Executable commands reference

Table of Contents

How simpleperf works

Modern CPUs have a hardware component called the performance monitoring unit (PMU). The PMU has several hardware counters, counting events like how many cpu cycles have happened, how many instructions have executed, or how many cache misses have happened.

The Linux kernel wraps these hardware counters into hardware perf events. In addition, the Linux kernel also provides hardware independent software events and tracepoint events. The Linux kernel exposes all events to userspace via the perf_event_open system call, which is used by simpleperf.

Simpleperf has three main commands: stat, record and report.

The stat command gives a summary of how many events have happened in the profiled processes in a time period. Here's how it works:

1. Given user options, simpleperf enables profiling by making a system call to the kernel.
2. The kernel enables counters while the profiled processes are running.
3. After profiling, simpleperf reads counters from the kernel, and reports a counter summary.

The record command records samples of the profiled processes in a time period. Here's how it works:

1. Given user options, simpleperf enables profiling by making a system call to the kernel.
2. Simpleperf creates mapped buffers between simpleperf and the kernel.
3. The kernel enables counters while the profiled processes are running.
4. Each time a given number of events happen, the kernel dumps a sample to the mapped buffers.
5. Simpleperf reads samples from the mapped buffers and stores profiling data in a file called perf.data.

The report command reads perf.data and any shared libraries used by the profiled processes, and outputs a report showing where the time was spent.

Commands

Simpleperf supports several commands, listed below:

```
The debug-unwind command: debug/test dwarf based offline unwinding, used for debugging
The dump command: dumps content in perf.data, used for debugging simpleperf.
The help command: prints help information for other commands.
The kmem command: collects kernel memory allocation information (will be replaced by P
The list command: lists all event types supported on the Android device.
The record command: profiles processes and stores profiling data in perf.data.
The report command: reports profiling data in perf.data.
The report-sample command: reports each sample in perf.data, used for supporting integ
                           simpleperf in Android Studio.
The stat command: profiles processes and prints counter summary.
```

Each command supports different options, which can be seen through help message.

```
# List all commands.
$ simpleperf --help

# Print help message for record command.
$ simpleperf record --help
```

Below describes the most frequently used commands, which are list, stat, record and report.

The list command

The list command lists all events available on the device. Different devices may support different events because they have different hardware and kernels.

```
$ simpleperf list
List of hw-cache events:
  branch-loads
  ...
List of hardware events:
  cpu-cycles
  instructions
  ...
List of software events:
  cpu-clock
  task-clock
  ...
```

On ARM/ARM64, the list command also shows a list of raw events, they are the events supported by the ARM PMU on the device. The kernel has wrapped part of them into hardware events and hw-cache events. For example, raw-cpu-cycles is wrapped into cpu-cycles, raw-instruction-retired is wrapped into instructions. The raw events are provided in case we want to use some events supported on the device, but unfortunately not wrapped by the kernel.

The stat command

The stat command is used to get event counter values of the profiled processes. By passing options, we can select which events to use, which processes/threads to monitor, how long to monitor and the print interval.

```
# Stat using default events (cpu-cycles,instructions,...), and monitor process 7394 fo
$ simpleperf stat -p 7394 --duration 10
Performance counter statistics:

 1,320,496,145   cpu-cycles         # 0.131736 GHz                     (100%)
   510,426,028   instructions       # 2.587047 cycles per instruction  (100%)
     4,692,338   branch-misses      # 468.118 K/sec                    (100%)
886.008130(ms)   task-clock         # 0.088390 cpus used               (100%)
           753   context-switches   # 75.121 /sec                      (100%)
           870   page-faults        # 86.793 /sec                      (100%)

Total test time: 10.023829 seconds.
```

Select events to stat

We can select which events to use via -e.

```
# Stat event cpu-cycles.
$ simpleperf stat -e cpu-cycles -p 11904 --duration 10

# Stat event cache-references and cache-misses.
$ simpleperf stat -e cache-references,cache-misses -p 11904 --duration 10
```

When running the stat command, if the number of hardware events is larger than the number of hardware counters available in the PMU, the kernel shares hardware counters between events, so each event is only monitored for part of the total time. In the example below, there is a percentage at the end of each row, showing the percentage of the total time that each event was actually monitored.

```
# Stat using event cache-references, cache-references:u,....
$ simpleperf stat -p 7394 -e cache-references,cache-references:u,cache-references:k \
      -e cache-misses,cache-misses:u,cache-misses:k,instructions --duration 1
Performance counter statistics:

4,331,018  cache-references     # 4.861 M/sec    (87%)
3,064,089  cache-references:u   # 3.439 M/sec    (87%)
1,364,959  cache-references:k   # 1.532 M/sec    (87%)
   91,721  cache-misses         # 102.918 K/sec  (87%)
   45,735  cache-misses:u       # 51.327 K/sec   (87%)
   38,447  cache-misses:k       # 43.131 K/sec   (87%)
9,688,515  instructions         # 10.561 M/sec   (89%)

Total test time: 1.026802 seconds.
```

