

Set \$wgLogo to the URL path to your own logo image.

navigation

- [Main page](#)
- [Community portal](#)
- [Current events](#)
- [Recent changes](#)
- [Random page](#)
- [Help](#)

search

toolbox

- [What links here](#)
- [Related changes](#)
- [Special pages](#)
- [Printable version](#)
- [Permanent link](#)

Tutorial

Linux kernel profiling with perf

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 commandline interface. Perf is based on the `perf_events` interface exported by recent versions of the Linux kernel. This article demonstrates the `perf` tool through example runs. Output was obtained on a Ubuntu 11.04 system with kernel 2.6.38-8-generic results running on an HP 6710b with dual-core Intel Core2 T7100 CPU). For readability, some output is abbreviated using ellipsis ([...]).

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`, [...]

The list of supported commands:

```
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 <tt>build-id</tt> cache.
buildid-list List the buildids in a perf.data file
diff      Read two perf.data files and display the differential profile
inject    Filter to augment the events stream with additional information
kmem      Tool to trace/measure kernel memory(slab) properties
kvm       Tool to trace/measure kvm guest os
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)
script    Read perf.data (created by perf record) and display trace output
stat      Run a command and gather performance counter statistics
test      Runs sanity tests.
timechart Tool to visualize total system behavior during a workload
top       System profiling tool.
```

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`:

```
perf stat -h

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

-e, --event <event>    event selector. use 'perf list' to list available events
-i, --no-inherit        child tasks do not inherit counters
-p, --pid <n>           stat events on existing process id
-t, --tid <n>           stat events on existing thread id
-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
-B, --big-num           print large numbers with thousands' separators
```

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 event 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 an actual events provided by the CPU, if they exists, 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:

```
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-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 (see 'perf list --help' on how to encode it) [Raw hardware
event descriptor]

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

kvmmmu:kvm_mmu_pagetable_walk [Tracepoint event]

[...]

sched:sched_stat_runtime    [Tracepoint event]
sched:sched_pi_setprio      [Tracepoint event]
syscalls:sys_enter_socket   [Tracepoint event]
syscalls:sys_exit_socket    [Tracepoint event]



[...]

```

An event can have sub-events (or unit masks). On some processors and for some events, it may be possible to combine unit masks and measure when either sub-event occurs. Finally, an event can have modifiers, i.e., filters which alter when or how the event is counted.

Hardware events

PMU hardware events are CPU specific and documented by the CPU vendor. The `perf` tool, if linked against the `libpfm4` library, provides some short description of the events. For a listing of PMU hardware events for Intel and AMD processors, see

- Intel PMU event tables: Appendix A of manual [here](#) 
- AMD PMU event table: section 3.14 of manual [here](#) 

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:

```
perf stat -B dd if=/dev/zero of=/dev/null count=1000000
```

```
1000000+0 records in
1000000+0 records out
512000000 bytes (512 MB) copied, 0.956217 s, 535 MB/s
```

```
Performance counter stats for 'dd if=/dev/zero of=/dev/null count=1000000':
```

5,099	cache-misses	#	0.005	M/sec	(scaled from 66.58%)
235,384	cache-references	#	0.246	M/sec	(scaled from 66.56%)
9,281,660	branch-misses	#	3.858	%	(scaled from 33.50%)
240,609,766	branches	#	251.559	M/sec	(scaled from 33.66%)
1,403,561,257	instructions	#	0.679	IPC	(scaled from 50.23%)
2,066,201,729	cycles	#	2160.227	M/sec	(scaled from 66.67%)
217	page-faults	#	0.000	M/sec	
3	CPU-migrations	#	0.000	M/sec	
83	context-switches	#	0.000	M/sec	
956.474238	task-clock-msecs	#	0.999	CPUs	
0.957617512 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).

Options controlling event selection

It is possible to measure one or more events per run of the `perf` tool. Events are designated using their symbolic names followed by optional unit masks and modifiers. Event names, unit masks, and modifiers are case insensitive.

By default, events are measured at **both** user and kernel levels:

```
perf stat -e cycles dd if=/dev/zero of=/dev/null count=100000
```

To measure only at the user level, it is necessary to pass a modifier:

```
perf stat -e cycles:u dd if=/dev/zero of=/dev/null count=100000
```

To measure both user and kernel (explicitly):

```
perf stat -e cycles:uk dd if=/dev/zero of=/dev/null count=100000
```

Modifiers

Modifiers have a type indicated in parenthesis. The type determines the valid values. The value is passed after the equal sign (no space). Booleans accept `0`, `1`, `y`, `n`. To set a boolean modifier to true, it is possible to use `u=1` or simply `u`. Integer may have range restrictions, see `c` modifier in the example above.

