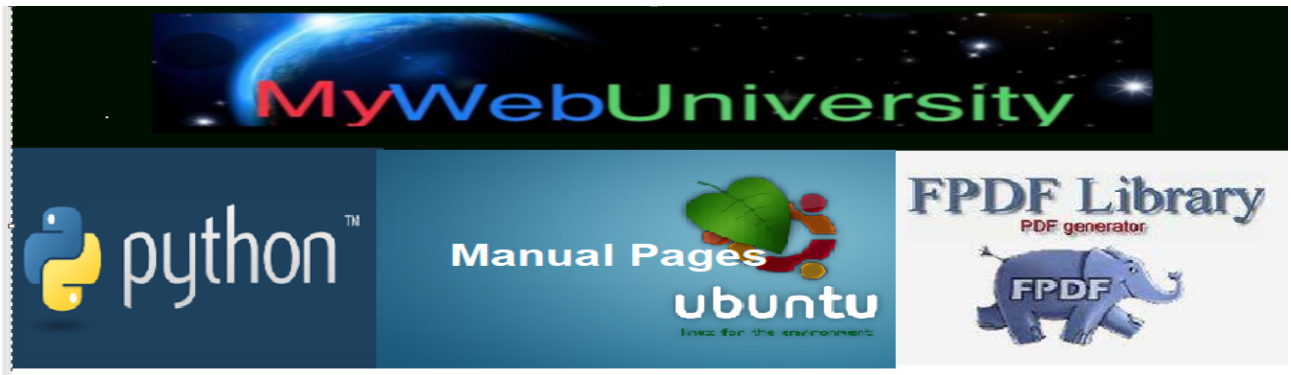

Linux Ubuntu 26.04 Manual Pages on command 'perf-intel-pt.1'

$ man perf-intel-pt.1

PERF-INTEL-PT(1) perf Manual PERF-INTEL-PT(1)

NAME

perf-intel-pt - Support for Intel Processor Trace within perf tools

SYNOPSIS

perf record -e intel_pt//

DESCRIPTION

Intel Processor Trace (Intel PT) is an extension of Intel Architecture that collects information about software execution such as control flow, execution modes and timings and formats it into highly compressed binary packets. Technical details are documented in the Intel 64 and IA-32 Architectures Software Developer Manuals, Chapter 36 Intel Processor Trace.

Intel PT is first supported in Intel Core M and 5th generation Intel Core processors that are based on the Intel micro-architecture code name Broadwell.

Trace data is collected by perf record and stored within the perf.data file. See below for options to perf record.

Trace data must be decoded which involves walking the object code and matching the trace data packets. For example a TNT packet only tells whether a conditional branch was taken or not taken, so to make use of that packet the decoder must know precisely which instruction was being executed.

Decoding is done on-the-fly. The decoder outputs samples in the same format as samples output by perf hardware events, for example as though the "instructions" or "branches" events had been recorded. Presently 3 tools support this: perf script, perf report and perf inject. See below for more information on using those tools.

The main distinguishing feature of Intel PT is that the decoder can determine the exact flow of software execution. Intel PT can be used to understand why and how did software get to a certain point, or behave a certain way. The software does not have to be recompiled, so Intel PT works with debug or release builds, however the executed images are needed - which makes use in JIT-compiled environments, or with self-modified code, a challenge. Also symbols need to be provided to make sense of addresses.

A limitation of Intel PT is that it produces huge amounts of trace data (hundreds of megabytes per second per core) which takes a long time to decode, for example two or three orders of magnitude longer than it took to collect. Another limitation is the performance impact of tracing, something that will vary depending on the use-case and architecture.

QUICKSTART

It is important to start small. That is because it is easy to capture vastly more data than can possibly be processed.

The simplest thing to do with Intel PT is userspace profiling of small programs. Data is captured with perf record e.g. to trace ls userspace-only:

```
perf record -e intel_pt//u ls
```

And profiled with perf report e.g.

```
perf report
```

To also trace kernel space presents a problem, namely kernel self-modifying code. A fairly good kernel image is available in /proc/kcore but to get an accurate image a copy of /proc/kcore needs to be made under the same conditions as the data capture. perf record can make a copy of /proc/kcore if the option --kcore is used, but access to /proc/kcore is restricted e.g.

```
sudo perf record -o pt_ls --kcore -e intel_pt// -- ls
```

which will create a directory named pt_ls and put the perf.data file (named simply data) and copies of /proc/kcore, /proc/kallsyms and /proc/modules into it. The other tools understand the directory format, so to use perf report becomes:

```
sudo perf report -i pt_ls
```

Because samples are synthesized after-the-fact, the sampling period can be selected for reporting. e.g. sample every microsecond

```
sudo perf report pt_ls --itrace=i1usge
```

See the sections below for more information about the --itrace option.

Beware the smaller the period, the more samples that are produced, and the longer it takes

to process them.

Also note that the coarseness of Intel PT timing information will start to distort the statistical value of the sampling as the sampling period becomes smaller.

To represent software control flow, "branches" samples are produced. By default a branch sample is synthesized for every single branch. To get an idea what data is available you can use the perf script tool with all itrace sampling options, which will list all the samples.

```
perf record -e intel_pt//u ls
perf script --itrace=iybxwpe
```

An interesting field that is not printed by default is flags which can be displayed as follows:

```
perf script --itrace=iybxwpe -F+flags
```

The flags are "bcrosyiABExghDt" which stand for branch, call, return, conditional, system, asynchronous, interrupt, transaction abort, trace begin, trace end, in transaction, VM-entry, VM-exit, interrupt disabled, and interrupt disable toggle respectively.

perf script also supports higher level ways to dump instruction traces:

```
perf script --insn-trace=disasm
```

or to use the xed disassembler, which requires installing the xed tool (see XED below):

```
perf script --insn-trace --xed
```

Dumping all instructions in a long trace can be fairly slow. It is usually better to start with higher level decoding, like

```
perf script --call-trace
```

or

```
perf script --call-ret-trace
```

and then select a time range of interest. The time range can then be examined in detail with

```
perf script --time starttime,stoptime --insn-trace=disasm
```

While examining the trace it?s also useful to filter on specific CPUs using the -C option

```
perf script --time starttime,stoptime --insn-trace=disasm -C 1
```

Dump all instructions in time range on CPU 1.

Another interesting field that is not printed by default is ipc which can be displayed as follows:

```
perf script --itrace=be -F+ipc
```

There are two ways that instructions-per-cycle (IPC) can be calculated depending on the recording.

If the cyc config term (see config terms section below) was used, then IPC and cycle events are calculated using the cycle count from CYC packets, otherwise MTC packets are used - refer to the mtc config term. When MTC is used, however, the values are less accurate because the timing is less accurate.

Because Intel PT does not update the cycle count on every branch or instruction, the values will often be zero. When there are values, they will be the number of instructions and number of cycles since the last update, and thus represent the average IPC cycle count since the last IPC for that event type. Note IPC for "branches" events is calculated separately from IPC for "instructions" events.

Even with the cyc config term, it is possible to produce IPC information for every change of timestamp, but at the expense of accuracy. That is selected by specifying the itrace A option. Due to the granularity of timestamps, the actual number of cycles increases even though the cycles reported does not. The number of instructions is known, but if IPC is reported, cycles can be too low and so IPC is too high. Note that inaccuracy decreases as the period of sampling increases i.e. if the number of cycles is too low by a small amount, that becomes less significant if the number of cycles is large. It may also be useful to use the A option in conjunction with dlfilter-show-cycles.so to provide higher granularity cycle information.

Also note that the IPC instruction count may or may not include the current instruction. If the cycle count is associated with an asynchronous branch (e.g. page fault or interrupt), then the instruction count does not include the current instruction, otherwise it does. That is consistent with whether or not that instruction has retired when the cycle count is updated.

Another note, in the case of "branches" events, non-taken branches are not presently sampled, so IPC values for them do not appear e.g. a CYC packet with a TNT packet that starts with a non-taken branch. To see every possible IPC value, "instructions" events can be used e.g. --itrace=i0ns

While it is possible to create scripts to analyze the data, an alternative approach is available to export the data to a sqlite or postgresql database. Refer to script export-to-sqlite.py or export-to-postgresql.py for more details, and to script exported-sql-viewer.py for an example of using the database.

There is also script intel-pt-events.py which provides an example of how to unpack the raw data for power events and PTWRITE. The script also displays branches, and supports 2

additional modes selected by option:

? --insn-trace - instruction trace

? --src-trace - source trace

The intel-pt-events.py script also has options:

? --all-switch-events - display all switch events, not only the last consecutive.

? --interleave [<n>] - interleave sample output for the same timestamp so that no more than n samples for a CPU are displayed in a row. n defaults to 4. Note this only affects the order of output, and only when the timestamp is the same.

As mentioned above, it is easy to capture too much data. One way to limit the data captured is to use snapshot mode which is explained further below. Refer to new snapshot option and Intel PT modes of operation further below.

Another problem that will be experienced is decoder errors. They can be caused by inability to access the executed image, self-modified or JIT-ed code, or the inability to match side-band information (such as context switches and mmaps) which results in the decoder not knowing what code was executed.

There is also the problem of perf not being able to copy the data fast enough, resulting in data lost because the buffer was full. See Buffer handling below for more details.

PERF RECORD

new event

The Intel PT kernel driver creates a new PMU for Intel PT. PMU events are selected by providing the PMU name followed by the "config" separated by slashes. An enhancement has been made to allow default "config" e.g. the option

```
-e intel_pt//
```

will use a default config value. Currently that is the same as

```
-e intel_pt/tsc,noretcomp=0/
```

which is the same as

```
-e intel_pt/tsc=1,noretcomp=0/
```

Note there are other config terms - see section config terms further below.

The config terms are listed in /sys/devices/intel_pt/format. They are bit fields within the config member of the struct perf_event_attr which is passed to the kernel by the perf_event_open system call. They correspond to bit fields in the IA32_RTIT_CTL MSR. Here is a list of them and their definitions:

```
$ grep -H . /sys/bus/event_source/devices/intel_pt/format/*
```

/sys/bus/event_source/devices/intel_pt/format/cyc:config:1

/sys/bus/event_source/devices/intel_pt/format/cyc_thresh:config:19-22

/sys/bus/event_source/devices/intel_pt/format/mtc:config:9

/sys/bus/event_source/devices/intel_pt/format/mtc_period:config:14-17

/sys/bus/event_source/devices/intel_pt/format/noretcomp:config:11

/sys/bus/event_source/devices/intel_pt/format/psb_period:config:24-27

/sys/bus/event_source/devices/intel_pt/format/tsc:config:10

Note that the default config must be overridden for each term i.e.

-e intel_pt/noretcomp=0/

is the same as:

-e intel_pt/tsc=1,noretcomp=0/

So, to disable TSC packets use:

-e intel_pt/tsc=0/

It is also possible to specify the config value explicitly:

-e intel_pt/config=0x400/

Note that, as with all events, the event is suffixed with event modifiers:

u userspace

k kernel

h hypervisor

G guest

H host

p precise ip

h, G and H are for virtualization which are not used by Intel PT. p is also not relevant to

Intel PT. So only options u and k are meaningful for Intel PT.

perf_event_attr is displayed if the -vv option is used e.g.

---

perf_event_attr:

type 6

size 112

config 0x400

{ sample_period, sample_freq } 1

sample_type IP|TID|TIME|CPU|IDENTIFIER

read_format ID

```
disabled                 1
inherit                  1
exclude_kernel           1
exclude_hv               1
enable_on_exec           1
sample_id_all            1
------------------------------------------------------------
sys_perf_event_open: pid 31104  cpu 0  group_fd -1  flags 0x8
sys_perf_event_open: pid 31104  cpu 1  group_fd -1  flags 0x8
sys_perf_event_open: pid 31104  cpu 2  group_fd -1  flags 0x8
sys_perf_event_open: pid 31104  cpu 3  group_fd -1  flags 0x8
------------------------------------------------------------
```

config terms

Config terms are parameters specified with the -e intel_pt// event option, for example:

```
-e intel_pt/cyc/
```

which selects cycle accurate mode. Each config term can have a value which defaults to 1, so the above is the same as:

```
-e intel_pt/cyc=1/
```

Some terms are set by default, so must be set to 0 to turn them off. For example, to turn off branch tracing:

```
-e intel_pt/branch=0/
```

Multiple config terms are separated by commas, for example:

```
-e intel_pt/cyc,mtc_period=9/
```

There are also common config terms, see perf-record(1) documentation.

Intel PT config terms are described below.

tsc

Always supported. Produces TSC timestamp packets to provide timing information. In some cases it is possible to decode without timing information, for example a per-thread context that does not overlap executable memory maps.

