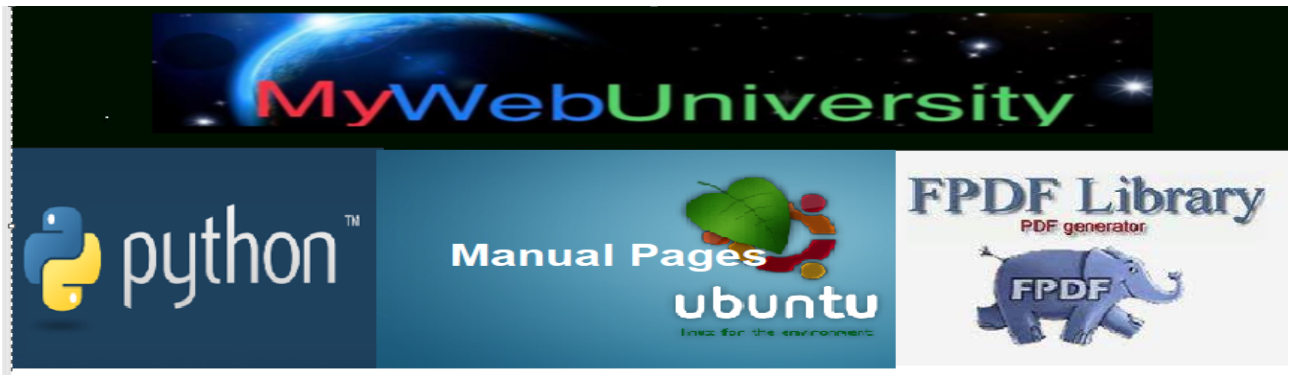

Full credit is given to the above companies including the OS that this PDF file was generated!

Linux Ubuntu 26.04 Manual Pages on command 'perf-report.1'

$ man perf-report.1

PERF-REPORT(1) perf Manual PERF-REPORT(1)

NAME

perf-report - Read perf.data (created by perf record) and display the profile

SYNOPSIS

perf report [-i <file> | --input=file]

DESCRIPTION

This command displays the performance counter profile information recorded via perf record.

OPTIONS

-i, --input=

Input file name. (default: perf.data unless stdin is a fifo)

-v, --verbose

Be more verbose. (show symbol address, etc)

-q, --quiet

Do not show any warnings or messages. (Suppress -v)

-n, --show-nr-samples

Show the number of samples for each symbol

--show-cpu-utilization

Show sample percentage for different cpu modes.

-T, --threads

Show per-thread event counters. The input data file should be recorded with -s option.

-c, --comms=

Only consider symbols in these comms. CSV that understands file://filename entries. This

option will affect the percentage of the overhead and latency columns. See --percentage

for more info.

--pid=

Only show events for given process ID (comma separated list).

--tid=

Only show events for given thread ID (comma separated list).

-d, --dsos=

Only consider symbols in these dsos. CSV that understands file://filename entries. This option will affect the percentage of the overhead and latency columns. See --percentage for more info.

-S, --symbols=

Only consider these symbols. CSV that understands file://filename entries. This option will affect the percentage of the overhead and latency columns. See --percentage for more info.

--symbol-filter=

Only show symbols that match (partially) with this filter.

-U, --hide-unresolved

Only display entries resolved to a symbol.

--parallelism

Only consider these parallelism levels. Parallelism level is the number of threads that actively run on CPUs at the time of sample. The flag accepts single number, comma-separated list, and ranges (for example: "1", "7,8", "1,64-128"). This is useful in understanding what a program is doing during sequential/low-parallelism phases as compared to high-parallelism phases. This option will affect the percentage of the overhead and latency columns. See --percentage for more info. Also see the ?CPU and latency overheads? section for more details.

--latency

Show latency-centric profile rather than the default CPU-consumption-centric profile (requires perf record --latency flag).

-s, --sort=

Sort histogram entries by given key(s) - multiple keys can be specified in CSV format. Following sort keys are available: pid, comm, dso, symbol, parent, cpu, socket, srcline, weight, local_weight, cgroup_id, addr.