Note: When using **hardware** events, e.g., `cycles`, only the `u` and `k` modifiers are accepted. To measure at both user and kernel level use `cycles:uk`. In other words, there is no colon separator between the modifiers.

To measure a PMU event and pass unit masks and modifiers:

```
perf stat -e inst_retired:any_p:u:c=1:i dd if=/dev/zero of=/dev/null count=100000
```

In this example, we are measuring the number of cycles at the user level in which less (i) than 1 (c=1) instruction is retired per cycles. Note that for actual events, the modifiers depends on the underlying PMU model. All modifiers can be combined at will. Here is a simple table to summarize the most common modifiers for Intel and AMD x86 processors.

Modifiers	Type	Description	Example
u	boolean	monitor at priv level 3, 2, 1 (user)	event:u=1 or event:u
k	boolean	monitor at priv level 0 (kernel)	event:k=1 or event:k
c	integer	threshold monitoring: number of cycles when n or more occurrences of event occur	event:c=2
i	boolean	invert the test of threshold: number of cycles in which less than n occurrences of the event occur	event:c=2:i
e	boolean	edge detect, increment the counter only when the condition goes from false -> true	event:e or event:e=1

Hardware events

To measure an actual PMU as provided by the HW vendor documentation, pass the hexadecimal parameter code:

```
perf stat -e r1a8 -a sleep 1
Performance counter stats for 'sleep 1':
      210,140 raw 0x1a8
    1.001213705 seconds time elapsed
```

multiple events

To measure more than one event, simply provide a comma-separated list with no space:

```
perf stat -e cycles,instructions,cache-misses [...]
```

There is no theoretical limit in terms of the number of events that can be provided. If there are more events than there are actual hw counters, the kernel will automatically multiplex them. There is no limit of the number of software events. It is possible to simultaneously measure events coming from different sources.

However, given that there is one file descriptor used per event and either per-thread (per-thread mode) or per-cpu (system-wide), it is possible to reach the maximum number of open file descriptor per process as imposed by the kernel. In that case, perf will report an error. See the troubleshooting section for help with this matter.

multiplexing and scaling events

If there are more events than counters, the kernel uses time multiplexing (switch frequency = `Hz`, generally 100 or 1000) to give each event a chance to access the monitoring hardware. Multiplexing only applies to PMU events. With multiplexing, an event is **not** measured all the time. At the end of the run, the tool **scales** the count based on total time enabled vs time running. The actual formula is:

```
final_count = raw_count * time_enabled/time_running
```

This provides an **estimate** of what the count would have been, had the event been measured during the entire run. It is **very** important to understand this is an **estimate** not an actual count. Depending on the workload, there will be blind spots which can introduce errors during scaling.

Events are currently managed in round-robin fashion. Therefore each event will eventually get a chance to run. If there are N counters, then up to the first N events on the round-robin list are programmed into the PMU. In certain situations it may be less than that because some events may not be measured together or they compete for the same counter. Furthermore, the perf_events interface allows multiple tools to measure the same thread or CPU at the same time. Each event is added to the same round-robin list. There is no guarantee that all events of a tool are stored sequentially in the list.

To avoid scaling (in the presence of only one active perf_event user), one can try and reduce the number of events. The following table provides the number of counters for a few common processors:

Processor	Generic counters	Fixed counters
Intel Core	2	3
Intel Nehalem	4	3

Generic counters can measure any events. Fixed counters can only measure one event. Some counters may be reserved for special purposes, such as a watchdog timer.

The following examples show the effect of scaling:

```
perf stat -B -e cycles,cycles ./noploop 1

Performance counter stats for './noploop 1':

    2,812,305,464 cycles
    2,812,305,464 cycles
    2,812,304,340 cycles

    1.302481065 seconds time elapsed
```

Here, there is no multiplexing and thus no scaling. Let's add one more event:

```
perf stat -B -e cycles,cycles,cycles ./noploop 1

Performance counter stats for './noploop 1':

    2,809,946,289 cycles          (scaled from 74.98%)
    2,809,725,593 cycles          (scaled from 74.98%)
    2,810,797,044 cycles          (scaled from 74.97%)
    2,809,315,647 cycles          (scaled from 75.09%)

    1.295007067 seconds time elapsed
```

There was multiplexing and thus scaling. It can be interesting to try and pack events in a way that guarantees that event A and B are always measured together. Although the perf_events kernel interface provides support for event grouping, the current perf tool does **not**.