The default config selects tsc (i.e. tsc=1).

noretcomp

Always supported. Disables "return compression" so a TIP packet is produced when a function returns. Causes more packets to be produced but might make decoding more

reliable.

The default config does not select noretcomp (i.e. noretcomp=0).

psb_period

Allows the frequency of PSB packets to be specified.

The PSB packet is a synchronization packet that provides a starting point for decoding or recovery from errors.

Support for psb_period is indicated by:

/sys/bus/event_source/devices/intel_pt/caps/psb_cyc

which contains "1" if the feature is supported and "0" otherwise.

Valid values are given by:

/sys/bus/event_source/devices/intel_pt/caps/psb_periods

which contains a hexadecimal value, the bits of which represent valid values e.g. bit 2 set means value 2 is valid.

The psb_period value is converted to the approximate number of trace bytes between PSB packets as:

2 ^ (value + 11)

e.g. value 3 means 16KiB bytes between PSBs

If an invalid value is entered, the error message will give a list of valid values e.g.

$ perf record -e intel_pt/psb_period=15/u uname

Invalid psb_period for intel_pt. Valid values are: 0-5

If MTC packets are selected, the default config selects a value of 3 (i.e. psb_period=3) or the nearest lower value that is supported (0 is always supported). Otherwise the default is 0.

If decoding is expected to be reliable and the buffer is large then a large PSB period can be used.

Because a TSC packet is produced with PSB, the PSB period can also affect the granularity to timing information in the absence of MTC or CYC.

mtc

Produces MTC timing packets.

MTC packets provide finer grain timestamp information than TSC packets. MTC packets record time using the hardware crystal clock (CTC) which is related to TSC packets using a TMA packet.

Support for this feature is indicated by:

/sys/bus/event_source/devices/intel_pt/caps/mtc

which contains "1" if the feature is supported and "0" otherwise.

The frequency of MTC packets can also be specified - see mtc_period below.

mtc_period

Specifies how frequently MTC packets are produced - see mtc above for how to determine if MTC packets are supported.

Valid values are given by:

/sys/bus/event_source/devices/intel_pt/caps/mtc_periods

which contains a hexadecimal value, the bits of which represent valid values e.g. bit 2 set means value 2 is valid.

The mtc_period value is converted to the MTC frequency as:

CTC-frequency / (2 ^ value)

e.g. value 3 means one eighth of CTC-frequency

Where CTC is the hardware crystal clock, the frequency of which can be related to TSC via values provided in cpuid leaf 0x15.

If an invalid value is entered, the error message will give a list of valid values e.g.

$ perf record -e intel_pt/mtc_period=15/u uname

Invalid mtc_period for intel_pt. Valid values are: 0,3,6,9

The default value is 3 or the nearest lower value that is supported (0 is always supported).

cyc

Produces CYC timing packets.

CYC packets provide even finer grain timestamp information than MTC and TSC packets. A CYC packet contains the number of CPU cycles since the last CYC packet. Unlike MTC and TSC packets, CYC packets are only sent when another packet is also sent.

Support for this feature is indicated by:

/sys/bus/event_source/devices/intel_pt/caps/psb_cyc

which contains "1" if the feature is supported and "0" otherwise.

The number of CYC packets produced can be reduced by specifying a threshold - see cyc_thresh below.

cyc_thresh

Specifies how frequently CYC packets are produced - see cyc above for how to determine if CYC packets are supported.

Valid cyc_thresh values are given by:

/sys/bus/event_source/devices/intel_pt/caps/cycle_thresholds

which contains a hexadecimal value, the bits of which represent valid values e.g. bit 2 set means value 2 is valid.

The cyc_thresh value represents the minimum number of CPU cycles that must have passed before a CYC packet can be sent. The number of CPU cycles is:

2 ^ (value - 1)

e.g. value 4 means 8 CPU cycles must pass before a CYC packet can be sent. Note a CYC packet is still only sent when another packet is sent, not at, e.g. every 8 CPU cycles.

If an invalid value is entered, the error message will give a list of valid values e.g.

$ perf record -e intel_pt/cyc,cyc_thresh=15/u uname

Invalid cyc_thresh for intel_pt. Valid values are: 0-12

CYC packets are not requested by default.

pt

Specifies pass-through which enables the branch config term.

The default config selects pt if it is available, so a user will never need to specify this term.

branch

Enable branch tracing. Branch tracing is enabled by default so to disable branch tracing use branch=0.

The default config selects branch if it is available.

ptw

Enable PTWRITE packets which are produced when a ptwrite instruction is executed.

Support for this feature is indicated by:

/sys/bus/event_source/devices/intel_pt/caps/ptwrite

which contains "1" if the feature is supported and "0" otherwise.

As an alternative, refer to "Emulated PTWRITE" further below.

fup_on_ptw

Enable a FUP packet to follow the PTWRITE packet. The FUP packet provides the address of the ptwrite instruction. In the absence of fup_on_ptw, the decoder will use the address of the previous branch if branch tracing is enabled, otherwise the address will be zero.

Note that fup_on_ptw will work even when branch tracing is disabled.

pwr_evt

Enable power events. The power events provide information about changes to the CPU C-state.

Support for this feature is indicated by:

/sys/bus/event_source/devices/intel_pt/caps/power_event_trace

which contains "1" if the feature is supported and "0" otherwise.

event

Enable Event Trace. The events provide information about asynchronous events.

Support for this feature is indicated by:

/sys/bus/event_source/devices/intel_pt/caps/event_trace

which contains "1" if the feature is supported and "0" otherwise.

notnt

Disable TNT packets. Without TNT packets, it is not possible to walk executable code to reconstruct control flow, however FUP, TIP, TIP.PGE and TIP.PGD packets still indicate asynchronous control flow, and (if return compression is disabled - see noretcomp) return statements. The advantage of eliminating TNT packets is reducing the size of the trace and corresponding tracing overhead.

Support for this feature is indicated by:

/sys/bus/event_source/devices/intel_pt/caps/tnt_disable

which contains "1" if the feature is supported and "0" otherwise.

aux-action=start-paused

Start tracing paused, refer to the section Pause or Resume Tracing

config terms on other events

Some Intel PT features work with other events, features such as AUX area sampling and PEBS-via-PT. In those cases, the other events can have config terms below:

aux-sample-size

Used to set the AUX area sample size, refer to the section AUX area sampling option

aux-output

Used to select PEBS-via-PT, refer to the section PEBS via Intel PT

aux-action

Used to pause or resume tracing, refer to the section Pause or Resume Tracing

AUX area sampling option

To select Intel PT "sampling" the AUX area sampling option can be used:

--aux-sample

Optionally it can be followed by the sample size in bytes e.g.

--aux-sample=8192

In addition, the Intel PT event to sample must be defined e.g.

-e intel_pt//u

Samples on other events will be created containing Intel PT data e.g. the following will create Intel PT samples on the branch-misses event, note the events must be grouped using {}:

perf record --aux-sample -e '{intel_pt//u,branch-misses:u}'

An alternative to --aux-sample is to add the config term aux-sample-size to events. In this case, the grouping is implied e.g.

perf record -e intel_pt//u -e branch-misses/aux-sample-size=8192/u

is the same as:

perf record -e '{intel_pt//u,branch-misses/aux-sample-size=8192/u}'

but allows for also using an address filter e.g.:

perf record -e intel_pt//u --filter 'filter * @/bin/ls' -e branch-misses/aux-sample-size=8192/u -- ls

It is important to select a sample size that is big enough to contain at least one PSB packet. If not a warning will be displayed:

Intel PT sample size (%zu) may be too small for PSB period (%zu)

The calculation used for that is: if sample_size ? psb_period + 256 display the warning.

When sampling is used, psb_period defaults to 0 (2KiB).

The default sample size is 4KiB.

The sample size is passed in aux_sample_size in struct perf_event_attr. The sample size is limited by the maximum event size which is 64KiB. It is difficult to know how big the event might be without the trace sample attached, but the tool validates that the sample size is not greater than 60KiB.

new snapshot option

The difference between full trace and snapshot from the kernel?s perspective is that in full trace we don?t overwrite trace data that the user hasn?t collected yet (and indicated that by advancing aux_tail), whereas in snapshot mode we let the trace run and overwrite older data in the buffer so that whenever something interesting happens, we can stop it and grab a snapshot of what was going on around that interesting moment.

To select snapshot mode a new option has been added:

-S

Optionally it can be followed by the snapshot size e.g.

-S0x100000

The default snapshot size is the auxtrace mmap size. If neither auxtrace mmap size nor snapshot size is specified, then the default is 4MiB for privileged users (or if /proc/sys/kernel/perf_event_paranoid < 0), 128KiB for unprivileged users. If an unprivileged user does not specify mmap pages, the mmap pages will be reduced as described in the new auxtrace mmap size option section below.

The snapshot size is displayed if the option -vv is used e.g.

Intel PT snapshot size: %zu

new auxtrace mmap size option

Intel PT buffer size is specified by an addition to the -m option e.g.

-m,16

selects a buffer size of 16 pages i.e. 64KiB.

Note that the existing functionality of -m is unchanged. The auxtrace mmap size is specified by the optional addition of a comma and the value.

The default auxtrace mmap size for Intel PT is 4MiB/page_size for privileged users (or if /proc/sys/kernel/perf_event_paranoid < 0), 128KiB for unprivileged users. If an unprivileged user does not specify mmap pages, the mmap pages will be reduced from the default 512KiB/page_size to 256KiB/page_size, otherwise the user is likely to get an error as they exceed their mlock limit (Max locked memory as shown in /proc/self/limits). Note that perf does not count the first 512KiB (actually /proc/sys/kernel/perf_event_mlock_kb minus 1 page) per cpu against the mlock limit so an unprivileged user is allowed 512KiB per cpu plus their mlock limit (which defaults to 64KiB but is not multiplied by the number of cpus).

In full-trace mode, powers of two are allowed for buffer size, with a minimum size of 2 pages. In snapshot mode or sampling mode, it is the same but the minimum size is 1 page.

The mmap size and auxtrace mmap size are displayed if the -vv option is used e.g.

mmap length 528384

auxtrace mmap length 4198400

Intel PT modes of operation

Intel PT can be used in 3 modes: full-trace mode sample mode snapshot mode

Full-trace mode traces continuously e.g.

perf record -e intel_pt//u uname

Sample mode attaches a Intel PT sample to other events e.g.

perf record --aux-sample -e intel_pt//u -e branch-misses:u

Snapshot mode captures the available data when a signal is sent or "snapshot" control command is issued. e.g. using a signal

```
perf record -v -e intel_pt//u -S ./loopy 1000000000 &
[1] 11435
kill -USR2 11435
Recording AUX area tracing snapshot
```

Note that the signal sent is SIGUSR2. Note that "Recording AUX area tracing snapshot" is displayed because the -v option is used.

The advantage of using "snapshot" control command is that the access is controlled by access to a FIFO e.g.

```
$ mkfifo perf.control
$ mkfifo perf.ack
$ cat perf.ack &
[1] 15235
$ sudo ~/bin/perf record --control fifo:perf.control,perf.ack -S -e intel_pt//u -- sleep 60 &
[2] 15243
$ ps -e | grep perf
15244 pts/1    00:00:00 perf
$ kill -USR2 15244
bash: kill: (15244) - Operation not permitted
$ echo snapshot > perf.control
ack
```

The 3 Intel PT modes of operation cannot be used together.

Buffer handling

There may be buffer limitations (i.e. single ToPa entry) which means that actual buffer sizes are limited to powers of 2 up to 4MiB (MAX_PAGE_ORDER). In order to provide other sizes, and in particular an arbitrarily large size, multiple buffers are logically concatenated. However an interrupt must be used to switch between buffers. That has two potential problems: a) the interrupt may not be handled in time so that the current buffer becomes full and some trace data is lost. b) the interrupts may slow the system and affect the performance results.

If trace data is lost, the driver sets truncated in the PERF_RECORD_AUX event which the

tools report as an error.

In full-trace mode, the driver waits for data to be copied out before allowing the (logical) buffer to wrap-around. If data is not copied out quickly enough, again truncated is set in the PERF_RECORD_AUX event. If the driver has to wait, the intel_pt event gets disabled. Because it is difficult to know when that happens, perf tools always re-enable the intel_pt event after copying out data.

Intel PT and build ids

By default "perf record" post-processes the event stream to find all build ids for executables for all addresses sampled. Deliberately, Intel PT is not decoded for that purpose (it would take too long). Instead the build ids for all executables encountered (due to mmap, comm or task events) are included in the perf.data file.

