15-213

"The course that gives CMU its Zip!"

Code Optimization I: Machine Independent Optimizations Feb 11, 2003

Topics

- Machine-Independent Optimizations
  - Code motion
  - Strength Reduction/Induction Var Elim
  - Common subexpression sharing
- Tuning
  - Identifying performance bottlenecks

Great Reality #4

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

The Bottom Line:

When in doubt, do nothing
i.e., The compiler must be conservative.

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

Machine-Independent Optimizations

- Optimizations that should be done 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

```
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
.L40:
  movl 12(%ebp),%edi      # b
  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++
  jl .L40                 # loop if j<n
```

Strength Reduction†

- 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 Pentium II or III, integer multiply only requires 4 CPU cycles
- Recognize sequence of products (induction var analysis)

```
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;
}
```

† As a result of Induction Variable Elimination

Make Use of Registers

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

Limitation

- Limited number of registers
- Compiler cannot always 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

```
/* 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 multiplies: i*n, (i–1)*n, (i+1)*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;
```

1 multiply: i*n

```
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
```

†AKA Common Subexpression Elimination (CSE)

Measuring Performance: 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 X 10^9 cycles per second
    - Clock period = 0.5ns
- Fish machines: 550 MHz (1.8 ns clock period)

Measuring Performance

For many programs, cycles per element (CPE)

- Especially true of programs that work on lists/vectors
- Total time = fixed overhead + CPE * length-of-list

```
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];
}
```

- vsum2 only works on even n.
- vsum2 is an example of loop unrolling.

Cycles Per Element

- Convenient way to express performance of a program that operates on vectors or lists
- Length = n
- T = CPE*n + Overhead