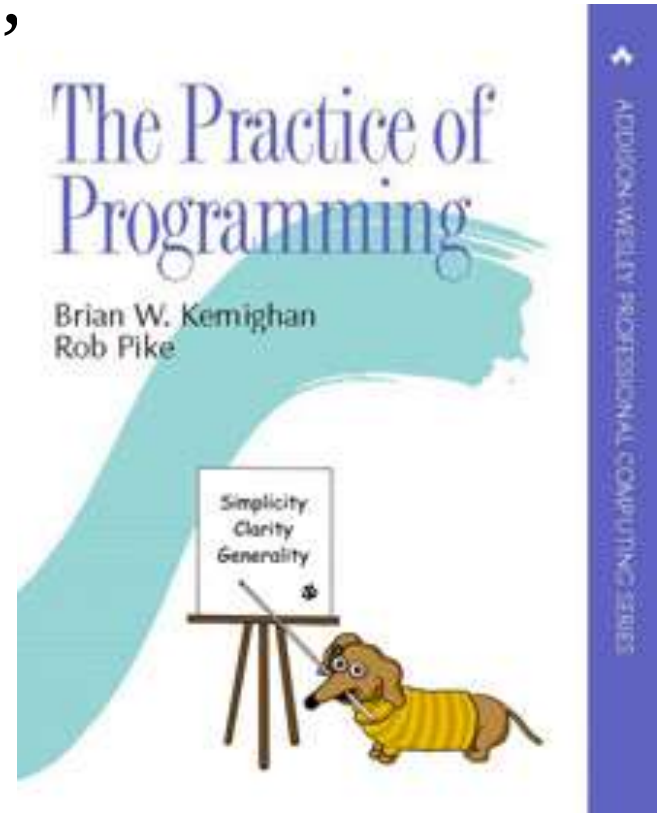
Performance Issues

*CIS*2750*

Advanced Programming Concepts

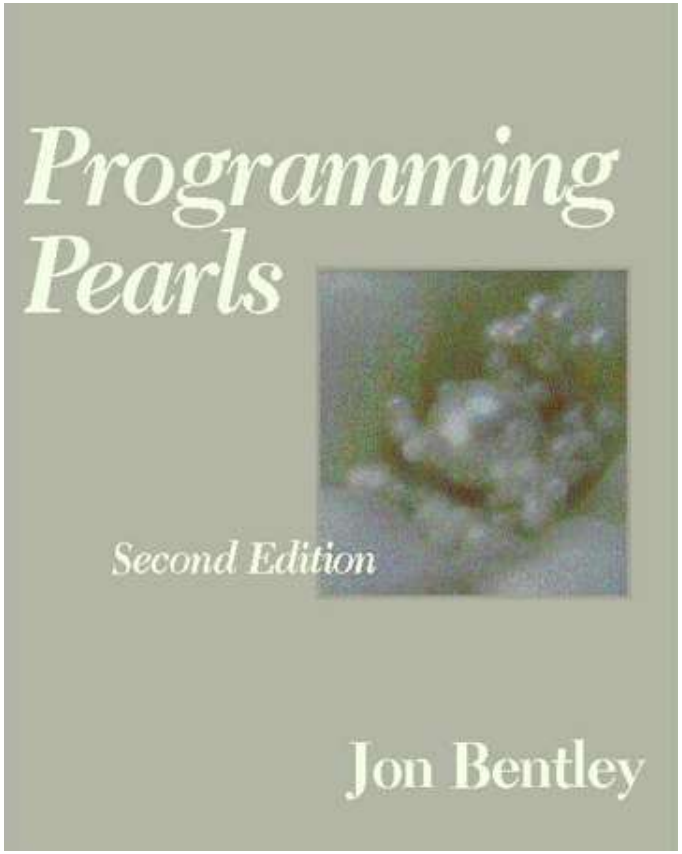
References

- **The Practice of Programming** by Brian W. Kernighan and Rob Pike,
  Addison-Wesley, 1999.



References

- **Programming Pearls**, 2nd Edition, by Jon Bentley, Addison-Wesley, 2000.



Performance Means *Speed!*

- Everyone loves a fast program
  - But is speed critical for your application?
  - Is it "fast enough" the way it is?
- Speeding up a program → **optimization**
  - **When** should you try to optimize a program?
  - **How** can this be done?
  - **What** are the costs vs. benefits?

The First Principle of Optimization

Don't!

