Software Lab, Assignment 4

Code Profiling, By Aagam Sancheti

1) What is code Profiling & Why it is important?

Overview:

Code profiling is a process of measuring the performance of software applications. With profiling, you get fine-grained information for the

components of an application, such as how often a function is called, how long a routine takes to execute, how much time is spent of different spots in the code.

Importance:

Using this, one can identify the performance bottlenecks and the poorly implemented parts in a software application, and find effective methods to improve them. Also, certain unnoticed bugs could also be identified.

2) What is the information collected by the Profilers?

There are various kinds of information that is collected by profilers. Few of them are listed below:

- **Call counts:** These are the number of times each function calls another function (or itself in case of recursion). By knowing this, one can verify if the functions are being called as expected, and/or may try to minimize the number of function calls to reduce function call overheads if needed.

- **Time intervals:** These are the time required for the execution of a code segment or a subroutine. There are different kinds of time measurements, such as wall-clock time (actual elapsed time), user-CPU time (time required for the execution of the program on CPU), system-CPU time (time spent in OS calls). By knowing this, one can decide which code segment/subroutine must be optimized (by using task-parallelism, data-parallelism or memory optimizations as suited for the application).
- **Memory allocation:** These are the details of memory allocation in stack or heap. This can be useful to identify if memory management is done correctly or if there is any memory leak in the application. Memory allocation profiling is especially useful for languages with automatic memory management (like Java).
- **Hardware counts:** Details like cache misses and page faults come under this category. These details can be useful to fine-grained tuning.

3) Explain the code profiling methods and their advantages and disadvantages in brief?

Sampling Code Profiling:

- **How it works:** Imagine you're at a busy intersection watching traffic. Instead of tracking every single car, you glance at the intersection every few seconds and count how many cars are passing by. This observation gives you a good idea of the overall traffic flow without needing to monitor every vehicle.

- **Advantages:** Sampling profiling works similarly. Instead of constantly monitoring every aspect of a program, it periodically checks what's happening at specific intervals. This approach lets you get insights into the program's behaviour without slowing it down too much.

- **Disadvantages:** However, just like glancing at traffic, sampling may miss some details. If something important happens between samples, you might not catch it. Also, the accuracy of sampling depends on how often you take samples and whether they represent the program's overall behaviour accurately.

Instrumentation Profiling:

- **How it works:** Imagine you want to find out how many people are using different entrances to a building. Instead of just watching from afar, you decide to stand at each entrance and count every person who enters. This hands-on approach gives you precise data about each entrance's usage.
- **Advantages:** Instrumentation profiling is like being hands-on with your program. Instead of guessing what's happening, you insert special "probes" or checkpoints into your code to track specific actions or events. This gives you detailed insights into how the program behaves under different conditions.

- **Disadvantages:** However, this approach requires more effort. You must modify your code to add these probes, which can be time-consuming. Also, the extra code might affect the program's performance, and you need to be careful not to skew the results by adding too many probes.

4) Explain the basics of 'gprof' and 'perf' profilers and the difference between them.

Gprof (GNU profiler):

gprof is a profiler that comes bundled with the GNU Compiler Collection (GCC). It's primarily used for profiling C and C++ programs. **gprof** works by inserting extra code into your program during compilation, which records the time spent in each function and how functions call each other.
Gprof uses a hybrid of instrumentation and sampling. Instrumentation is used to collect function call information, and sampling is used to gather runtime profiling information.
gprof is easy to use and provides insights into function-level performance but has limitations in its accuracy, especially for multithreaded or complex programs.

Perf profiler:

Perf provides some advanced profiling options, hence a bit complex than gprof. Perf profiler comes included in the Linux kernel & is a sampling-based profiler that works on event-based sampling. It uses CPU performance counters to profile the application. It can instrument hardware counters, static trace points, and dynamic trace points. It also provides per task, per

CPU and per-workload counters, sampling on top of these and source code event annotation. As it does not instrument the application code, it has a really fast speed & generates fairly precise results.

Difference between gprof and perf:

- **Scope:** **gprof** primarily focuses on function-level profiling within individual programs, while **perf** provides system-wide performance monitoring and analysis.
- **Granularity:** **gprof** offers function-level granularity, while **perf** provides detailed insights into various system-level events and performance metrics.
- **Integration:** **gprof** is closely integrated with GCC and primarily used for profiling C/C++ programs, while **perf** is a standalone tool available on Linux systems and can be used for profiling various types of applications and system-level activities.

5) Demonstrate (in short) profiling with 'gprof' for a C program with a couple of function calls (recursive and non-recursive) and loops.

```
#include <stdio.h>

int factorial_recursion(int n);
int factorial_loops(int n);

int main()
{
int n;
printf("Enter n of which factorial has to be calculated:\n");
scanf("%d", &n);
printf("Factorial using recursion is: %d\n", factorial_recursion(n));
factorial_loops(n);
}

int factorial_recursion(int n)
{
if (n == 1 || n == 0)
{
return 1;
}
int factorialNminus1 = factorial_recursion(n - 1);
int factorial = factorialNminus1 * n;
return factorial;
}

int factorial_loops(int n)
{
int factorial = 1;
for (int i = n; i >= 1; i--)
{
factorial = factorial * i;
}
printf("Factorial using loops is: %d\n", factorial);

return 0;
}
```

- Terminal commands to Execute and generate the gprof profiling report:

`gcc -o program program.c -pg` (will add extra overhead to the existing code to accumulate the required profiling data)

`./program` (this command will run the code as it is, with the extra code overhead added)

`gprof program gmon.out > profile_report.txt` (this command will generate a proper report of detailed information about the time spent in each function, including both recursive and non-recursive function calls, as well as loops.)