In the example above, each event is monitored about 87% of the total time. But there is no guarantee that any pair of events are always monitored at the same time. If we want to have some events monitored at the same time, we can use --group.

```
# Stat using event cache-references, cache-references:u,....
$ simpleperf stat -p 7964 --group cache-references,cache-misses \
      --group cache-references:u,cache-misses:u --group cache-references:k,cache-misse
      -e instructions --duration 1
Performance counter statistics:

3,638,900  cache-references     # 4.786 M/sec          (74%)
   65,171  cache-misses         # 1.790953% miss rate  (74%)
2,390,433  cache-references:u   # 3.153 M/sec          (74%)
   32,280  cache-misses:u       # 1.350383% miss rate  (74%)
  879,035  cache-references:k   # 1.251 M/sec          (68%)
   30,303  cache-misses:k       # 3.447303% miss rate  (68%)
8,921,161  instructions         # 10.070 M/sec         (86%)

Total test time: 1.029843 seconds.
```

Select target to stat

We can select which processes or threads to monitor via -p or -t. Monitoring a process is the same as monitoring all threads in the process. Simpleperf can also fork a child process to run the new command and then monitor the child process.

```
# Stat process 11904 and 11905.
$ simpleperf stat -p 11904,11905 --duration 10

# Stat thread 11904 and 11905.
$ simpleperf stat -t 11904,11905 --duration 10

# Start a child process running `ls`, and stat it.
$ simpleperf stat ls

# Stat the process of an Android application. This only works for debuggable apps on r
# devices.
$ simpleperf stat --app com.example.simpleperf.simpleperfexamplewithnative

# Stat system wide using -a.
$ simpleperf stat -a --duration 10
```

Decide how long to stat

When monitoring existing threads, we can use --duration to decide how long to monitor. When monitoring a child process running a new command, simpleperf monitors until the child process ends. In this case, we can use Ctrl-C to stop monitoring at any time.

```
# Stat process 11904 for 10 seconds.
$ simpleperf stat -p 11904 --duration 10

# Stat until the child process running `ls` finishes.
$ simpleperf stat ls

# Stop monitoring using Ctrl-C.
$ simpleperf stat -p 11904 --duration 10
^C
```

If you want to write a script to control how long to monitor, you can send one of SIGINT, SIGTERM, SIGHUP signals to simpleperf to stop monitoring.

Decide the print interval

When monitoring perf counters, we can also use --interval to decide the print interval.

```
# Print stat for process 11904 every 300ms.
$ simpleperf stat -p 11904 --duration 10 --interval 300

# Print system wide stat at interval of 300ms for 10 seconds. Note that system wide pr
# root privilege.
$ su 0 simpleperf stat -a --duration 10 --interval 300
```

Display counters in systrace

Simpleperf can also work with systrace to dump counters in the collected trace. Below is an example to do a system wide stat.

```
# Capture instructions (kernel only) and cache misses with interval of 300 millisecond
# seconds.
$ su 0 simpleperf stat -e instructions:k,cache-misses -a --interval 300 --duration 15
# On host launch systrace to collect trace for 10 seconds.
(HOST)$ external/chromium-trace/systrace.py --time=10 -o new.html sched gfx view
# Open the collected new.html in browser and perf counters will be shown up.
```

The record command

The record command is used to dump samples of the profiled processes. Each sample can contain information like the time at which the sample was generated, the number of events since last sample, the program counter of a thread, the call chain of a thread.

By passing options, we can select which events to use, which processes/threads to monitor, what frequency to dump samples, how long to monitor, and where to store samples.

```
# Record on process 7394 for 10 seconds, using default event (cpu-cycles), using defau
# frequency (4000 samples per second), writing records to perf.data.
$ simpleperf record -p 7394 --duration 10
simpleperf I cmd_record.cpp:316] Samples recorded: 21430. Samples lost: 0.
```

Select events to record

By default, the cpu-cycles event is used to evaluate consumed cpu cycles. But we can also use other events via -e.

```
# Record using event instructions.
$ simpleperf record -e instructions -p 11904 --duration 10

# Record using task-clock, which shows the passed CPU time in nanoseconds.
$ simpleperf record -e task-clock -p 11904 --duration 10
```

