Assignment 1

CS 531
Due January 16, 2019

Jason Graalum

January 13, 20190

- Read the Class Syllabus (the class webpage is cs.pdx.edu/ karavan/perf). ✓
- We will follow these steps in the Linux Lab during our class period:
- Read and Step through the examples in the Introduction to Gprof link from the class Lectures page
- Read the gprof paper (the 2004 version):
  Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. 2004. gprof: a call graph execution profiler. SIGPLAN Not. 39, 4 (April 2004), 49-57. ✓

Submit your answers to the following questions:

Question 3A gprof hands-on
In this exercise you will use gprof to examine and improve a piece of code called simpleCode.c. Turn in your answer to question 4.

1. Look over the source code provided. Build and run the code. Step through the code in the debugger, examining the values of number and searchkey. ✓
2. Evaluate the code using gprof ✓
3. Define and try a few changes to the code to improve the performance ✓

4. Write a short summary of what you learned about the code, what you tried and why, the results, and whether or not the results matched your expectations.

   The simpleCode.c program fills an array of integers with values equal to the location index. Then the program repeatedly chooses a random integer within the bounds of the array and then executes a iterative binary search until the number is found.

   I run the simpleCode program in four configurations: original, recursive call, flat program, and optimized search. For the recursive call version, I simply replaced the iteration in the BinarySearch function with a recursive call. For the flat program version, I removed the BinarySearch function and included the search iteration in the main program. Finally, for the optimize search version, I modified the iteration to move one of the comparisons outside the search loop.

   I initially ran each program with 10^6 searches. However, after running each several times, I notice a large deviation between runs. I decided to increase each run to 10^7 searches with expectations that the variation between runs would decrease.

Sample Results

Orginal Code

Flat profile:
Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	self calls	total us/call	us/call	name
97.01	23.03	23.03	9999999	2.30	2.30	BinarySearch
2.99	23.74	0.71				main

Call graph
granularity: each sample hit covers 4 byte(s) for 0.04% of 23.74 seconds

index	% time	self	children	called	name
					<spontaneous>
[1]	100.0	0.71	23.03		main [1]
		23.03	0.00	9999999/9999999	BinarySearch [2]
		23.03	0.00	9999999/9999999	main [1]
[2]	97.0	23.03	0.00	9999999	BinarySearch [2]

Recursive Code

Flat profile:
Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	self calls	total us/call	us/call	name
97.48	25.90	25.90	9999999	2.59	2.59	BinarySearch
2.52	26.57	0.67				main

Call graph
granularity: each sample hit covers 4 byte(s) for 0.04% of 26.57 seconds

index	% time	self	children	called	name
					<spontaneous>
[1]	100.0	0.67	25.90		main [1]
	25.90	0.00		9999999/9999999	BinarySearch [2]
				179514089	BinarySearch [2]
	25.90	0.00		9999999/9999999	main [1]
[2]	97.5	25.90	0.00	9999999+179514089	BinarySearch [2]
				179514089	BinarySearch [2]

Flat Code

Flat profile:
Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	self calls	total Ts/call	Ts/call	name
100.72	23.52	23.52				main

Optimized Search Code

Flat profile:
Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	self calls	total us/call	us/call	name
97.57	21.24	21.24	9999999	2.12	2.12	BinarySearch
3.10	21.92	0.68				main

Call graph
granularity: each sample hit covers 2 byte(s) for 0.05% of 21.94 seconds

index	% time	self	children	called	name
					<spontaneous>
[1]	99.9	0.68	21.24		main [1]
		21.24	0.00	9999999/9999999	BinarySearch [2]
		21.24	0.00	9999999/9999999	main [1]
[2]	96.8	21.24	0.00	9999999	BinarySearch [2]
					<spontaneous>
[3]	0.1	0.02	0.00		frame_dummy [3]

Question 3B Questions on the reading:

1. What is the difference between their "dynamic call graph" and "static call graph"? Can you give an example of something that would not appear in a dynamic call graph but would appear in a static call graph?

A dynamic call graph is built from execution of the program. Only functions called during execution are included in the graph. A static call graph is built directly from the program text by including all possible function calls(not including any functional parameters.)

Any function in the program which is not called during execution, for instance a function which prints a help message or error message, will not be included in the dynamic call tree while it will be included in the static call tree.

2. [old paper alert] What is a "time-sharing" system? We don't use this term much anymore, but we use time-sharing systems all the time. What is an example of one you use at PSU?

   Time-sharing is a term from the 1960s which described concurrently sharing of computer resources between multiple users and/or running programs. Today, we take for granted that most OS operating systems support some level of resource sharing. In face, because current OSs share time in non-deterministic ways(usually), real-time operating systems(RTOSs) are now needed for applications which are sensitive to this non-determinism - for instance many embedded sensor systems use RTOSs.

   Any of the linux systems, such as ruby, are time-sharing systems.

3. How accurate are the % time amounts reported by gprof?

   Because gprof does not actually use a timer to report a functions running time, but a sampling of the program counter, the reported times are subject to typical sampling errors. Per the gprof paper, an appropriate sampling period must be selected which provides enough samples for an accurate estimate while not including so many samples such that the profiler has a significant impact on the performance of the program. In addition to the sampling period, the total number of samples impacts the accuracy. This total number is related to both the total running time of the program and individual functions and the size(in machine instructions) of each functions. So, we can see that the accuracy of a specific functions timing is proportional to the size of the function times the number of times the function is called during

program execution - the total number of times instructions from that function are executed.