Repeated measurement

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.

```
perf stat -r 5 sleep 1
```

```
Performance counter stats for 'sleep 1' (5 runs):
```

```
<not counted> cache-misses
20,676 cache-references      #    13.046 M/sec    ( +-   0.658% )
 6,229 branch-misses        #     0.000 %      ( +-  40.825% )
<not counted> branches
<not counted> instructions
<not counted> cycles
144 page-faults             #     0.091 M/sec    ( +-   0.139% )
 0 CPU-migrations           #     0.000 M/sec    ( +-   -nan% )
 1 context-switches          #     0.001 M/sec    ( +-   0.000% )
1.584872 task-clock-msecs    #     0.002 CPUs     ( +-  12.480% )

1.002251432 seconds time elapsed ( +-   0.025% )
```

Here, `sleep` is run 5 times and the mean count for each event, along with ratio of std-dev/mean is printed.

Options controlling environment selection

The `perf` tool can be used to count events on a per-thread, per-process, per-cpu or system-wide basis. In *per-thread* mode, the counter only monitors the execution of a designated thread. When the thread is scheduled out, monitoring stops. When a thread migrated from one processor to another, counters are saved on the current processor and are restored on the new one.

The *per-process* mode is a variant of per-thread where **all** threads of the process are monitored. Counts and samples are aggregated at the process level. The `perf_events` interface allows for automatic inheritance on `fork()` and `pthread_create()`. By default, the `perf` tool **activates** inheritance.

In *per-cpu* mode, all threads running on the designated processors are monitored. Counts and samples are thus aggregated per CPU. An event is only monitoring one CPU at a time. To monitor across multiple processors, it is necessary to create multiple events. The `perf` tool can aggregate counts and samples across multiple processors. It can also monitor only a subset of the processors.

Counting and inheritance

By default, `perf stat` counts for all threads of the process and subsequent child processes and threads. This can be altered using the `-i` option. It is not possible to obtain a count breakdown per-thread or per-process.

Processor-wide mode

By default, `perf stat` counts in per-thread mode. To count on a per-cpu basis pass the `-a` option. When it is specified by itself, all online processors are monitored and counts are aggregated. For instance:

```
perf stat -B -ecycles:u,instructions:u -a dd if=/dev/zero of=/dev/null count=2000000
```

```
2000000+0 records in
2000000+0 records out
1024000000 bytes (1.0 GB) copied, 1.91559 s, 535 MB/s
```

```
Performance counter stats for 'dd if=/dev/zero of=/dev/null count=2000000':
```

```
1,993,541,603 cycles
764,086,803 instructions      #    0.383 IPC
```

```
1.916930613 seconds time elapsed
```

This measurement collects `events` `cycles` and `instructions` across all CPUs. The duration of the measurement is determined by the execution of `dd`. In other words, this measurement captures execution of the `dd` process **and** anything else than runs at the user level on all CPUs.

To time the duration of the measurement without actively consuming cycles, it is possible to use the `=/usr/bin/sleep=` command:

```
perf stat -B -e cycles:u,instructions:u -a sleep 5

Performance counter stats for 'sleep 5':

    766,271,289 cycles
    596,796,091 instructions          #      0.779 IPC

    5.001191353 seconds time elapsed
```

It is possible to restrict monitoring to a subset of the CPUs using the `-c` option. A list of CPUs to monitor can be passed. For instance, to measure on CPU0, CPU2 and CPU3:

```
perf stat -B -e cycles:u,instructions:u -a -C 0,2-3 sleep 5
```

The demonstration machine has only two CPUs, but we can limit to CPU 1.

```
perf stat -B -e cycles:u,instructions:u -a -C 1 sleep 5

Performance counter stats for 'sleep 5':

    301,141,166 cycles
    225,595,284 instructions          #      0.749 IPC

    5.002125198 seconds time elapsed
```

Counts are aggregated across all the monitored CPUs. Notice how the number of counted cycles and instructions are both halved when measuring a single CPU.

