Systems I

Code Optimization I: Machine Independent Optimizations

Topics

- Machine-Independent Optimizations
  - Code motion
  - Reduction in strength
  - Common subexpression sharing

- Tuning
  - Identifying performance bottlenecks
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 and generality
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

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 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

When in doubt, the compiler must be conservative
Machine-Independent 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

```c
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

- Most compilers do a good job with array code + simple loop structures

**Code Generated by GCC**

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

```
imull %ebx,%eax      # i*n
movl 8(%ebp),%edi    # a
leal (%edi,%eax,4),%edx # p = a+i*n (scaled by 4)
    # Inner Loop
movl 12(%ebp),%edi   # b
.L40:
    movl (%edi,%ecx,4),%eax # b+j  (scaled by 4)
movl %eax,(%edx)      # *p = b[j]
addl $4,%edx          # p++  (scaled by 4)
incl %ecx             # j++
cmpl %ebx,%ecx        # loop if j<n
jl .L40
```
Reduction in Strength

- Replace costly operation with simpler one
- Shift, add instead of multiply or divide
  \[16 \times x \rightarrow x \ll 4\]
  - Utility machine dependent
  - Depends on cost of multiply or divide instruction
  - On Pentium II or III, integer multiply only requires 4 CPU cycles
- Recognize sequence of products

```c
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        a[n*i + 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;
}
```
Make Use of Registers

- Reading and writing registers much faster than reading/writing memory

Limitation

- Compiler not always able to determine whether variable can be held in register
- Possibility of Aliasing
- See example later
Machine-Independent Opts. (Cont.)

Share Common Subexpressions

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

```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;
```

```c
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;
```

3 multiplications: i*n, (i−1)*n, (i+1)*n
1 multiplication: i*n

```assembly
leal -1(%edx),%ecx  # i-1
imull %ebx,%ecx     # (i-1)*n
leal 1(%edx),%eax   # i+1
imull %ebx,%eax     # (i+1)*n
imull %ebx,%edx     # i*n
```
Time Scales

Absolute Time

- Typically use nanoseconds
  - $10^{-9}$ seconds
- Time scale of computer instructions

Clock Cycles

- Most computers controlled by high frequency clock signal
- Typical Range
  - 100 MHz
    - $10^8$ cycles per second
    - Clock period = 10ns
  - 2 GHz
    - $2 \times 10^9$ cycles per second
    - Clock period = 0.5ns
Example of Performance Measurement

Loop unrolling
  - Assume even number of elements

```c
void vsum1(int n) {
    int i;
    for(i=0; i<n; i++)
        c[i] = a[i] + b[i];
}

void vsum2(int n) {
    int i;
    for(i=0; i<n; i+=2) {
        c[i] = a[i] + b[i];
        c[i+1] = a[i+1] + b[i+1];
    }
}
Cycles Per Element

- Convenient way to express performance of program that operators on vectors or lists
- Length = n
- \( T = \text{CPE} \times n + \text{Overhead} \)
void lower(char *s)
{
    int i;
    for (i = 0; i < strlen(s); i++)
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
}
Lower Case Conversion Performance

- **Time quadruples when string length doubles**
- **Quadratic performance**

![Graph showing CPU seconds vs. string length](image)
Convert Loop To Goto Form

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:
}

- `strlen` executed every iteration
- `strlen` linear in length of string
  - Must scan string until finds `\0`
- Overall performance is quadratic
Improving Performance

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

```c
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');
}
```
Lower Case Conversion Performance

- Time doubles when double string length
- Linear performance

![Bar Chart]

**CPU Seconds**

**String Length**

- lower1
- lower2
Optimization Blocker: Procedure Calls

Why couldn’t the compiler move `strlen` out of the 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`

Why doesn’t compiler look at code for `strlen`?
- Linker may overload with different version
  - Unless declared static
- Interprocedural optimization is not used extensively due to cost

Warning:
- Compiler treats procedure call as a black box
- Weak optimizations in and around them
Summary

Today
- Improving program performance (machine independent)
- Mostly focusing on instruction count

Next time
- Optimization blocker: procedure calls
- Optimization blocker: memory aliasing
- Tools (profiling) for understanding performance