - Sharing of common subexpressions
  - Impediments to compiler optimization:
    - Procedure calls
    - Memory aliasing

Context

- **We've looked at the ISA and talked about compiling C programs to it**
- **Most of our compilation has been very direct**
  - As thought a C statement represented a template for machine code
  - Implying that the programmer's job is primarily to tell the compiler what machine code to generate
- **It's more complicated than that...**

Context (cont.)

- **Who are you talking to when you're writing code?**
  - To the compiler, so it can generate code that can run on the hardware
    - This is motivation to write code that runs fast
  - To programmers, who have to maintain the code
    - This is motivation to write simple, clear code

- **Ideally, we can have both...**
  - You write clear, simple code for other programmers to read
  - The compiler transforms your code into something that runs fast
    - i.e., optimizes it

Does this work?

- **Well...**
  - The compiler can do a lot, but...
  - It can't do everything you might hope for
- **It's useful to have some understanding of the kinds of things it can do**
  - And can't do

The general sense of what the compiler can/can't do

- **It can't do major redesign**
  - It can't pick an entirely different algorithm
  - It can't pick better data structures

- **It's good at**
  - “local” optimizations
    - a short sequence of instructions
    - a procedure
  - bookkeeping – it's better than we are
    - E.g., register allocation

Another example – no optimization

```c
int rescaleValue(int val) {
    int a = 30;
    int b = 9 - a / 5;
    int c;

    c = b * 4;
    if (c > 10) {
        c = c - 10;
    }
    return val * c * (60 / a);
}
```

$ gcc -S rescale.c

```
        pushl   %ebp
        movl    %esp, %ebp
        subl    $16, %esp
        movl    $30, -4(%ebp)
        movl    -4(%ebp), %ecx
        movl    $1717986919, %edx
        movl    %ecx, %eax
        imull   %edx
        sarl    %edx
        movl    %ecx, %eax
        sarl    $31, %eax
        subl    %eax, %edx
        movl    $9, %eax
        subl    %edx, %eax
        movl    %eax, -8(%ebp)
        movl    -8(%ebp), %eax
        sall    $2, %eax
        movl    %eax, -12(%ebp)
        cmpl    $10, -12(%ebp)
        jle     .L2
        subl    $10, -12(%ebp)
.L2:
        movl    8(%ebp), %eax
        movl    %eax, %ecx
        imull   -12(%ebp), %ecx
        movl    $60, %eax
        movl    %eax, %edx
        sarl    $31, %edx
        idivl   -4(%ebp)
        imull   %ecx, %eax
        leave
        ret
```

Another example – optimized

```
int rescaleValue(int val) {
    int a = 30;
    int b = 9 - a / 5;
    int c;

    c = b * 4;
    if (c > 10) {
        c = c - 10;
    }
    return val * c * (60 / a);
}
```

$ gcc -O2 -S rescale.c

```
     pushl    %ebp
     movl     %esp, %ebp
     movl     8(%ebp), %eax
     popl     %ebp
     sall     $2, %eax
     ret
```

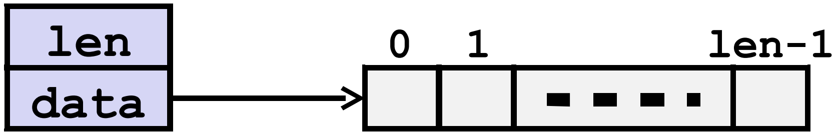
```
int rescaleValue(int val) {
    return val * 4;
}
```

Limitations of Optimizing Compilers

- ***If in doubt, the compiler is conservative***
- **Operate under fundamental constraints**
  - Must not change program behavior under any possible condition
  - Often prevents it from making optimizations when would only affect behavior under pathological conditions.
- **Behavior that may be 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
- **Most analysis is based only on *static* information**
  - Compiler has difficulty anticipating run-time inputs

Example: Data Type for Vectors

```
/* data structure for vectors */
typedef struct{
    int len;
    double *data;
} vec;
```



```
/* retrieve vector element and store at val */
int get_vec_element(vec *v, int idx, double *val)
{
    if (idx < 0 || idx >= v->len)
        return 0;
    *val = v->data[idx];
    return 1;
}
```

Example: Summing Vector Elements

```
double get_vec_element(vec *v, int idx,
                                double *val)
{
  if (idx < 0 || idx >= v->len)
    return 0;
  *val = v->data[idx];
  return 1;
}
```