Attaching to a running process

It is possible to use `perf` to attach to an already running thread or process. This requires the permission to attach along with the thread or process ID. To attach to a process, the `-p` option must be the process ID. To attach to the `sshd` service that is commonly running on many Linux machines, issue:

```
ps ax | fgrep sshd

2262 ?        Ss      0:00 /usr/sbin/sshd -D
2787 pts/0    S+      0:00 fgrep --color=auto sshd

perf stat -e cycles -p 2262 sleep 2

Performance counter stats for process id '2262':

    <not counted> cycles
```



```
2.001263149 seconds time elapsed
```

What determines the duration of the measurement is the command to execute. Even though we are attaching to a process, we can still pass the name of a command. It is used to time the measurement. Without it, `perf` monitors until it is killed. Also note that when attaching to a process, all threads of the process are monitored. Furthermore, given that inheritance is on by default, child processes or threads will also be monitored. To turn this off, you must use the `-i` option. It is possible to attach a specific thread within a process. By thread, we mean kernel visible thread. In other words, a thread visible by the `ps` or `top` commands. To attach to a thread, the `-t` option must be used. We look at `rsyslogd`, because it always runs on Ubuntu 11.04, with multiple threads.

```
ps -L ax | fgrep rsyslogd | head -5
889  889 ?      Sl      0:00 rsyslogd -c4
889  932 ?      Sl      0:00 rsyslogd -c4
889  933 ?      Sl      0:00 rsyslogd -c4
2796 2796 pts/0  S+      0:00 fgrep --color=auto rsyslogd

perf stat -e cycles -t 932 sleep 2

Performance counter stats for thread id '932':

    <not counted> cycles

    2.001037289 seconds time elapsed
```

In this example, the thread 932 did not run during the 2s of the measurement. Otherwise, we would see a count value. Attaching to kernel threads is possible, though not really recommended. Given that kernel threads tend to be pinned to a specific CPU, it is best to use the `cpu-wide` mode.

Options controlling output

`perf stat` can modify output to suit different needs.

Pretty printing large numbers

For most people, it is hard to read large numbers. With `perf stat`, it is possible to print large numbers using the comma separator for thousands (US-style). For that the `-B` option and the correct locale for `LC_NUMERIC` must be set. As the above example showed, Ubuntu already sets the locale information correctly. An explicit call looks as follows:

```
LC_NUMERIC=en_US.UTF8 perf stat -B -e cycles:u,instructions:u dd if=/dev/zero of=/dev/null count=10000000

100000+0 records in
100000+0 records out
51200000 bytes (51 MB) copied, 0.0971547 s, 527 MB/s

Performance counter stats for 'dd if=/dev/zero of=/dev/null count=1000000':

    96,551,461 cycles
    38,176,009 instructions          #      0.395 IPC

    0.098556460 seconds time elapsed
```

Machine readable output

`perf stat` can also print counts in a format that can easily be imported into a spreadsheet or parsed by scripts. The `-x` option alters the format of the output and allows users to pass a field delimiter. This makes it easy to produce CSV-style output:

```
perf stat -x, date
Thu May 26 21:11:07 EDT 2011
884,cache-misses
32559,cache-references
<not counted>,branch-misses
<not counted>,branches
<not counted>,instructions
<not counted>,cycles
188,page-faults
2,CPU-migrations
0,context-switches
2.350642,task-clock-msecs
```

Note that the `-x` option is not compatible with `-b`.

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.

Event-based sampling overview

`Perf_events` is based on event-based sampling. The period is expressed as the number of occurrences of an event, not the number of timer ticks. A sample is recorded when the sampling counter overflows, i.e., wraps from 2^{64} back to 0. No PMU implements 64-bit hardware counters, but `perf_events` emulates such counters in software.

The way `perf_events` emulates 64-bit counter is limited to expressing sampling periods using the number of bits in the actual hardware counters. If this is smaller than 64, the kernel **silently** truncates the period in this case. Therefore, it is best if the period is always smaller than 2^{31} if running on 32-bit systems.

On counter overflow, the kernel records information, i.e., a sample, about the execution of the program. What gets recorded depends on the type of measurement. This is all specified by the user and the tool. But the key information that is common in all samples is the instruction pointer, i.e. where was the program when it was interrupted.