To see buildids included in the perf.data file use the command:

```
perf buildid-list
```

If the perf.data file contains Intel PT data, that is the same as:

```
perf buildid-list --with-hits
```

Snapshot mode and event disabling

In order to make a snapshot, the intel_pt event is disabled using an IOCTL, namely PERF_EVENT_IOC_DISABLE. However doing that can also disable the collection of side-band information. In order to prevent that, a dummy software event has been introduced that permits tracking events (like mmaps) to continue to be recorded while intel_pt is disabled. That is important to ensure there is complete side-band information to allow the decoding of subsequent snapshots.

A test has been created for that. To find the test:

```
perf test list
...
23: Test using a dummy software event to keep tracking
```

To run the test:

```
perf test 23
23: Test using a dummy software event to keep tracking     : Ok
```

perf record modes (nothing new here)

perf record essentially operates in one of three modes: per thread per cpu workload only "per thread" mode is selected by -t or by --per-thread (with -p or -u or just a workload). "per cpu" is selected by -C or -a. "workload only" mode is selected by not using the other

options but providing a command to run (i.e. the workload).

In per-thread mode an exact list of threads is traced. There is no inheritance. Each thread has its own event buffer.

In per-cpu mode all processes (or processes from the selected cgroup i.e. -G option, or processes selected with -p or -u) are traced. Each cpu has its own buffer. Inheritance is allowed.

In workload-only mode, the workload is traced but with per-cpu buffers. Inheritance is allowed. Note that you can now trace a workload in per-thread mode by using the --per-thread option.

Privileged vs non-privileged users

Unless /proc/sys/kernel/perf_event_paranoid is set to -1, unprivileged users have memory limits imposed upon them. That affects what buffer sizes they can have as outlined above.

The v4.2 kernel introduced support for a context switch metadata event, PERF_RECORD_SWITCH, which allows unprivileged users to see when their processes are scheduled out and in, just not by whom, which is left for the PERF_RECORD_SWITCH_CPU_WIDE, that is only accessible in system wide context, which in turn requires CAP_PERFMON or CAP_SYS_ADMIN.

Please see the 45ac1403f564 ("perf: Add PERF_RECORD_SWITCH to indicate context switches") commit, that introduces these metadata events for further info.

When working with kernels < v4.2, the following considerations must be taken, as the sched:sched_switch tracepoints will be used to receive such information:

Unless /proc/sys/kernel/perf_event_paranoid is set to -1, unprivileged users are not permitted to use tracepoints which means there is insufficient side-band information to decode Intel PT in per-cpu mode, and potentially workload-only mode too if the workload creates new processes.

Note also, that to use tracepoints, read-access to debugfs is required. So if debugfs is not mounted or the user does not have read-access, it will again not be possible to decode Intel PT in per-cpu mode.

sched_switch tracepoint

The sched_switch tracepoint is used to provide side-band data for Intel PT decoding in kernels where the PERF_RECORD_SWITCH metadata event isn?t available.

The sched_switch events are automatically added. e.g. the second event shown below:

```
$ perf record -vv -e intel_pt//u uname
------------------------------------------------------------
```

```
perf_event_attr:
type                    6
size                    112
config                  0x400
{ sample_period, sample_freq }  1
sample_type             IP|TID|TIME|CPU|IDENTIFIER
read_format             ID
disabled                1
inherit                 1
exclude_kernel          1
exclude_hv              1
enable_on_exec          1
sample_id_all           1
------------------------------------------------------------
sys_perf_event_open: pid 31104  cpu 0  group_fd -1  flags 0x8
sys_perf_event_open: pid 31104  cpu 1  group_fd -1  flags 0x8
sys_perf_event_open: pid 31104  cpu 2  group_fd -1  flags 0x8
sys_perf_event_open: pid 31104  cpu 3  group_fd -1  flags 0x8
------------------------------------------------------------
perf_event_attr:
type                    2
size                    112
config                  0x108
{ sample_period, sample_freq }  1
sample_type             IP|TID|TIME|CPU|PERIOD|RAW|IDENTIFIER
read_format             ID
inherit                 1
sample_id_all           1
exclude_guest           1
------------------------------------------------------------
sys_perf_event_open: pid -1  cpu 0  group_fd -1  flags 0x8
sys_perf_event_open: pid -1  cpu 1  group_fd -1  flags 0x8
sys_perf_event_open: pid -1  cpu 2  group_fd -1  flags 0x8
```

```
sys_perf_event_open: pid -1  cpu 3  group_fd -1  flags 0x8
------------------------------------------------------------
perf_event_attr:
type                      1
size                      112
config                    0x9
{ sample_period, sample_freq }    1
sample_type               IP|TID|TIME|IDENTIFIER
read_format               ID
disabled                  1
inherit                   1
exclude_kernel            1
exclude_hv                1
mmap                      1
comm                      1
enable_on_exec            1
task                      1
sample_id_all             1
mmap2                     1
comm_exec                 1
------------------------------------------------------------
sys_perf_event_open: pid 31104  cpu 0  group_fd -1  flags 0x8
sys_perf_event_open: pid 31104  cpu 1  group_fd -1  flags 0x8
sys_perf_event_open: pid 31104  cpu 2  group_fd -1  flags 0x8
sys_perf_event_open: pid 31104  cpu 3  group_fd -1  flags 0x8
mmap size 528384B
AUX area mmap length 4194304
perf event ring buffer mmapped per cpu
Synthesizing auxtrace information
Linux
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.042 MB perf.data ]
```

Note, the sched_switch event is only added if the user is permitted to use it and only in

per-cpu mode.

Note also, the sched_switch event is only added if TSC packets are requested. That is because, in the absence of timing information, the sched_switch events cannot be matched against the Intel PT trace.

PERF SCRIPT

By default, perf script will decode trace data found in the perf.data file. This can be further controlled by new option --itrace.

New --itrace option

Having no option is the same as

--itrace

which, in turn, is the same as

--itrace=cepwxy

The letters are:

i synthesize "instructions" events

y synthesize "cycles" events

b synthesize "branches" events

x synthesize "transactions" events

w synthesize "ptwrite" events

p synthesize "power" events (incl. PSB events)

c synthesize branches events (calls only)

r synthesize branches events (returns only)

o synthesize PEBS-via-PT events

l synthesize Event Trace events

e synthesize tracing error events

d create a debug log

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)

"Instructions" events look like they were recorded by "perf record -e instructions".
"Cycles" events look like they were recorded by "perf record -e cycles" (ie., the default).
Note that even with CYC packets enabled and no sampling, these are not fully accurate, since CYC packets are not emitted for each instruction, only when some other event (like an indirect branch, or a TNT packet representing multiple branches) happens causes a packet to be emitted. Thus, it is more effective for attributing cycles to functions (and possibly basic blocks) than to individual instructions, although it is not even perfect for functions (although it becomes better if the noretcomp option is active).
"Branches" events look like they were recorded by "perf record -e branches". "c" and "r" can be combined to get calls and returns.
"Transactions" events correspond to the start or end of transactions. The flags field can be used in perf script to determine whether the event is a transaction start, commit or abort.
Note that "instructions", "cycles", "branches" and "transactions" events depend on code flow packets which can be disabled by using the config term "branch=0". Refer to the config terms section above.
"ptwrite" events record the payload of the ptwrite instruction and whether "fup_on_ptw" was used. "ptwrite" events depend on PTWRITE packets which are recorded only if the "ptw" config term was used. Refer to the config terms section above. perf script "synth" field displays "ptwrite" information like this: "ip: 0 payload: 0x123456789abcdef0" where "ip" is 1 if "fup_on_ptw" was used.
"Power" events correspond to power event packets and CBR (core-to-bus ratio) packets. While CBR packets are always recorded when tracing is enabled, power event packets are recorded only if the "pwr_evt" config term was used. Refer to the config terms section above. The power events record information about C-state changes, whereas CBR is indicative of CPU frequency. perf script "event,synth" fields display information like this:

```
    cbr:  cbr: 22 freq: 2189 MHz (200%)
    mwait:  hints: 0x60 extensions: 0x1
    pwre:  hw: 0 cstate: 2 sub-cstate: 0
    exstop:  ip: 1
    pwrx:  deepest cstate: 2 last cstate: 2 wake reason: 0x4
```

Where:

"cbr" includes the frequency and the percentage of maximum non-turbo
"mwait" shows mwait hints and extensions

"pwre" shows C-state transitions (to a C-state deeper than C0) and whether initiated by hardware

"exstop" indicates execution stopped and whether the IP was recorded exactly,

"pwrx" indicates return to C0

For more details refer to the Intel 64 and IA-32 Architectures Software Developer Manuals.

PSB events show when a PSB+ occurred and also the byte-offset in the trace. Emitting a PSB+ can cause a CPU a slight delay. When doing timing analysis of code with Intel PT, it is useful to know if a timing bubble was caused by Intel PT or not.

Error events show where the decoder lost the trace. Error events are quite important. Users must know if what they are seeing is a complete picture or not. 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 supported by Intel PT are:

-o Suppress overflow errors

-l Suppress trace data lost errors

For example, for errors but not overflow or data lost errors:

--itrace=e-o-l

The "d" option will cause the creation of a file "intel_pt.log" containing all decoded packets and instructions. Note that this option slows down the decoder and that the resulting file may be very large. 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 support by Intel PT are:

-a Suppress logging of perf events

+a Log all perf events

+e Output only on decoding errors (size configurable)

+o Output to stdout instead of "intel_pt.log"

By default, logged perf events are filtered by any specified time ranges, but flag +a overrides that. The +e flag can be useful for analyzing errors. By default, the log size in that case is 16384 bytes, but can be altered by perf-config(1) e.g. perf config itrace.debug-log-buffer-size=30000

In addition, the period of the "instructions" event can be specified. e.g.

--itrace=i10us

sets the period to 10us i.e. one instruction sample is synthesized for each 10 microseconds

of trace. Alternatives to "us" are "ms" (milliseconds), "ns" (nanoseconds), "t" (TSC ticks) or "i" (instructions).

"ms", "us" and "ns" are converted to TSC ticks.

The timing information included with Intel PT does not give the time of every instruction. Consequently, for the purpose of sampling, the decoder estimates the time since the last timing packet based on 1 tick per instruction. The time on the sample is not adjusted and reflects the last known value of TSC.

For Intel PT, the default period is 100us.

Setting it to a zero period means "as often as possible".

In the case of Intel PT that is the same as a period of 1 and a unit of instructions (i.e. --itrace=i1i).

Also the call chain size (default 16, max. 1024) for instructions or transactions events can be specified. e.g.

```
--itrace=ig32
--itrace=xg32
```

Also the number of last branch entries (default 64, max. 1024) for instructions or transactions events can be specified. e.g.

```
--itrace=il10
--itrace=xl10
```

Note that last branch entries are cleared for each sample, so there is no overlap from one sample to the next.

The G and L options are designed in particular for sample mode, and work much like g and l but add call chain and branch stack to the other selected events instead of synthesized events. For example, to record branch-misses events for ls and then add a call chain derived from the Intel PT trace:

```
perf record --aux-sample -e '{intel_pt//u,branch-misses:u}' -- ls
perf report --itrace=Ge
```

Although in fact G is a default for perf report, so that is the same as just:

```
perf report
```

One caveat with the G and L options is that they work poorly with "Large PEBS". Large PEBS means PEBS records will be accumulated by hardware and the written into the event buffer in one go. That reduces interrupts, but can give very late timestamps. Because the Intel PT trace is synchronized by timestamps, the PEBS events do not match the trace. Currently,

Large PEBS is used only in certain circumstances: - hardware supports it - PEBS is used -
event period is specified, instead of frequency - the sample type is limited to the
following flags: PERF_SAMPLE_IP | PERF_SAMPLE_TID | PERF_SAMPLE_ADDR | PERF_SAMPLE_ID |
PERF_SAMPLE_CPU | PERF_SAMPLE_STREAM_ID | PERF_SAMPLE_DATA_SRC | PERF_SAMPLE_IDENTIFIER
|
PERF_SAMPLE_TRANSACTION | PERF_SAMPLE_PHYS_ADDR | PERF_SAMPLE_REGS_INTR |
PERF_SAMPLE_REGS_USER | PERF_SAMPLE_PERIOD (and sometimes) | PERF_SAMPLE_TIME Because Intel
PT sample mode uses a different sample type to the list above, Large PEBS is not used with
Intel PT sample mode. To avoid Large PEBS in other cases, avoid specifying the event period
i.e. avoid the perf record -c option, --count option, or period config term.
To disable trace decoding entirely, use the option --no-itrace.
It is also possible to skip events generated (instructions, branches, transactions) at the
beginning. This is useful to ignore initialization code.