Select target to record

The way to select target in record command is similar to that in the stat command.

```
# Record process 11904 and 11905.
$ simpleperf record -p 11904,11905 --duration 10

# Record thread 11904 and 11905.
$ simpleperf record -t 11904,11905 --duration 10

# Record a child process running `ls`.
$ simpleperf record ls

# Record the process of an Android application. This only works for debuggable apps or
# devices.
$ simpleperf record --app com.example.simpleperf.simpleperfexamplewithnative

# Record system wide.
$ simpleperf record -a --duration 10
```

Set the frequency to record

We can set the frequency to dump records via -f or -c. For example, -f 4000 means dumping approximately 4000 records every second when the monitored thread runs. If a monitored thread runs 0.2s in one second (it can be preempted or blocked in other times), simpleperf dumps about 4000 * 0.2 / 1.0 = 800 records every second. Another way is using -c. For example, -c 10000 means dumping one record whenever 10000 events happen.

```
# Record with sample frequency 1000: sample 1000 times every second running.
$ simpleperf record -f 1000 -p 11904,11905 --duration 10

# Record with sample period 100000: sample 1 time every 100000 events.
$ simpleperf record -c 100000 -t 11904,11905 --duration 10
```

To avoid taking too much time generating samples, kernel >= 3.10 sets the max percent of cpu time used for generating samples (default is 25%), and decreases the max allowed sample frequency when hitting that limit. Simpleperf uses --cpu-percent option to adjust it, but it needs either root privilege or to be on Android >= Q.

```
# Record with sample frequency 10000, with max allowed cpu percent to be 50%.
$ simpleperf record -f 1000 -p 11904,11905 --duration 10 --cpu-percent 50
```

Decide how long to record

The way to decide how long to monitor in record command is similar to that in the stat command.

```
# Record process 11904 for 10 seconds.
$ simpleperf record -p 11904 --duration 10

# Record until the child process running `ls` finishes.
$ simpleperf record ls

# Stop monitoring using Ctrl-C.
$ simpleperf record -p 11904 --duration 10
^C
```

If you want to write a script to control how long to monitor, you can send one of SIGINT, SIGTERM, SIGHUP signals to simpleperf to stop monitoring.

Set the path to store profiling data

By default, simpleperf stores profiling data in perf.data in the current directory. But the path can be changed using -o.

```
# Write records to data/perf2.data.
$ simpleperf record -p 11904 -o data/perf2.data --duration 10
```

Record call graphs

A call graph is a tree showing function call relations. Below is an example.

```
main() {
    FunctionOne();
    FunctionTwo();
}
FunctionOne() {
    FunctionTwo();
    FunctionThree();
}
a call graph:
    main-> FunctionOne
       |     |
       |     |-> FunctionTwo
       |     |-> FunctionThree
       |
       |-> FunctionTwo
```

A call graph shows how a function calls other functions, and a reversed call graph shows how a function is called by other functions. To show a call graph, we need to first record it, then report it.

There are two ways to record a call graph, one is recording a dwarf based call graph, the other is recording a stack frame based call graph. Recording dwarf based call graphs needs support of debug information in native binaries. While recording stack frame based call graphs needs support of stack frame registers.

```
# Record a dwarf based call graph
$ simpleperf record -p 11904 -g --duration 10

# Record a stack frame based call graph
$ simpleperf record -p 11904 --call-graph fp --duration 10
```

Here are some suggestions about recording call graphs.

Record both on CPU time and off CPU time

Simpleperf is a CPU profiler, it generates samples for a thread only when it is running on a CPU. However, sometimes we want to figure out where the time of a thread is spent, whether it is running on a CPU, or staying in the kernel's ready queue, or waiting for something like I/O events.

To support this, the record command uses --trace-offcpu to trace both on CPU time and off CPU time. When --trace-offcpu is used, simpleperf generates a sample when a running thread is scheduled out, so we know the callstack of a thread when it is scheduled out. And when reporting a perf.data generated with --trace-offcpu, we use time to the next sample (instead of event counts from the previous sample) as the weight of the current sample. As a result, we can get a call graph based on timestamps, including both on CPU time and off CPU time.

trace-offcpu is implemented using sched:sched_switch tracepoint event, which may not be supported on old kernels. But it is guaranteed to be supported on devices >= Android O MR1. We can check whether trace-offcpu is supported as below.

```
$ simpleperf list --show-features
dwarf-based-call-graph
trace-offcpu
```

If trace-offcpu is supported, it will be shown in the feature list. Then we can try it.

```
# Record with --trace-offcpu.
$ simpleperf record -g -p 11904 --duration 10 --trace-offcpu

# Record with --trace-offcpu using app_profiler.py.
$ python app_profiler.py -p com.example.simpleperf.simpleperfexamplewithnative -a .Sle
    -r "-g -e task-clock:u -f 1000 --duration 10 --trace-offcpu"
```