Interrupt-based sampling introduces skids on modern processors. That means that the instruction pointer stored in each sample designates the place where the program was interrupted to process the PMU interrupt, not the place where the counter actually overflows, i.e., where it was at the end of the sampling period. In some case, the distance between those two points may be several dozen instructions or more if there were taken branches. When the program cannot make forward progress, those two locations are indeed identical. *For this reason, care must be taken* when interpreting profiles.

Default event: cycle counting

By default, `perf record` uses the `cycles` event as the sampling event. This is a generic hardware event that is mapped to a hardware-specific PMU event by the kernel. For Intel, it is mapped to `UNHALTED_CORE_CYCLES`. This event does not maintain a constant correlation to time in the presence of CPU frequency scaling. Intel provides another event, called `UNHALTED_REFERENCE_CYCLES` but this event is NOT currently available with `perf_events`.

On AMD systems, the event is mapped to `CPU_CLK_UNHALTED` and this event is also subject to frequency scaling. On any Intel or AMD processor, the `cycle` event does not count when the processor is idle, i.e., when it calls `mwait()`.

Period and rate

The `perf_events` interface allows two modes to express the sampling period:

- the number of occurrences of the event (period)
- the average rate of samples/sec (frequency)

The `perf` tool defaults to the average rate. It is set to 1000Hz, or 1000 samples/sec. That means that the kernel is dynamically adjusting the sampling period to achieve the target average rate. The adjustment in period is reported in the raw profile data. In contrast, with the other mode, the sampling period is set by the user and does not vary between samples. There is currently no support for sampling period randomization.

Collecting samples

By default, `perf record` operates in per-thread mode, with `inherit` mode enabled. The simplest mode looks as follows, when executing a simple program that busy loops:

```
perf record ./noploop 1
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.002 MB perf.data (~89 samples) ]
```

The example above collects samples for event `cycles` at an average target rate of 1000Hz. The resulting samples are saved into the `perf.data` file. If the file already existed, you may be prompted to pass `-f` to overwrite it. To put the results in a specific file, use the `-o` option.

WARNING: The number of reported samples is only an **estimate**. It does not reflect the actual number of samples collected. The estimate is based on the number of bytes written to the `perf.data` file and the minimal sample size. But the size of each sample depends on the type of measurement. Some samples are generated by the counters themselves but others are recorded to support symbol correlation during post-processing, e.g., `mmap()` information.

To get an accurate number of samples for the `perf.data` file, it is possible to use the `perf report` command:

```
perf record ./noploop 1
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.058 MB perf.data (~2526 samples) ]
perf report -D -i perf.data | fgrep RECORD_SAMPLE | wc -l
1280
```

To specify a custom rate, it is necessary to use the `-F` option. For instance, to sample on event `instructions` only at the user level and

at an average rate of 250 samples/sec:

```
perf record -e instructions:u -F 250 ./noploop 4

[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.049 MB perf.data (~2160 samples) ]
```

To specify a sampling period, instead, the `-c` option must be used. For instance, to collect a sample every 2000 occurrences of event `instructions` only at the user level only:

```
perf record -e retired_instructions:u -c 2000 ./noploop 4

[ perf record: Woken up 55 times to write data ]
[ perf record: Captured and wrote 13.514 MB perf.data (~590431 samples) ]
```

Processor-wide mode

In per-cpu mode mode, samples are collected for all threads executing on the monitored CPU. To switch `perf record` in per-cpu mode, the `-a` option must be used. By default in this mode, **ALL** online CPUs are monitored. It is possible to restrict to the a subset of CPUs using the `-c` option, as explained with `perf stat` above.

To sample on `cycles` at both user and kernel levels for 5s on all CPUS with an average target rate of 1000 samples/sec:

```
perf record -a -F 1000 sleep 5

[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.523 MB perf.data (~22870 samples) ]
```

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.

```
perf report
# Events: 1K cycles
#
# Overhead      Command          Shared Object
# .....
#
# 28.15%      firefox-bin  libxul.so          [.] 0xd10b45
#  4.45%      swapper   [kernel.kallsyms]  [k] mwait_idle_with_hints
#  4.26%      swapper   [kernel.kallsyms]  [k] read_hpet
#  2.13%      firefox-bin  firefox-bin        [.] 0x1e3d
```

```
1.40%  unity-panel-ser  libglib-2.0.so.0.2800.6      [.] 0x886f1
[...]
```

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.