--itrace=i0nss1000000

skips the first million instructions.
The q option changes the way the trace is decoded. The decoding is much faster but much less
detailed. Specifically, with the q option, the decoder does not decode TNT packets, and does
not walk object code, but gets the ip from FUP and TIP packets. The q option can be used
with the b and i options but the period is not used. The q option decodes more quickly, but
is useful only if the control flow of interest is represented or indicated by FUP, TIP,
TIP.PGE, or TIP.PGD packets (refer below). However the q option could be used to find time
ranges that could then be decoded fully using the --time option.
What will not be decoded with the (single) q option:

? direct calls and jmps
? conditional branches
? non-branch instructions

What will be decoded with the (single) q option:

? asynchronous branches such as interrupts
? indirect branches
? function return target address if the noretcomp config term (refer config terms section)
was used
? start of (control-flow) tracing
? end of (control-flow) tracing, if it is not out of context

? power events, ptwrite, transaction start and abort

? instruction pointer associated with PSB packets

Note the q option does not specify what events will be synthesized e.g. the p option must be used also to show power events.

Repeating the q option (double-q i.e. qq) results in even faster decoding and even less detail. The decoder decodes only extended PSB (PSB+) packets, getting the instruction pointer if there is a FUP packet within PSB+ (i.e. between PSB and PSBEND). Note PSB packets occur regularly in the trace based on the psb_period config term (refer config terms section). There will be a FUP packet if the PSB+ occurs while control flow is being traced.

What will not be decoded with the qq option:

? everything except instruction pointer associated with PSB packets

What will be decoded with the qq option:

? instruction pointer associated with PSB packets

The Z option is equivalent to having recorded a trace without TSC (i.e. config term tsc=0). It can be useful to avoid timestamp issues when decoding a trace of a virtual machine.

dlfilter-show-cycles.so

Cycles can be displayed using dlfilter-show-cycles.so in which case the itrace A option can be useful to provide higher granularity cycle information:

```
perf script --itrace=A --call-trace --dlfilter dlfilter-show-cycles.so
```

To see a list of dlfilters:

```
perf script -v --list-dlfilters
```

See also perf-dlfilters(1)

dump option

perf script has an option (-D) to "dump" the events i.e. display the binary data.

When -D is used, Intel PT packets are displayed. The packet decoder does not pay attention to PSB packets, but just decodes the bytes - so the packets seen by the actual decoder may not be identical in places where the data is corrupt. One example of that would be when the buffer-switching interrupt has been too slow, and the buffer has been filled completely. In that case, the last packet in the buffer might be truncated and immediately followed by a PSB as the trace continues in the next buffer.

To disable the display of Intel PT packets, combine the -D option with --no-itrace.

PERF REPORT

By default, perf report will decode trace data found in the perf.data file. This can be

further controlled by new option --itrace exactly the same as perf script, with the exception that the default is --itrace=igxe.

PERF INJECT

perf inject also accepts the --itrace option in which case tracing data is removed and replaced with the synthesized events. e.g.

```
perf inject --itrace -i perf.data -o perf.data.new
```

Below is an example of using Intel PT with autofdo. It requires autofdo (https://github.com/google/autofdo) and gcc version 5. The bubble sort example is from the AutoFDO tutorial (https://gcc.gnu.org/wiki/AutoFDO/Tutorial) amended to take the number of elements as a parameter.

```
$ gcc-5 -O3 sort.c -o sort_optimized
$ ./sort_optimized 30000
Bubble sorting array of 30000 elements
2254 ms
$ cat ~/.perfconfig
[intel-pt]
        mispred-all = on
$ perf record -e intel_pt//u ./sort 3000
Bubble sorting array of 3000 elements
58 ms
[ perf record: Woken up 2 times to write data ]
[ perf record: Captured and wrote 3.939 MB perf.data ]
$ perf inject -i perf.data -o inj --itrace=i100usle --strip
$ ./create_gcov --binary=./sort --profile=inj --gcov=sort.gcov -gcov_version=1
$ gcc-5 -O3 -fauto-profile=sort.gcov sort.c -o sort_autofdo
$ ./sort_autofdo 30000
Bubble sorting array of 30000 elements
2155 ms
```

Note there is currently no advantage to using Intel PT instead of LBR, but that may change in the future if greater use is made of the data.

PEBS VIA INTEL PT

Some hardware has the feature to redirect PEBS records to the Intel PT trace. Recording is selected by using the aux-output config term e.g.

perf record -c 10000 -e '{intel_pt/branch=0/,cycles/aux-output/ppp}' uname

Originally, software only supported redirecting at most one PEBS event because it was not able to differentiate one event from another. To overcome that, more recent kernels and perf tools add support for the PERF_RECORD_AUX_OUTPUT_HW_ID side-band event. To check for the presence of that event in a PEBS-via-PT trace:

perf script -D --no-itrace | grep PERF_RECORD_AUX_OUTPUT_HW_ID

To display PEBS events from the Intel PT trace, use the itrace o option e.g.

perf script --itrace=oe

XED

For --xed the xed tool is needed. Here is how to install it:

```
$ git clone https://github.com/intelxed/mbuild.git mbuild
$ git clone https://github.com/intelxed/xed
$ cd xed
$ ./mfile.py --share
$ ./mfile.py examples
$ sudo ./mfile.py --prefix=/usr/local install
$ sudo ldconfig
$ sudo cp obj/examples/xed /usr/local/bin
```

Basic xed testing:

```
$ xed | head -3
ERROR: required argument(s) were missing
Copyright (C) 2017, Intel Corporation. All rights reserved.
XED version: [v10.0-328-g7d62c8c49b7b]
$
```

TRACING VIRTUAL MACHINES (KERNEL ONLY)

Currently, kernel tracing is supported with either "timeless" decoding (i.e. no TSC timestamps) or VM Time Correlation. VM Time Correlation is an extra step using perf inject and requires unchanging VMX TSC Offset and no VMX TSC Scaling.

Other limitations and caveats

VMX controls may suppress packets needed for decoding resulting in decoding errors

VMX controls may block the perf NMI to the host potentially resulting in lost trace data

Guest kernel self-modifying code (e.g. jump labels or JIT-compiled eBPF) will result in decoding errors

Guest thread information is unknown

Guest VCPU is unknown but may be able to be inferred from the host thread

Callchains are not supported

Example using "timeless" decoding

Start VM

$ sudo virsh start kubuntu20.04

Domain kubuntu20.04 started

Mount the guest file system. Note sshfs needs -o direct_io to enable reading of proc files.

root access is needed to read /proc/kcore.

$ mkdir vm0

$ sshfs -o direct_io root@vm0:/ vm0

Copy the guest /proc/kallsyms, /proc/modules and /proc/kcore

$ perf buildid-cache -v --kcore vm0/proc/kcore

kcore added to build-id cache directory /home/user/.debug/[kernel.kcore]/9600f316a53a0f54278885e8d9710538ec5f6a08/2021021807494306

$ KALLSYMS=/home/user/.debug/[kernel.kcore]/9600f316a53a0f54278885e8d9710538ec5f6a08/2021021807494306/kallsyms

Find the VM process

$ ps -eLl | grep 'KVM\|PID'

F S UID PID PPID LWP C PRI NI ADDR SZ WCHAN TTY TIME CMD

3 S 64055 1430 1 1440 1 80 0 - 1921718 - ? 00:02:47 CPU 0/KVM

3 S 64055 1430 1 1441 1 80 0 - 1921718 - ? 00:02:41 CPU 1/KVM

3 S 64055 1430 1 1442 1 80 0 - 1921718 - ? 00:02:38 CPU 2/KVM

3 S 64055 1430 1 1443 2 80 0 - 1921718 - ? 00:03:18 CPU 3/KVM

Start an open-ended perf record, tracing the VM process, do something on the VM, and then ctrl-C to stop. TSC is not supported and tsc=0 must be specified. That means mtc is useless, so add mtc=0. However, IPC can still be determined, hence cyc=1 can be added. Only kernel decoding is supported, so k must be specified. Intel PT traces both the host and the guest so --guest and --host need to be specified. Without timestamps, --per-thread must be specified to distinguish threads.

$ sudo perf kvm --guest --host --guestkallsyms $KALLSYMS record --kcore -e intel_pt/tsc=0,mtc=0,cyc=1/k -p 1430 --per-thread

^C

[perf record: Woken up 1 times to write data]

[perf record: Captured and wrote 5.829 MB]

perf script can be used to provide an instruction trace

$ perf script --guestkallsyms $KALLSYMS --insn-trace=disasm -F+ipc | grep -C10 vmresume | head -21

CPU 0/KVM 1440 ffffffff82133cdd __vmx_vcpu_run+0x3d ([kernel.kallsyms]) movq 0x48(%rax), %r9

CPU 0/KVM 1440 ffffffff82133ce1 __vmx_vcpu_run+0x41 ([kernel.kallsyms]) movq 0x50(%rax), %r10

CPU 0/KVM 1440 ffffffff82133ce5 __vmx_vcpu_run+0x45 ([kernel.kallsyms]) movq 0x58(%rax), %r11

CPU 0/KVM 1440 ffffffff82133ce9 __vmx_vcpu_run+0x49 ([kernel.kallsyms]) movq 0x60(%rax), %r12

CPU 0/KVM 1440 ffffffff82133ced __vmx_vcpu_run+0x4d ([kernel.kallsyms]) movq 0x68(%rax), %r13

CPU 0/KVM 1440 ffffffff82133cf1 __vmx_vcpu_run+0x51 ([kernel.kallsyms]) movq 0x70(%rax), %r14

CPU 0/KVM 1440 ffffffff82133cf5 __vmx_vcpu_run+0x55 ([kernel.kallsyms]) movq 0x78(%rax), %r15

CPU 0/KVM 1440 ffffffff82133cf9 __vmx_vcpu_run+0x59 ([kernel.kallsyms]) movq (%rax), %rax

CPU 0/KVM 1440 ffffffff82133cfc __vmx_vcpu_run+0x5c ([kernel.kallsyms]) callq 0xffffffff82133c40

CPU 0/KVM 1440 ffffffff82133c40 vmx_vmenter+0x0 ([kernel.kallsyms]) jz 0xffffffff82133c46

CPU 0/KVM 1440 ffffffff82133c42 vmx_vmenter+0x2 ([kernel.kallsyms]) vmresume IPC: 0.11 (50/445)

:1440 1440 ffffffffbb678b06 native_write_msr+0x6 ([guest.kernel.kallsyms]) nopl %eax, (%rax,%rax,1)

:1440 1440 ffffffffbb678b0b native_write_msr+0xb ([guest.kernel.kallsyms]) retq IPC: 0.04 (2/41)

:1440 1440 ffffffffbb666646 lapic_next_deadline+0x26 ([guest.kernel.kallsyms]) data16 nop

:1440 1440 ffffffffbb666648 lapic_next_deadline+0x28 ([guest.kernel.kallsyms]) xor %eax, %eax

:1440 1440 ffffffffbb66664a lapic_next_deadline+0x2a ([guest.kernel.kallsyms]) popq %rbp

:1440 1440 ffffffffbb66664b lapic_next_deadline+0x2b ([guest.kernel.kallsyms]) retq IPC: 0.16 (4/25)

:1440 1440 ffffffffbb74607f clockevents_program_event+0x8f ([guest.kernel.kallsyms]) test %eax, %eax

:1440 1440 ffffffffbb746081 clockevents_program_event+0x91 ([guest.kernel.kallsyms]) jz 0xffffffffbb74603c IPC: 0.06 (2/30)

:1440 1440 ffffffffbb74603c clockevents_program_event+0x4c ([guest.kernel.kallsyms]) popq %rbx

:1440 1440 ffffffffbb74603d clockevents_program_event+0x4d ([guest.kernel.kallsyms]) popq %r12

Example using VM Time Correlation

Start VM

$ sudo virsh start kubuntu20.04

Domain kubuntu20.04 started

Mount the guest file system. Note sshfs needs -o direct_io to enable reading of proc files.

root access is needed to read /proc/kcore.

```
$ mkdir -p vm0
$ sshfs -o direct_io root@vm0:/ vm0
```

Copy the guest /proc/kallsyms, /proc/modules and /proc/kcore

```
$ perf buildid-cache -v --kcore vm0/proc/kcore
```

same kcore found in /home/user/.debug/[kernel.kcore]/cc9c55a98c5e4ec0aeda69302554aabed5cd6491/2021021312450777

```
$ KALLSYMS=/home/user/.debug/\[kernel.kcore\]/cc9c55a98c5e4ec0aeda69302554aabed5cd6491/2021021312450777/kallsyms
```

