/***
** Notes on Linux perf tool
**
** Intended audience: Those who would like to learn more about
** Linux perf performance analysis and profiling tool.
**
** Used: CPE 631 Advanced Computer Systems and Architectures
**   CPE 619 Modeling and Analysis of Computer and Communication Systems
**
** ver 0.1, Spring 2012
**
** @Aleksandar Milenkovic, milenkovic@computer.org
***/

Perf Tool: Performance Analysis Tool for Linux

1. Introduction

Perf is a profiler tool for Linux 2.6+ based systems that abstracts away CPU hardware differences in Linux performance measurements and presents a simple command line interface. It covers hardware level (CPU/PMU, Performance Monitoring Unit) features and software features (software counters, tracepoints) as well.

To learn more about perf type in man perf.

```
<<~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
[milenka@eb136i-nsf02] man perf

[milenka@eb136i-nsf02 perf.tool]$ man perf-stat

[milenka@eb136i-nsf02 perf.tool]$ man perf-top

...

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~>>
```

2. Commands

The perf tool offers a rich set of commands to collect and analyze performance and trace data.
The command line usage is reminiscent of git in that there is a generic tool, perf, which implements a set of commands: stat, record, report, [...].

* To see the list of all options, please type in perf.

```
<<~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
[milenka@eb136i-nsf02 perf.tool]$ perf

 usage: perf [--version] [--help] COMMAND [ARGS]
```

```
 The most commonly used perf commands are:
   annotate        Read perf.data (created by perf record) and display annotated code
   archive         Create archive with object files with build-ids found in perf.data
file
   bench           General framework for benchmark suites
   buildid-cache   Manage build-id cache.
   buildid-list    List the buildids in a perf.data file
   diff            Read two perf.data files and display the differential profile
   kmem            Tool to trace/measure kernel memory(slab) properties
   list            List all symbolic event types
   lock            Analyze lock events
   probe           Define new dynamic tracepoints
   record          Run a command and record its profile into perf.data
   report          Read perf.data (created by perf record) and display the profile
   sched           Tool to trace/measure scheduler properties (latencies)
   stat            Run a command and gather performance counter statistics
   timechart       Tool to visualize total system behavior during a workload
   top             System profiling tool.
   trace           Read perf.data (created by perf record) and display trace output

 See 'perf help COMMAND' for more information on a specific command.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~>>
```

* Certain commands require special support in the kernel and may not be available. To obtain the list of options for each command, simply type the command name followed by -h, e.g.:

```
<<~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
[milenka@eb136i-nsf02 perf.tool]$ perf stat -h

 usage: perf stat [<options>] [<command>]

    -e, --event <event>   event selector. use 'perf list' to list available events
    -i, --inherit         child tasks inherit counters
    -p, --pid <n>         stat events on existing pid
    -a, --all-cpus        system-wide collection from all CPUs
    -c, --scale           scale/normalize counters
    -v, --verbose         be more verbose (show counter open errors, etc)
    -r, --repeat <n>      repeat command and print average + stddev (max: 100)
    -n, --null            null run - dont start any counters
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~>>
```

3. Events

The perf tool supports a list of measurable events. The tool and underlying kernel interface can measure events coming from different sources. For instance, some events are pure kernel counters; in this case they are called software events. Examples include: context-switches, minor-fault.
Another source of events is the processor itself and its Performance Monitoring Unit (PMU). It provides a list of events to measure micro-architectural events such as the number of cycles, instructions retired, L1 cache misses and so on. Those events are called PMU hardware events or hardware events for short. They vary with each processor type and model. The perf_events interface also provides a small set of common hardware events monikers.
On each processor, those events get mapped onto actual events provided by the CPU, if they exist, otherwise the event cannot be used. Somewhat confusingly, these are also called hardware events and hardware cache events.

Finally, there are also tracepoint events which are implemented by the kernel ftrace infrastructure. Those are only available with the 2.6.3x and newer kernels.