There are many different ways samples can be presented, i.e., sorted. To sort by shared objects, i.e., dsos:

```
perf report --sort=dso
# Events: 1K cycles
#
# Overhead          Shared Object
# .....
#
38.08% [kernel.kallsyms]
28.23% libxul.so
3.97%  libglib-2.0.so.0.2800.6
3.72%  libc-2.13.so
3.46%  libpthread-2.13.so
2.13%  firefox-bin
1.51%  libdrm_intel.so.1.0.0
1.38%  dbus-daemon
1.36%  [drm]
[...]
```

Options controlling output

To make the output easier to parse, it is possible to change the column separator to a single character:

```
perf report -t
```

Options controlling kernel reporting

The `perf` tool does not know how to extract symbols from compressed kernel images (`vmlinuz`). Therefore, users must pass the path of the uncompressed kernel using the `-k` option:

```
perf report -k /tmp/vmlinux
```

Of course, this works only if the kernel is compiled to with debug symbols.

Processor-wide mode

In per-cpu mode, samples are recorded from all threads running on the monitored CPUs. As a result, samples from many different processes may be collected. For instance, if we monitor across all CPUs for 5s:

```
perf record -a sleep 5
perf report

# Events: 354 cycles
#
# Overhead      Command      Shared Object      Symbol
# .....
#
# 13.20%        swapper [kernel.kallsyms] [k] read_hpet
#  7.53%        swapper [kernel.kallsyms] [k] mwait_idle_with_hints
#  4.40%    perf_2.6.38-8 [kernel.kallsyms] [k] _raw_spin_unlock_irqrestore
#  4.07%    perf_2.6.38-8 perf_2.6.38-8      [.] 0x34e1b
#  3.88%    perf_2.6.38-8 [kernel.kallsyms] [k] format_decode
# [...]
```

When the symbol is printed as an hexadecimal address, this is because the ELF image does not have a symbol table. This happens when binaries are stripped. We can sort by cpu as well. This could be useful to determine if the workload is well balanced:

```
perf report --sort=cpu

# Events: 354 cycles
#
# Overhead  CPU
# .....
#
# 65.85%    1
# 34.15%    0
```

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. All the functions with samples will be disassembled and each instruction will have its relative percentage of samples reported:

```
perf record ./noploop 5
perf annotate -d ./noploop

-----
Percent | Source code & Disassembly of noploop.nogddb
-----
:
:
:
: Disassembly of section .text:
:
:
: 08048484 <main>:
0.00 : 8048484: 55                push    %ebp
```

```

0.00 : 8048485: 89 e5          mov    %esp,%ebp
[...]
0.00 : 8048530: eb 0b          jmp    804853d <main+0xb9>
15.08 : 8048532: 8b 44 24 2c    mov    0x2c(%esp),%eax
0.00 : 8048536: 83 c0 01       add    $0x1,%eax
14.52 : 8048539: 89 44 24 2c    mov    %eax,0x2c(%esp)
14.27 : 804853d: 8b 44 24 2c    mov    0x2c(%esp),%eax
56.13 : 8048541: 3d ff e0 f5 05 cmp    $0x5f5e0ff,%eax
0.00 : 8048546: 76 ea          jbe    8048532 <main+0xae>
[...]
```

The first column reports the percentage of samples for function `==noploop()==` captured for at that instruction. As explained earlier, you should interpret this information carefully.

`perf annotate` can generate sourcecode level information if the application is compiled with `-ggdb`. The following snippet shows the much more informative output for the same execution of `noploop` when compiled with this debugging information.

```

-----
Percent | Source code & Disassembly of noploop
-----
:
:
: Disassembly of section .text:
:
: 08048484 <main>:
: #include <string.h>
: #include <unistd.h>
: #include <sys/time.h>
:
: int main(int argc, char **argv)
: {
0.00 : 8048484: 55             push   %ebp
0.00 : 8048485: 89 e5          mov    %esp,%ebp
[...]
0.00 : 8048530: eb 0b          jmp    804853d <main+0xb9>
:
:             count++;
14.22 : 8048532: 8b 44 24 2c    mov    0x2c(%esp),%eax
0.00 : 8048536: 83 c0 01       add    $0x1,%eax
14.78 : 8048539: 89 44 24 2c    mov    %eax,0x2c(%esp)
:
:             memcpy(&tv_end, &tv_now, sizeof(tv_now));
:             tv_end.tv_sec += strtol(argv[1], NULL, 10);
:             while (tv_now.tv_sec < tv_end.tv_sec ||
:                 tv_now.tv_usec < tv_end.tv_usec) {
:                 count = 0;
:                 while (count < 1000000000UL)
14.78 : 804853d: 8b 44 24 2c    mov    0x2c(%esp),%eax
56.23 : 8048541: 3d ff e0 f5 05 cmp    $0x5f5e0ff,%eax
0.00 : 8048546: 76 ea          jbe    8048532 <main+0xae>
[...]
```