Bound check unnecessary in sum_elements
Why?

```
/* sum elements of vector */
double sum_elements(vec *v, double *res)
{
  int i;
  n = v->len;
  *res = 0.0;
  double val;

  for (i = 0; i < n; i++) {
    get_vec_element(v, i, &val);
     *res += val;
  }
  return res;
}
```

Overhead for every fp +:

- One fct call
- One <
- One >=
- One ||
- One memory variable access

Slowdown:
probably 10x or more

Manually Removing Procedure Call

```
/* sum elements of vector */
double sum_elements(vec *v, double *res)
{
  int i;
  n = v->len;
  *res = 0.0;
  double val;

  for (i = 0; i < n; i++) {
    get_vec_element(v, i, &val);
      *res += val;
  }
  return res;
}
```

```
/* sum elements of vector */
double sum_elements(vec *v, double *res)
{
  int i;
  n = v->len;
  *res = 0.0;
  double *data = get_vec_start(v);

  for (i = 0; i < n; i++)
      *res += data[i];
  return res;
}
```

Inlining

- **Inlining is the notion of inserted the subroutine call into the call site**
  - Rather than generate the procedure call convention code (which is overhead), simply generate the body of the procedure
- **Does inlining make code faster?**
  - It's complicated...
  - Eliminates procedure call convention overhead
  - May make code larger
    - May make code smaller

Compiler Assisted Inlining

- **C**
  - #define get_vec_element(v, idx) (v->data[idx])
    - preprocessor rewrites `*res += get_vec_element(v,i);` as `*res = (v->data[idx])`
    - Why write the code this way?
  - gcc has the `-finline-functions` switch
- **C++**
  - has `inline` keyword
    - `inline int get_vec_element(vector* v, index idx);`

Code Motion

- **Reduce frequency with which computation is performed**
  - If it will always produce same result
  - Especially moving code out of loop
- **Sometimes also called pre-computation**

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

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

Compiler-Generated Code Motion

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

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

```
copy_row:
    xorl %r8d, %r8d           #  j = 0
    cmpq %rcx, %r8            #  j:n
    jge  .L7                  #  if >= goto done
    movq %rcx, %rax           #  n
    imulq    %rdx, %rax       #  n*i outside of inner loop
    leaq (%rdi,%rax,8), %rdx  #  rowp = A + n*i*8
.L5:                                  # loop:
    movq (%rsi,%r8,8), %rax   #  t = b[j]
    incq %r8                  #  j++
    movq %rax, (%rdx)         #  *rowp = t
    addq $8, %rdx             #  rowp++
    cmpq %rcx, %r8            #  j:n
    jl   .L5                  #  if < goto loop
.L7:                                  # done:
    rep ; ret                    #  return
```

Strength Reduction

- **Replace costly operation with simpler one**
- **Example: Shift/add instead of multiply or divide**
  - `16*x → x << 4`
  - Depends on cost of multiply or divide instruction
  - On Pentium IV, integer multiply requires 10 CPU cycles
- **Example: 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;
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**
- **Compilers often not very sophisticated in exploiting arithmetic properties**

3 mults: i*n, (i–1)*n, (i+1)*n

```
/* 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;
```

```
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 mult: i*n