4. Gprof samples the program counter – what does this contain? What is the key benefit of this method? What is an important downside to this method?

   The program counter(PC) contains the address of the instruction currently under execution. By sampling this value and by knowing the instruction address space for each function, the gprof tool can estimate how often a program was executing each function. This method provides the advantage of less impact to the overall program performance over actual timing of the function calls. Such timing would require many calls/interrupts into the kernels timing module. The disadvantage of the sampling method, as describe in the answer to the above question, is that some accuracy(due to sampling error) is exchanged for the lesser impact on performance.

5. [From the 2003 update] What was one challenge the authors mention to using the profiler on the operating system kernel?

   In the update to the original article, the author highlight that, since the kernel runs continuously, there is no point for the original gprof program to exit and report out results. Therefore, the authors modified gprof to include a programmers interface which could control the start, stop and reporting of the tool independent of the lifetime of the program being profiled.
   Another challenge they encountered was "interaction major subsystems" with "large cycles". These large cycles - i.e. a cycle in the call graph with many nodes - also had a low number of call counts resulting in less useful profiling data. To account for this challenge, the authors added filtering(both manual and automatic) to the tool. Filtering these large, infrequently called cycles provided a cleaner set of profiling data set.

Question 3c Showcasing gprof

Develop a code in C or C++ that yields interesting results with gprof. You may use "example code 2" as a starting point. Think about what characteristics in the source code might lead to any of the following: an illustration of a case where gprof results are not quite accurate; an illustration of a difference between a longer function called a few times versus a shorter function called a large number of times; recursion; or any other gprof feature you want to highlight. Turn in your code plus a brief description of what it shows.

I wrote a short program with two mutually recursive functions - each which call a common function. The common function is a simple loop which takes an integer as an argument defining the number of loops to execute.

I ran the program in two configurations. In the first, funcA and funcB call loop an equal number of times. As we see from the output of gprof, the percentage of time in loop is equally divided between calls from funcA and funcB.

Equal Loop Count between funcA and funcB

Flat profile:
Each sample counts as 0.01 seconds.

index	% time	self	children	called	name
[1]	100.0	0.01	19.15	1+80000	<cycle 1 as a whole> [1]
		0.01	9.57	40001	funcA <cycle 1> [4]
		0.00	9.57	40000	funcB <cycle 1> [5]
					<spontaneous>
[2]	100.0	0.00	19.16	main [2]	
		0.01	19.15	1/1	funcA <cycle 1> [4]
		9.57	0.00	40000/80000	funcA <cycle 1> [4]
		9.57	0.00	40000/80000	funcB <cycle 1> [5]
[3]	99.9	19.15	0.00	80000	loop [3]
				40000	funcB <cycle 1> [5]
[4]	50.0	0.01	9.57	40001	funcA <cycle 1> [4]
		9.57	0.00	40000/80000	loop [3]
				40000	funcB <cycle 1> [5]
				40000	funcA <cycle 1> [4]
[5]	50.0	0.00	9.57	40000	funcB <cycle 1> [5]
		9.57	0.00	40000/80000	loop [3]
				40000	funcA <cycle 1> [4]

In the second configuration, I fixed the number of times loop is called from funcB to a small constant. The expectation is that gprof will show that the vast majority of calls to loop are from funcA. However, because gprof must group mutually recursive calls, it even distributes calls to functions outside of the recursion. In the case, gprof still reports an equal share of calls to loop between funcA and funcB.

Offset Loop Count between funcA and funcB

Flat profile:
Each sample counts as 0.01 seconds.

index	% time	self	children	called	name
[1]	100.0	0.01	9.58	1+80000	<cycle 1 as a whole> [1]
	0.01	4.79	40000		funcB <cycle 1> [4]
	0.00	4.79	40001		funcA <cycle 1> [5]
					<spontaneous>
[2]	100.0	0.00	9.59		main [2]
		0.01	9.58	1/1	funcA <cycle 1> [5]
		4.79	0.00	40000/80000	funcA <cycle 1> [5]
		4.79	0.00	40000/80000	funcB <cycle 1> [4]
[3]	99.9	9.58	0.00	80000	loop [3]
				40000	funcA <cycle 1> [5]
[4]	50.1	0.01	4.79	40000	funcB <cycle 1> [4]
		4.79	0.00	40000/80000	loop [3]
				40000	funcA <cycle 1> [5]
				40000	funcB <cycle 1> [4]
		0.01	9.58	1/1	main [2]
[5]	49.9	0.00	4.79	40001	funcA <cycle 1> [5]
		4.79	0.00	40000/80000	loop [3]
				40000	funcB <cycle 1> [4]

This example shows the limitation gprof has when estimating the performance of recursive functions. Being unable to correctly segment the calls to the recursion, gprof combines the recursive functions into a single node. It then splits any calls out of the recursive functions evenly between those functions. Correctly optimizing the performance of recursive routines requires manual examination of the functions to identify the correct call-site.

Code Listing

```
#include <stdio.h>
#include <stdlib.h>
int funcA(int);
int funcB(int);

void loop(int c)
{
        int x = 0;
        while (c--) { x++; }
        return;
}
int funcA(int a) {
        if (a > 0) {
                loop(a);
                return funcB(--a);
        } else return 0;
}

int funcB(int b) {
        if (b > 0) {
                loop(b); // Replace with loop(10); for offset loop case
                return funcA(--b);
        } else return 0;
}

int main() {
        int count = 100;
        printf("Enter loop count: ");
        scanf("%d", &count);
        funcA(count);
        return 0;
}
```