Performance and Optimization

CSC 230 : C and Software Tools

NC State Department of Computer Science

Topics for Today

- Leftover Topic: Environment Variables
- Code coverage
- Optimization Techniques
  - Timing program execution
  - Profiling
  - Optimizing by Hand
  - Compiler support for optimization

Environment Variables

- Your execution environment supports variables
  - We can access these from the shell

```
$ echo $HOME
/home/jerry
$ HOME=/home/linda
$ echo $HOME
/home/linda
```

- The shell uses these a lot for configuration and to customize behavior
  - `HOME`, `PATH`, `SHELL`, `PS1`, `EDITOR`, …
- But, these aren't just available to the shell, all programs have environment variables.

Using the Environment

- The standard library lets us get/set these variables
  - We need our good friend, `stdlib.h`
  - Then, we can gen environment variable values

```
char *str = getenv( “HOME” );
printf( “HOME = $s\n”, str );
```

  - And, we can change them

```
setenv( “HOME”, “/some/other/place”, 1 );
```

Variable to set.

new value.

Should I overwrite?

- These changes are local to the program, they go away when it exits.

Testing Testing

- How do you know if your tests are adequate?
  - Well, there are different types of tests you can perform
    - Unit testing, integration testing, system testing, regression testing, acceptance testing, beta testing
    - black-box vs white box
  - Each type intended to expose a different cause or type of bug.
- For acceptance testing, we could make sure we have a test or two for every (functional) requirement
- For black box testing, we can test output on typical and boundary cases
  - To make sure we cover the range of possible inputs

Code Coverage

- For white-box testing, we can make sure we cover the code itself.
- Ideally, all the source code should get a chance to run under test
  - Every statement executed
  - Every branch direction of every conditional
  - Every sub-expression of every compound conditional

These are details, but they could hide errors.

Tools for Code Coverage

- The compiler will help us measure this
- Compile with the -fprofile-arcs and -ftest-coverage flags

```
gcc -Wall -std=c99 -fprofile-arcs \
    -ftest-coverage bubble1.c -o bubble1
```

- Run your program all you like

```
./bubble1 < small_input
./bubble1 < some_other_input
```

Using gcov

- Notice, you get some extra files.

```
eos$ ls
bubble1          bubble1.c
bubble1.gcno     bubble1.gcda
```

written during execution

written by the compiler

- Run gcov to extract the coverage information

```
eos$ gcov bubble1.c
File 'bubble1.c'
Lines executed:66.67% of 36
bubble1.c:creating 'bubble1.c.gcov'
```

Reading gcov Output

- In addition to the summary, gcov writes a text file giving coverage details.

```
    2:   42:  for ( int i=0; !done && i<len; i++ ) {
    -:   43:    // If we make it through without …
    1:   44:    done = true;
    -:   45:
    5:   46:    for ( int j=0; j<len-i-1; j++ )
    4:   47:      if ( list[j ] > list[j+1] ) {
#####:   48:        swap( list + j, list + j + 1 );
#####:   49:        done = false;
    -:   50:      }
    -:   51:  }
    1:   52:}
```

number of times executed … line number

Reading gcov Output

- gcov has some codes for reporting execution count

5	I was executed 5 times
-	I could never be executed (this doesn't count against coverage)
#####	I was never executed (but I could have been)

- Let's see if we can get code coverage up to 100 %

Getting More from gcov

- By default, gcov tells you about statements executed
- It can also tell you how branches went

```
eos$ gcov -b bubble1.c
File 'bubble1.c'
Lines executed:66.67% of 36
Branches executed:100.00% of 18
Taken at least once:77.78% of 18
Calls executed:61.54% of 13
bubble1.c:creating 'bubble1.c.gcov'
```

Reading gcov Output

- With the -b option, gcov reports branching directions for each part of each conditional.

```
        1:   40:  bool done = false;
        -:   41:
        2:   42:  for ( int i=0; !done && i<len; i++ ){
branch  0 taken 50% (fallthrough)
branch  1 taken 50%
branch  2 taken 100%
branch  3 taken 0% (fallthrough)
        -:   43:    // If we make it through …
        1:   44:    done = true;
```

Which way did !done go?

