

目录

- 1. 背景..... 2
 - 1.1. 基于 perf2
 - 1.2. 基于 ebpfn.....3
 - 1.3. 基础知识.....3
 - 1.3.1. 必备的内核功能.....3
 - 1.3.2. 内核模块分析.....3
 - 1.4. perf-prof3
- 2. 框架介绍..... 4
 - 2.1. 内核态.....4
 - 2.1.1. 事件源.....4
 - 2.1.2. filter5
 - 2.1.3. perf_event.....5
 - 2.2. 用户态.....5
 - 2.2.1. 基础功能.....6
 - 2.2.2. 分析单元.....6
 - 2.2.3. 联合分析.....10
- 3. 基础功能..... 11
 - 3.1. Help 帮助系统.....11
 - 3.1.1. perf-prof11
 - 3.1.2. perf-prof profiler -h11
 - 3.1.3. perf-prof -e event help13
 - 3.1.4. bash-completion14
 - 3.2. 事件指定.....14
 - 3.3. 事件的输出格式.....15
 - 3.4. Attach.....15
 - 3.4.1. Attach to CPU15
 - 3.4.2. Attach to PID/TID.....15
 - 3.4.3. Attach to workload.....15
 - 3.4.4. Attach to cgroups16
 - 3.5. 栈与火焰图.....16
 - 3.5.1. 网络丢包火焰图.....17
 - 3.5.2. CPU 性能火焰图.....17
 - 3.6. Filter 过滤器.....17
 - 3.6.1. ebpf 过滤器17
 - 3.6.2. pmu 过滤器18
 - 3.6.3. trace events 过滤器.....18
 - 3.7. Order.....18
 - 3.7.1. 排序原理.....19
 - 3.8. Convert19
 - 3.9. Expr.....19

3.10.	Event-spread	21
3.11.	Perf kprobe、uprobe.....	21
3.12.	USDT	22
4.	分析单元.....	23
4.1.	计数分析	23
4.1.1.	Stat	23
4.1.2.	Percpu-stat	24
4.1.3.	Top.....	24
4.2.	延迟分析	29
4.2.1.	Multi-trace	29
4.2.2.	Syscalls.....	41
4.2.3.	Nested-trace.....	42
4.2.4.	Rundelay.....	42
4.3.	进程	44
4.3.1.	Task-state.....	44
4.3.2.	Oncpu	46
4.4.	内存	47
4.4.1.	Kmemleak.....	47
4.4.2.	Kmemprof.....	48
4.5.	虚拟化	49
4.5.1.	Kvm-exit.....	49
4.6.	块设备	50
4.6.1.	Blktrace	50
4.7.	硬件与调试.....	50
4.7.1.	Profile	51
4.7.2.	Breakpoint.....	52
4.7.3.	Page-faults.....	53
5.	联合分析	55
5.1.	Multi-trace	56
5.2.	Trace	57
5.3.	更多的可能性	58
6.	其他主题	59
6.1.	事件丢失	59
6.2.	虚拟化场景	59
6.2.1.	时间戳对齐.....	59
6.2.2.	事件传播.....	59
6.3.	Soft lockup 分析.....	60
6.3.1.	Guest	60
6.3.2.	Host	61
6.3.3.	过滤.....	63
6.3.4.	整体逻辑.....	64

1. 背景

行为上，一个复杂问题的分析，往往是使用多个单元分析一步一步找到方向。如：经常使用的监控工具（top、mpstat、iostat、atop、Grafana）、压测工具（netperf、fio）、跟踪工具（perf、strace）、状态统计(/proc、/sys)等，都是基本的分析单元。通过多个分析单元组合使用，逐渐找到方向。如：经常使用的分析方法：USE 方法，分析各个资源的使用率、饱和度、错误等情况，找到性能瓶颈；延迟分析，各个链路拆解，找到耗时长部分。也是启动多个监控工具，经过多次单元分析，找到方向。

单元分析进入 linux 内核之后，问题会变得更加复杂。再叠加虚拟化层，问题分析更加困难。已有的一些工具 perf、systemtap、ftrace、bcc 等，在性能、安全性、易用性、兼容性等方面存在各种差异。

- ftrace 功能开关过于复杂，使用繁琐。开发新功能需要增加内核代码。
- trace-cmd 基于 ftrace。trace-cmd record/report 需要先存盘，再读取，性能不行。
- perf record/script，也需要先存盘，再读取。存盘，数据量大，性能不行，不能长时间运行。
- systemtap，是基于内核模块的，经常编译不过。内核模块是不安全的。
- perf 命令支持 perl、python 脚本扩展，数据量大时性能不行，不能长期运行。
- perf 命令功能过于庞大，不支持简单的扩展，不够模块化。
- bcc、bpftrace 工具在 3.10 内核使用不了。ebpf 不支持旧内核，兼容性不行。3.10 内核只支持 perf，不支持 ebpf。
- 内核模块，开发周期长，安全性得不到保障，导致内核故障。

