

Goals of this Lecture



8

• Help you learn about:

- Techniques for improving program performance
 How to make your programs run faster and/or use less memory
- The GPROF execution profiler

• Why?

- In a large program, typically a small fragment of the code consumes most of the CPU time and/or memory
- A power programmer knows how to identify such code fragments
- A power programmer knows techniques for improving the performance of such code fragments

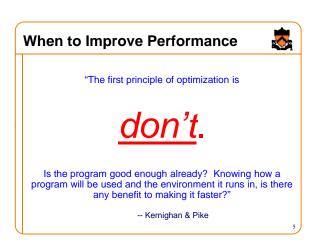
Performance Improvement Pros

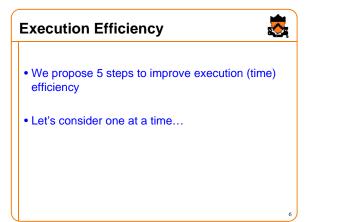
• Techniques described in this lecture can yield answers to questions such as:

- How slow is my program?
- Where is my program slow?
- Why is my program slow?
- How can I make my program run faster?
- How can I make my program use less memory?

- Is less clear/maintainable Might confuse debuggers
- Might contain bugs
- Requires regression testing

• So...





Timing Studies



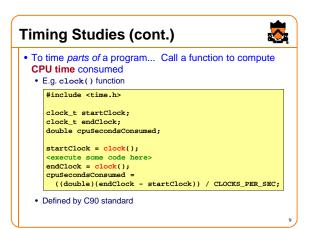
(1) Do timing studies

To time a program... Run a tool to time program execution
 E.g., Unix time command
 \$ time sort < bigfile.txt > output.txt

o crme	Sort < Digitie.txt > Output.txt	
real	0m12.977s	
user	0m12.860s	
sys	0m0.010s	

• Output:

- Real: Wall-clock time between program invocation and termination
 User: CPU time spent executing the program
- System: CPU time spent within the OS on the program's behalf
- But, which *parts* of the code are the most time consuming?





Identify Hot Spots

(2) Identify hot spots

Gather statistics about your program's execution

2

10

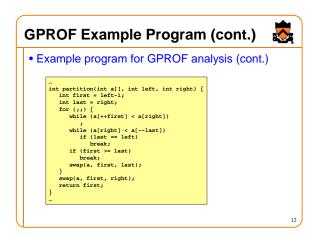
11

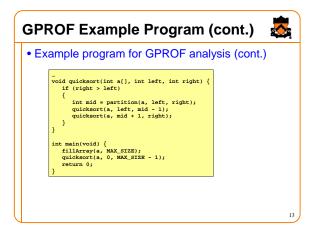
- How much time did execution of a function take?
- How many times was a particular function called?
- · How many times was a particular line of code executed? • Which lines of code used the most time?
- Etc.

• How? Use an execution profiler

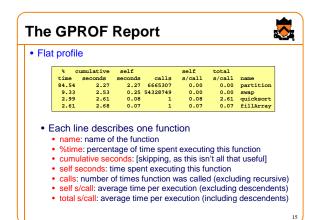
• Example: gprof (GNU Performance Profiler)

. **GPROF Example Program** • Example program for GPROF analysis • Sort an array of 10 million random integers · Artificial: consumes much CPU time, generates no output #include <string.h> #include <stdio.h> #include <stdlib.h> enum {MAX_SIZE = 10000000}; int a[MAX_SIZE]; /* Too big to fit in stack! */ void fillArray(int a[], int size) { int i; int i; int i; for (i = 0; i < size; i++) a[i] = rand(); 3 void swap(int a[], int i, int j) { int temp = a[i]; a[i] = a[j]; a[j] = temp;









	aph prof				
	apii piui				
inde	x % time	self	childre	n called	name
					<pre><spontaneous></spontaneous></pre>
[1]	100.0				main [1]
					quicksort [2]
		0.07	0.00	1/1	fillArray [5]
				12220614	quicksort [2]
		0 08		1/1	
[2]					14 guicksort [2]
	27.11				7 partition [3]
				13330614	
		2.27	0.25	6665307/666530	7 quicksort [2]
[3]	94.4	2.27	0.25	6665307	partition [3]
		0.25	0.00	54328749/54328	749 swap [4]
					749 partition [3]
				54328749	
				1/1	
					fillArray [5]



The GPROF Report (cont.)



