System Performance Tools: perf

CS165: Data Systems — Fall 2015

October 28, 2015

This document provides a short introduction to tools commonly used for systems profiling. We cover profiling with *perf* through examples. This tutorial is based amongst other sources on `https://perf.wiki.kernel.org/index.php/Tutorial`. Other good tutorials can be found at `http://www.brendangregg.com/perf.html` and `http://sandsoftwaresound.net/perf/perf-tutorial-hot-spots/`.

Profiling with perf

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. Perf is based on the perf_events interface exported by recent versions of the Linux kernel. Here we demonstrates the perf tool through example runs. **Output was obtained on a Debian Wheezy 7.9 system with kernel 3.18.11 results running on a 4-socket Intel Xeon E7-4820 v2 Ivy Bridge)**[1]. For readability, some output is abbreviated using ellipsis ([...]).

[1] This is in fact the machine we are using for the tests

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.
```

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-faults.

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[2]:

```
$ perf list
```

[2] All available events in our server can be found here `http://daslab.seas.harvard.edu/servers/adama/perf_events.txt`

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.

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
```

would give:

```
1000000+0 records in
1000000+0 records out
512000000 bytes (512 MB) copied, 0.407795 s, 1.3 GB/s

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

        408.479346      task-clock (msec)         #    0.998 CPUs utilized
                 8      context-switches          #    0.020 K/sec
                 0      cpu-migrations            #    0.000 K/sec
                76      page-faults               #    0.186 K/sec
     1,020,659,628      cycles                    #    2.499 GHz
       400,205,489      stalled-cycles-frontend   #   39.21% frontend cycles idle
   <not supported>      stalled-cycles-backend
     1,643,498,017      instructions              #    1.61
insns per cycle
                                                  #    0.24
stalled cycles per insn
       332,324,072      branches                  #  813.564 M/sec
         1,012,543      branch-misses             #    0.30% of all branches

       0.409323105 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
```

Measure 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 (the testing server is an Ivy Bridge):

Processor	Generic Counters	Fixed Counters
Intel Core	2	3
Intel Nehalem	4	3
Intel Ivy Bridge (HT on)	4	3
Intel Ivy Bridge (HT off)	8	3

Generic counters can measure any event. 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[3]:

[3] Here we use the noploop benchmark: http://www.brendangregg.com/blog/2014-04-26/the-noploop-cpu-benchmark.html

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

 Performance counter stats for './noploop':

     5,044,187,042      cycles
     5,044,187,030      cycles
     5,044,187,030      cycles

       2.019098080 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,cycles,cycles,cycles ./noploop

 Performance counter stats for './noploop':

     5,042,263,226      cycles                    [83.36%]
     5,042,138,165      cycles                    [83.36%]
     5,042,174,429      cycles                    [83.36%]
     5,041,937,370      cycles                    [83.36%]
     5,042,137,362      cycles                    [83.36%]
     5,033,444,804      cycles                    [83.33%]
```

```
       2.020170625 seconds time elapsed
```

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):

          0.465979      task-clock (msec)         #    0.000 CPUs utilized
( +-  2.15% )
                 1      context-switches          #    0.002 M/sec
                 0      cpu-migrations            #    0.000 K/sec
                56      page-faults               #    0.119 M/sec
( +-  1.22% )
         1,134,026      cycles                    #    2.434 GHz
( +-  2.17% )
           826,792      stalled-cycles-frontend   #   72.91% frontend cycles idle
( +-  3.08% )
   <not supported>      stalled-cycles-backend
           617,402      instructions              #    0.54
insns per cycle
                                                  #    1.34
stalled cycles per insn  ( +-  1.02% )
           137,798      branches                  #  295.716 M/sec
( +-  1.01% )
             6,991      branch-misses             #    5.07% of all branches
( +-  0.62% )

       1.001362075 seconds time elapsed
( +-  0.00% )
```

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, 0.811336 s, 1.3 GB/s

 Performance counter stats for 'system wide':

       439,673,000      cycles:u           [100.00%]
       633,677,536      instructions:u     #1.44  IPC