Each key has following meaning:

? comm: command (name) of the task which can be read via /proc/<pid>/comm

? pid: command and tid of the task

? tgid: command and tgid of the task

? dso: name of library or module executed at the time of sample

? dso_size: size of library or module executed at the time of sample

? symbol: name of function executed at the time of sample

? symbol_size: size of function executed at the time of sample

? parent: name of function matched to the parent regex filter. Unmatched entries are displayed as "[other]".

? cpu: cpu number the task ran at the time of sample

? socket: processor socket number the task ran at the time of sample

? parallelism: number of running threads at the time of sample

? srcline: filename and line number executed at the time of sample. The DWARF debugging info must be provided.

? srcfile: file name of the source file of the samples. Requires dwarf information.

? weight: Event specific weight, e.g. memory latency or transaction abort cost. This is the global weight.

? local_weight: Local weight version of the weight above.

? cgroup_id: ID derived from cgroup namespace device and inode numbers.

? cgroup: cgroup pathname in the cgroupfs.

? transaction: Transaction abort flags.

? overhead: CPU overhead percentage of sample.

? latency: latency (wall-clock) overhead percentage of sample. See the ?CPU and latency overheads? section for more details.

? overhead_sys: CPU overhead percentage of sample running in system mode

? overhead_us: CPU overhead percentage of sample running in user mode

? overhead_guest_sys: CPU overhead percentage of sample running in system mode on guest machine

? overhead_guest_us: CPU overhead percentage of sample running in user mode on guest machine

? sample: Number of sample

? period: Raw number of event count of sample

? time: Separate the samples by time stamp with the resolution specified by

--time-quantum (default 100ms). Specify with overhead and before it.

? code_page_size: the code page size of sampled code address (ip)

? ins_lat: Instruction latency in core cycles. This is the global instruction latency

? local_ins_lat: Local instruction latency version

? p_stage_cyc: On powerpc, this presents the number of cycles spent in a pipeline stage. And currently supported only on powerpc.

? addr: (Full) virtual address of the sampled instruction

? retire_lat: On X86, this reports pipeline stall of this instruction compared to the previous instruction in cycles. And currently supported only on X86

? simd: Flags describing a SIMD operation. "e" for empty Arm SVE predicate. "p" for partial Arm SVE predicate

? type: Data type of sample memory access.

? typeoff: Offset in the data type of sample memory access.

? symoff: Offset in the symbol.

? weight1: Average value of event specific weight (1st field of weight_struct).

? weight2: Average value of event specific weight (2nd field of weight_struct).

? weight3: Average value of event specific weight (3rd field of weight_struct).

By default, overhead, comm, dso and symbol keys are used.

(i.e. --sort overhead,comm,dso,symbol).

If --branch-stack option is used, following sort keys are also available:

? dso_from: name of library or module branched from

? dso_to: name of library or module branched to

? symbol_from: name of function branched from

? symbol_to: name of function branched to

? srcline_from: source file and line branched from

? srcline_to: source file and line branched to

? mispredict: "N" for predicted branch, "Y" for mispredicted branch

? in_tx: branch in TSX transaction

? abort: TSX transaction abort.

? cycles: Cycles in basic block

And default sort keys are changed to comm, dso_from, symbol_from, dso_to and symbol_to, see '--branch-stack'.

When the sort key symbol is specified, columns "IPC" and "IPC Coverage"
are enabled automatically. Column "IPC" reports the average IPC per function
and column "IPC coverage" reports the percentage of instructions with
sampled IPC in this function. IPC means Instruction Per Cycle. If it's low,
it indicates there may be a performance bottleneck when the function is
executed, such as a memory access bottleneck. If a function has high overhead
and low IPC, it's worth further analyzing it to optimize its performance.
If the --mem-mode option is used, the following sort keys are also available
(incompatible with --branch-stack):
symbol_daddr, dso_daddr, locked, tlb, mem, snoop, dcacheline, blocked.

