Profilers

Three important considerations while developing a software application are correctness, maintainability, and fast execution speed. To make a software application run faster, it is important to measure the performance of the application and understand what part of the code can be optimized to make the application run faster. Profilers are used to measure 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, etc. With this information, one can identify the performance bottlenecks and the poorly implemented parts in a software application, and find effective methods to improve them. Sometimes, profiling can also be useful to identify bugs that had otherwise been unnoticed.

There are various kinds of information that can be 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 has to 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.

However, *it is important to understand how this information is collected and hence how precise it is*. Mainly, there are two ways to obtain profiling information: code instrumentation and statistical sampling. The below paragraphs give a brief introduction to each of these methods along with their advantages/disadvantages and inaccuracies in these methods.

1. Instrumentation:

The instrumentation method inserts special code at the beginning and end of each routine to record when the routine starts and ends. The time spent on calling other routines within a routine may also be recorded. The profiling result shows the actual time taken by the routine on each call. Instrumentation based profilers can either insert the instrumenting code in the source code (source code modifying profilers) or into an application's executable code once it is loaded in memory (binary profilers).

Instrumentation profiling is actually disruptive to the program execution, but it allows the profiler to record all the events it is interested in. So the good thing about the instrumentation method is it gives you the actual execution times and precise invocation order (call graph). The inserted instrumentation code (timer calls) takes some time themselves. To reduce the impact of that, at the start of each run, profilers measure the overhead incurred from the instrumenting process and later subtract this overhead from the measurement result. But the instrumenting process could still significantly affect an application's performance in some cases, for example when the routine is very short and frequently called, as the inserted instrumentation would disturb the way the routine executes on the CPU.

As a matter of fact, the timer code that programers often include in code to time specific code segments is the simplest type of instrumentation profiling.

2. Sampling:

Sampling measures applications without inserting any modifications. Sampling profilers interrupts the CPU at regular intervals and collects the function call stack. Exclusive sample counts are incremented for the function that is executing and inclusive counts are incremented for all of the calling functions on the call stack. Sampling reports present the totals of these counts for the profiled module, function, source code line, and instruction.

Sampling-based profiling is less disruptive to program execution, but it cannot provide completely accurate information. The accuracy of the sampling-based profilers depends on the sampling interval and the nature of the program. Sampling profilers causes only a little overhead to the application run process, and they work well on small and often-called routines. The overheads in sampling-based profiling are often stable and depend on the sampling interval. One drawback is the evaluations of time spent are statistical approximations rather than actual time. Also, sampling could only tell what routine is executing currently, not where it was called from. As a result, sampling profilers can't accurately report call traces of an application if the sampling interval is not sufficiently small.

There is a huge list of various profiling tools for various platforms, some of which are dedicated to specific profiling tasks where others provide a vast number of profiling options. Also, many IDEs are integrated with their own profilers (e.g. visual studio, android studio, etc.). **Some good profiling tools** are gprof, perf, Valgrind, gperftools, etc. From these profilers, gprof and perf are basic profilers and Valgrind and gperftools are advanced profilers with various options (e.g. memory profiling). For starters, let us focus on gprof and perf tools, and students can explore other advanced profiling tools as and when needed. Like many other profilers, these two profilers (gprof and perf) can be used on Ubuntu terminal or via eclipse IDE.

1. Gprof (GNU profiler):

Gprof profiler is very easy to use which makes it suitable for beginners. 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. Being developed under the GNU project, it works very well with the GCC compiler. Details about gprof can be studied from the official documentation links provided below.

Documentation: https://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html_mono/gprof.html#SEC19

Eclipse user guide: https://wiki.eclipse.org/Linux_Tools_Project/GProf/User_Guide

2. Perf:

Perf provides some advanced profiling options, hence a bit complex than gprof. Perf profiler comes included in the Linux kernel. Perf 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 tracepoints, and dynamic tracepoints. 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. It also generates fairly precise results. Details about perf can be studied from the official documentation links provided below. Some command-specific details about perf can also be found in man pages of commands like – perf, perf-stat, perf-top, perf-record, perf-report, perf-list, etc.

Introduction: https://perf.wiki.kernel.org/index.php/Main_Page

Eclipse user guide: https://wiki.eclipse.org/Linux_Tools_Project/PERF/User_Guide

Official tutorial page: https://perf.wiki.kernel.org/index.php/Tutorial