Using perf annotate on kernel code

The `perf` tool does not know how to extract symbols from compressed kernel images (`vmlinuz`). As in the case of `perf report`, users must pass the path of the uncompressed kernel using the `-k` option:

```
perf annotate -k /tmp/vmlinux -d symbol
```

Again, this only works if the kernel is compiled to with debug symbols.

Live analysis with perf top

The perf tool can operate in a mode similar to the Linux top tool, printing sampled functions in real time. The default sampling event is `cycles` and default order is descending number of samples per symbol, thus `perf top` shows the functions where most of the time is spent. By default, `perf top` operates in processor-wide mode, monitoring all online CPUs at both user and kernel levels. It is possible to monitor only a subset of the CPUs using the `-c` option.

```
perf top
-----
PerfTop: 260 irqs/sec kernel:61.5% exact: 0.0% [1000Hz
cycles], (all, 2 CPUs)
-----

```

	samples	pcnt	function	DSO
	80.00	23.7%	read_hpet	[kernel.kallsyms]
	14.00	4.2%	system_call	[kernel.kallsyms]
	14.00	4.2%	__ticket_spin_lock	[kernel.kallsyms]
	14.00	4.2%	__ticket_spin_unlock	[kernel.kallsyms]
	8.00	2.4%	hpet_legacy_next_event	[kernel.kallsyms]
	7.00	2.1%	i8042_interrupt	[kernel.kallsyms]
	7.00	2.1%	strcmp	[kernel.kallsyms]
	6.00	1.8%	_raw_spin_unlock_irqrestore	[kernel.kallsyms]
	6.00	1.8%	pthread_mutex_lock	/lib/i386-linux-gnu/libpthread-2.13.so
	6.00	1.8%	fget_light	[kernel.kallsyms]
	6.00	1.8%	__pthread_mutex_unlock_usercnt	/lib/i386-linux-gnu/libpthread-2.13.so
	5.00	1.5%	native_sched_clock	[kernel.kallsyms]
	5.00	1.5%	drm_addbufs_sg	/lib/modules/2.6.38-8-generic/kernel/drivers/gpu/drm/drm.ko

By default, the first column shows the aggregated number of samples since the beginning of the run. By pressing the 'Z' key, this can be changed to print the number of samples since the last refresh. Recall that the `cycle` event counts CPU cycles when the processor is not in halted state, i.e. not idle. Therefore this is **not** equivalent to wall clock time. Furthermore, the event is also subject to frequency scaling.

It is also possible to drill down into single functions to see which instructions have the most samples. To drill down into a specify function, press the 's' key and enter the name of the function. Here we selected the top function `noploop` (not shown above):

```
-----
PerfTop: 2090 irqs/sec kernel:50.4% exact: 0.0% [1000Hz cycles], (all, 16 CPUs)
-----
Showing cycles for noploop
Events Pcnt (>=5%)
  0 0.0% 00000000004003a1 <noploop>:
  0 0.0% 4003a1: 55                push    %rbp
  0 0.0% 4003a2: 48 89 e5          mov     %rsp,%rbp
3550 100.0% 4003a5: eb fe          jmp     4003a5 <noploop+0x4>
```

Troubleshooting and Tips

This section lists a number of tips to avoid common pitfalls when using perf.

Open file limits

The design of the `perf_event` kernel interface which is used by the `perf` tool, is such that it uses one file descriptor per event per-thread or per-cpu.

On a 16-way system, when you do:

```
perf stat -e cycles sleep 1
```

You are effectively creating 16 events, and thus consuming 16 file descriptors.

In per-thread mode, when you are sampling a process with 100 threads on the same 16-way system:

```
perf record -e cycles my_hundred_thread_process
```

Then, once all the threads are created, you end up with $100 * 1 \text{ (event)} * 16 \text{ (cpus)} = 1600$ file descriptors. `Perf` creates one instance of the event on each CPU. Only when the thread executes on that CPU does the event effectively measure. This approach enforces sampling buffer locality and thus mitigates sampling overhead. At the end of the run, the tool aggregates all the samples into a single output file.