? symbol_daddr: name of data symbol being executed on at the time of sample

? dso_daddr: name of library or module containing the data being executed on at the
time of the sample

? locked: whether the bus was locked at the time of the sample

? tlb: type of tlb access for the data at the time of the sample

? mem: type of memory access for the data at the time of the sample

? snoop: type of snoop (if any) for the data at the time of the sample

? dcacheline: the cacheline the data address is on at the time of the sample

? phys_daddr: physical address of data being executed on at the time of sample

? data_page_size: the data page size of data being executed on at the time of sample

? blocked: reason of blocked load access for the data at the time of the sample

And the default sort keys are changed to local_weight, mem, sym, dso,
symbol_daddr, dso_daddr, snoop, tlb, locked, blocked, local_ins_lat,
see '--mem-mode'.
If the data file has tracepoint event(s), following (dynamic) sort keys
are also available:
trace, trace_fields, [<event>.]<field>[/raw]

? trace: pretty printed trace output in a single column

? trace_fields: fields in tracepoints in separate columns

? <field name>: optional event and field name for a specific field

The last form consists of event and field names. If event name is
omitted, it searches all events for matching field name. The matched
field will be shown only for the event has the field. The event name

supports substring match so user doesn't need to specify full subsystem
and event name everytime. For example, 'sched:sched_switch' event can
be shortened to 'switch' as long as it's not ambiguous. Also event can
be specified by its index (starting from 1) preceded by the '%'.
So '%1' is the first event, '%2' is the second, and so on.
The field name can have '/raw' suffix which disables pretty printing
and shows raw field value like hex numbers. The --raw-trace option
has the same effect for all dynamic sort keys.
The default sort keys are changed to 'trace' if all events in the data
file are tracepoint.

-F, --fields=

Specify output field - multiple keys can be specified in CSV format. Following fields are available: overhead, latency, overhead_sys, overhead_us, overhead_children, sample, period, weight1, weight2, weight3, ins_lat, p_stage_cyc and retire_lat. The last 3 names are alias for the corresponding weights. When the weight fields are used, they will show the average value of the weight.

Also it can contain any sort key(s).
By default, every sort keys not specified in -F will be appended
automatically.
If the keys starts with a prefix '+', then it will append the specified
field(s) to the default field order. For example: perf report -F +period,sample.

-p, --parent=<regex>

A regex filter to identify parent. The parent is a caller of this function and searched through the callchain, thus it requires callchain information recorded. The pattern is in the extended regex format and defaults to "^sys_|^do_page_fault", see --sort parent.

-x, --exclude-other

Only display entries with parent-match.

-w, --column-widths=<width[,width...]>

Force each column width to the provided list, for large terminal readability. 0 means no limit (default behavior).

-t, --field-separator=

Use a special separator character and don?t pad with spaces, replacing all occurrences of this separator in symbol names (and other output) with a . character, that thus it?s

the only non valid separator.

-D, --dump-raw-trace

Dump raw trace in ASCII.

--disable-order

Disable raw trace ordering.

-g, --call-graph=<print_type,threshold[,print_limit],order,sort_key[,branch],value>

Display call chains using type, min percent threshold, print limit, call order, sort key, optional branch and value. Note that ordering is not fixed so any parameter can be given in an arbitrary order. One exception is the print_limit which should be preceded by threshold.

print_type can be either:

- flat: single column, linear exposure of call chains.
- graph: use a graph tree, displaying absolute overhead rates. (default)
- fractal: like graph, but displays relative rates. Each branch of the tree is considered as a new profiled object.
- folded: call chains are displayed in a line, separated by semicolons
- none: disable call chain display.

threshold is a percentage value which specifies a minimum percent to be included in the output call graph. Default is 0.5 (%).

print_limit is only applied when stdio interface is used. It's to limit number of call graph entries in a single hist entry. Note that it needs to be given after threshold (but not necessarily consecutive).

Default is 0 (unlimited).

order can be either:

- callee: callee based call graph.
- caller: inverted caller based call graph.

Default is 'caller' when --children is used, otherwise 'callee'.

sort_key can be:

- function: compare on functions (default)
- address: compare on individual code addresses
- srcline: compare on source filename and line number

branch can be:

- branch: include last branch information in callgraph when available.

Usually more convenient to use --branch-history for this.

value can be:

- percent: display overhead percent (default)

- period: display event period

- count: display event count

--children

Accumulate callchain of children to parent entry so that then can show up in the output. The output will have a new "Children" column and will be sorted on the data. It requires callchains are recorded. See the ?Overhead calculation? section for more details. Enabled by default, disable with --no-children.