6 Steps to Optimization

1. Start with the **simplest**, **cleanest** algorithms and data structures appropriate for the task.
    - Much better initial goals for coding than *speed:* easy to understand, simple to code
    - Fewer bugs, easier to maintain
    - **Exceptions:** systems with realtime constraints must plan to meet execution deadlines with given CPU

6 Steps to Optimization

2. **Measure** performance to see if changes are necessary (see below).
3. Enable **compiler options** to generate the fastest possible code.
    - More applies to “number crunching” apps
    - CIS*3090 Parallel Programming students shocked to find 20-30% faster run times simply by enabling full compiler optimization!

Steps to Optimization

4. **Assess** possible changes to the program and their effect -- choose the ones that have the most effect.
5. Make changes **one** at a time.
6. **Re-assess**. Compare to the original version for correctness and performance.
   - Critical to have "gold output" to compare to!

Performance Tools

Automated Timing Measurements

```
#include < time.h >
int main ( int argc, char *argv[] ) {
   clock_t before;        // 32-bit integer datatype from time.h
   double elapsed;
   before = clock();
   count = FileInput( f_input, &allstrings );
   elapsed = clock() - before;
   printf("FileInput function used %.3f seconds\n",
      elapsed/CLOCKS_PER_SEC);
```

Automated Timing Measurements

```
before = clock();
for ( j=0; j < count; j++ ) {
   i = PatternSearch ( j, allstrings, pattern );
   if ( i != -1 ) {
      printf("Found: line %d (%d)\n",j,i+1);
   }
}
elapsed = clock() - before;
printf("%d calls to the PatternSearch function used %.3f
    seconds\n", count,elapsed/CLOCKS_PER_SEC);
```

Timing Issues

- *What if the times are **too small** relative to clock resolution?*
  - **man clock_getres** → resolution in nanosec.
    - Typical clock() res: 1/100th sec; so millisec. is good!
  - Inflate time by doing N reps; divide t by N
- *Will the time be the same every time I measure it?*
  - Many sources of small variations
  - Measure several times and take average

Profilers

- A profile is a measurement of where a program spends its time.
- A profiler is an effective tool for finding **hot spots** in a program
  - functions
  - areas of sections of code

  which are consuming most of the processing time.

Parallel Profilers

- Multicore computers raise new challenge
  - Keeping *all* the cores busy!
    - Otherwise, going to waste, program could conceivably be faster by utilizing them.
  - Parallel profilers not only find hot spots, they keep track of core utilization
  - May guide programmer to more even work distribution aka *load balancing*
    - With all cores in use, program finishes sooner

Profilers

> gcc -pg slow1.c -o slow1
> slow1 Big_pattern
> gprof slow1

gmon.out file

Every 1/100 sec. it "samples" to see what function the program is currently in.

- Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self us/call	total us/call	name
97.92	0.47	0.47	28008	16.78	16.78	**PatternSearch**
2.08	0.48	0.01				mcount
0.00	0.48	0.00	1	0.00	0.00	**FileInput**
0.00	0.48	0.00	1	0.00	470000.00	main

Profilers

- In 1971, Donald Knuth wrote,

"...less than 4 per cent of a program generally accounts for more than half of its running time."

3 Strategies for Speed

Enable Compiler Optimization

- Without -O, the compiler's goal is to reduce the cost of compilation and to make execution match source code for debugging.
- *Optimizing compilation takes somewhat more time, and a lot more memory for large functions.*
- With -O2, etc., compiler tries to reduce code size and/or execution time.
- *Different compilers have different options!*

Use a **Better** *Algorithm* or *Data Structure*

- Most important factor in making a program faster.

Hot Spot Code

```
/* Search jth string in char all[][100] for *pat */
int PatternSearch( int j, char *all, char *pat ) {
    int i;
    for ( i=0; i < strlen(all+j*100); i++ ) {
       if ( strncmp(pat, ((all+j*100)+i), strlen(pat)) == 0 ) {
          return(i);          // *pat found at ith byte of all[j]
       }
    }
    return(-1);               // *pat not found in all[j]
}
```

IDEA: strncmp() is expensive to call, so test 1st letter before calling it!

Hot Spot after Modification

```
int PatternSearch ( int j, char *all, char *pat) {
   int i;
   char firstletter = *(pat+0);
   for ( i=0; i < strlen(all+j*100); i++ ) {
      if ( *((all+j*100)+i) == firstletter ) {
         if ( strncmp(pat, ((all+j*100)+i), strlen(pat)) == 0 ) {
            return(i);
         }
      }
   }
   return(-1);
}
```