17

8

• Call graph profile (cont.) • Each section describes one function

- Which functions called it, and how much time was consumed?
 Which functions it calls, how many times, and for how long?
 Usually overkill; we won't look at this output in any detail

GPROF Report Analysis

• Observations

• swap() is called very many times; each call consumes little time; swap() consumes only 9% of the time overall

- partition() is called many times; each call consumes little time; but partition() consumes 85% of the time overall
- Conclusions
 - To improve performance, try to make partition() faster • Don't even think about trying to make fillArray() or
 - quicksort() faster

GPROF Design

- Incidentally...
- How does GPROF work?
 - Good question!
 - Essentially, by randomly sampling the code as it runs
 - $\ensuremath{\,\bullet\,}$... and seeing what line is running, & what function it's in

Algorithms and Data Structures

ures 🙇

.

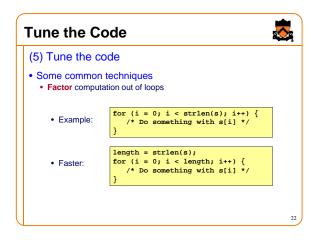
19

20

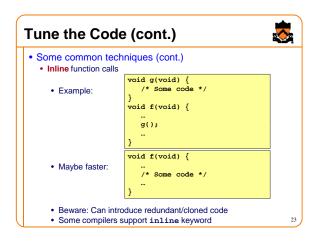
(3) Use a better algorithm or data structure

- Example:
 - For mysort, would mergesort work better than quicksort?
- Depends upon:
 - Data
 - HardwareOperating system
 - ...

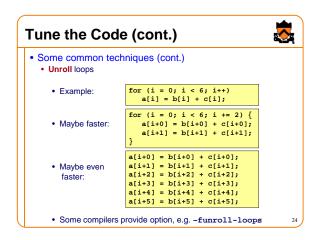
- Compiler Speed Optimization (4) Enable compiler speed optimization gcc217 -Ox mysort.c -o mysort • Compiler spends more time compiling your code so... • Your code spends less time executing • x can be: • 1: optimize • 2: optimize more • 3: optimize yet more • See "man gcc" for details
 - Beware: Speed optimization can affect debugging
 E.g. Optimization eliminates variable => GDB cannot print value of variable















- Some common techniques (cont.):
 - Rewrite in a lower-level language
 - Write key functions in assembly language instead of C · Use registers instead of memory
 - Use instructions (e.g. adc) that compiler doesn't know
 - Beware: Modern optimizing compilers generate fast code • Hand-written assembly language code could be slower than compiler-generated code, especially when compiled with speed optimization

Execution Efficiency Summary



25

.

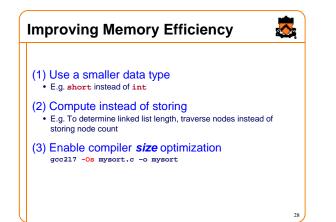
- Steps to improve execution (time) efficiency: (1) Do timing studies
 - (2) Identify hot spots

 - (3) Use a better algorithm or data structure (4) Enable compiler speed optimization
 - (5) Tune the code

Improving Memory Efficiency



- These days, memory is cheap, so...
- Memory (space) efficiency typically is less important than execution (time) efficiency
- Techniques to improve memory (space) efficiency...



Summary



- Steps to improve execution (time) efficiency:
 - (1) Do timing studies
 - (2) Identify hot spots *
 - (3) Use a better algorithm or data structure(4) Enable compiler speed optimization
 - (5) Tune the code
 - * Use GPROF
- Techniques to improve memory (space) efficiency: (1) Use a smaller data type (2) Occurrent interact of station
 - (2) Compute instead of storing(3) Enable compiler size optimization
- And, most importantly...