```
int 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;
```

```
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
```

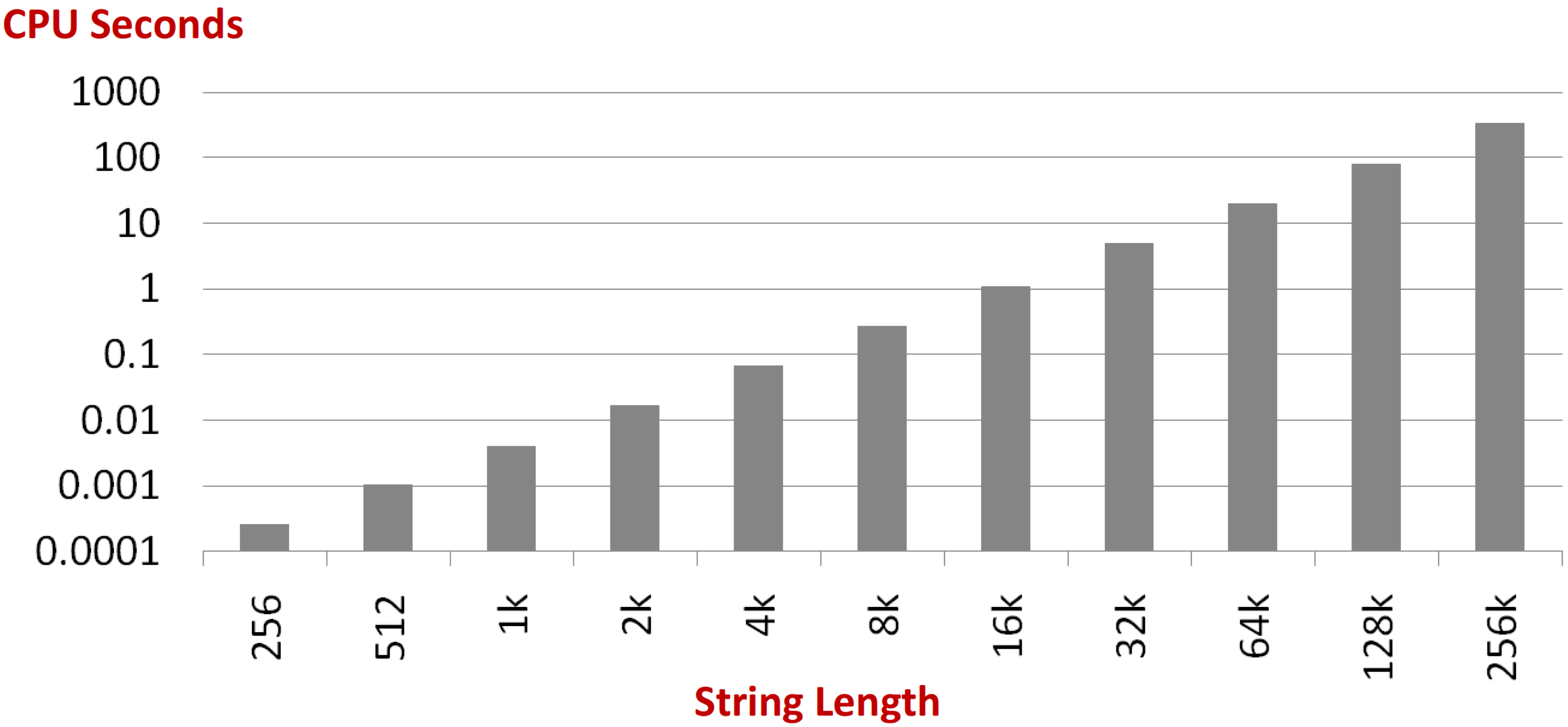
Optimization Blocker: Procedure Calls

- **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');
}
```

Performance

- Time quadruples when double string length
- Quadratic performance



Why is That?

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

- **String length is called in every iteration!**
  - And **`strlen`** is O(n), so **`lower`** is O(n^2)