Hot Spot

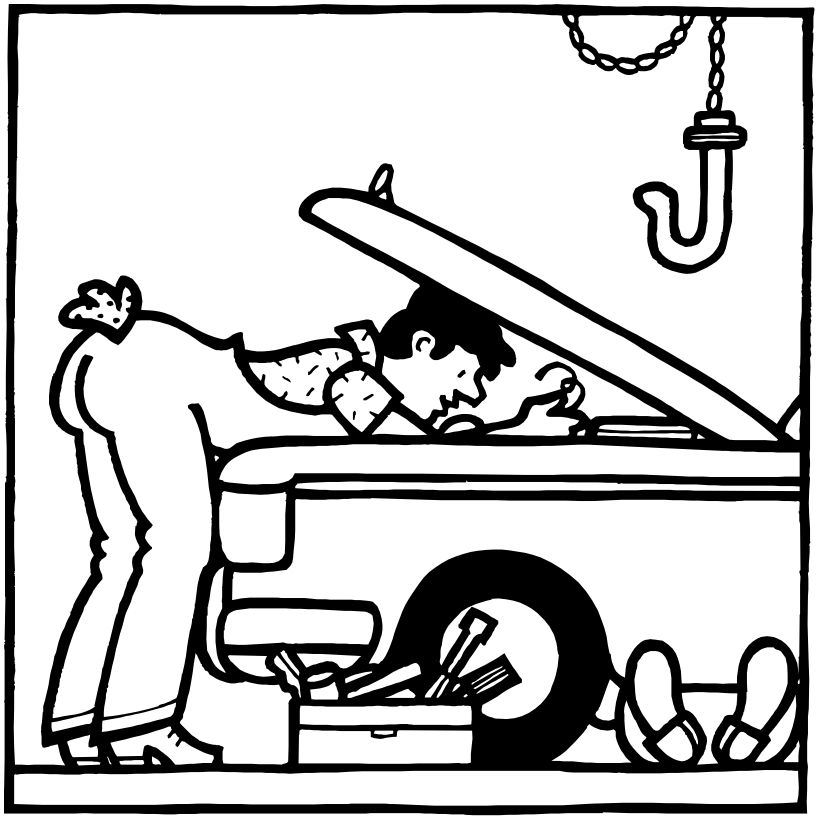
Timing Differences

- *Before*:
  - FileInput function used 0.250 seconds
  - 28008 calls to the PatternSearch function used **1.970** seconds
- *After*:
  - FileInput function used 0.250 seconds
  - 28008 calls to the PatternSearch function used **1.480** seconds

25% improvement

Tune the Code

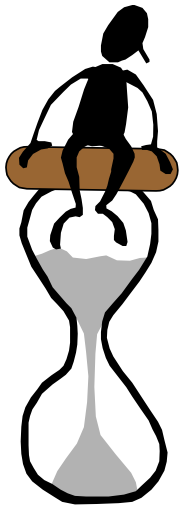
- Read the **Rules for Code Tuning** in *Programming Pearls* by Bentley.



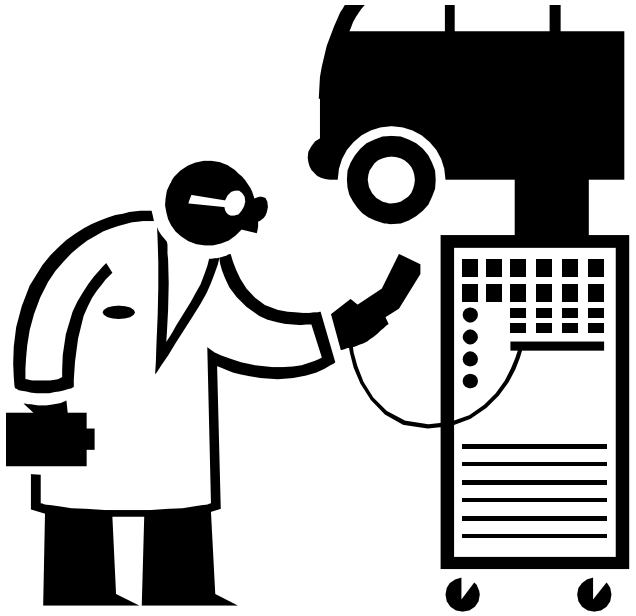
Don't Optimize What Doesn't Matter.

- Remember the *first principle of optimization*!