我们需要一个能够长期运行的，更安全易用，兼容性更广，性能又不太差的工具。

- *长期运行*，一般是监控类工具，这类工具必须不能对系统业务产生影响。对于不能稳定复现的问题，也必须部署这类监控。对工具自身的性能有一定要求。
- *安全易用*，不能导致系统故障。使用方便，多条命令配合使用(如：perf record、perf script)，要压缩成单条命令。命令结束，要还原所有的内核配置。
- *兼容性广*，要求能支持更低版本的内核。
- *性能*，一个最基本的观察：工具自身的性能，决定定位的精度。如：tcpdump 一秒只能存 10 万个包，就不能用于定位很高 pps 的网络问题。Perf record 每秒只能记录 10 万个事件，就不能用于跟踪高事件量的内核问题。

综上，目前只能选择 perf 内核态接口。perf-prof 由此而来。

1.1. 基于 perf

原生 perf 命令不满足需求，但内核 perf_event 能力，非常简单。

perf-prof 基于 libperf 和 libtraceevent 库实现简单的分析框架，提供比 perf 更灵活的特性。

- 数据不落盘。
- 数据过滤，基于 tracepoint 的过滤机制，减少数据量。

- 数据实时处理并输出，不需要存盘后再处理。
- 兼容更多内核版本。
- 用户态实现更安全，能快速迭代。

虽然比 perf 更灵活，但 perf-prof 不能替代 perf。perf 灵活的符号处理，支持大量的 event，支持很多硬件 PMU 特性。

perf-prof 选择性的支持全部的 tracepoint 事件，以及部分常用的硬件 pmu 事件。

1.2. 基于 ebpf

ebpf 虽然不兼容旧内核，但其性能是最好的。对于有性能要求，且内核支持，perf-prof 会启用 ebpf。使用 libbpf 库来装载 bpf 程序。

1.3. 基础知识

1.3.1. 必备的内核功能

- tracepoint。perf list tracepoint 能够看到所有的 tracepoint 点。
- ftrace。工作目录：/sys/kernel/debug/tracing/。 [内核文档](#)
- perf。
- kprobe。接口：/sys/kernel/debug/tracing/kprobe_events。 [内核文档](#)
- uprobe。接口：/sys/kernel/debug/tracing/uprobe_events。 [内核文档](#)
- ebpf。

1.3.2. 内核模块分析

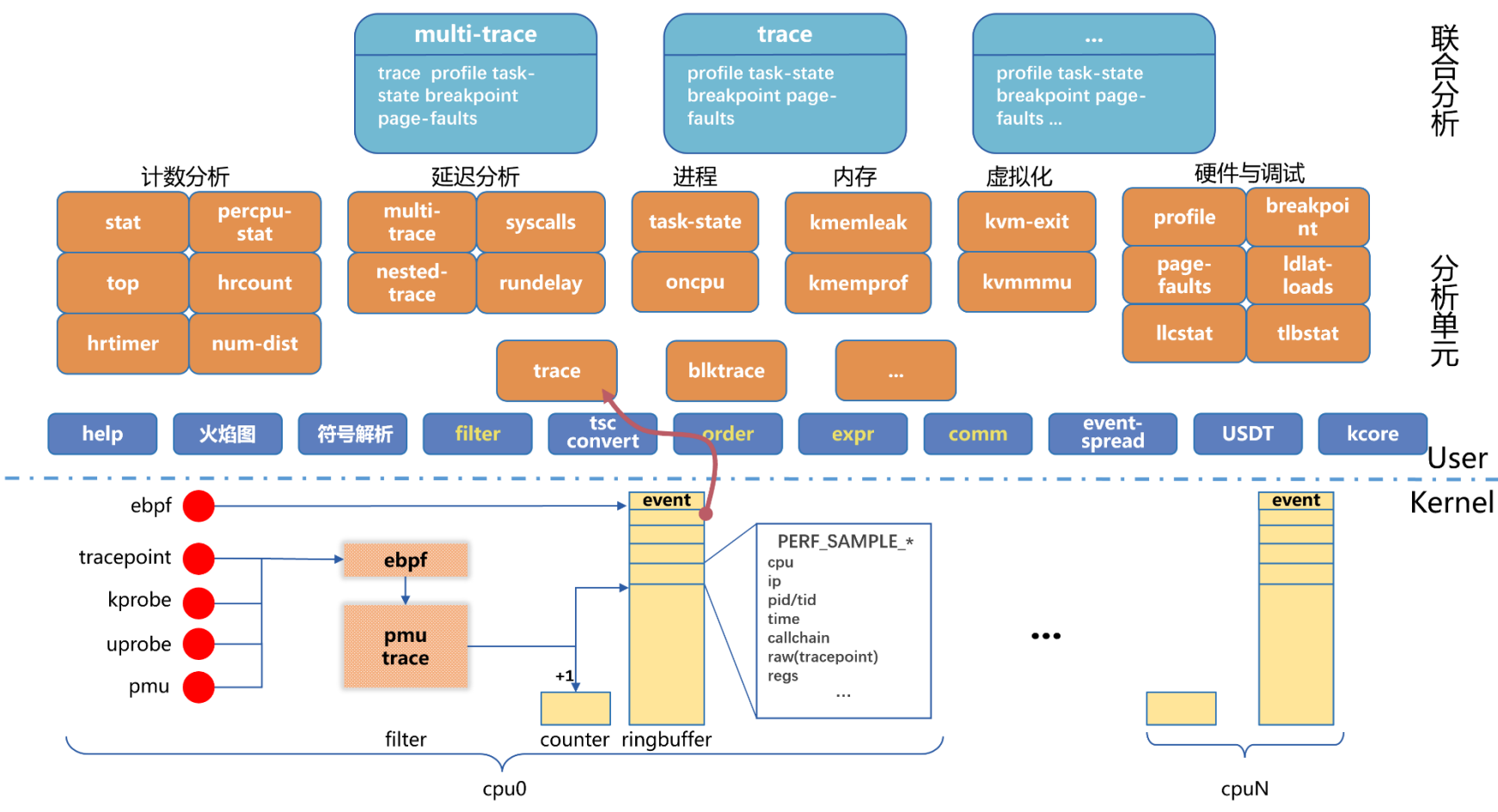
对于每一个内核模块，都应该尝试去了解原理。以及内部的 tracepoint 点、启用的内核线程、在 /proc、/sys 导出的状态。