--max-stack

Set the stack depth limit when parsing the callchain, anything beyond the specified depth will be ignored. This is a trade-off between information loss and faster processing especially for workloads that can have a very long callchain stack. Note that when using the --itrace option the synthesized callchain size will override this value if the synthesized callchain size is bigger.

Default: 127

-G, --inverted

alias for inverted caller based call graph.

--ignore-callees=<regex>

Ignore callees of the function(s) matching the given regex. This has the effect of collecting the callers of each such function into one place in the call-graph tree.

--pretty=<key>

Pretty printing style. key: normal, raw

--stdio

Use the stdio interface.

--stdio-color

always, never or auto, allowing configuring color output via the command line, in addition to via "color.ui" .perfconfig. Use --stdio-color always to generate color even when redirecting to a pipe or file. Using just --stdio-color is equivalent to using always.

--tui

Use the TUI interface, that is integrated with annotate and allows zooming into DSOs or

threads, among other features. Use of --tui requires a tty, if one is not present, as

when piping to other commands, the stdio interface is used.

--gtk

Use the GTK2 interface.

-k, --vmlinux=<file>

vmlinux pathname

--ignore-vmlinux

Ignore vmlinux files.

--kallsyms=<file>

kallsyms pathname

-m, --modules

Load module symbols. WARNING: This should only be used with -k and a LIVE kernel.

-f, --force

Don?t do ownership validation.

--symfs=<directory>

Look for files with symbols relative to this directory.

-C, --cpu

Only report samples for the list of CPUs provided. Multiple CPUs can be provided as a

comma-separated list with no space: 0,1. Ranges of CPUs are specified with -: 0-2.

Default is to report samples on all CPUs.

-M, --disassembler-style=

Set disassembler style for objdump.

--source

Interleave source code with assembly code. Enabled by default, disable with --no-source.

--asm-raw

Show raw instruction encoding of assembly instructions.

--show-total-period

Show a column with the sum of periods.

-I, --show-info

Display extended information about the perf.data file. This adds information which may

be very large and thus may clutter the display. It currently includes: cpu and numa

topology of the host system.

-b, --branch-stack

Use the addresses of sampled taken branches instead of the instruction address to build the histograms. To generate meaningful output, the perf.data file must have been obtained using perf record -b or perf record --branch-filter xxx where xxx is a branch filter option. perf report is able to auto-detect whether a perf.data file contains branch stacks and it will automatically switch to the branch view mode, unless --no-branch-stack is used.

--branch-history

Add the addresses of sampled taken branches to the callstack. This allows to examine the path the program took to each sample. The data collection must have used -b (or -j) and -g.

Also show with some branch flags that can be:

- Predicted: display the average percentage of predicated branches.

(predicated number / total number)

- Abort: display the number of tsx aborted branches.

- Cycles: cycles in basic block.

? iterations: display the average number of iterations in callchain list.

--addr2line=<path>

Path to addr2line binary.

--objdump=<path>

Path to objdump binary.

--prefix=PREFIX, --prefix-strip=N

Remove first N entries from source file path names in executables and add PREFIX. This allows to display source code compiled on systems with different file system layout.

--group

Show event group information together. It forces group output also if there are no groups defined in data file.

--group-sort-idx

Sort the output by the event at the index n in group. If n is invalid, sort by the first event. It can support multiple groups with different amount of events. WARNING: This should be used on grouped events.

--demangle

Demangle symbol names to human readable form. It?s enabled by default, disable with --no-demangle.

--demangle-kernel

Demangle kernel symbol names to human readable form (for C++ kernels).

--mem-mode

Use the data addresses of samples in addition to instruction addresses to build the histograms. To generate meaningful output, the perf.data file must have been obtained using perf record -d -W and using a special event -e cpu/mem-loads/p or -e cpu/mem-stores/p. See perf mem for simpler access.

--percent-limit

Do not show entries which have an overhead under that percent. (Default: 0). Note that this option also sets the percent limit (threshold) of callchains. However the default value of callchain threshold is different than the default value of hist entries. Please see the --call-graph option for details.

--percentage

Determine how to display the CPU and latency overhead percentage of filtered entries. Filters can be applied by --comms, --dsos, --symbols and/or --parallelism options and Zoom operations on the TUI (thread, dso, etc).