Which way did i<len go?

Performance and Optimization

- Performance depends on
  - Coding Style
  - Choice of programming language
  - Compiler and Options
  - **Choice of Algorithm and Data Structure**

Code Profiling

- We can measure our program's performance using the time command.

```
time ./program program-options
```

- This reports how long it took the program to execute, CPU time consumed, etc.
- It's fairly coarse grained, we'd like to know more about where our program is spending its time.
- Code profiling tools let us do that.

Code Profiling

- gprof is a tool for profiling during execution
  - It counts the number of times each function is called
  - How much time we spend in each function
- To use gprof, we need to compile with the -pg option:

```
gcc -pg -Wall -std=c99 program.c -o program
```

Generating Profile Information

- Then, you can run your program as normal.

```
./program program-options
```

- This also generates some profiling output

```
$ ls
gmon.out        myProgram           myProgram.c
```

Where did this file come from?

Examining Profile Output

- The gmon.out file is in a binary format.
- But, the gprof command will print it in a way that's human readable.

```
$ gprof ./myProgram
<lots of output>
```

- Maybe it would be better to send this output to a file.

```
$ gprof ./myProgram > perf.txt
```

Fun With Time and Gprof

- Let’s run bubble4.c against the big_input.txt input file.
  - This file is large enough to take a few seconds to bubble sort.
- First, let’s time the execution
- Then, let’s profile:
  - Compile with profiling support
  - Run to collect profiling results
  - Examine results with gprof

DIY Optimizations : Code Motion

- You can optimize code yourself
  - By moving duplicate expression evaluation outside loops

```
for ( int i = 0; i < n; i++ )
  for ( int j = 0; j < n; j++ )
    a[ n * i + j ] = f() + g(i) + h(j);
```

```
int c1 = f();
for ( int i = 0; i < n; i++ ){
  int c2 = c1 + g(i);
  int ri = n * i;
  for ( int j = 0; j < n; j++ )
    a[ ri + j ] = c2 + h(j);
}
```

DIY Optimizations : Strength Reduction

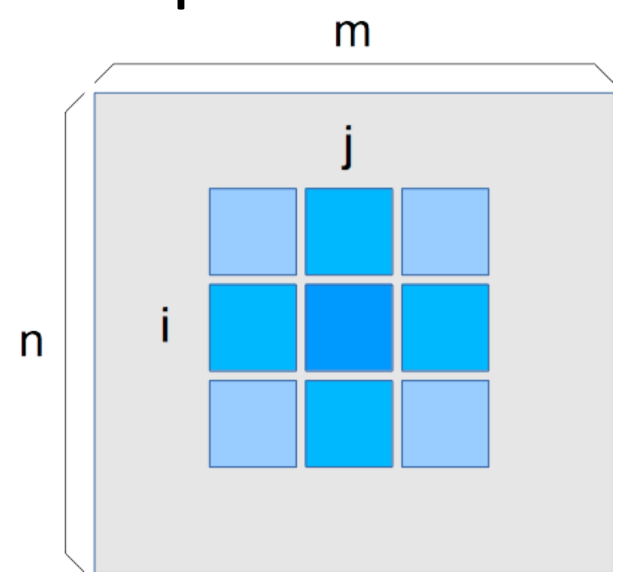
```
c1 = f();
for ( int i = 0; i < n; i++ ){
  c2 = c1 + g(i);
  int ri = n * i;
  for ( int j = 0; j < n; j++ )
    a[ ri + j ] = c2 + h(j);
}
```

```
c1 = f();
*p = a;
for ( int i = 0; i < n; i++ ){
  c2 = c1 + g(i);
  for ( int j = 0; j < n; j++ )
    *p++ = c2 + h(j);
}
```