- Determine which time is more important:
  - Time to delivery
  - Execution time



Tuning



The PatternSearch Function

```
int PatternSearch ( int j, char *all, char *pat) {
   int i; char firstletter = *(pat+0);
   int n = strlen(all+j*100);
   int p = strlen(pat);
   for ( i=0; i < n; i++ ) {
      if ( *((all+j*100)+i) == firstletter ) {
         if ( strncmp(pat, ((all+j*100)+i), p) == 0 ) {
            return(i);
         }
      }
   }
   return(-1);
}
```

Moving invariant calc out of loop

Compiler optimization may move these out

"Code motion"

```
int PatternSearch ( int j, char *all, char *pat) {
    int i;
    for ( i=0; i < strlen(all+j*100); i++ ) {
      if ( strncmp(pat, ((all+j*100)+i), strlen(pat)) == 0 ) {
        return(i); …
```

Good, Better, Best?

```
int PatternSearch ( int j, char *all, char *pat) {
    int i;
    char firstletter = *(pat+0);
    for ( i=0; i < strlen(all+j*100); i++ ) {
      if ( *((all+j*100)+i) == firstletter ) {
        if ( strncmp(pat, ((all+j*100)+i), strlen(pat)) == 0 ) {
          return(i); …
```

```
int PatternSearch ( int j, char *all, char *pat) {
    int i; char firstletter = *(pat+0);
    int n = strlen(all+j*100);
    int p = strlen(pat);
    for ( i=0; i < n; i++ ) {
      if ( *((all+j*100)+i) == firstletter ) {
        if ( strncmp(pat, ((all+j*100)+i), p) == 0 ) {
          return(i); …
```

Timing

- FileInput function used 0.250 seconds
- 28008 calls to the PatternSearch func used **1.970** secs

- FileInput function used 0.250 seconds
- 28008 calls to the PatternSearch func used **1.480** secs

- FileInput function used 0.260 seconds
- 28008 calls to the PatternSearch func used **0.280** secs

Profiling

% time	cumulative seconds	self seconds	calls	self us/call	total us/call	name
97.92	0.47	0.47	28008	16.78	16.78	PatternSearch
2.08	0.48	0.01				mcount
0.00	0.48	0.00	1	0.00	470000.00	main
100.00	0.37	0.37	28008	13.21	13.21	PatternSearch
0.00	0.37	0.00	1	0.00	370000.00	main
92.59	0.25	0.25	28008	8.93	8.93	PatternSearch
3.70	0.27	0.01				mcount
0.00	0.27	0.00	1	0.00	260000.00	main

Tuning for *Speed*

Unroll or Eliminate Loops

- There is overhead involved in setting up and running a loop.
- If the body of the loop is not too long and there are not too many iterations, it can be more efficient to *eliminate* the loop.

Unroll or Eliminate Loops

- ***The Loop***

```
for ( j=0; j < 1000000; j++ ) {
   for ( i=0; i < 3; i++ ) {
      a[i] = b[i] + c[i];
   }
}
```

- ***No Loop***

```
for ( j=0; j < 1000000; j++ ) {
   a[0] = b[0] + c[0];
   a[1] = b[1] + c[1];
   a[2] = b[2] + c[2];
}
```

Unroll or Eliminate Loops

Timing

> loop1

- 1000000 repetitions of the inner loop used **0.290** seconds

> loop2

- 1000000 repetitions of the additions used **0.080** seconds

Unroll or Eliminate Loops

- **Longer Loops**
  - If the loop is longer, you can transform the loop to eliminate *some* of the looping
  - "Partial unrolling"

```
for ( j=0; j < 1000000; j++ )
  for ( i=0; i < 99; i++ )
    a[i] = b[i] + c[i];
```

- *Becomes...*

```
for ( j=0; j < 1000000; j++ ) {
  for ( i=0; i < 99; i += 3 ) {
    a[i]   =  b[i]   + c[i];
    a[i+1] =  b[i+1] + c[i+1];
    a[i+2] =  b[i+2] + c[i+2];
  }
}
```

cuts no. of inner reps. by 2/3

Unroll or Eliminate Loops

Timing

- If inner loop had only 9 (not 99) iterations:
  - The loop used 0.720 seconds
  - The modified loop used 0.820 seconds
- For 99 iterations:
  - The loop used 10.670 seconds
  - The modified loop used 10.210 seconds
- If 198 iterations:
  - The loop used 24.010 seconds
  - The modified loop used 21.160 seconds

Parting Words

- How do you know which techniques help?
  - Try several and *measure* with typical data (=benchmarks)
- Don't underestimate cleverness of compiler optimization! *e.g., gcc profile-directed opt:*
  - Profile generation (-fprofile-generate)
  - Training run → profile saved to file
  - Feedback optimization (-fprofile-use)
- Always verify correctness (make sure you didn't break program)

Suppose one CPU still too slow

- It's a multicore world!
  - Requires *parallel programming* to take advantage of 2, 4, 8, + processors
  - Use language that supports threads: **Java**
  - Write multithreaded code: **pthreads**
  - Use compiler-organized threads: **OpenMP**
- **CIS*3090 Parallel Programming course**
  - Also includes programming for (non-shared memory) high-performance clusters (SHARCNET)