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

Improving Performance

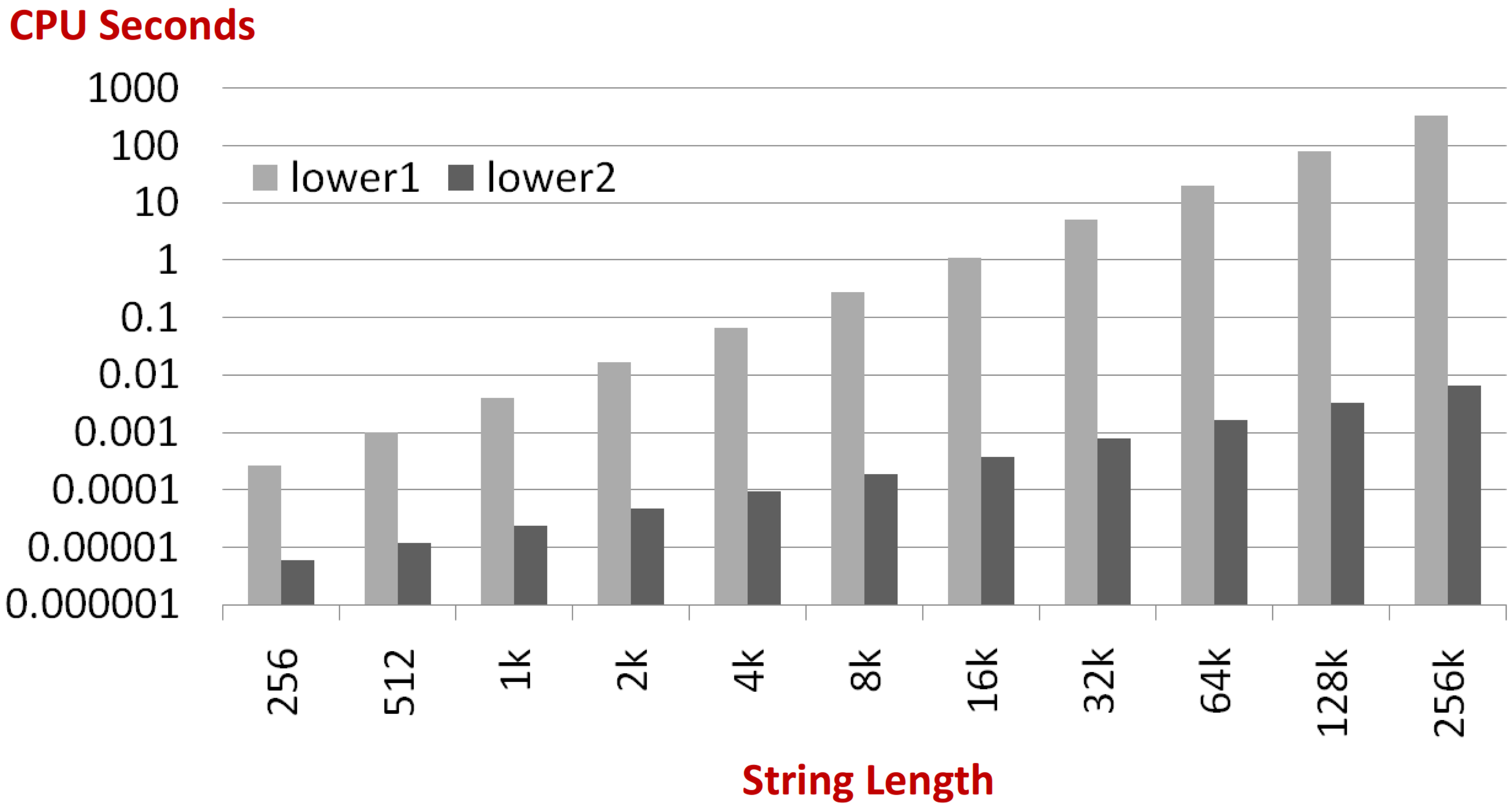
```
void lower(char *s)
{
  int i;
  for (i = 0; i < strlen(s); i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}
```

```
void lower(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');
}
```

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

Performance

- Lower2: Time doubles when double string length
- Linear performance



Optimization Blocker: Procedure Calls

- **Why couldn't compiler move `strlen` out of inner loop?**
  - Procedure may have side effects
    - For all the compiler knows, strlen could modify the string!
  - Function may not return same value for given arguments
    - Could depend on other parts of global state
  - Procedure **`lower`** could interact with **`strlen`**

- **Compiler usually treats procedure call as a black box that cannot be analyzed**
  - Consequence: conservative in optimizations
    - Can only do things that it can be sure always give the same results as the original code

Optimization Blocker: Memory Aliasing

```
// add twice the value stored at yp to the value stored at xp

void twiddle1(int *xp, int *yp)
{
    *xp += *yp;
    *xp += *yp;
}

void twiddle2(int *xp, int *yp)
{
    *xp += 2*(*yp);
}
```

- **twiddle1 appears to be less efficient**
  - 6 memory references: two reads each of *yp and *xp, two writes of *xp
- **twiddle2 appears to be more efficient**
  - 3 memory references: read *yp, read *xp, write *xp
- **Can a compiler come up with twiddle2 if given twiddle1?**

Optimization Blocker: Memory Aliasing

```
// add twice the value stored at yp to the value stored at xp
// *xp = *xp + 2 * *yp;

void twiddle1(int *xp, int *yp)
{
    *xp += *yp;
    *xp += *yp;
}

void twiddle2(int *xp, int *yp)
{
    *xp += 2*(*yp);
}
```

- **But what if xp == yp?**
  - twiddle1 quadruples value at xp
  - twiddle2 triples value at xp
- **Because of this ‘aliasing’, compiler does not optimize twiddle1**
  - Could lead to different result
- Assume twiddle1 is programmer’s intent

Optimization Blocker: Memory Aliasing

```
x = 1000;
y = 3000;
*q = y;
*p = x;
return *q;
```

- **What is the return value?**
- **Two cases:**
  - q and p are different addresses
  - q and p are aliases for the same address

A Final Thought

- **Source code optimization can muddle/destroy code clarity and program structure**
  - Certain optimizations are pretty easy and not too messy, so do them – e.g, move strlen(s) outside the loop
  - But it’s not always that simple…
- **Worth doing when it actually buys you something**
  - Use profiling tools to find out where the code is spending its time (it’s often not where you think!)
    (Alas, we probably won’t see gprof and other tools in this course)

“Premature optimization is the root of all evil”
Donald Knuth