

Optimization: Introduction

CPSC 501: Advanced Programming Techniques
Fall 2020

Jonathan Hudson, Ph.D
Instructor
Department of Computer Science
University of Calgary

Wednesday, August 5, 2020



Definition

Optimization

- ***Optimization*** is the process of modifying a program to improve its efficiency
 - Increase its speed
 - Reduce its size (memory usage)

- **Optimization can often be seen as de-factoring**
 - **Program gets faster but...**
 - **Harder to understand, upkeep, read**

Efficiency

- Efficiency can be viewed in terms of:
 1. Program requirements
 - Does the program really need to run at a certain speed? Is it worth the extra effort
 2. Program design
 - If performance is important, design a performance-oriented architecture
 - Set resource goals for individual subsystems and classes
 3. Class and routine design
 - Choose efficient algorithms and datatypes
 - E.g. Quicksort vs. bubble sort
 - E.g. Binary search vs. linear search

Efficiency (cont'd)

4. Operating system interactions
 - Working with files, dynamic memory, or I/O devices means using system calls
 - May be slow or fast
5. Code compilation
 - Good compilers produce optimized machine code
 - May have options for different optimization levels

Efficiency (cont'd)

6. Hardware

- A hardware upgrade may be the cheapest way to improve a program's performance
 - Not always possible

7. Code tuning

- Small-scale changes made to code to make it run more efficiently
 - At the level of a single routine, or a few lines of code
- Tends to produce hard-to-understand code
 - Obscures design

Guide to the galaxy of optimization

General Guidelines

- **Don't** optimize as you go
 - Focusing on optimization during initial development detracts from achieving correctness, readability, and design quality

General Guidelines (cont'd)

- Jackson's Rules of Optimization:
 - Rule 1. **Don't do it.**
 - Rule 2 (for experts only). **Don't do it yet**—that is, not until you have a perfectly clear and unoptimized solution.
- Code tuning should be done only as a last step
 - Knuth: **Pre-mature optimization is the root of all evil**

General Guidelines (cont'd)

- Optimize bottlenecks
 - **The 80/20 rule:** 20% of program's routines consume 80% of its execution time
 - Knuth found 4% of a FORTRAN program accounted for over 50% of its run time
 - Spend your time fixing these 'bottlenecks'
 - **Don't waste effort on the other parts**

General Guidelines (cont'd)

- Measure performance when optimizing
 - Use a profiler to find bottlenecks
 - Use timers to measure CPU time
 - Make sure a change actually improves speed
 - May actually make things worse when using a different compiler, OS, or processor
- Run regression tests after each optimization
 - Make sure your program is still correct

Swipe right

Profiling

- **Profiling**
 - Is used to find how much time is spent in each function of a program
 - Helps find bottlenecks
 - Helps you compare the performance of algorithms or programs

Profiling (cont'd)

- Works by sampling the program counter (PC register)
 - Periodically queries the program, recording the function in which it is running
- Is statistical in nature
 - i.e. is somewhat inexact, and will vary from run to run
- Also the act of enabling profiling will generally slow down operation of code, this slowdown can be different for varying classes

C++ profiling

Profilers – gcc

- Available UNIX profilers for programs compiled with gcc:
 - prof (most commonly used)
 - gprof
 - pixie

Profilers – gcc (cont'd)

- Using prof:
 - Compile the program with the -p option
 - E.g.
`gcc -c myprog.c`
`gcc -o myprog -p myprog.o`
 - Run the program
 - E.g.
`./myprog`
 - Produces the file mon.out
 - Print the profile report to stdout
 - E.g. `prof myprog mon.out`

Profilers – gcc (cont'd)

- Example output:

Each sample covers 8.00 byte(s) for 0.17% of 0.5742 seconds

%time	seconds	cum %	cum sec	procedure (file)
38.1	0.2188	38.1	0.22	mycompare (<myprog>)
36.1	0.2070	74.1	0.43	unique (<myprog>)
22.1	0.1270	96.3	0.55	bubble_sort (<myprog>)
3.6	0.0205	99.8	0.57	extract (<myprog>)
0.2	0.0010	100.0	0.57	main (<mysort>)

Timestamp profiling

Profilers - Timing

- Timing measurements
 - In UNIX, can use the time command to time an entire program
 - E.g.

```
time java Test
```

```
1.09u 0.12s 0:01.27 95.2%
```

```
<user CPU time> <system CPU time> <real_time>
```

Profilers – C/C++ clock()

- In C and C++, use the clock() function to measure the CPU time used by a function or section of code
 - E.g.

```
#include <time.h>
#include <stdio.h>
. . .
    clock_t before;
    double elapsed;

    before = clock();
    long_running_function();
    elapsed = clock() - before;
    printf("function used %.3f seconds\n",
          elapsed/CLOCKS_PER_SEC);
```