Find the VM process

```
$ ps -eLl | grep 'KVM\|PID'
F S   UID     PID    PPID     LWP  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
3 S 64055   16998       1   17005 13  80   0 - 1818189 -    ?        00:00:16 CPU 0/KVM
3 S 64055   16998       1   17006  4  80   0 - 1818189 -    ?        00:00:05 CPU 1/KVM
3 S 64055   16998       1   17007  3  80   0 - 1818189 -    ?        00:00:04 CPU 2/KVM
3 S 64055   16998       1   17008  4  80   0 - 1818189 -    ?        00:00:05 CPU 3/KVM
```

Start an open-ended perf record, tracing the VM process, do something on the VM, and then ctrl-C to stop. IPC can be determined, hence cyc=1 can be added. Only kernel decoding is supported, so k must be specified. Intel PT traces both the host and the guest so --guest and --host need to be specified.

```
$ sudo perf kvm --guest --host --guestkallsyms $KALLSYMS record --kcore -e intel_pt/cyc=1/k -p 16998
^C[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 9.041 MB perf.data.kvm ]
```

Now perf inject can be used to determine the VMX TCS Offset. Note, Intel PT TSC packets are only 7-bytes, so the TSC Offset might differ from the actual value in the 8th byte. That will have no effect i.e. the resulting timestamps will be correct anyway.

```
$ perf inject -i perf.data.kvm --vm-time-correlation=dry-run
ERROR: Unknown TSC Offset for VMCS 0x1bff6a
VMCS: 0x1bff6a  TSC Offset 0xffffe42722c64c41
ERROR: Unknown TSC Offset for VMCS 0x1cbc08
```

VMCS: 0x1cbc08 TSC Offset 0xffffe42722c64c41

ERROR: Unknown TSC Offset for VMCS 0x1c3ce8

VMCS: 0x1c3ce8 TSC Offset 0xffffe42722c64c41

ERROR: Unknown TSC Offset for VMCS 0x1cbce9

VMCS: 0x1cbce9 TSC Offset 0xffffe42722c64c41

Each virtual CPU has a different Virtual Machine Control Structure (VMCS) shown above with the calculated TSC Offset. For an unchanging TSC Offset they should all be the same for the same virtual machine.

Now that the TSC Offset is known, it can be provided to perf inject

$ perf inject -i perf.data.kvm --vm-time-correlation="dry-run 0xffffe42722c64c41"

Note the options for perf inject --vm-time-correlation are:

[dry-run] [<TSC Offset> [: <VMCS> [, <VMCS>]...]]...

So it is possible to specify different TSC Offsets for different VMCS. The option "dry-run" will cause the file to be processed but without updating it. Note it is also possible to get a intel_pt.log file by adding option --itrace=d

There were no errors so, do it for real

$ perf inject -i perf.data.kvm --vm-time-correlation=0xffffe42722c64c41 --force

perf script can be used to see if there are any decoder errors

$ perf script -i perf.data.kvm --guestkallsyms $KALLSYMS --itrace=e-o

There were none.

perf script can be used to provide an instruction trace showing timestamps

$ perf script -i perf.data.kvm --guestkallsyms $KALLSYMS --insn-trace=disasm -F+ipc | grep -C10 vmresume | head -21

CPU 1/KVM 17006 [001] 11500.262865593: ffffffff82133cdd __vmx_vcpu_run+0x3d ([kernel.kallsyms])
movq 0x48(%rax), %r9

CPU 1/KVM 17006 [001] 11500.262865593: ffffffff82133ce1 __vmx_vcpu_run+0x41 ([kernel.kallsyms])
movq 0x50(%rax), %r10

CPU 1/KVM 17006 [001] 11500.262865593: ffffffff82133ce5 __vmx_vcpu_run+0x45 ([kernel.kallsyms])
movq 0x58(%rax), %r11

CPU 1/KVM 17006 [001] 11500.262865593: ffffffff82133ce9 __vmx_vcpu_run+0x49 ([kernel.kallsyms])
movq 0x60(%rax), %r12

CPU 1/KVM 17006 [001] 11500.262865593: ffffffff82133ced __vmx_vcpu_run+0x4d ([kernel.kallsyms])
movq 0x68(%rax), %r13

CPU 1/KVM 17006 [001] 11500.262865593: ffffffff82133cf1 __vmx_vcpu_run+0x51 ([kernel.kallsyms])
movq 0x70(%rax), %r14
CPU 1/KVM 17006 [001] 11500.262865593: ffffffff82133cf5 __vmx_vcpu_run+0x55 ([kernel.kallsyms])
movq 0x78(%rax), %r15
CPU 1/KVM 17006 [001] 11500.262865593: ffffffff82133cf9 __vmx_vcpu_run+0x59 ([kernel.kallsyms])
movq (%rax), %rax
CPU 1/KVM 17006 [001] 11500.262865593: ffffffff82133cfc __vmx_vcpu_run+0x5c ([kernel.kallsyms])
callq 0xffffffff82133c40
CPU 1/KVM 17006 [001] 11500.262865593: ffffffff82133c40 vmx_vmenter+0x0 ([kernel.kallsyms]) jz
0xffffffff82133c46
CPU 1/KVM 17006 [001] 11500.262866075: ffffffff82133c42 vmx_vmenter+0x2 ([kernel.kallsyms])
vmresume IPC: 0.05 (40/769)
:17006 17006 [001] 11500.262869216: ffffffff82200cb0 asm_sysvec_apic_timer_interrupt+0x0
([guest.kernel.kallsyms]) clac
:17006 17006 [001] 11500.262869216: ffffffff82200cb3 asm_sysvec_apic_timer_interrupt+0x3
([guest.kernel.kallsyms]) pushq $0xffffffffffffffff
:17006 17006 [001] 11500.262869216: ffffffff82200cb5 asm_sysvec_apic_timer_interrupt+0x5
([guest.kernel.kallsyms]) callq 0xffffffff82201160
:17006 17006 [001] 11500.262869216: ffffffff82201160 error_entry+0x0 ([guest.kernel.kallsyms]) cld
:17006 17006 [001] 11500.262869216: ffffffff82201161 error_entry+0x1 ([guest.kernel.kallsyms])
pushq %rsi
:17006 17006 [001] 11500.262869216: ffffffff82201162 error_entry+0x2 ([guest.kernel.kallsyms])
movq 0x8(%rsp), %rsi
:17006 17006 [001] 11500.262869216: ffffffff82201167 error_entry+0x7 ([guest.kernel.kallsyms])
movq %rdi, 0x8(%rsp)
:17006 17006 [001] 11500.262869216: ffffffff8220116c error_entry+0xc ([guest.kernel.kallsyms])
pushq %rdx
:17006 17006 [001] 11500.262869216: ffffffff8220116d error_entry+0xd ([guest.kernel.kallsyms])
pushq %rcx
:17006 17006 [001] 11500.262869216: ffffffff8220116e error_entry+0xe ([guest.kernel.kallsyms])
pushq %rax

TRACING VIRTUAL MACHINES (INCLUDING USER SPACE)

It is possible to use perf record to record sideband events within a virtual machine, so

that an Intel PT trace on the host can be decoded. Sideband events from the guest perf.data

file can be injected into the host perf.data file using perf inject.

Here is an example of the steps needed:

On the guest machine:

Check that no-kvmclock kernel command line option was used to boot:

Note, this is essential to enable time correlation between host and guest machines.

$ cat /proc/cmdline

BOOT_IMAGE=/boot/vmlinuz-5.10.0-16-amd64 root=UUID=cb49c910-e573-47e0-bce7-79e293df8e1d ro no-kvmclock

There is no BPF support at present so, if possible, disable JIT compiling:

$ echo 0 | sudo tee /proc/sys/net/core/bpf_jit_enable

0

Start perf record to collect sideband events:

$ sudo perf record -o guest-sideband-testing-guest-perf.data --sample-identifier --buildid-all --switch-events --kcore -a -e dummy

On the host machine:

Start perf record to collect Intel PT trace:

Note, the host trace will get very big, very fast, so the steps from starting to stopping

the host trace really need to be done so that they happen in the shortest time possible.

$ sudo perf record -o guest-sideband-testing-host-perf.data -m,64M --kcore -a -e intel_pt/cyc/

On the guest machine:

Run a small test case, just uname in this example:

$ uname

Linux

On the host machine:

Stop the Intel PT trace:

^C

[perf record: Woken up 1 times to write data]

[perf record: Captured and wrote 76.122 MB guest-sideband-testing-host-perf.data]

On the guest machine:

Stop the Intel PT trace:

^C

[perf record: Woken up 1 times to write data]

[perf record: Captured and wrote 1.247 MB guest-sideband-testing-guest-perf.data]

And then copy guest-sideband-testing-guest-perf.data to the host (not shown here).

On the host machine:

With the 2 perf.data recordings, and with their ownership changed to the user.

Identify the TSC Offset:

```
$ perf inject -i guest-sideband-testing-host-perf.data --vm-time-correlation=dry-run
VMCS: 0x103fc6  TSC Offset 0xfffffa6ae070cb20
VMCS: 0x103ff2  TSC Offset 0xfffffa6ae070cb20
VMCS: 0x10fdaa  TSC Offset 0xfffffa6ae070cb20
VMCS: 0x24d57c  TSC Offset 0xfffffa6ae070cb20
```

Correct Intel PT TSC timestamps for the guest machine:

```
$ perf inject -i guest-sideband-testing-host-perf.data --vm-time-correlation=0xfffffa6ae070cb20 --force
```

Identify the guest machine PID:

```
$ perf script -i guest-sideband-testing-host-perf.data --no-itrace --show-task-events | grep KVM
    CPU 0/KVM     0 [000]     0.000000: PERF_RECORD_COMM: CPU 0/KVM:13376/13381
    CPU 1/KVM     0 [000]     0.000000: PERF_RECORD_COMM: CPU 1/KVM:13376/13382
    CPU 2/KVM     0 [000]     0.000000: PERF_RECORD_COMM: CPU 2/KVM:13376/13383
    CPU 3/KVM     0 [000]     0.000000: PERF_RECORD_COMM: CPU 3/KVM:13376/13384
```

Note, the QEMU option -name debug-threads=on is needed so that thread names can be used to determine which thread is running which VCPU as above. libvirt seems to use this by default.

Create a guestmount, assuming the guest machine is vm_to_test:

```
$ mkdir -p ~/guestmount/13376
$ sshfs -o direct_io vm_to_test:/ ~/guestmount/13376
```

Inject the guest perf.data file into the host perf.data file:

Note, due to the guestmount option, guest object files and debug files will be copied into the build ID cache from the guest machine, with the notable exception of VDSO. If needed, VDSO can be copied manually in a fashion similar to that used by the perf-archive script.

```
$ perf inject -i guest-sideband-testing-host-perf.data -o inj --guestmount ~/guestmount --guest-data=guest-sideband-testing-guest-perf.data,13376,0xfffffa6ae070cb20
```

Show an excerpt from the result. In this case the CPU and time range have been to chosen to show interaction between guest and host when uname is starting to run on the guest machine:

Notes:

? the CPU displayed, [002] in this case, is always the host CPU

? events happening in the virtual machine start with VM:13376 VCPU:003, which shows the hypervisor PID 13376 and the VCPU number

? only calls and errors are displayed i.e. --itrace=ce

? branches entering and exiting the virtual machine are split, and show as 2 branches to/from "0 [unknown] ([unknown])"

$ perf script -i inj --itrace=ce -F+machine_pid,+vcpu,+addr,+pid,+tid,-period --ns --time 7919.408803365,7919.408804631 -C 2

CPU 3/KVM 13376/13384 [002] 7919.408803365: branches: ffffffffc0f8ebe0 vmx_vcpu_enter_exit+0xc0 ([kernel.kallsyms]) => ffffffffc0f8edc0 __vmx_vcpu_run+0x0 ([kernel.kallsyms])

CPU 3/KVM 13376/13384 [002] 7919.408803365: branches: ffffffffc0f8edd5 __vmx_vcpu_run+0x15 ([kernel.kallsyms]) => ffffffffc0f8eca0 vmx_update_host_rsp+0x0 ([kernel.kallsyms])

CPU 3/KVM 13376/13384 [002] 7919.408803365: branches: ffffffffc0f8ee1b __vmx_vcpu_run+0x5b ([kernel.kallsyms]) => ffffffffc0f8ed60 vmx_vmenter+0x0 ([kernel.kallsyms])