1.4. perf-prof

代码仓库。 <https://github.com/OpenCloudOS/perf-prof>

编译。 `yum install -y xz-devel elfutils-libelf-devel && make`

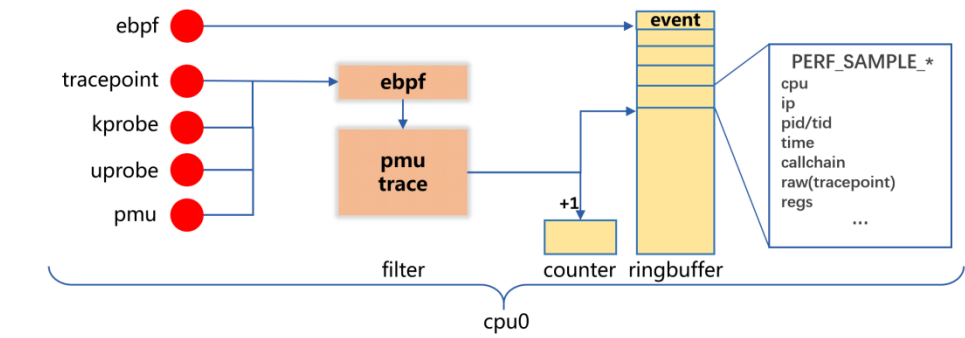
2. 框架介绍



整体框架由内核态和用户态 2 部分组成。

2.1. 内核态

内核态分为几部分：事件源，filter，perf_event。



2.1.1. 事件源

内核态的事件源目前有 5 种。

1. ebpf。bpf 程序可以调用 bpf_perf_event_output()直接往 perf 的 ringbuffer 内写入数据。数据的格式是 bpf 程序自己定义的。

- 2. tracepoint。内核代码路径上，执行到 tracepoint 点的位置，就会执行该事件点对应的函数。打开 perf_event，其对应的 tracepoint 点会被激活。
- 3. kprobe。动态 tracepoint 点。可以在内核函数的任何位置添加 kprobe 点。其功能跟 tracepoint 点一致。
- 4. uprobe。跟 kprobe 类似，作用于用户态的二进制文件。
- 5. pmu。硬件事件源。源自 CPU 内部，PMC 计数溢出后触发采样。

2.1.2. filter

tracepoint、kprobe、uprobe 事件源经过 ebpf、trace event 过滤器。pmu 事件源经过 ebpf、pmu 过滤器。

这三个过滤器，全是内核态过滤器，事件在内核直接过滤，过滤出的事件才会放到 ringbuffer，被用户态使用。

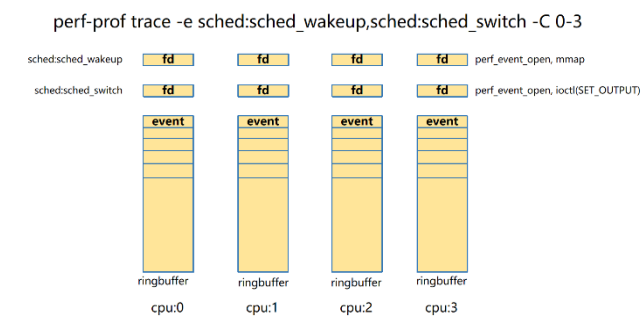
- 1. ebpf 过滤器。可以在事件源 perf_event 上添加 bpf 程序。bpf 程序返回 1，可以继续采样；bpf 程序返回 0，终止采样。
- 2. trace event 过滤器。内核可以对 tracepoint 点的字段进行过滤。只有满足条件的事件会进行下一步处理。
- 3. pmu 过滤器。仅支持 user、kernel 过滤。