Profilers – C/C++ clock() (cont'd)

- If the function takes a fraction of a second, run it in a loop to get a more accurate measurement

- E.g.

```
before = clock();  
for (i = 0; i < 1000; i++)  
    short_running_function();  
elapsed = (clock() - before) / (double) i;
```

Profilers – nanoTime()

- In Java, use the nanoTime() method
 - E.g.

```
long startTime = System.nanoTime();
longRunningMethod();
long elapsedTime = System.nanoTime() - startTime;
```
 - Note: result is in nanoseconds (10^{-9} s)

Java profiling

Profilers – Java

- Standard JVM Profilers

- VisualVM, JProfiler, YourKit and Java Mission Control
- method calls and memory usage

- **Pros:**

- Great for tracking down memory leaks, standard profilers detail out all memory usage by the JVM and which classes/objects are responsible.
- Good for tracking CPU usage and zero in on hot spots.

Profilers – Java

- Standard JVM Profilers

- VisualVM, JProfiler, YourKit and Java Mission Control
- method calls and memory usage
- **Cons:**
 - Requires a direct connection to the monitored JVM; this ends up limiting usage to development environments in most cases.
 - They slow down your application; a good deal of processing power is required for the high level of detail provided.

Profilers – Java

- Lightweight Java Transaction Profilers
 - XRebel and Stackify Prefix
 - Aspect Profilers
 - use aspect-oriented programming (AOP) to inject code into the start and end of specified methods.
 - Java Agent profilers (ex. Netbeans built-in)
 - use the Java Instrumentation API to inject code into your application. This method has greater access to your application since the code is being rewritten at the bytecode level.

Profilers – Java

- Lightweight Java Transaction Profilers
 - Aspect profilers are pretty easy to setup but are limited in what they can monitor and are encumbered by detailing out everything you want to be tracked.
 - Java Agents have a big advantage in their tracking depth but are much more complicated to write.

Profilers – Java

- Low Overhead, Java JVM Profiling in Production (APM – APplication Monitoring)
 - New Relic, AppDynamics, Stackify Retrace, Dynatrace
 - how your system performs in production is critical
 - Java APM tools typically use the Java Agent profiler method
 - different instrumentation rules to allow them to run without affecting performance in productions.

Algorithm Based Optimization

Algorithm-Based Optimization

- Choosing a more efficient algorithm or data structure is often the best way to improve program efficiency
 - Look for algorithms that reduce the order of complexity
 - E.g. Binary search $O(\log n)$ vs. linear search $O(n)$
 - E.g. Merge sort $O(n \log n)$ vs. bubble sort $O(n^2)$

Algorithm-Based Optimization

- Do this first before attempting other optimizations
 - Hand tuning an $O(n^2)$ algorithm won't yield near the same gains as using an $O(n \log n)$ algorithm
- **Beware of worst-case performance**
 - Some algorithms may not achieve their average Big-O performance under certain conditions
 - E.g. The quicksort degenerates to $O(n^2)$ with nearly-sorted inputs

Algorithm-Based Optimization

- Sometimes an inefficient algorithm is fine for small inputs
 - The overhead of a complicated algorithm may make it slower than a simple one
 - And harder to debug and maintain!
 - Measure performance to make sure you've made the right choice

Algorithm-Based Optimization

- Sometimes an inefficient algorithm is fine for small inputs
 - **Java's own internal Quick Sort uses an Insertion Sort below a specific array size**

Compiler Based Optimization

Compiler-Level Optimization

- Enabling compiler optimization can improve speed by as much as 2 times
- Most compilers turn off optimization by default
 - Optimized code tends to confuse debuggers
- Works best with straightforward code
 - Hand tuned code may actually be harder for the compiler to optimize

Compiler-Level Optimization

- Some compilers optimize better than others
- E.g.

Compiler (C++)	Time without optimization	Time with optimization	Savings
1	2.21	1.05	52%
2	2.78	1.15	59%
3	2.43	1.25	49%

Compiler-Level Optimization

- Aggressive optimizers may introduce bugs
 - Rerun regression tests to ensure correctness
- gcc optimization flags:
 - Optimize: **-O** or **-O1**
 - Optimize even more: **-O2**
 - Optimize yet more: **-O3**
 - Don't optimize (default): **-O0**

Compiler-Level Optimization

- E.g. `gcc -O2 -o myprog myfile.c`

Onward to ... logic optimization.

Jonathan Hudson
jwhudson@ucalgary.ca
<https://pages.cpsc.ucalgary.ca/~hudsonj/>



UNIVERSITY OF
CALGARY