Below is an example comparing the profiling result with / without --trace-offcpu. First we record without --trace-offcpu.

```
$ python app_profiler.py -p com.example.simpleperf.simpleperfexamplewithnative -a .Sle
$ python report_html.py --add_disassembly --add_source_code --source_dirs ../demo
```

The result is here. In the result, all time is taken by RunFunction(), and sleep time is ignored. But if we add --trace-offcpu, the result changes.

```
$ python app_profiler.py -p com.example.simpleperf.simpleperfexamplewithnative -a .Sle
    -r "-g -e task-clock:u --trace-offcpu -f 1000 --duration 10"

$ python report_html.py --add_disassembly --add_source_code --source_dirs ../demo
```

The result is here. In the result, half of the time is taken by RunFunction(), and the other half is taken by SleepFunction(). So it traces both on CPU time and off CPU time.

The report command

The report command is used to report profiling data generated by the record command. The report contains a table of sample entries. Each sample entry is a row in the report. The report command groups samples belong to the same process, thread, library, function in the same sample entry. Then sort the sample entries based on the event count a sample entry has.

By passing options, we can decide how to filter out uninteresting samples, how to group samples into sample entries, and where to find profiling data and binaries.

Below is an example. Records are grouped into 4 sample entries, each entry is a row. There are several columns, each column shows piece of information belonging to a sample entry. The first column is Overhead, which shows the percentage of events inside the current sample entry in total events. As the perf event is cpu-cycles, the overhead is the percentage of CPU cycles used in each function.

```
# Reports perf.data, using only records sampled in libsudo-game-jni.so, grouping recor
# thread name(comm), process id(pid), thread id(tid), function name(symbol), and showi
# count for each row.
$ simpleperf report --dsos /data/app/com.example.sudogame-2/lib/arm64/libsudo-game-jni
      --sort comm,pid,tid,symbol -n
Cmdline: /data/data/com.example.sudogame/simpleperf record -p 7394 --duration 10
Arch: arm64
Event: cpu-cycles (type 0, config 0)
Samples: 28235
Event count: 546356211

Overhead  Sample  Command   Pid   Tid   Symbol
59.25%    16680   sudogame  7394  7394  checkValid(Board const&, int, int)
20.42%    5620    sudogame  7394  7394  canFindSolution_r(Board&, int, int)
13.82%    4088    sudogame  7394  7394  randomBlock_r(Board&, int, int, int, int, int)
6.24%     1756    sudogame  7394  7394  @plt
```

Set the path to read profiling data

By default, the report command reads profiling data from perf.data in the current directory. But the path can be changed using -i.

```
$ simpleperf report -i data/perf2.data
```

Set the path to find binaries

To report function symbols, simpleperf needs to read executable binaries used by the monitored processes to get symbol table and debug information. By default, the paths are the executable binaries used by monitored processes while recording. However, these binaries may not exist when reporting or not contain symbol table and debug information. So we can use --symfs to redirect the paths.

```
# In this case, when simpleperf wants to read executable binary /A/b, it reads file ir
$ simpleperf report

# In this case, when simpleperf wants to read executable binary /A/b, it prefers file
# /debug_dir/A/b to file in /A/b.
$ simpleperf report --symfs /debug_dir

# Read symbols for system libraries built locally. Note that this is not needed since
# which ships symbols for system libraries on device.
$ simpleperf report --symfs $ANDROID_PRODUCT_OUT/symbols
```

Filter samples

When reporting, it happens that not all records are of interest. The report command supports four filters to select samples of interest.

```
# Report records in threads having name sudogame.
$ simpleperf report --comms sudogame

# Report records in process 7394 or 7395
$ simpleperf report --pids 7394,7395

# Report records in thread 7394 or 7395.
$ simpleperf report --tids 7394,7395

# Report records in libsudo-game-jni.so.
$ simpleperf report --dsos /data/app/com.example.sudogame-2/lib/arm64/libsudo-game-jni
```

Group samples into sample entries

The report command uses --sort to decide how to group sample entries.

```
# Group records based on their process id: records having the same process id are in t
# sample entry.
$ simpleperf report --sort pid

# Group records based on their thread id and thread comm: records having the same thre
# thread name are in the same sample entry.
$ simpleperf report --sort tid,comm

# Group records based on their binary and function: records in the same binary and fun
# the same sample entry.
$ simpleperf report --sort dso,symbol

# Default option: --sort comm,pid,tid,dso,symbol. Group records in the same thread, an
# the same function in the same binary.
$ simpleperf report
```

Report call graphs

To report a call graph, please make sure the profiling data is recorded with call graphs, as here.

```
$ simpleperf report -g
```