2.1.3. perf_event

perf_event 内包含 counter 和 ringbuffer 两部分。

- 1. counter，计数器，对事件发生次数进行计数。
- 2. ringbuffer，环形缓冲区。用于存放过滤后的事件。采样的事件格式，由 perf_event_attr::sample_type 字段指定，由 PERF_SAMPLE_*宏定义各个位的含义。包含基本的 CPU、pid、tid、时间戳、堆栈、raw、寄存器等信息。

perf_event 使用 perf_event_open 系统调用打开，并返回文件描述符。**读**文件描述符得到计数器。对文件描述符进行 **mmap** 得到 ringbuffer，ringbuffer 大小由 mmap 映射长度决定。内核态往 ringbuffer 写入数据，用户态从 ringbuffer 读取数据。用户态读取不及时，ringbuffer 满之后，再写入的事件都会丢失，内核会生成 lost 记录，记录丢失的事件量。对文件描述符进行 **ioctl**(PERF_EVENT_IOC_SET_OUTPUT)可以设置 perf_event 采样的事件输出到别的 perf_event 的 ringbuffer 上。

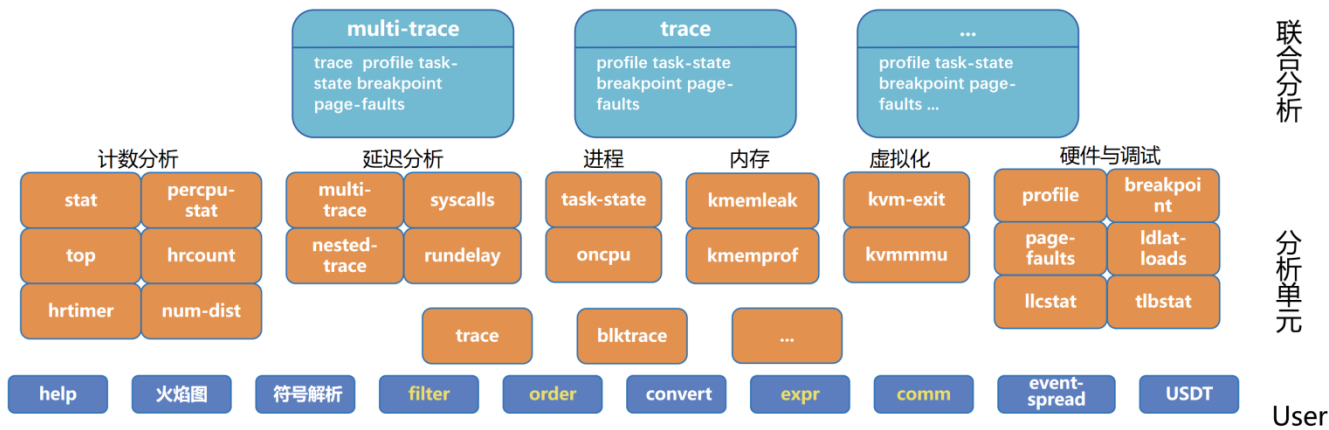


ringbuffer 是共用的，每个 cpu 或每个 tid 打开的所有事件共用一个 ringbuffer。如：在 cpu0 上打开 sched:sched_wakeup，sched:sched_switch 两个事件，其共用 cpu0 的 ringbuffer，但计数器是独立的。内部打开事件的过程是，先打开 sched:sched_wakeup 事件，得到文件描述符，映射 ringbuffer；再打开 sched:sched_switch，得到文件描述符，设置为 sched:sched_wakeup 的 ringbuffer。

参考：man perf_event_open

2.2. 用户态

用户态分为 3 部分：基础功能、分析单元、联合分析。



2.2.1. 基础功能

基础功能给分析单元提供基础服务。

1. Help。帮助系统，每个分析单元都有独立的帮助系统。
2. 火焰图。栈处理的一种形式，直接输出折叠栈格式。
3. 符号解析。根据 ip 获取符号名。支持内核态符号和用户态符号解析。
4. Filter。过滤器。
5. Order。事件排序，对采样的事件按时间排序。
6. Tsc convert。事件时间由 ns 转换为 tsc 时间戳。用于虚拟化场景。
7. Expr。表达式编译器。
8. Comm。Pid->comm 映射服务。
9. Event-spread。事件传播，事件可以通过网络发送和接收。用于虚拟化场景。
10. USDT。用户态 trace 格式，用于添加 uprobe 点。