In case `perf` aborts with 'too many open files' error, there are a few solutions:

- increase the number of per-process open files using `ulimit -n`. Caveat: you must be root
- limit the number of events you measure in one run
- limit the number of CPU you are measuring

increasing open file limit

The superuser can override the per-process open file limit using the `==ulimit==` shell builtin command:

```
ulimit -a
[...]  
open files                (-n) 1024  
[...]  
  
ulimit -n 2048  
ulimit -a  
[...]  
open files                (-n) 2048  
[...]
```

Binary identification with build-id

The `perf record` command saves in the `perf.data` unique identifiers for all ELF images relevant to the measurement. In per-thread mode, this includes all the ELF images of the monitored processes. In cpu-wide mode, it includes all running processes running on the system. Those unique identifiers are generated by the linker if the `-Wl,--build-id` option is used. Thus, they are called `build-id`. The `build-id` are a helpful tool when correlating instruction addresses to ELF images. To extract all `build-id` entries used in a `perf.data` file, issue:

```
perf buildid-list -i perf.data
```

```
06cb68e95ccef1ff4e80a3663ad339d9d6f0e43 [kernel.kallsyms]
e445a2c74bc98ac0c355180a8d770cd35deb7674 /lib/modules/2.6.38-8-generic/kernel/drivers/gpu/drm/i915/i915.ko
83c362c95642c3013196739902b0360d5cbb13c6 /lib/modules/2.6.38-8-generic/kernel/drivers/net/wireless/iwlwifi/iwlcore.ko
1b71b1dd65a7734e7aa960efbde449c430bc4478 /lib/modules/2.6.38-8-generic/kernel/net/mac80211/mac80211.ko
ae4d6ec2977472f40b6871fb641e45efd408fa85 /lib/modules/2.6.38-8-generic/kernel/drivers/gpu/drm/drm.ko
fafad827c43e34b538aea792cc98ecfd8d387e2f /lib/i386-linux-gnu/ld-2.13.so
0776add23cf3b95b4681e4e875ba17d62d30c7ae /lib/i386-linux-gnu/libdbus-1.so.3.5.4
f22f8e683907b95384c5799b40daa455e44e4076 /lib/i386-linux-gnu/libc-2.13.so
[...]
```

The build-id cache

At the end of each run, the `perf record` command updates a build-id cache, with new entries for ELF images with samples. The cache contains:

- build-id for ELF images with samples
- copies of the ELF images with samples

Given that build-id are immutable, they uniquely identify a binary. If a binary is recompiled, a new build-id is generated and a new copy of the ELF images is saved in the cache. The cache is saved on disk in a directory which is by default `$HOME/.debug`. There is a global configuration file `==/etc/perfconfig==` which can be used by sysadmin to specify an alternate global directory for the cache:

```
$ cat /etc/perfconfig
[buildid]
dir = /var/tmp/.debug
```

In certain situations it may be beneficial to turn off the build-id cache updates altogether. For that, you must pass the `-N` option to `perf record`

```
perf record -N dd if=/dev/zero of=/dev/null count=100000
```

Access Control

For some events, it is necessary to be root to invoke the `perf` tool. This document assumes that the user has root privileges. If you try to run `perf` with insufficient privileges, it will report

```
No permission to collect system-wide stats.
```

Other Resources


Linux sourcecode

The `perf` tools sourcecode lives in the Linux kernel tree under `/tools/perf`. You will find much more documentation in `/tools/perf/Documentation`. To build manpages, info pages and more, install these tools:

- asciidoc
- tetex-fonts
- tetex-dvips

- `dialog`
- `tetex`
- `tetex-latex`
- `xmltex`
- `passivetex`
- `w3m`
- `xmlto`

and issue a `make install-man` from `/tools/perf`. This step is also required to be able to run `perf help <command>`.

This guide is adapted from a tutorial by Stephane Eranian at Google, with contributions from Eric Gouriou, Tipp Moseley and Willem de Bruijn. The original content imported into wiki.perf.google.com is made available under the [Creative Commons attribution sharealike 3.0 license](https://creativecommons.org/licenses/by-sa/3.0/) .

This page was last modified 19:39, 29 June 2011 by Willem de Bruijn?

[Privacy policy](#)

[About Perf Wiki](#)

[Disclaimers](#)