"relative" means it's relative to filtered entries only so that the sum of shown entries will be always 100%. "absolute" means it retains the original value before and after the filter is applied.

--header

Show header information in the perf.data file. This includes various information like hostname, OS and perf version, cpu/mem info, perf command line, event list and so on. Currently only --stdio output supports this feature.

--header-only

Show only perf.data header (forces --stdio).

--time

Only analyze samples within given time window: <start>,<stop>. Times have the format seconds.nanoseconds. If start is not given (i.e. time string is ,x.y) then analysis starts at the beginning of the file. If stop time is not given (i.e. time string is x.y,) then analysis goes to end of file. Multiple ranges can be separated by spaces, which requires the argument to be quoted e.g. --time "1234.567,1234.789 1235,"

Also support time percent with multiple time ranges. Time string is 'a%/n,b%/m,...' or 'a%-b%,c%-%d,...'.

For example:

Select the second 10% time slice:

perf report --time 10%/2

Select from 0% to 10% time slice:

perf report --time 0%-10%

Select the first and second 10% time slices:

perf report --time 10%/1,10%/2

Select from 0% to 10% and 30% to 40% slices:

perf report --time 0%-10%,30%-40%

--switch-on EVENT_NAME

Only consider events after this event is found.

This may be interesting to measure a workload only after some initialization

phase is over, i.e. insert a perf probe at that point and then using this

option with that probe.

--switch-off EVENT_NAME

Stop considering events after this event is found.

--show-on-off-events

Show the --switch-on/off events too. This has no effect in perf report now but probably

we?ll make the default not to show the switch-on/off events on the --group mode and if

there is only one event besides the off/on ones, go straight to the histogram browser,

just like perf report with no events explicitly specified does.

--itrace

Options for decoding instruction tracing data. The options are:

i synthesize instructions events

y synthesize cycles events

b synthesize branches events

c synthesize branches events (calls only)

r synthesize branches events (returns only)

x synthesize transactions events

w synthesize ptwrite events

p synthesize power events (incl. PSB events for Intel PT)

o synthesize other events recorded due to the use

of aux-output (refer to perf record)

I synthesize interrupt or similar (asynchronous) events
(e.g. Intel PT Event Trace)
e synthesize error events
d create a debug log
f synthesize first level cache events
m synthesize last level cache events
M synthesize memory events
t synthesize TLB events
a synthesize remote access events
g synthesize a call chain (use with i or x)
G synthesize a call chain on existing event records
l synthesize last branch entries (use with i or x)
L synthesize last branch entries on existing event records
s skip initial number of events
q quicker (less detailed) decoding
A approximate IPC
Z prefer to ignore timestamps (so-called "timeless" decoding)
T use the timestamp trace as kernel time
The default is all events i.e. the same as --itrace=iybxwpe,
except for perf script where it is --itrace=ce
In addition, the period (default 100000, except for perf script where it is 1)
for instructions events can be specified in units of:
i instructions
t ticks
ms milliseconds
us microseconds
ns nanoseconds (default)
Also the call chain size (default 16, max. 1024) for instructions or
transactions events can be specified.
Also the number of last branch entries (default 64, max. 1024) for
instructions or transactions events can be specified.
Similar to options g and l, size may also be specified for options G and L.
On x86, note that G and L work poorly when data has been recorded with

large PEBS. Refer linkperf:perf-intel-pt[1] man page for details.

It is also possible to skip events generated (instructions, branches, transactions, ptwrite, power) at the beginning. This is useful to ignore initialization code.

--itrace=i0nss1000000

skips the first million instructions.

The 'e' option may be followed by flags which affect what errors will or will not be reported. Each flag must be preceded by either '+' or '-'.

The flags are:

- o overflow
- l trace data lost

If supported, the 'd' option may be followed by flags which affect what debug messages will or will not be logged. Each flag must be preceded by either '+' or '-'. The flags are:

- a all perf events
- e output only on errors (size configurable - see linkperf:perf-config[1])
- o output to stdout

If supported, the 'q' option may be repeated to increase the effect.

To disable decoding entirely, use --no-itrace.

--full-source-path

Show the full path for source files for srcline output.

