How to use Perf in UNIX (Linux)

Mateusz Ferenc

2023
May

Introduction

Perf is a tool that uses the hardware PMU (Performance Measurement Unit) to count system statistics.
Using Perf, the user is able to obtain performance counters, tracepoints, kprobes and uprobes.

Performance counters are CPU hardware registers that count hardware events such as instructions executed, cache misses, or branch failures.
They provide a basis for profiling applications to trace dynamic control flow and identify hotspots.
Perf provides rich generalized abstractions over hardware-specific capabilities.
Among other things, it provides per-task, per-CPU, and per-workload counters, sampling on top of these, and source code event annotation.

A few preliminary clarifications:
Almost all of the commands below require administrative privileges [i.e., superuser, sudo].
Text enclosed in '[]' means it is optional, and the '|' character means the first or second option is possible.

1 How to setup

First of all Perf needs to be installed. (also you can check if it not already installed using: **which perf** or by simply typing **perf** in your command line to get perf help)

You can install Perf using your package menager.
Package name is ***linux-tools-*your kernel version****.

On Debian based systems:
apt install linux-tools-‘uname -r‘

When Perf is finally installed you should enable system profiling in kernel configuration.
To do this you need to edit in **/etc/sysctl.conf** file, value after **kernel.perf_event_paranoid** to **-1** .

Now you are able to use Perf properly.

2 Preparing the program for testing

You can measure system statistics using pre-existing trace points, but using custom trace points gives you more flexibility with Perf, resulting in better results.

To be able to use tracepoints you need to prepare you program (written in c/c++).
The preparation consists of compiling program using gcc with debug flag (like below).
gcc -g -o *output-file-name file-name*
"-g" flag used to produce debugging information in the operating system's native format

3 Creating and using tracepoints

Now that we have compiled program, we can create tracepoints according to functions or specified location in the code.

With the help of the following command, you can get functions that can be probed.
perf probe –funcs –exec *executable-name*

Also it is possible to list probe-able code lines of any function <function-name>
perf probe –line *function-name* –exec *executable-name*

With the information from the above commands, you can create your own trace points. To do this, you can use the following commands.

To create a trace point at the entry of the specified function (in this case probe-definition), use:
perf probe –exec *executable-name probe-definition*
Format of probing 'probe-definition' can be different, what affects where probe is placed:

[EVENT=]FUNC[%return | :RL]

- **EVENT=** - Specify event name, optional. If specified, overrides the Perf naming convention.
- **FUNC** - function name to probe, moderatory.
- **%return** - place probe at exit of the function, optional.
- **:RL** - Relative line number from function input, obtained by using the –line switch with the above command, optional.

If '**EVENT=**' is not specified, Perf will create tracepoint (name) by using following convention:

probe_*executable-name*:*function-name*[___return | :*line-number*]

otherwise:

probe_*executable-name*:*event-name*

i.e.: probe_test:TestFunction___return, probe_test:TestEvent, probe_test:TestFunction:15

To test if newly created tracepoints, you can simply run Perf stat like below:
perf stat –event *your-event-name*
Output of Perf stat command should contain number how many times your event/events occured.

4 Recording and obtaining results

When you have finally prepared your tracepoints, it is time to get the results.

My example will show measurements using two self-defined tracepoints.
Because I need to show the time difference between tracepoints pointing to the entrance and exit of the function being measured, i.e. I was measuring how much time it takes to execute code in my function.

To do this, you must have at least two tracepoints and use the following command:
perf record –event *your-event-entrance* –event *your-event-exit* *./your-executable*
i.e.: perf record -e probe_test:TestFunction -e probe_test:TestFunction___return ./test_executable

Now using Perf script, you can get data from Perf record:
perf script –ns –deltatime
The '–ns' flag is used to display time in nanosecond resolution, the '–deltatime' flag is used to calculate the difference (in time) between the occurrence of events.

Perf script have ability to get only needed columns, using '*-F, –fields*' flag,
using fields '*time,event*', Perf script output should look like below:

```
0.000000000:         probe_test_delay:funcB:
0.100086322: probe_test_delay:funcB__return:
0.000004126:         probe_test_delay:funcB:
0.100069059: probe_test_delay:funcB__return:
... and so on...
```

Of course, the names (and values) will be different.

5 How to interpret the output

Analyze data generated by Perf script.

6 More about the tracepoints

It is possible to create events group (leader event, i.e. event that occur causes reading other connected events)
Best use is when leader even is tracepoint (placed in code).

Adding leader event:
perf record –event *your-event-entrance* –event "{*your-event-exit*,ref-cycles:u,cpu-cycles:u}" ./*your-executable*

redirect to file:
perf script –ns –deltatime > *file-name*

Generated file should look like below:

```
simple_loop  9298 [000]     0.000000000:         probe_simple_loop:funcB: (56309a67d149)
simple_loop  9298 [000]     1.087778453: probe_simple_loop:funcB__return: (56309a67d149
<- 56309a67d1b7)
simple_loop  9298 [000]     0.000000000: 1148077744                    ref-cycles:u:
56309a67d1b7 main+0x38 (executable-path)
simple_loop  9298 [000]     0.000000000: 1030436445                    cpu-cycles:u:
56309a67d1b7 main+0x38 (executable-path)
simple_loop  9298 [000]     0.000007534:         probe_simple_loop:funcB: (56309a67d149)
simple_loop  9298 [000]     1.095117039: probe_simple_loop:funcB__return: (56309a67d149
<- 56309a67d1b7)
simple_loop  9298 [000]     0.000000000: 1146679864                    ref-cycles:u:
56309a67d1b7 main+0x38 (executable-path)
simple_loop  9298 [000]     0.000000000: 1029181631                    cpu-cycles:u:
56309a67d1b7 main+0x38 (executable-path)
... and so on...
```

7 Adjusting hardware te get better results

When it comes to the results, it is interesting to note that the actual measured cycles depend on the CPU statistics.
First of all, most important thing to do is to restrict CPU context-switching, i.e. force process to run on single core, without switching to other cores.
Importance of this move is that when process is moved between cores (context switch occurs), actual data from PMU is lost due to the fact that each core has its own performance registers and counted values differ between cores.
I.e. if cycles are counted on first core and then process is switched to second core, the value of measured cycles on first core and second core is not added. So finally the amount of measured cycles will be much different than it should be.

To restrict context-switching the Kernel should be informed to run the process on specified core only.
This is achieved by running command/program using '**taskset**'
taskset ./run_all *switches*

To further improve measurements quality, you can change CPU governor to performance:
cpupower frequency-set –governor performance or **cpufreq-set –governor performance**
Also changing CPU behavior can make further improvements:
Disable SpeedStep technology (disable underclocking option of the cpu)
Disable HyperThreading technology (disable virtual cores)
Disable TurboBoost technology (disable higher cpu frequencies)