CPU 3/KVM 13376/13384 [002] 7919.408803461: branches: ffffffffc0f8ed62 vmx_vmenter+0x2 ([kernel.kallsyms]) => 0 [unknown] ([unknown])

VM:13376 VCPU:003 uname 3404/3404 [002] 7919.408803461: branches: 0 [unknown] ([unknown]) => 7f851c9b5a5c init_cacheinfo+0x3ac (/usr/lib/x86_64-linux-gnu/libc-2.31.so)

VM:13376 VCPU:003 uname 3404/3404 [002] 7919.408803567: branches: 7f851c9b5a5a init_cacheinfo+0x3aa (/usr/lib/x86_64-linux-gnu/libc-2.31.so) => 0 [unknown] ([unknown])

CPU 3/KVM 13376/13384 [002] 7919.408803567: branches: 0 [unknown] ([unknown]) => ffffffffc0f8ed80 vmx_vmexit+0x0 ([kernel.kallsyms])

CPU 3/KVM 13376/13384 [002] 7919.408803596: branches: ffffffffc0f6619a vmx_vcpu_run+0x26a ([kernel.kallsyms]) => ffffffffb2255c60 x86_virt_spec_ctrl+0x0 ([kernel.kallsyms])

CPU 3/KVM 13376/13384 [002] 7919.408803801: branches: ffffffffc0f66445 vmx_vcpu_run+0x515 ([kernel.kallsyms]) => ffffffffb2290b30 native_write_msr+0x0 ([kernel.kallsyms])

CPU 3/KVM 13376/13384 [002] 7919.408803850: branches: ffffffffc0f661f8 vmx_vcpu_run+0x2c8 ([kernel.kallsyms]) => ffffffffc1092300 kvm_load_host_xsave_state+0x0 ([kernel.kallsyms])

CPU 3/KVM 13376/13384 [002] 7919.408803850: branches: ffffffffc1092327 kvm_load_host_xsave_state+0x27 ([kernel.kallsyms]) => ffffffffc1092220 kvm_load_host_xsave_state.part.0+0x0 ([kernel.kallsyms])

CPU 3/KVM 13376/13384 [002] 7919.408803862: branches: ffffffffc0f662cf vmx_vcpu_run+0x39f ([kernel.kallsyms]) => ffffffffc0f63f90 vmx_recover_nmi_blocking+0x0 ([kernel.kallsyms])

CPU 3/KVM 13376/13384 [002] 7919.408803862: branches: ffffffffc0f662e9 vmx_vcpu_run+0x3b9

([kernel.kallsyms]) => ffffffffc0f619a0 __vmx_complete_interrupts+0x0 ([kernel.kallsyms])

CPU 3/KVM 13376/13384 [002] 7919.408803872: branches: ffffffffc109cfb2 vcpu_enter_guest+0x752 ([kernel.kallsyms]) => ffffffffc0f5f570 vmx_handle_exit_irqoff+0x0 ([kernel.kallsyms])

CPU 3/KVM 13376/13384 [002] 7919.408803881: branches: ffffffffc109d028 vcpu_enter_guest+0x7c8 ([kernel.kallsyms]) => ffffffffb234f900 __srcu_read_lock+0x0 ([kernel.kallsyms])

CPU 3/KVM 13376/13384 [002] 7919.408803897: branches: ffffffffc109d06f vcpu_enter_guest+0x80f ([kernel.kallsyms]) => ffffffffc0f72e30 vmx_handle_exit+0x0 ([kernel.kallsyms])

CPU 3/KVM 13376/13384 [002] 7919.408803897: branches: ffffffffc0f72e3d vmx_handle_exit+0xd ([kernel.kallsyms]) => ffffffffc0f727c0 __vmx_handle_exit+0x0 ([kernel.kallsyms])

CPU 3/KVM 13376/13384 [002] 7919.408803897: branches: ffffffffc0f72b15 __vmx_handle_exit+0x355 ([kernel.kallsyms]) => ffffffffc0f60ae0 vmx_flush_pml_buffer+0x0 ([kernel.kallsyms])

CPU 3/KVM 13376/13384 [002] 7919.408803903: branches: ffffffffc0f72994 __vmx_handle_exit+0x1d4 ([kernel.kallsyms]) => ffffffffc10b7090 kvm_emulate_cpuid+0x0 ([kernel.kallsyms])

CPU 3/KVM 13376/13384 [002] 7919.408803903: branches: ffffffffc10b70f1 kvm_emulate_cpuid+0x61 ([kernel.kallsyms]) => ffffffffc10b6e10 kvm_cpuid+0x0 ([kernel.kallsyms])

CPU 3/KVM 13376/13384 [002] 7919.408803941: branches: ffffffffc10b7125 kvm_emulate_cpuid+0x95 ([kernel.kallsyms]) => ffffffffc1093110 kvm_skip_emulated_instruction+0x0 ([kernel.kallsyms])

CPU 3/KVM 13376/13384 [002] 7919.408803941: branches: ffffffffc109311f kvm_skip_emulated_instruction+0xf ([kernel.kallsyms]) => ffffffffc0f5e180 vmx_get_rflags+0x0 ([kernel.kallsyms])

CPU 3/KVM 13376/13384 [002] 7919.408803951: branches: ffffffffc109312a kvm_skip_emulated_instruction+0x1a ([kernel.kallsyms]) => ffffffffc0f5fd30 vmx_skip_emulated_instruction+0x0 ([kernel.kallsyms])

CPU 3/KVM 13376/13384 [002] 7919.408803951: branches: ffffffffc0f5fd79 vmx_skip_emulated_instruction+0x49 ([kernel.kallsyms]) => ffffffffc0f5fb50 skip_emulated_instruction+0x0 ([kernel.kallsyms])

CPU 3/KVM 13376/13384 [002] 7919.408803956: branches: ffffffffc0f5fc68 skip_emulated_instruction+0x118 ([kernel.kallsyms]) => ffffffffc0f6a940 vmx_cache_reg+0x0 ([kernel.kallsyms])

CPU 3/KVM 13376/13384 [002] 7919.408803964: branches: ffffffffc0f5fc11 skip_emulated_instruction+0xc1 ([kernel.kallsyms]) => ffffffffc0f5f9e0 vmx_set_interrupt_shadow+0x0 ([kernel.kallsyms])

CPU 3/KVM 13376/13384 [002] 7919.408803980: branches: ffffffffc109f8b1 vcpu_run+0x71 ([kernel.kallsyms]) => ffffffffc10ad2f0 kvm_cpu_has_pending_timer+0x0 ([kernel.kallsyms])

CPU 3/KVM 13376/13384 [002] 7919.408803980: branches: ffffffffc10ad2fb kvm_cpu_has_pending_timer+0xb ([kernel.kallsyms]) => ffffffffc10b0490 apic_has_pending_timer+0x0 ([kernel.kallsyms])

CPU 3/KVM 13376/13384 [002] 7919.408803991: branches: ffffffffc109f899 vcpu_run+0x59 ([kernel.kallsyms]) => ffffffffc109c860 vcpu_enter_guest+0x0 ([kernel.kallsyms])

CPU 3/KVM 13376/13384 [002] 7919.408803993: branches: ffffffffc109cd4c vcpu_enter_guest+0x4ec ([kernel.kallsyms]) => ffffffffc0f69140 vmx_prepare_switch_to_guest+0x0 ([kernel.kallsyms])

CPU 3/KVM 13376/13384 [002] 7919.408803996: branches: ffffffffc109cd7d vcpu_enter_guest+0x51d ([kernel.kallsyms]) => ffffffffb234f930 __srcu_read_unlock+0x0 ([kernel.kallsyms])

CPU 3/KVM 13376/13384 [002] 7919.408803996: branches: ffffffffc109cd9c vcpu_enter_guest+0x53c ([kernel.kallsyms]) => ffffffffc0f609b0 vmx_sync_pir_to_irr+0x0 ([kernel.kallsyms])

CPU 3/KVM 13376/13384 [002] 7919.408803996: branches: ffffffffc0f60a6d vmx_sync_pir_to_irr+0xbd ([kernel.kallsyms]) => ffffffffc10adc20 kvm_lapic_find_highest_irr+0x0 ([kernel.kallsyms])

CPU 3/KVM 13376/13384 [002] 7919.408804010: branches: ffffffffc0f60abd vmx_sync_pir_to_irr+0x10d ([kernel.kallsyms]) => ffffffffc0f60820 vmx_set_rvi+0x0 ([kernel.kallsyms])

CPU 3/KVM 13376/13384 [002] 7919.408804019: branches: ffffffffc109ceca vcpu_enter_guest+0x66a ([kernel.kallsyms]) => ffffffffb2249840 fpregs_assert_state_consistent+0x0 ([kernel.kallsyms])

CPU 3/KVM 13376/13384 [002] 7919.408804021: branches: ffffffffc109cf10 vcpu_enter_guest+0x6b0 ([kernel.kallsyms]) => ffffffffc0f65f30 vmx_vcpu_run+0x0 ([kernel.kallsyms])

CPU 3/KVM 13376/13384 [002] 7919.408804024: branches: ffffffffc0f6603b vmx_vcpu_run+0x10b ([kernel.kallsyms]) => ffffffffb229bed0 __get_current_cr3_fast+0x0 ([kernel.kallsyms])

CPU 3/KVM 13376/13384 [002] 7919.408804024: branches: ffffffffc0f66055 vmx_vcpu_run+0x125 ([kernel.kallsyms]) => ffffffffb2253050 cr4_read_shadow+0x0 ([kernel.kallsyms])

CPU 3/KVM 13376/13384 [002] 7919.408804030: branches: ffffffffc0f6608d vmx_vcpu_run+0x15d ([kernel.kallsyms]) => ffffffffc10921e0 kvm_load_guest_xsave_state+0x0 ([kernel.kallsyms])

CPU 3/KVM 13376/13384 [002] 7919.408804030: branches: ffffffffc1092207 kvm_load_guest_xsave_state+0x27 ([kernel.kallsyms]) => ffffffffc1092110 kvm_load_guest_xsave_state.part.0+0x0 ([kernel.kallsyms])

CPU 3/KVM 13376/13384 [002] 7919.408804032: branches: ffffffffc0f660c6 vmx_vcpu_run+0x196 ([kernel.kallsyms]) => ffffffffb22061a0 perf_guest_get_msrs+0x0 ([kernel.kallsyms])

CPU 3/KVM 13376/13384 [002] 7919.408804032: branches: ffffffffb22061a9 perf_guest_get_msrs+0x9 ([kernel.kallsyms]) => ffffffffb220cda0 intel_guest_get_msrs+0x0 ([kernel.kallsyms])

CPU 3/KVM 13376/13384 [002] 7919.408804039: branches: ffffffffc0f66109 vmx_vcpu_run+0x1d9 ([kernel.kallsyms]) => ffffffffc0f652c0 clear_atomic_switch_msr+0x0 ([kernel.kallsyms])

CPU 3/KVM 13376/13384 [002] 7919.408804040: branches: ffffffffc0f66119 vmx_vcpu_run+0x1e9 ([kernel.kallsyms]) => ffffffffc0f73f60 intel_pmu_lbr_is_enabled+0x0 ([kernel.kallsyms])

CPU 3/KVM 13376/13384 [002] 7919.408804042: branches: ffffffffc0f73f81 intel_pmu_lbr_is_enabled+0x21 ([kernel.kallsyms]) => ffffffffc10b68e0 kvm_find_cpuid_entry+0x0 ([kernel.kallsyms])

CPU 3/KVM 13376/13384 [002] 7919.408804045: branches: ffffffffc0f66454 vmx_vcpu_run+0x524 ([kernel.kallsyms]) => ffffffffc0f61ff0 vmx_update_hv_timer+0x0 ([kernel.kallsyms])

CPU 3/KVM 13376/13384 [002] 7919.408804057: branches: ffffffffc0f66142 vmx_vcpu_run+0x212 ([kernel.kallsyms]) => ffffffffc10af100 kvm_wait_lapic_expire+0x0 ([kernel.kallsyms])

CPU 3/KVM 13376/13384 [002] 7919.408804057: branches: ffffffffc0f66156 vmx_vcpu_run+0x226 ([kernel.kallsyms]) => ffffffffb2255c60 x86_virt_spec_ctrl+0x0 ([kernel.kallsyms])

CPU 3/KVM 13376/13384 [002] 7919.408804057: branches: ffffffffc0f66161 vmx_vcpu_run+0x231 ([kernel.kallsyms]) => ffffffffc0f8eb20 vmx_vcpu_enter_exit+0x0 ([kernel.kallsyms])

