Program Optimization

Slides adapted from Bryant and O’Hallaron
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

Programmers must optimize at multiple levels
- Big-O: algorithm, data representations
- Systems: optimize memory access, I/O, parallelize execution

Steps of manual optimization

1. Identify bottlenecks
   - Bottleneck is CPU? Disk/SSD? Network? others?

2. Measure program performance
   - If CPU is the bottleneck, profile a program’s execution to figure out which code path takes the most time

The challenge:
- How to improve performance without destroying code modularity and readability
- Premature optimization.

What about automatic optimization?
**Optimizing Compilers**

- **Goal is to generate efficient, correct machine code**
  - register allocation
  - code selection and ordering
  - dead code elimination
- **Don’t improve asymptotic efficiency**
  - Programmer should select best overall algorithm
Limitations of Optimizing Compilers

- Constraint: Must not change program behavior in *any* circumstances
  → *When in doubt, the compiler must be conservative*

- *Most analysis is performed only within procedures*
  - Whole-program analysis is too expensive in most cases
  - Newer versions of GCC do inter-procedural analysis within individual files
    - But, not between code in different files

- *Analysis is based only on static information*
  - Compiler does not know run-time inputs
Generally Useful Optimizations

**Code Motion**
- Reduce frequency with which computation performed
  - If it will always produce same result

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

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:
    movsrd (%rsi,%rax,8), %xmm0 # t = b[j]
    movsrd %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:
    rep ; ret                  # done:
Reduction in Strength

- Replace costly operation with simpler one
  - Shift, add instead of multiply or divide
    \[ 16 \times x \quad \rightarrow \quad x \ll 4 \]
  - Recognize sequence of products

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

```c
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

```c
/* 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: i*n, (i-1)*n, (i+1)*n

```assembly
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

```assembly
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;
```

```assembly
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

```c
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');
}
```

Question: What’s the big-O runtime of lower, O(n)?
Lower Case Conversion Performance

- Quadratic performance!
Calling strlen in loop

- Strlen takes $O(n)$ to finish
- Strlen is called $n$ times

```c
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');
}
```
Improving Performance

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

```c
void lower2(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');
}
```
Lower Case Conversion Performance

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

![Graph showing lower case conversion performance with CPU seconds on the y-axis and string length on the x-axis. The graph illustrates the linear performance of lower2 and the exponential growth of lower1.]
**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

**Warning:**
- Compiler treats procedure call as a black box
- Weak optimizations near them

**Remedies:**
- 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

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

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

```
int A[9] =
    { 0,   1,   2,
      4,   8,  16},
    32,  64, 128};
int *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]
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++) {
        int val = 0;
        for (j = 0; j < n; j++)
            val += a[i*n + j];
        b[i] = val;
    }
}

No need to store intermediate results
Optimization Blocker: Memory Aliasing

Aliasing

- Two different memory references specify single location
- Easy to 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
Getting High Performance

- Use compiler optimization flags
- Watch out for:
  - hidden algorithmic inefficiencies
  - Watch out for optimization blockers: procedure calls & memory aliasing
- Profile the program’s performance