2.2.2. 分析单元

分析单元分为一些大类：计数分析、延迟分析、进程、内存、虚拟化、硬件与调试、块设备、trace。

每一个分析单元都是一个独立的 profiler(剖析器)。基于 profiler 可以创建很多独立的 prof_dev 设备，不同的 prof_dev 设备可以使用相同的 profiler 也可以使用不同的 profiler。prof_dev 提供 profiler 的工作环境。

```
1. typedef struct monitor {
2.     struct monitor *next;
3.     const char *name; // 名字
4.     const char **desc; // 描述, 用于-h 帮助
5.     const char **argv; // 参数列表, 用于-h 帮助
6.     const char *compgen; // bash-completion
7.     int pages; // 默认 ringbuffer 大小
8.     bool order; // default enable order
9. }
```

```

10. void (*help)(struct help_ctx *ctx);
11.
12. int (*argc_init)(int argc, char *argv[]);
13.
14. int (*init)(struct prof_dev *dev);
15. int (*filter)(struct prof_dev *dev);
16. void (*enabled)(struct prof_dev *dev);
17. void (*deinit)(struct prof_dev *dev);
18. void (*sigusr)(struct prof_dev *dev, int signum);
19. void (*interval)(struct prof_dev *dev);
20.
21. // Profiler minimum event time. tsc or ns.
22. u64 (*minevtime)(struct prof_dev *dev);
23.
24. // return 0:continue; 1:break;
25. int (*read)(struct prof_dev *dev, struct perf_evsel *evsel,
26.             struct perf_counts_values *count, int instance);
27.
28. /* PERF_RECORD_* */
29.
30. //PERF_RECORD_SAMPLE = 9,
31. void (*sample)(struct prof_dev *dev, union perf_event *event,
32.               int instance);
33.} profiler;
34.
35./*
36. * Profiler device
37. * Contains sampling ringbuffer, environment, timer, profiler-
38. * specific memory, convert, order, etc.
39.**/
40.struct prof_dev {
41.    profiler *prof; // 指向profiler
42.    struct perf_cpu_map *cpus; // profiler 工作在哪些cpu 上
43.    struct perf_thread_map *threads; // profiler 工作在哪些线程上
44.    struct perf_evlist *evlist; // profiler 打开事件列表
45.    struct timer timer; // 间隔定时器
46.    struct env *env; // 指定的选项参数
47.    void *private; // profiler 私有内存
48.    int pages; // ringbuffer 大小
49.    struct perf_event_convert {
50.        // tsc convert
51.        bool need_tsc_conv; // 是否可以转换为tsc
52.        struct perf_tsc_conversion tsc_conv;
53.        char *event_copy;
54.    } convert;
55.    struct order_ctx {
56.        profiler *base;

```



```

57.     profiler order; // order 排序
58.     struct ordered_events oe; // 排序缓存的事件
59. } order;
60. struct prof_dev_links {
61.     /*
62.      * profiler[/option/ATTR/ATTR/...] opens the device as a
63.      * child prof_dev. Will be linked to the parent device.
64.      */
65.     struct prof_dev *parent;
66.     struct list_head child_list;
67.     struct list_head link_to_parent;
68. } links; // 用于联合分析
69. struct perf_event_forward_to {
70.     struct prof_dev *target;
71.     struct list_head source_list;
72.     struct list_head link_to_target;
73.     struct perf_record_dev *event_dev;
74. } forward; // 用于联合分析
75. };

```

Profiler 与 prof_dev 之间的关系，可以看做驱动与设备、类与对象。

2.2.2.1. profiler

每个剖析器，都需要定义一个 profiler 结构。

- name 定义 profiler 的名字。
- desc 定义帮助信息，包含 profiler 的基本原理，可用参数，一些示例等。
- argv 定义选项参数，目前所有的 profiler 共用一套选项参数，单个 profiler 启用的参数只是其中一个子集。这个字段用于选择 profiler 使用到的选项参数。
- pages 用于设置 profiler 工作时，perf ringbuffer 的默认大小，只能是 2 的幂次。
- init 初始化。分配 profiler 私有内存并赋初值，存到 prof_dev::private 字段。同时打开事件，并挂接到 prof_dev::evlist 链表上。事件可以是 profiler 内部定义的，也可以是通过选项参数传递的。
- filter 过滤器。对打开的事件设置过滤条件。可以不指定。
- deinit 销毁。销毁 profiler 私有内存结构。
- interval 间隔输出。用于指定间隔输出 profiler 内解析的数据。
- read 计数器。给 profiler 传递读到的计数器值。由 prof_dev 的间隔定时器周期性读取后传递到 read 接口。
- sample 事件。给 profiler 传递采样事件，并由 profiler 内部来处理。由 prof_dev 读取 ringbuffer 的事件后传递到 sample 接口。