CPU 3/KVM 13376/13384 [002] 7919.408804057: branches: ffffffffc0f8eb44 vmx_vcpu_enter_exit+0x24 ([kernel.kallsyms]) => ffffffffb2353e10 rcu_note_context_switch+0x0 ([kernel.kallsyms])

CPU 3/KVM 13376/13384 [002] 7919.408804057: branches: ffffffffb2353e1c rcu_note_context_switch+0xc ([kernel.kallsyms]) => ffffffffb2353db0 rcu_qs+0x0 ([kernel.kallsyms])

CPU 3/KVM 13376/13384 [002] 7919.408804066: branches: ffffffffc0f8ebe0 vmx_vcpu_enter_exit+0xc0 ([kernel.kallsyms]) => ffffffffc0f8edc0 __vmx_vcpu_run+0x0 ([kernel.kallsyms])

CPU 3/KVM 13376/13384 [002] 7919.408804066: branches: ffffffffc0f8edd5 __vmx_vcpu_run+0x15 ([kernel.kallsyms]) => ffffffffc0f8eca0 vmx_update_host_rsp+0x0 ([kernel.kallsyms])

CPU 3/KVM 13376/13384 [002] 7919.408804066: branches: ffffffffc0f8ee1b __vmx_vcpu_run+0x5b ([kernel.kallsyms]) => ffffffffc0f8ed60 vmx_vmenter+0x0 ([kernel.kallsyms])

CPU 3/KVM 13376/13384 [002] 7919.408804162: branches: ffffffffc0f8ed62 vmx_vmenter+0x2 ([kernel.kallsyms]) => 0 [unknown] ([unknown])

VM:13376 VCPU:003 uname 3404/3404 [002] 7919.408804162: branches: 0 [unknown] ([unknown]) => 7f851c9b5a5c init_cacheinfo+0x3ac (/usr/lib/x86_64-linux-gnu/libc-2.31.so)

VM:13376 VCPU:003 uname 3404/3404 [002] 7919.408804273: branches: 7f851cb7c0e4 _dl_init+0x74 (/usr/lib/x86_64-linux-gnu/ld-2.31.so) => 7f851cb7bf50 call_init.part.0+0x0 (/usr/lib/x86_64-linux-gnu/ld-2.31.so)

VM:13376 VCPU:003 uname 3404/3404 [002] 7919.408804526: branches: 55e0c00136f0 _start+0x0 (/usr/bin/uname) => ffffffff83200ac0 asm_exc_page_fault+0x0 ([kernel.kallsyms])

VM:13376 VCPU:003 uname 3404/3404 [002] 7919.408804526: branches: ffffffff83200ac3 asm_exc_page_fault+0x3 ([kernel.kallsyms]) => ffffffff83201290 error_entry+0x0 ([kernel.kallsyms])

VM:13376 VCPU:003 uname 3404/3404 [002] 7919.408804534: branches: ffffffff832012fa error_entry+0x6a ([kernel.kallsyms]) => ffffffff830b59a0 sync_regs+0x0 ([kernel.kallsyms])

VM:13376 VCPU:003 uname 3404/3404 [002] 7919.408804631: branches: ffffffff83200ad9 asm_exc_page_fault+0x19 ([kernel.kallsyms]) => ffffffff830b8210 exc_page_fault+0x0 ([kernel.kallsyms])

VM:13376 VCPU:003 uname 3404/3404 [002] 7919.408804631: branches: ffffffff830b82a4 exc_page_fault+0x94 ([kernel.kallsyms]) => ffffffff830b80e0 __kvm_handle_async_pf+0x0 ([kernel.kallsyms])

VM:13376 VCPU:003 uname 3404/3404 [002] 7919.408804631: branches: ffffffff830b80ed __kvm_handle_async_pf+0xd ([kernel.kallsyms]) => ffffffff830b80c0 kvm_read_and_reset_apf_flags+0x0 ([kernel.kallsyms])

TRACING VIRTUAL MACHINES - GUEST CODE

A common case for KVM test programs is that the test program acts as the hypervisor, creating, running and destroying the virtual machine, and providing the guest object code from its own object code. In this case, the VM is not running an OS, but only the functions loaded into it by the hypervisor test program, and conveniently, loaded at the same virtual addresses. To support that, option "--guest-code" has been added to perf script and perf kvm report.

Here is an example tracing a test program from the kernel?s KVM selftests:

```
# perf record --kcore -e intel_pt/cyc/ -- tools/testing/selftests/kselftest_install/kvm/tsc_msrs_test
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.280 MB perf.data ]
# perf script --guest-code --itrace=bep --ns -F-period,+addr,+flags
[SNIP]
```

tsc_msrs_test 18436 [007] 10897.962087733: branches: call ffffffffc13b2ff5 __vmx_vcpu_run+0x15 (vmlinux) => ffffffffc13b2f50 vmx_update_host_rsp+0x0 (vmlinux)

tsc_msrs_test 18436 [007] 10897.962087733: branches: return ffffffffc13b2f5d vmx_update_host_rsp+0xd (vmlinux) => ffffffffc13b2ffa __vmx_vcpu_run+0x1a (vmlinux)

tsc_msrs_test 18436 [007] 10897.962087733: branches: call ffffffffc13b303b __vmx_vcpu_run+0x5b (vmlinux) => ffffffffc13b2f80 vmx_vmenter+0x0 (vmlinux)

tsc_msrs_test 18436 [007] 10897.962087836: branches: vmentry ffffffffc13b2f82 vmx_vmenter+0x2 (vmlinux) => 0 [unknown] ([unknown])

[guest/18436] 18436 [007] 10897.962087836: branches: vmentry 0 [unknown] ([unknown]) => 402c81 guest_code+0x131 (/home/user/git/work/tools/testing/selftests/kselftest_install/kvm/tsc_msrs_test)

[guest/18436] 18436 [007] 10897.962087836: branches: call 402c81 guest_code+0x131 (/home/user/git/work/tools/testing/selftests/kselftest_install/kvm/tsc_msrs_test) => 40dba0 ucall+0x0 (/home/user/git/work/tools/testing/selftests/kselftest_install/kvm/tsc_msrs_test)

[guest/18436] 18436 [007] 10897.962088248: branches: vmexit 40dba0 ucall+0x0

(/home/user/git/work/tools/testing/selftests/kselftest_install/kvm/tsc_msrs_test) => 0 [unknown] ([unknown])

tsc_msrs_test 18436 [007] 10897.962088248: branches: vmexit 0 [unknown] ([unknown]) => ffffffffc13b2fa0 vmx_vmexit+0x0 (vmlinux)

tsc_msrs_test 18436 [007] 10897.962088248: branches: jmp ffffffffc13b2fa0 vmx_vmexit+0x0 (vmlinux) => ffffffffc13b2fd2 vmx_vmexit+0x32 (vmlinux)

tsc_msrs_test 18436 [007] 10897.962088256: branches: return ffffffffc13b2fd2 vmx_vmexit+0x32 (vmlinux) => ffffffffc13b3040 __vmx_vcpu_run+0x60 (vmlinux)

tsc_msrs_test 18436 [007] 10897.962088270: branches: return ffffffffc13b30b6 __vmx_vcpu_run+0xd6 (vmlinux) => ffffffffc13b2f2e vmx_vcpu_enter_exit+0x4e (vmlinux)

[SNIP]

tsc_msrs_test 18436 [007] 10897.962089321: branches: call ffffffffc13b2ff5 __vmx_vcpu_run+0x15 (vmlinux) => ffffffffc13b2f50 vmx_update_host_rsp+0x0 (vmlinux)

tsc_msrs_test 18436 [007] 10897.962089321: branches: return ffffffffc13b2f5d vmx_update_host_rsp+0xd (vmlinux) => ffffffffc13b2ffa __vmx_vcpu_run+0x1a (vmlinux)

tsc_msrs_test 18436 [007] 10897.962089321: branches: call ffffffffc13b303b __vmx_vcpu_run+0x5b (vmlinux) => ffffffffc13b2f80 vmx_vmenter+0x0 (vmlinux)

tsc_msrs_test 18436 [007] 10897.962089424: branches: vmentry ffffffffc13b2f82 vmx_vmenter+0x2 (vmlinux) => 0 [unknown] ([unknown])

[guest/18436] 18436 [007] 10897.962089424: branches: vmentry 0 [unknown] ([unknown]) => 40dba0 ucall+0x0 (/home/user/git/work/tools/testing/selftests/kselftest_install/kvm/tsc_msrs_test)

[guest/18436] 18436 [007] 10897.962089701: branches: jmp 40dc1b ucall+0x7b (/home/user/git/work/tools/testing/selftests/kselftest_install/kvm/tsc_msrs_test) => 40dc39 ucall+0x99 (/home/user/git/work/tools/testing/selftests/kselftest_install/kvm/tsc_msrs_test)

[guest/18436] 18436 [007] 10897.962089701: branches: jcc 40dc3c ucall+0x9c (/home/user/git/work/tools/testing/selftests/kselftest_install/kvm/tsc_msrs_test) => 40dc20 ucall+0x80 (/home/user/git/work/tools/testing/selftests/kselftest_install/kvm/tsc_msrs_test)

[guest/18436] 18436 [007] 10897.962089701: branches: jcc 40dc3c ucall+0x9c (/home/user/git/work/tools/testing/selftests/kselftest_install/kvm/tsc_msrs_test) => 40dc20 ucall+0x80 (/home/user/git/work/tools/testing/selftests/kselftest_install/kvm/tsc_msrs_test)

[guest/18436] 18436 [007] 10897.962089701: branches: jcc 40dc37 ucall+0x97 (/home/user/git/work/tools/testing/selftests/kselftest_install/kvm/tsc_msrs_test) => 40dc50 ucall+0xb0 (/home/user/git/work/tools/testing/selftests/kselftest_install/kvm/tsc_msrs_test)

[guest/18436] 18436 [007] 10897.962089878: branches: vmexit 40dc55 ucall+0xb5

(/home/user/git/work/tools/testing/selftests/kselftest_install/kvm/tsc_msrs_test) => 0 [unknown] ([unknown])

tsc_msrs_test 18436 [007] 10897.962089878: branches: vmexit 0 [unknown] ([unknown]) => ffffffffc13b2fa0 vmx_vmexit+0x0 (vmlinux)

tsc_msrs_test 18436 [007] 10897.962089878: branches: jmp ffffffffc13b2fa0 vmx_vmexit+0x0 (vmlinux) => ffffffffc13b2fd2 vmx_vmexit+0x32 (vmlinux)

tsc_msrs_test 18436 [007] 10897.962089887: branches: return ffffffffc13b2fd2 vmx_vmexit+0x32 (vmlinux) => ffffffffc13b3040 __vmx_vcpu_run+0x60 (vmlinux)

tsc_msrs_test 18436 [007] 10897.962089901: branches: return ffffffffc13b30b6 __vmx_vcpu_run+0xd6 (vmlinux) => ffffffffc13b2f2e vmx_vcpu_enter_exit+0x4e (vmlinux)

[SNIP]

```
# perf kvm --guest-code --guest --host report -i perf.data --stdio | head -20
# To display the perf.data header info, please use --header/--header-only options.
#
#
# Total Lost Samples: 0
#
# Samples: 12  of event 'instructions'
# Event count (approx.): 2274583
#
# Children      Self  Command       Shared Object        Symbol
# ........  ........  ............  ...................  .........................................
#
   54.70%     0.00%  tsc_msrs_test  [kernel.vmlinux]     [k] entry_SYSCALL_64_after_hwframe
            |
            ---entry_SYSCALL_64_after_hwframe
               do_syscall_64
               |
               |--29.44%--syscall_exit_to_user_mode
               |          exit_to_user_mode_prepare
               |          task_work_run
               |          __fput
```

EVENT TRACE

Event Trace records information about asynchronous events, for example interrupts, faults,

VM exits and entries. The information is recorded in CFE and EVD packets, and also the Interrupt Flag is recorded on the MODE.Exec packet. The CFE packet contains a type field to identify one of the following:

1	INTR	interrupt, fault, exception, NMI
2	IRET	interrupt return
3	SMI	system management interrupt
4	RSM	resume from system management mode
5	SIPI	startup interprocessor interrupt
6	INIT	INIT signal
7	VMENTRY	VM-Entry
8	VMEXIT	VM-Entry
9	VMEXIT_INTR	VM-Exit due to interrupt
10	SHUTDOWN	Shutdown

For more details, refer to the Intel 64 and IA-32 Architectures Software Developer Manuals (version 076 or later).

The capability to do Event Trace is indicated by the /sys/bus/event_source/devices/intel_pt/caps/event_trace file.