--show-ref-call-graph

When multiple events are sampled, it may not be needed to collect callgraphs for all of them. The sample sites are usually nearby, and it?s enough to collect the callgraphs on a reference event. So user can use "call-graph=no" event modifier to disable callgraph for other events to reduce the overhead. However, perf report cannot show callgraphs for the event which disable the callgraph. This option extends the perf report to show reference callgraphs, which collected by reference event, in no callgraph event.

--stitch-lbr

Show callgraph with stitched LBRs, which may have more complete callgraph. The perf.data file must have been obtained using perf record --call-graph lbr. Disabled by default. In common cases with call stack overflows, it can recreate better call stacks than the default lbr call stack output. But this approach is not foolproof. There can be cases where it creates incorrect call stacks from incorrect matches. The known limitations

include exception handing such as setjmp/longjmp will have calls/returns not match.

--socket-filter

Only report the samples on the processor socket that match with this filter

--samples=N

Save N individual samples for each histogram entry to show context in perf report tui browser.

--raw-trace

When displaying traceevent output, do not use print fmt or plugins.

-H, --hierarchy

Enable hierarchical output. In the hierarchy mode, each sort key groups samples based on the criteria and then sub-divide it using the lower level sort key.

For example:

In normal output:

```
perf report -s dso,sym
# Overhead  Shared Object      Symbol
   50.00%  [kernel.kallsyms]  [k] kfunc1
   20.00%  perf               [.] foo
   15.00%  [kernel.kallsyms]  [k] kfunc2
   10.00%  perf               [.] bar
    5.00%  libc.so            [.] libcall
```

In hierarchy output:

```
perf report -s dso,sym --hierarchy
#   Overhead  Shared Object / Symbol
   65.00%    [kernel.kallsyms]
     50.00%    [k] kfunc1
     15.00%    [k] kfunc2
   30.00%    perf
     20.00%    [.] foo
     10.00%    [.] bar
    5.00%    libc.so
     5.00%    [.] libcall
```

--inline

If a callgraph address belongs to an inlined function, the inline stack will be printed.

Each entry is function name or file/line. Enabled by default, disable with --no-inline.

--mmaps

Show --tasks output plus mmap information in a format similar to /proc/<PID>/maps.

Please note that not all mmaps are stored, options affecting which ones

are include 'perf record --data', for instance.

--ns

Show time stamps in nanoseconds.

--stats

Display overall events statistics without any further processing. (like the one at the

end of the perf report -D command)

--tasks

Display monitored tasks stored in perf data. Displaying pid/tid/ppid plus the command

string aligned to distinguish parent and child tasks.

--percent-type

Set annotation percent type from following choices: global-period, local-period,

global-hits, local-hits

The local/global keywords set if the percentage is computed

in the scope of the function (local) or the whole data (global).

The period/hits keywords set the base the percentage is computed

on - the samples period or the number of samples (hits).

--time-quantum

Configure time quantum for time sort key. Default 100ms. Accepts s, us, ms, ns units.

--total-cycles

When --total-cycles is specified, it supports sorting for all blocks by Sampled Cycles%.

This is useful to concentrate on the globally hottest blocks. In output, there are some

new columns:

'Sampled Cycles%' - block sampled cycles aggregation / total sampled cycles

'Sampled Cycles' - block sampled cycles aggregation

'Avg Cycles%' - block average sampled cycles / sum of total block average

sampled cycles

'Avg Cycles' - block average sampled cycles

'Branch Counter' - block branch counter histogram (with -v showing the number)

--skip-empty

Do not print 0 results in the --stat output.

CPU AND LATENCY OVERHEADS

There are two notions of time: wall-clock time and CPU time. For a single-threaded program, or a program running on a single-core machine, these notions are the same. However, for a multi-threaded/multi-process program running on a multi-core machine, these notions are significantly different. Each second of wall-clock time we have number-of-cores seconds of CPU time. Perf can measure overhead for both of these times (shown in overhead and latency columns for CPU and wall-clock time correspondingly).