2.2.2.2. prof_dev

prof_dev 提供 profiler 的工作环境。prof_dev 由命令行自动打开。

```
perf-prof trace -e sched:sched_wakeup -N 10
```

打开一个 prof_dev 设备，为 trace 提供工作环境。

每个 prof_dev 都是事件驱动的，不需要独立的线程。prof_dev 内的定时器、perf_event 全都是文件描述符，会加到 epoll 内，进行事件监听。由此，可以保证 prof_dev 相互独立，可以同时打开多个 prof_dev 设备。

Comm 服务，提供 pid->comm 的映射关系，就是一个内部定义的 prof_dev 设备，会默认打开，在后台持续收集 comm 信息，直到 perf-prof 命令退出。

2.2.2.3. trace

trace 是最基本的分析单元。直接显示采样的事件，不做任何其他处理。一般用于 profiler 开发初期，直接观察事件的原始信息，并配合脚本处理事件。对于事件量比较少时，可以直观显示。

2.2.2.4. 计数分析

1. stat。间隔输出事件的计数器。
2. percpu-stat。精选好的一些事件，不需要指定。
3. top。对事件的某些字段进行 top 分析。把字段的所有可能值拆分成独立的计数器，并按从大到小显示。
4. hrcount。高精度计数器。可以观察到 ms 粒度事件的发生次数。用于事件发生密度分析，是否有一定的集中性。
5. hrtimer。高精度条件采样。采样间隔内，事件发生一定次数时，输出采样的堆栈。
6. num-dist。数值分布。事件的某字段的分布情况，最小值，平均值，p99，最大值。

2.2.2.5. 延迟分析

1. multi-trace。多功能分析，主要用于事件延迟分析，并确定延迟的中间细节。
2. syscalls。系统调用耗时分析。基于 multi-trace。只分析 sys_enter->sys_exit 之间的延迟。
3. nested-trace。嵌套的耗时分析。用于函数的发生关系分析，并统计每个函数的耗时。
4. rundelay。调度延迟分析。基于 multi-trace，自动设置 filter。

2.2.2.6. 进程

1. task-state。进程状态。可以统计进程 R, RD, S, D 等状态的分布情况。
2. oncpu。进程运行在哪些 cpu 上。cpu 上运行过哪些进程。

2.2.2.7. 内存

1. kmemleak。内存泄露分析。支持内核态多种内存分配器，以及用户态的内存分配器。
2. kmemprof。内存分配热点分析。能够采集到哪些路径会密集分配内存。

2.2.2.8. 虚拟化

- 1. kvm-exit。虚拟化退出耗时。
- 2. kvmmmu。跟踪 mmu page 的分配和建立过程。

2.2.2.9. 块设备

- 1. blktrace。跟踪块设备 request 在每个阶段的耗时。

2.2.2.10. 硬件与调试

收集常用的硬件 PMU 事件。

- 1. profile。指定频率采样。Cpu-cycles 事件。硬件 PMU 采样 NMI 中断，不会受到关中断影响。
- 2. breakpoint。硬件断点。可以捕获对某个虚拟地址的读、写、执行。如：全局变量被修改。
- 3. page-faults。缺页异常。跟踪系统发生的缺页异常。
- 4. ldlat-loads。采样内存访问延迟。基于 PEBS。
- 5. llcstat。L3 缓存状态。命中率。
- 6. tlbstat。Tlb 状态。命中率。

2.2.3.联合分析

联合分析，是指把多个分析单元联合起来一起参与分析，使其成为一个整体。

当前只有 multi-trace、trace 可以作为联合分析的主体，接受其他分析单元的事件。

2.2.3.1. multi-trace

multi-trace

trace profile task-state breakpoint page-faults

multi-trace 可以接受 trace、profile、task-state、breakpoint、page-faults 这几个。只有这些 profiler 是有意义的，其他的暂时未涉及。

这些 profiler 的事件会送到 multi-trace 内部，跟 multi-trace 自身的事件混合到一起，参与延迟分析。

2.2.3.2. trace

trace

profile task-state breakpoint page-faults

trace 可以接受 profile、task-state、breakpoint、page-faults 这几个。只有这些 profiler 是有意义的，其他的暂时未涉及。