Event trace is selected for recording using the "event" config term. e.g.

```
perf record -e intel_pt/event/u uname
```

Event trace events are output using the --itrace I option. e.g.

```
perf script --itrace=Ie
```

perf script displays events containing CFE type, vector and event data, in the form:

```
evt:   hw int          (t)  cfe: INTR IP: 1 vector: 3 PFA: 0x8877665544332211
```

The IP flag indicates if the event binds to an IP, which includes any case where flow control packet generation is enabled, as well as when CFE packet IP bit is set.

perf script displays events containing changes to the Interrupt Flag in the form:

```
iflag:   t                  IFLAG: 1->0 via branch
```

where "via branch" indicates a branch (interrupt or return from interrupt) and "non branch" indicates an instruction such as CFI, STI or POPF).

In addition, the current state of the interrupt flag is indicated by the presence or absence of the "D" (interrupt disabled) perf script flag. If the interrupt flag is changed, then the "t" flag is also included i.e.

```
        no flag, interrupts enabled IF=1
```

t interrupts become disabled IF=1 -> IF=0

D interrupts are disabled IF=0

Dt interrupts become enabled IF=0 -> IF=1

The intel-pt-events.py script illustrates how to access Event Trace information using a Python script.

TNT DISABLE

TNT packets are disabled using the "notnt" config term. e.g.

```
perf record -e intel_pt/notnt/u uname
```

In that case the --itrace q option is forced because walking executable code to reconstruct the control flow is not possible.

EMULATED PTWRITE

Later perf tools support a method to emulate the ptwrite instruction, which can be useful if hardware does not support the ptwrite instruction.

Instead of using the ptwrite instruction, a function is used which produces a trace that encodes the payload data into TNT packets. Here is an example of the function:

```
#include <stdint.h>
void perf_emulate_ptwrite(uint64_t x)
__attribute__((externally_visible, noipa, no_instrument_function, naked));
#define PERF_EMULATE_PTWRITE_8_BITS \
        "1: shl %rax\n"     \
        "   jc 1f\n"        \
        "1: shl %rax\n"     \
        "   jc 1f\n"        \
        "1: shl %rax\n"     \
        "   jc 1f\n"        \
        "1: shl %rax\n"     \
        "   jc 1f\n"        \
        "1: shl %rax\n"     \
        "   jc 1f\n"        \
        "1: shl %rax\n"     \
        "   jc 1f\n"        \
        "1: shl %rax\n"     \
        "   jc 1f\n"        \
        "1: shl %rax\n"     \
        "   jc 1f\n"        \
```

```
            "1: shl %rax\n"     \
            "   jc 1f\n"
    /* Undefined instruction */
    #define PERF_EMULATE_PTWRITE_UD2        ".byte 0x0f, 0x0b\n"
    #define PERF_EMULATE_PTWRITE_MAGIC      PERF_EMULATE_PTWRITE_UD2 ".ascii \"perf,ptwrite  \"\n"
    void perf_emulate_ptwrite(uint64_t x __attribute__ ((__unused__)))
    {
        /* Assumes SysV ABI : x passed in rdi */
        __asm__ volatile (
            "jmp 1f\n"
            PERF_EMULATE_PTWRITE_MAGIC
            "1: mov %rdi, %rax\n"
            PERF_EMULATE_PTWRITE_8_BITS
            PERF_EMULATE_PTWRITE_8_BITS
            PERF_EMULATE_PTWRITE_8_BITS
            PERF_EMULATE_PTWRITE_8_BITS
            PERF_EMULATE_PTWRITE_8_BITS
            PERF_EMULATE_PTWRITE_8_BITS
            PERF_EMULATE_PTWRITE_8_BITS
            PERF_EMULATE_PTWRITE_8_BITS
            "1: ret\n"
        );
    }
```

For example, a test program with the function above:

```
    #include <stdio.h>
    #include <stdint.h>
    #include <stdlib.h>
    #include "perf_emulate_ptwrite.h"
    int main(int argc, char *argv[])
    {
        uint64_t x = 0;
        if (argc > 1)
            x = strtoull(argv[1], NULL, 0);
```

```
        perf_emulate_ptwrite(x);
        return 0;
    }
```

Can be compiled and traced:

```
    $ gcc -Wall -Wextra -O3 -g -o eg_ptw eg_ptw.c
    $ perf record -e intel_pt//u ./eg_ptw 0x1234567890abcdef
    [ perf record: Woken up 1 times to write data ]
    [ perf record: Captured and wrote 0.017 MB perf.data ]
    $ perf script --itrace=ew
            eg_ptw 19875 [007]  8061.235912:      ptwrite:  IP: 0 payload: 0x1234567890abcdef      55701249a196
perf_emulate_ptwrite+0x16 (/home/user/eg_ptw)
    $
```

PIPE MODE

Pipe mode is a problem for Intel PT and possibly other auxtrace users. It?s not recommended to use a pipe as data output with Intel PT because of the following reason.

Essentially the auxtrace buffers do not behave like the regular perf event buffers. That is because the head and tail are updated by software, but in the auxtrace case the data is written by hardware. So the head and tail do not get updated as data is written.

In the Intel PT case, the head and tail are updated only when the trace is disabled by software, for example: - full-trace, system wide : when buffer passes watermark - full-trace, not system-wide : when buffer passes watermark or context switches - snapshot mode : as above but also when a snapshot is made - sample mode : as above but also when a sample is made

That means finished-round ordering doesn?t work. An auxtrace buffer can turn up that has data that extends back in time, possibly to the very beginning of tracing.

For a perf.data file, that problem is solved by going through the trace and queuing up the auxtrace buffers in advance.

For pipe mode, the order of events and timestamps can presumably be messed up.

PAUSE OR RESUME TRACING

With newer Kernels, it is possible to use other selected events to pause or resume Intel PT tracing. This is configured by using the "aux-action" config term:

"aux-action=pause" is used with events that are to pause Intel PT tracing.

"aux-action=resume" is used with events that are to resume Intel PT tracing.

"aux-action=start-paused" is used with the Intel PT event to start in a paused state.

For example, to trace only the uname system call (sys_newuname) when running the command line utility uname:

$ perf record --kcore -e intel_pt/aux-action=start-paused/k,syscalls:sys_enter_newuname/aux-action=resume/,syscalls:sys_exit_newuname/aux-action=pause/ uname

```
Linux
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.043 MB perf.data ]
$ perf script --call-trace
uname   30805 [000] 24001.058782799: name: 0x7ffc9c1865b0
uname   30805 [000] 24001.058784424:  psb offs: 0
uname   30805 [000] 24001.058784424:  cbr: 39 freq: 3904 MHz (139%)
uname   30805 [000] 24001.058784629: ([kernel.kallsyms])       debug_smp_processor_id
uname   30805 [000] 24001.058784629: ([kernel.kallsyms])       __x64_sys_newuname
uname   30805 [000] 24001.058784629: ([kernel.kallsyms])          down_read
uname   30805 [000] 24001.058784629: ([kernel.kallsyms])             __cond_resched
uname   30805 [000] 24001.058784629: ([kernel.kallsyms])             preempt_count_add
uname   30805 [000] 24001.058784629: ([kernel.kallsyms])                in_lock_functions
uname   30805 [000] 24001.058784629: ([kernel.kallsyms])             preempt_count_sub
uname   30805 [000] 24001.058784629: ([kernel.kallsyms])          up_read
uname   30805 [000] 24001.058784629: ([kernel.kallsyms])             preempt_count_add
uname   30805 [000] 24001.058784838: ([kernel.kallsyms])                in_lock_functions
uname   30805 [000] 24001.058784838: ([kernel.kallsyms])             preempt_count_sub
uname   30805 [000] 24001.058784838: ([kernel.kallsyms])          _copy_to_user
uname   30805 [000] 24001.058784838: ([kernel.kallsyms])       syscall_exit_to_user_mode
uname   30805 [000] 24001.058784838: ([kernel.kallsyms])          syscall_exit_work
uname   30805 [000] 24001.058784838: ([kernel.kallsyms])             perf_syscall_exit
uname   30805 [000] 24001.058784838: ([kernel.kallsyms])                debug_smp_processor_id
uname   30805 [000] 24001.058785046: ([kernel.kallsyms])                perf_trace_buf_alloc
uname   30805 [000] 24001.058785046: ([kernel.kallsyms])                   perf_swevent_get_recursion_context
uname   30805 [000] 24001.058785046: ([kernel.kallsyms])                      debug_smp_processor_id
uname   30805 [000] 24001.058785046: ([kernel.kallsyms])                   debug_smp_processor_id
```

```
uname   30805 [000] 24001.058785046: ([kernel.kallsyms])                perf_tp_event
uname   30805 [000] 24001.058785046: ([kernel.kallsyms])                  perf_trace_buf_update
uname   30805 [000] 24001.058785046: ([kernel.kallsyms])                    tracing_gen_ctx_irq_test
uname   30805 [000] 24001.058785046: ([kernel.kallsyms])                  perf_swevent_event
uname   30805 [000] 24001.058785046: ([kernel.kallsyms])                    __perf_event_account_interrupt
uname   30805 [000] 24001.058785046: ([kernel.kallsyms])                      __this_cpu_preempt_check
uname   30805 [000] 24001.058785046: ([kernel.kallsyms])                    perf_event_output_forward
uname   30805 [000] 24001.058785046: ([kernel.kallsyms])                      perf_event_aux_pause
uname   30805 [000] 24001.058785046: ([kernel.kallsyms])                        ring_buffer_get
uname   30805 [000] 24001.058785046: ([kernel.kallsyms])                          __rcu_read_lock
uname   30805 [000] 24001.058785046: ([kernel.kallsyms])                          __rcu_read_unlock
uname   30805 [000] 24001.058785254: ([kernel.kallsyms])                        pt_event_stop
uname   30805 [000] 24001.058785254: ([kernel.kallsyms])                          debug_smp_processor_id
uname   30805 [000] 24001.058785254: ([kernel.kallsyms])                          debug_smp_processor_id
uname   30805 [000] 24001.058785254: ([kernel.kallsyms])                          native_write_msr
uname   30805 [000] 24001.058785463: ([kernel.kallsyms])                          native_write_msr
uname   30805 [000] 24001.058785639: 0x0
```

The example above uses tracepoints, but any kind of sampled event can be used.

For example:

Tracing between arch_cpu_idle_enter() and arch_cpu_idle_exit() using breakpoint events:

```
$ sudo cat /proc/kallsyms | sort | grep ' arch_cpu_idle_enter\| arch_cpu_idle_exit'
ffffffffb605bf60 T arch_cpu_idle_enter
ffffffffb614d8a0 W arch_cpu_idle_exit
 $ sudo perf record --kcore -a -e intel_pt/aux-action=start-paused/k -e mem:0xffffffffb605bf60:x/aux-action=resume/ -e mem:0xffffffffb614d8a0:x/aux-action=pause/ -- sleep 1
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 1.387 MB perf.data ]
```

Tracing __alloc_pages() using kprobes:

```
$ sudo perf probe --add '__alloc_pages order'
Added new event:  probe:__alloc_pages  (on __alloc_pages with order)
$ sudo perf probe --add __alloc_pages%return
Added new event:  probe:__alloc_pages__return (on __alloc_pages%return)
 $ sudo perf record --kcore -aR -e intel_pt/aux-action=start-paused/k -e probe:__alloc_pages/aux-action=resume/ -e
```

probe:__alloc_pages__return/aux-action=pause/ -- sleep 1

[perf record: Woken up 1 times to write data]

[perf record: Captured and wrote 1.490 MB perf.data]

Tracing starting at main() using a uprobe event:

$ sudo perf probe -x /usr/bin/uname main

Added new event: probe_uname:main   (on main in /usr/bin/uname)

$ sudo perf record -e intel_pt/-aux-action=start-paused/u -e probe_uname:main/aux-action=resume/ -- uname

Linux

[perf record: Woken up 1 times to write data]

[perf record: Captured and wrote 0.031 MB perf.data]

Tracing occasionally using cycles events with different periods:

$ perf record --kcore -a -m,64M -e intel_pt/aux-action=start-paused/k -e cycles/aux-action=pause,period=1000000/Pk -e cycles/aux-action=resume,period=10500000/Pk -- firefox

[perf record: Woken up 19 times to write data]

[perf record: Captured and wrote 16.561 MB perf.data]

EXAMPLE

Examples can be found on perf wiki page "Perf tools support for Intel? Processor Trace":

https://perf.wiki.kernel.org/index.php/Perf_tools_support_for_Intel%C2%AE_Processor_Trace

SEE ALSO

perf-record(1), perf-script(1), perf-report(1), perf-inject(1)