Optimizing CPU overhead is useful to improve throughput, while optimizing latency overhead is useful to improve latency. It?s important to understand which one is useful in a concrete situation at hand. For example, the former may be useful to improve max throughput of a CI build server that runs on 100% CPU utilization, while the latter may be useful to improve user-perceived latency of a single interactive program build. These overheads may be significantly different in some cases. For example, consider a program that executes function foo for 9 seconds with 1 thread, and then executes function bar for 1 second with 128 threads (consumes 128 seconds of CPU time). The CPU overhead is: foo - 6.6%, bar - 93.4%. While the latency overhead is: foo - 90%, bar - 10%. If we try to optimize running time of the program looking at the (wrong in this case) CPU overhead, we would concentrate on the function bar, but it can yield only 10% running time improvement at best.

By default, perf shows only CPU overhead. To show latency overhead, use perf record --latency and perf report:

.ft C

```
Overhead  Latency  Command
  93.88%   25.79%  cc1
   1.90%   39.87%  gzip
   0.99%   10.16%  dpkg-deb
   0.57%    1.00%  as
   0.40%    0.46%  sh
```

.ft

To sort by latency overhead, use perf report --latency:

.ft C

```
Latency  Overhead  Command
 39.87%     1.90%  gzip
```

```
    25.79%    93.88%  cc1
    10.16%     0.99%  dpkg-deb
     4.17%     0.29%  git
     2.81%     0.11%  objtool
```

To get insight into the difference between the overheads, you may check parallelization histogram with --sort=latency,parallelism,comm,symbol --hierarchy flags. It shows fraction of (wall-clock) time the workload utilizes different numbers of cores (Parallelism column). For example, in the following case the workload utilizes only 1 core most of the time, but also has some highly-parallel phases, which explains significant difference between CPU and wall-clock overheads:

```
  Latency  Overhead     Parallelism / Command / Symbol
+  56.98%     2.29%     1
+  16.94%     1.36%     2
+   4.00%    20.13%     125
+   3.66%    18.25%     124
+   3.48%    17.66%     126
+   3.26%     0.39%     3
+   2.61%    12.93%     123
```

By expanding corresponding lines, you may see what commands/functions run at the given parallelism level:

```
  Latency  Overhead     Parallelism / Command / Symbol
-  56.98%     2.29%     1
      32.80%     1.32%     gzip
       4.46%     0.18%     cc1
       2.81%     0.11%     objtool
       2.43%     0.10%     dpkg-source
       2.22%     0.09%     ld
       2.10%     0.08%     dpkg-genchanges
```

To see the normal function-level profile for particular parallelism levels (number of threads actively running on CPUs), you may use --parallelism filter. For example, to see the profile only for low parallelism phases of a workload use --latency --parallelism=1-2 flags.

OVERHEAD CALCULATION

The CPU overhead can be shown in two columns as Children and Self when perf collects callchains (and corresponding Wall columns for wall-clock overhead). 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; there are three functions like below.

```
.ft C
void foo(void) {
    /* do something */
}
void bar(void) {
    /* do something */
    foo();
}
int main(void) {
    bar()
    return 0;
}
.ft
```

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:

```
.ft C
Overhead  Symbol
........  ......................
  60.00%  foo
          |
          --- foo
              bar
              main
              __libc_start_main
  40.00%  bar
          |
          --- bar
              main
              __libc_start_main
.ft
```

When the --children option is enabled, the self overhead values of child functions (i.e. foo and bar) are added to the parents to calculate the children overhead. In this case the report could be displayed as:

```
.ft C
Children      Self  Symbol
........  ........  ......................
 100.00%     0.00%  __libc_start_main
          |
          --- __libc_start_main
 100.00%     0.00%  main
          |
          --- main
              __libc_start_main
 100.00%    40.00%  bar
          |
```

```
            --- bar
                main
                __libc_start_main
       60.00%    60.00%  foo
            |
            --- foo
                bar
                main
                __libc_start_main
    .ft
```

In the above output, the self overhead of foo (60%) was add to the children overhead of bar, main and __libc_start_main. Likewise, the self overhead of bar (40%) was added to the children overhead of main and __libc_start_main.

So __libc_start_main and main are shown first since they have same (100%) children overhead (even though they have zero self overhead) and they are the parents of foo and bar.

Since v3.16 the children overhead is shown by default and the output is sorted by its values. The children overhead is disabled by specifying --no-children option on the command line or by adding report.children = false or top.children = false in the perf config file.

SEE ALSO

perf-stat(1), perf-annotate(1), perf-record(1), perf-intel-pt(1)