在 trace 内部统一排序之后输出显示。

3. 基础功能

基础功能使用 trace 这个最基本的 profiler 来演示。

3.1.Help 帮助系统

合理使用帮助系统，能够快速的拼写命令，极大提升效率。

3.1.1.perf-prof

观察所有可用的 profiler。

```
[root@kvm ~]# perf-prof

Usage: perf-prof profiler [PROFILER OPTION...] [help] [cmd [args...]]
       or: perf-prof --symbols /path/to/bin

Profiling based on perf_event and ebpf

Available Profilers:
tlbstat      dTLB state on x86 platform.
breakpoint   Kernel/user-space hardware breakpoint facility.
expr         Expression compiler and simulator.
usdt         User Statically-Defined Tracing.
kvmmmu       Observe the kvm_mmu_page mapping on x86 platforms.
stat         Periodic counter with lower resolution.
hrcount      High-resolution counter.
event-lost   Determine if any events are lost.
irq-off      Detect the hrtimer latency to determine if the irq is off.
hrtimer      High-resolution conditional timing sampling.
page-faults  Print the user mode regs and stack when a page fault occurs.
ldlat-stores PEBS Data Address Profiling on Intel Platform.
ldlat-loads  Load Latency Performance Monitoring on Intel Platform.
help         Helps writing profiler commands, event attrs, event filters.
oncpu        Determine which processes are running on which CPUs.
rundelay     Schedule rundelay.
nested-trace Nested-event trace: delay, call, call-delay.
syscalls     Syscalls latency analysis.
kmemprof     Memory allocation profile. Both user and kernel allocators are supported.
multi-trace  Multipurpose trace: delay, pair, kmemprof, syscalls.
blktrace     Track IO latency on block devices.
top          Display key-value counters in top mode.
sched-migrate Monitor system process migrations.
llcstat      Last level cache state on x86 platform.
num-dist     Numerical distribution. Get 'num' data from the event itself.
kvm-exit     Count the delay from kvm_exit to kvm_entry.
percpu-stat  Handpicked event statistics.
kmemleak     Memory leak analysis. Both user and kernel allocators are supported.
watchdog     Detect hard lockup and soft lockup.
task-state   Trace task state, wakeup, switch, INTERRUPTIBLE, UNINTERRUPTIBLE.
signal       Demo
trace        Trace events and print them directly.
cpu-util     Report CPU utilization for guest or host.
profile      Sampling at the specified frequency to profile high CPU utilization.
split-lock   Split-lock on x86 platform.

See 'perf-prof profiler -h' for more information on a specific profiler.
```

3.1.1.1. 版本号

perf-prof -V 显示版本号

3.1.2.perf-prof profiler -h

每个 profiler 支持的选项参数都不一样，只能使用-h 列出的参数，使用其他参数会报错。

```
[root@kvm ~]# perf-prof trace -h

Usage: perf-prof trace [OPTION...] -e EVENT [--overwrite] [-g [--flame-graph file [-i INT]]]
```

```
Trace events and print them directly.

EXAMPLES
perf-prof trace -e sched:sched_wakeup -C 0 -g
perf-prof trace -e sched:sched_wakeup,sched:sched_switch --overwrite

OPTION:
-C, --cpus <cpu[-cpu],...> Monitor the specified CPU, Dflt: all cpu
-p, --pids <pid,...> Attach to processes
-t, --tids <tid,...> Attach to threads
--cgroups <cgroup,...> Attach to cgroups, support regular expression.
--inherit Child tasks do inherit counters.
--watermark <0-100> Wake up perf-prof watermark.
-i, --interval <ms> Interval, Unit: ms
-o, --output <file> Output file name
--order Order events by timestamp.
-m, --mmap-pages <pages> Number of mmap data pages and AUX area tracing mmap pages
-N, --exit-N <N> Exit after N events have been sampled.
--tsc Convert perf clock to tsc.
--kvmclock <uuid> Convert perf clock to Guest's kvmclock.
--clock-offset <n> Sum with clock-offset to get the final clock.
--usage-self <ms> Periodically output the CPU usage of perf-prof itself, Unit: ms
-V, --version Version info
-v, --verbose be more verbose
-q, --quiet be more quiet
-h, --help Give this help list

FILTER OPTION:
--user-callchain include user callchains, no- prefix to exclude
--kernel-callchain include kernel callchains, no- prefix to exclude

PROFILER OPTION:
-e, --event <EVENT,...> Event selector. use 'perf list tracepoint' to list available tp events.
EVENT,EVENT,...
EVENT: sys:name[/filter/ATTR/ATTR/...]
profiler[/option/ATTR/ATTR/...]
kprobe:func[/filter/ATTR/ATTR/...]
uprobe:func@"file"[/filter/ATTR/ATTR/...]
filter: trace events filter
ATTR:
stack: sample_type PERF_SAMPLE_CALLCHAIN
max-stack=int : sample_max_stack
alias=str: event alias
exec=EXPR: a public expression executed by any profiler
EXPR:
C expression. See `perf-prof expr -h` for more information.
-g, --call-graph Enable call-graph recording
--flame-graph <file> Specify the folded stack file.
--overwrite use overwrite mode
```