       0.812883452 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 -ecycles:u,instructions:u -a sleep 5

 Performance counter stats for 'system wide':

        61,339,632      cycles:u          [100.00%]
        73,528,386      instructions:u    #1.20 IPC

       5.001298458 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

 Performance counter stats for 'system wide':

           51,150      cycles:u          [100.00%]
           12,189      instructions:u    #0.24 IPC

       5.001239100 seconds time elapsed
```

Counts are aggregated across all the monitored CPUs. Notice how the number of counted cycles and instructions are reduced dramatically when measuring only 3 CPUs (out of 64).

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
  4625 ?        Ss     0:01 /usr/sbin/sshd
104078 ?        Ss     0:00 sshd: manos [priv]
104083 ?        S      0:00 sshd: manos@pts/1
104349 pts/1    S+     0:00 fgrep sshd
$ perf stat -e cycles -p 4625 sleep 2

 Performance counter stats for process id '4625':

     <not counted>      cycles

       2.001250972 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 Debian, with multiple threads.

```
$ ps -L ax | fgrep rsyslogd | head -5
  3735   3735 ?        Sl     0:00 /usr/sbin/rsyslogd -c5
  3735   3742 ?        Sl     0:33 /usr/sbin/rsyslogd -c5
  3735   3743 ?        Sl     0:07 /usr/sbin/rsyslogd -c5
  3735   3744 ?        Sl     0:00 /usr/sbin/rsyslogd -c5
104358 104358 pts/1    S+     0:00 fgrep rsyslogd
$ perf stat -e cycles -t 3742 sleep 2

 Performance counter stats for thread id '3742':

     <not counted>      cycles

       2.001415190 seconds time elapsed
```

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.[4]

[4] 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: (1) the number of occurrences of the event (period), and (2) 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 the noloop benchmark (which is busy looping):

```
$ perf record ./noploop
[ perf record: Woken up 2 times to write data ]
[ perf record: Captured and wrote 0.322 MB perf.data (~14055 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 report -D -i perf.data | fgrep RECORD_SAMPLE | wc -l
8151
```

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
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.030 MB perf.data (~1322 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 instructions:u -c 2000 ./noploop
[ perf record: Woken up 7 times to write data ]
[ perf record: Captured and wrote 1.699 MB perf.data (~74222 samples) ]
```

Processor-wide mode. In per-cpu 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.

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.

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
```

and the report:

```
$ perf report
# Samples: 527  of event 'cycles'
# Event count (approx.): 456470131
# Overhead  Command       Shared Object      Symbol
# ........  ............  .................  ......................
#
    27.54%  swapper       [kernel.kallsyms]  [k] 0xffffffff8124897a
```

```
3.90%  tmux          [kernel.kallsyms]  [k] 0xffffffff8113ac61
3.86%  perf_3.18     [kernel.kallsyms]  [k] 0xffffffff810908f1
3.21%  swapper       [kernel.kallsyms]  [k] 0xffffffff8107be14
3.06%  swapper       [kernel.kallsyms]  [k] 0xffffffff8103908a
2.58%  swapper       [kernel.kallsyms]  [k] 0xffffffff81042d2c
2.56%  sleep         [kernel.kallsyms]  [k] 0xffffffff81124aff
1.90%  swapper       [kernel.kallsyms]  [k] 0xffffffff8106310d
1.07%  swapper       [kernel.kallsyms]  [k] 0xffffffff8106fca1
1.07%  htop          [kernel.kallsyms]  [k] 0xffffffff81137422
```

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
# Samples: 527  of event 'cycles'
# Event count (approx.): 456470131
#
# Overhead  Shared Object
# ........  ..................
    88.18%  [kernel.kallsyms]
    11.82%  [unknown]
```

