EECS 750 Homework #3

Playing with the *perf* tool in Linux

In this homework, you will learn to interact with Linux's perf tool.

You should submit two files: hw3-1.png, hw3-2.png.

Part 0. Preparation

You need to have an access to a Linux computer. In addition, you need a root shell access to the computer to complete the homework. Virtual machines are not ideal due to their limited hardware counter support.

On a terminal, download the IsolBench benchmark suite, which is a collection of synthetic benchmarks to measure memory performance.

```
$ git clone https://github.com/CSL-KU/IsolBench
$ cd IsolBench/bench
$ make
```

Then, configure the kernel as follow to be able to access the hardware performance counters at the user-level.

```
$ sudo bash
# echo 0 > /proc/sys/kernel/perf_event_paranoid
```

The following is also needed to be able to use the kernel symbols

```
# echo 0 > /proc/sys/kernel/kptr_restrict
```

If perf is not already installed in your machine, you need to install it to do this homework, either using the package manager of your system or building from the source code. The following shows the latter case.

```
$ cd <linux-source-code-directory>
$ cd tools/perf/
$ make -j12
```

Part 1. Basic event counting.

In this part of the homework, we will learn to use ‘perf stat’ module, which monitor and report selected performance related event counters while running a program.

Before begin, let’s first see what are the available events counters---both hardware and software counters---on your system.

```
$ perf list
List of pre-defined events (to be used in -e):

  branch-instructions OR branches                    [Hardware event]
  branch-misses                                      [Hardware event]
  bus-cycles                                         [Hardware event]
  cache-misses                                       [Hardware event]
  cache-references                                   [Hardware event]
  cpu-cycles OR cycles                               [Hardware event]
  instructions                                       [Hardware event]
  ref-cycles                                         [Hardware event]

  alignment-faults                                   [Software event]
  bpf-output                                         [Software event]
  context-switches OR cs                             [Software event]
  cpu-clock                                          [Software event]
  cpu-migrations OR migrations                       [Software event]
  dummy                                              [Software event]
  emulation-faults                                   [Software event]
  major-faults                                       [Software event]
      ..
      ..
```

Depending on architecture and specific cpu model, the observable events may differ. On the rightmost column, “[Hardware event]” means that it is counted by a hardware performance counter, whereas “[Software event]” refers to an event that is accounted by the OS.

Next, we will monitor the behavior of the ‘latency’ benchmark using ‘perf stat’ command as follows. Note that the ‘latency’ benchmark is a so called ‘pointer-chasing’ application as it traverses a randomly created linked-list over a large memory space (-m 32768 means 32768 KB, or 32MB memory space is used for the linked list).

```
$ perf stat ./latency -m 32768
average 74.04 ns | bandwidth 864.40 MB (824.35 MiB)/s
```

```
...
 Performance counter stats for './latency -m 32768':

       3917.705675      task-clock (msec)         #    1.000 CPUs utilized
...
    13,901,105,074      cycles                    #    3.548 GHz
       366,597,875      instructions              #    0.03  insn per cycle
        63,919,959      branches                  #   16.316 M/sec
            30,055      branch-misses             #    0.05% of all branches

       3.917884912 seconds time elapsed
```

```
Among the several measured counters, the highlighted bottom four events are
based on hardware counters.
```

Due to inherent data dependency in pointer chasing, the program can measure true memory access latency (as long as your LLC size is less than 32MB; if bigger than that, increase the allocation size with ‘-m’ option.) In the example above, the measure (worst-case) memory access latency is 74ns.

You can choose different events to monitor by using ‘-e’ option. The following example monitors two events, LLC-load-misses and LLC-loads, of the program.

```
$ perf stat -e instructions,LLC-load-misses,LLC-loads ./latency -m 32768
...
 Performance counter stats for './latency -m 32768':

       366,486,929      instructions
        52,098,592      LLC-load-misses
        53,304,589      LLC-loads

       3.896608828 seconds time elapsed
```

What you see above is that almost all last-level cache (LLC) accesses were misses. This is expected because the allocated memory size (32M) of the linked list is bigger than the CPU’s LLC size. You can also compute the LLC MPKI (miss-per-killo-instructions) of the program as follows: MPKI = LLC-load-misses / (instructions / 1000).