* To obtain a list of supported events type in perf list.

```
<<~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
[milenka@eb136i-nsf02 perf.tool]$ perf list

List of pre-defined events (to be used in -e):

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

  cpu-clock                                  [Software event]
  task-clock                                 [Software event]
  page-faults OR faults                      [Software event]
  minor-faults                               [Software event]
  major-faults                               [Software event]
  context-switches OR cs                     [Software event]
  cpu-migrations OR migrations               [Software event]
  alignment-faults                           [Software event]
  emulation-faults                           [Software event]

  L1-dcache-loads                            [Hardware cache event]
  L1-dcache-load-misses                      [Hardware cache event]
  L1-dcache-stores                           [Hardware cache event]
  L1-dcache-store-misses                     [Hardware cache event]
  L1-dcache-prefetches                       [Hardware cache event]
  L1-dcache-prefetch-misses                  [Hardware cache event]
  L1-icache-loads                            [Hardware cache event]
  L1-icache-load-misses                      [Hardware cache event]
  L1-icache-prefetches                       [Hardware cache event]
  L1-icache-prefetch-misses                  [Hardware cache event]
  LLC-loads                                  [Hardware cache event]
  LLC-load-misses                            [Hardware cache event]
  LLC-stores                                 [Hardware cache event]
  LLC-store-misses                           [Hardware cache event]
  LLC-prefetches                             [Hardware cache event]
  LLC-prefetch-misses                        [Hardware cache event]
  dTLB-loads                                 [Hardware cache event]
  dTLB-load-misses                           [Hardware cache event]
  dTLB-stores                                [Hardware cache event]
  dTLB-store-misses                          [Hardware cache event]
  dTLB-prefetches                            [Hardware cache event]
  dTLB-prefetch-misses                       [Hardware cache event]
  iTLB-loads                                 [Hardware cache event]
  iTLB-load-misses                           [Hardware cache event]
  branch-loads                               [Hardware cache event]
  branch-load-misses                         [Hardware cache event]

  rNNN                                       [Raw hardware event descriptor]

  mem:<addr>[:access]                        [Hardware breakpoint]
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~>>
```

4. Counting with perf stat

For any of the supported events, perf can keep a running count during process execution. In counting modes, the occurrences of events are simply aggregated and presented on standard output at the end of an application run.
To generate these statistics, use the stat command of perf. For instance:

* Perform perf stat on a program arrsum from time measurement tutorial.

```
<<~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
[milenka@eb136i-nsf02 perf.tool]$ perf stat arrsum.exe 16384
array sum is 17488290749289.000000

 Performance counter stats for 'arrsum.exe 16384':

       1.165253  task-clock-msecs         #      0.745 CPUs
              0  context-switches         #      0.000 M/sec
              0  CPU-migrations           #      0.000 M/sec
            145  page-faults              #      0.124 M/sec
        1847059  cycles                   #   1585.114 M/sec
        2160526  instructions             #      1.170 IPC
         505524  branches                 #    433.832 M/sec
          15382  branch-misses            #      3.043 %
          11489  cache-references         #      9.860 M/sec
           2405  cache-misses             #      2.064 M/sec

    0.001563730  seconds time elapsed
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~>>
```

* With no events specified, perf stat collects the common events listed above.
Some are software events, such as context-switches, others are generic hardware events such as cycles.
After the hash sign, derived metrics may be presented, such as 'IPC' (instructions per cycle).

* We can specify specific events to monitor for both user and kernel level code (uk):

```
<<~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
[milenka@eb136i-nsf02 perf.tool]$ perf stat -e cycles:uk arrsum.exe 16384
array sum is 17488290749289.000000

 Performance counter stats for 'arrsum.exe 16384':

        1803076  cycles

    0.001603354  seconds time elapsed
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~>>
```

```
<<~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
[milenka@eb136i-nsf02 perf.tool]$ perf stat -e cycles:u arrsum.exe 16384
array sum is 17488290749289.000000

 Performance counter stats for 'arrsum.exe 16384':

         878116  cycles

    0.001515271  seconds time elapsed
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~>>
```

```
<<~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
[milenka@eb136i-nsf02 perf.tool]$ perf stat -e cycles:k arrsum.exe 16384
array sum is 17488290749289.000000

 Performance counter stats for 'arrsum.exe 16384':

         925197  cycles

    0.001493215  seconds time elapsed
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~>>
```