Processor-wide mode. In per-cpu mode, samples are recorded from all threads running on the monitored CPUs. As as 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
# Samples: 1K of event 'cycles'
# Event count (approx.): 367549016
#
# Overhead  Command       Shared Object      Symbol
# ........  ............  .................  ......................
#
    12.01%  swapper       [kernel.kallsyms]  [k] 0xffffffff8124897a
     2.17%  htop          [kernel.kallsyms]  [k] 0xffffffff8106fca1
     1.32%  perf_3.18     [kernel.kallsyms]  [k] 0xffffffff810908ee
     0.99%  htop          [unknown]          [.] 0x00007f17db4a2910
     0.95%  perf_3.18     [kernel.kallsyms]  [k] 0xffffffff810908f1
     0.73%  htop          [kernel.kallsyms]  [k] 0xffffffff8117460d
     0.67%  htop          [kernel.kallsyms]  [k] 0xffffffff811262af
     0.67%  htop          [kernel.kallsyms]  [k] 0xffffffff811fb565
     0.67%  htop          [unknown]          [.] 0x00007f17db418aaa
     0.60%  swapper       [kernel.kallsyms]  [k] 0xffffffff8105e49f
     0.51%  htop          [kernel.kallsyms]  [k] 0xffffffff8106f636
     0.51%  htop          [kernel.kallsyms]  [k] 0xffffffff81137392
     0.51%  htop          [unknown]          [.] 0x00007f17db421a04
     0.51%  htop          [kernel.kallsyms]  [k] 0xffffffff813d08cb
     0.50%  htop          [unknown]          [.] 0x00007f17db418aac
         [...]
```

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:

```
$ # Samples: 1K of event 'cycles'
# Event count (approx.): 367549016
#
# Overhead  CPU
# ........  ...
#
    68.84%  001
     4.68%  000
     3.06%  029
     3.00%  026
     1.25%  017
     1.10%  004
     0.93%  043
     0.87%  015
     0.64%  032
```

```
0.57%  008
0.56%  013
0.54%  014
0.52%  022
    [...]
```

Overhead calculation. The overhead can be shown in two columns as 'Children' and 'Self' when perf collects callchains. The 'self' overhead is simply calculated by adding all period values of the entry - usually a function (symbol). This is the value that perf shows traditionally and sum of all the 'self' overhead values should be 100%.

The 'children' overhead is calculated by adding all period values of the child functions so that it can show the total overhead of the higher level functions even if they don't directly execute much. 'Children' here means functions that are called from another (parent) function.

It might be confusing that the sum of all the 'children' overhead values exceeds 100% since each of them is already an accumulation of 'self' overhead of its child functions. But with this enabled, users can find which function has the most overhead even if samples are spread over the children.

Consider the following example:

```
1  void foo(void)
2  {
3      int i=0,j=0;
4      for (i=0;i<10000000;i++)
5          j=i%13;
6  }
7
8  void bar(void)
9  {
10     int i=0,j=0;
11     for (i=0;i<10000000;i++)
12         j=i%13;
13     foo();
14 }
15
16 int main(void)
17 {
18     bar();
19     return 0;
20 }
```

In this case 'foo' is a child of 'bar', and 'bar' is an immediate child of 'main' so 'foo' also is a child of 'main'. In other words, 'main' is a parent of 'foo' and 'bar', and 'bar' is a parent of 'foo'.

Suppose all samples are recorded in 'foo' and 'bar' only. When it's recorded with callchains the output will show something like below in the usual (self-overhead-only) output of perf report:

```
$ perf record -g ./overhead
$ perf report -g
# Overhead   Command      Shared Object  Symbol
# ........  ........  ..................  ................
#
    53.93%  overhead  overhead           [.] bar
            |
            --- bar
                main
                __libc_start_main

    41.01%  overhead  overhead           [.] foo
            |
            --- foo
                bar
                main
                __libc_start_main

     2.25%  overhead  [kernel.kallsyms]  [k] context_tracking_user_enter
            |
            --- context_tracking_user_enter
               |
               |--50.00%-- syscall_trace_leave
               |          int_check_syscall_exit_work
               |          |
               |          |--50.00%-- mmap64
               |          |
               |           --50.00%-- open64
               |                     _dl_map_object
               |
               |--25.00%-- do_page_fault
               |          page_fault
               |          __cxa_atexit
               |          0x41d589495541f689
               |
                --25.00%-- do_notify_resume
                          int_signal
                          munmap
                          _dl_sysdep_start
```