DIY Optimization : Expression Reuse

- Consider this neighborhood computation:

```
sum  = a[ (i-1)*m + j    ];
sum += a[ (i+1)*m + j    ];
sum += a[ i*m +      j-1 ];
sum += a[ i*m +      j+1 ];
```




```
int ctr = i*m + j;
sum  = a[ ctr - m ];
sum += a[ ctr + m ];
sum += a[ ctr - 1 ];
sum += a[ ctr + 1 ];
```

DIY Optimization : Inlining

```
double prod( int i, int j )
{
  double sum = 0.0;
  for ( int k = 0; k<n; k++)
    sum += a[i][k] * b[k][j];
  return sum;
}
```

```
for (i=0; i<n; i++)
  for (j=0; j<n; j++)
    c[i][j] = prod(i, j);
```

DIY Optimization : Inlining

```
double prod( int i, int j )
{
  double sum = 0.0;
  for ( int k = 0; k<n; k++)
    sum += a[i][k] * b[k][j];
  return sum;
}
```

```
for (i=0; i<n; i++)
  for (j=0; j<n; j++) {
    double sum = 0.0;
    for (k=0; k<n; k++ )
       sum += a[i][k] * b[k][j];
    }
    c[i][j] = sum;
  }
```

Inlining Help from the Language

- C can inline your functions for you
  - Introduced in C99

```
inline double prod( int i, int j )
{
  double sum = 0.0;
  for ( int k = 0; k<n; k++)
    sum += a[i][k] * b[k][j];
  return sum;
}
```

```
for (i=0; i<n; i++)
  for (j=0; j<n; j++)
    c[i][j] = prod(i, j);
```

Help from the Compiler

- Many of these refinements can be done automatically by the compiler
- It can try to optimize for execution speed
  - Options -O, -O2 and -O3, in order of increasing optimization.
- Increasing optimization level can:
  - Increase code size
  - Possibly decreasing performance
- But, the compiler can't do everything

DIY Optimization : Reordering Tests

```
if ( height > 84 )             // rare
   reallyTall();
else if ( height > 72 )        // unusual
   tall();
else                           // typical
   average();
```

- Place more frequent branches earlier

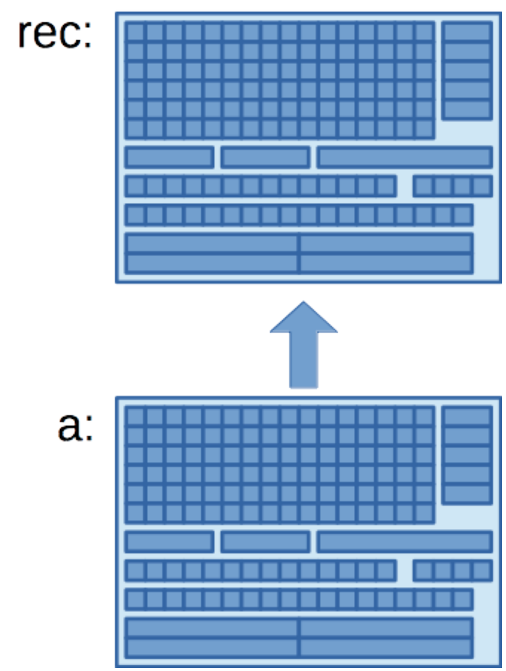
```
if ( height <= 72 )            // typical
   average();
else if ( height <= 84 )       // unusual
   tall();
else                           // rare
   reallyTall();
```

DIY Optimization : Pass by Reference

- C lets us use value semantics with most types
  - … maybe it's too easy.
  - Passing large objects by value can be expensive.

```
int f( struct BigRecord rec )
{
  int sum = rec.x + rec.y;
  return sum;
}
```

```
  struct BigRecord a;
  ...
  v = f( a );
```



DIY Optimization : Pass by Reference

- Pass by reference/address can reduce overhead copying large objects
- Const lets the function say what it plans to do with the object.

```
int f( const struct BigRecord *rec )
{
  int sum = rec->x + rec->y;
  return sum;
}
```

```
  struct BigRecord a;
  ...
  v = f( &a );
```