* It is possible to use perf stat to run the same test workload multiple times
and get for each count, the standard deviation from the mean.

```
<<~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
[milenka@eb136i-nsf02 perf.tool]$ perf stat -r 5 -e cycles arrsum.exe 16384
array sum is 17488290749289.000000
array sum is 17488290749289.000000
array sum is 17488290749289.000000
array sum is 17488290749289.000000
array sum is 17488290749289.000000

 Performance counter stats for 'arrsum.exe 16384' (5 runs):

        1775046  cycles                    ( +-   0.587% )

    0.001543764  seconds time elapsed   ( +-   1.420% )
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~>>
```

5. Sampling with perf record

The perf tool can be used to collect profiles on per-thread, per-process and per-cpu basis.
There are several commands associated with sampling: record, report, annotate.
You must first collect the samples using perf record. This generates an output file called perf.data.
That file can then be analyzed, possibly on another machine, using the perf report and perf annotate commands.
The model is fairly similar to that of OProfile.

*

```
<<~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
[milenka@eb136i-nsf02 perf.tool]$ perf record arrsum.exe 16384
array sum is 17488290749289.000000
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.007 MB perf.data (~308 samples) ]
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~>>
```

6. Sample analysis with perf report

Samples collected by perf record are saved into a binary file called, by default, perf.data. The perf report command reads this file and generates a concise execution profile. By default, samples are sorted by functions with the most samples first. It is possible to customize the sorting order and therefore to view the data differently.

```
<<~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
[milenka@eb136i-nsf02 perf.tool]$ perf report
[kernel.kallsyms] with build id 24538a6c96fd2bf066815a8b229b156d930a14ea not found,
continuing without
# Samples: 41823770
#
# Overhead          Command                             Shared Object  Symbol
# ........  ...............  ........................................  ......
#
    94.75%       arrsum.exe  [kernel.kallsyms]                         [k]
inode_has_perm
     3.26%       arrsum.exe  [kernel.kallsyms]                         [k] pgd_alloc
     1.95%       arrsum.exe  perf.2.6.34.9-69.fc13.x86_64              [.]
__cmd_record
     0.04%       arrsum.exe  [kernel.kallsyms]                         [k]
native_write_msr_safe
     0.00%       arrsum.exe  [kernel.kallsyms]                         [k]
0xffffffff8102a6fb


#
# (For a higher level overview, try: perf report --sort comm,dso)
#
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~>>
```

The column 'Overhead' indicates the percentage of the overall samples collected in the corresponding function.
The second column reports the process from which the samples were collected.
In per-thread/per-process mode, this is always the name of the monitored command.
But in cpu-wide mode, the command can vary.
The third column shows the name of the ELF image where the samples came from.
If a program is dynamically linked, then this may show the name of a shared library.
When the samples come from the kernel, then the pseudo ELF image name [kernel.kallsyms] is used.
The fourth column indicates the privilege level at which the sample was taken,
i.e. when the program was running when it was interrupted:
[.] : user level
[k]: kernel level
[g]: guest kernel level (virtualization)
[u]: guest os user space
[H]: hypervisor
The final column shows the symbol name.

```
<<~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
[milenka@eb136i-nsf02 perf.tool]$ perf report --sort comm,dso
[kernel.kallsyms] with build id 24538a6c96fd2bf066815a8b229b156d930a14ea not found,
continuing without
# Samples: 41823770
#
```

```
# Overhead          Command                              Shared Object
# ........  ................  .........................................
#
    98.05%       arrsum.exe  [kernel.kallsyms]
     1.95%       arrsum.exe  perf.2.6.34.9-69.fc13.x86_64
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~>>
```

7. Source level analysis with perf annotate

It is possible to drill down to the instruction level with perf annotate. For that, you need to invoke perf annotate with the name of the command to annotate. Perf annotate can generate source code level information if the application is compiled with -ggdb.