Capture the terminal screen showing the statistics above kind and save it as ‘hw3-1.png’. You should return the file as a proof.

Part 2. Detailed analysis via sampled recording.

Now, we will do more detailed performance analysis by profiling the program.

First, profile the latency program as follows.

```
 $ perf record -e cycles:pp -g ./latency -m 32768
allocated: wokingsetsize=524288 entries
initialized.
duration 3906315 us
average 74.51 ns | bandwidth 858.98 MB (819.19 MiB)/s
readsum  13743869132800
[ perf record: Woken up 4 times to write data ]
[ perf record: Captured and wrote 0.977 MB perf.data (15786 samples) ]
```

This use the ‘cycles’ hardware counter to profile the program. The “:pp” option is added here to use Intel’s PEBS (Precise Event Based Sampling). AMD or other architectures may not support this option.

The profiled data is stored in the ‘perf.data’ file. We can then

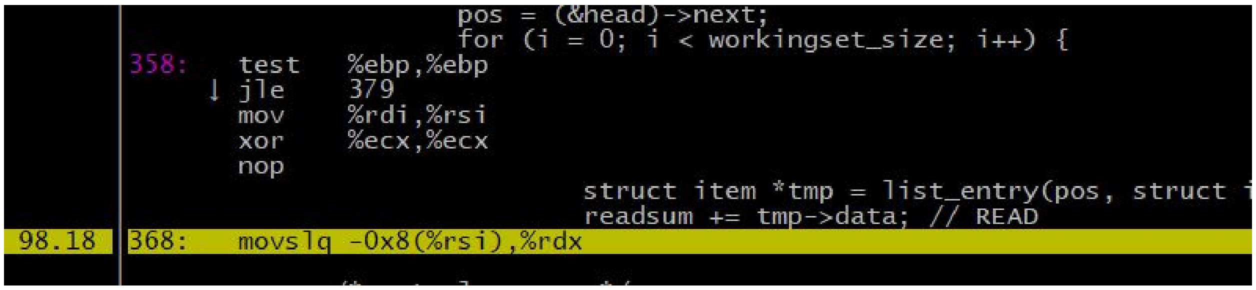
```
$ perf report

Samples: 15K of event 'cycles:pp', Event count (approx.): 13956722034
  Children      Self  Command  Shared Object      Symbol
+   99.93%    99.81%  latency  latency            [.] main
     0.05%     0.00%  latency  [kernel.kallsyms]  [k] handle_mm_fault
     0.05%     0.00%  latency  [kernel.kallsyms]  [k] __do_page_fault
     0.05%     0.00%  latency  [kernel.kallsyms]  [k] do_page_fault
...
```

The result shows the percentage of the time (via ‘cycles’ counter overflows) the program spent on each function (symbol). Note that the symbols marked as ‘[k]’ are kernel symbols whereas those with ‘[.]’ are user-level symbols. In this case, the program spent almost all time within the user-level ‘main’ function.

Next, we will drill-down the profiled data in more detail, using ‘perf annotate’, to see exactly which part of the code the program spent most of the time.

```
$ perf annotate
```



```
                        pos = (&head)->next;
                        for (i = 0; i < workingset_size; i++) {
       358:   test   %ebp,%ebp
            ↓ jle    379
              mov    %rdi,%rsi
              xor    %ecx,%ecx
              nop
                                struct item *tmp = list_entry(pos, struct i
                                readsum += tmp->data; // READ
 98.18 368:   movslq -0x8(%rsi),%rdx
```

The result shows how much time is spent on each instruction (with matching C code; You can toggle C source code view by pressing ‘s’ key.)

Capture the terminal screen showing the statistics above kind and save it as ‘hw3-2.png’. You should return the file as a proof.

Note that this is based on profiling and thus not 100% accurate. In particular, if you don’t use Intel PEBS mentioned earlier, identified instructions may be slightly off by a few instructions from the real ones. For a related discussion, please read the following.
https://stackoverflow.com/questions/29528550/perf-annotated-assembly-seems-off