选项参数分为三部分：

3.1.2.1. OPTION

这部分是公共的选项参数，这些参数的含义对每个 profiler 都一样。

```
-i, --interval <ms> Interval, Unit: ms
-o, --output <file> Output file name
-m, --mmap-pages <pages> Number of mmap data pages and AUX area tracing mmap pages
-N, --exit-N <N> Exit after N events have been sampled.
--usage-self <ms> Periodically output the CPU usage of perf-prof itself, Unit: ms
-v, --verbose be more verbose
```

- i 指定输出间隔。单位：ms。
- o 输出到文件，stdout 和 stderr 输出都会输出到同一个文件。
- N 只处理 N 个事件。一般用于少量测试。
- usage-self 周期性输出 perf-prof 的 cpu 和内存消耗。
- v 冗余输出。-v 输出内部调试信息。-vv 输出采样的事件。-vvv 全部输出。

◆ ringbuffer 大小

-m 调整 ringbuffer 大小，必须是 2 的幂次，单位是页。

可以设置很大的 ringbuffer，如：-m 65536。ringbuffer 越大 perf-prof 占用的内存就越多。

```
2024-02-18 12:19:38.243733 rundelay: lost 3836 events on CPU #24
```

当出现 lost 事件时，应调大 ringbuffer，保证结果的正确性。

3.1.2.2. FILTER OPTION

这部分是过滤器选项。有部分 profiler 没有。

3.1.2.3. PROFILER OPTION

这部分是 profiler 特有的选项。不同的 profiler 可能会复用同一个选项参数，其含义也可能会不同。

◆ event

-e 用于指定事件，目前有 2 种类型：

```
-e, --event <EVENT,...>      Event selector. use 'perf list tracepoint' to list available tp events.
                                EVENT,EVENT,...
                                EVENT: sys:name[/filter/ATTR/ATTR/.../]
                                      profiler[/option/ATTR/ATTR/.../]
                                      kprobe:func[/filter/ATTR/ATTR/.../]
                                      uprobe:func@"file"[/filter/ATTR/ATTR/.../]
                                filter: trace events filter # 参考 3.6.3 见下方
                                ATTR:
```

sys:name：指定 tracepoint、kprobe、uprobe。

profiler：指定 profiler，用于联合分析。

kprobe:func：指定 perf kprobe 事件。kretprobe:func：指定 perf kretprobe 事件。

uprobe:func@"file"：指定 perf uprobe 事件。uretprobe:func@"file"：指定 perf uretprobe 事件。

Perf kprobe、uprobe 事件，不需要通过 ftrace 接口创建，由 perf_event 打开时自动创建，perf_event 关闭时自动销毁。但不能指定 kprobe、uprobe 函数的参数，如果需要请使用 ftrace 接口创建。

3.1.3. perf-prof -e event help

-e event help 显示事件帮助。help 可以加到命令的末尾。

```
[root@kvm ~]# perf-prof -e sched:sched_wakeup help

perf-prof expr -e "sched:sched_wakeup/." {expression} [-C .] [-p .] [-m .] [-v]
perf-prof stat -e "sched:sched_wakeup/.[alias=.]" [--period ns] [-C .] [-i .] [-m .] [-v]
perf-prof hrcount -e "sched:sched_wakeup/.[alias=.]" [--period ns] [-C .] [-i .] [-m .] [-v]
perf-prof hrtimer -e "sched:sched_wakeup/." [--period ns] [-F freq] [-g] [-C .] [--order] [-m .]
perf-prof top -e "sched:sched_wakeup/.[alias=./top-by=./top-add=./key=./comm=.]" [-k .] [-C .] [-p .] [-i .] [-m .] [-v]
perf-prof num-dist -e "sched:sched_wakeup./num=./alias=./[stack/]" [--perins] [--than .] [--heatmap .] [-C .] [-p .] [-i .] [-v]
perf-prof trace -e "sched:sched_wakeup/.[stack/]" [-g] [--flame-graph .] [-C .] [-p .] [-i .] [--order] [-m .]

sched:sched_wakeup
name: sched_wakeup
ID: 340
format:
    field:unsigned short common_type;      offset:0;      size:2; signed:0;
    field:unsigned char common_flags;      offset:2;      size:1; signed:0;
    field:unsigned char common_preempt_count; offset:3;      size:1; signed:0;
    field:int common_pid; offset:4;      size:4; signed:1;

    field:char comm[16]; offset:8;      size:16; signed:1;
    field:pid_t pid; offset:24;      size:4; signed:1;
    field:int prio; offset:28;      size:4; signed:1;
    field:int success; offset:32;      size:4; signed:1;
    field:int target_cpu; offset:36;      size:4; signed:1;
```

```
print fmt: "comm=%s pid=%d prio=%d success=%d target_cpu=%03d", REC->comm, REC->pid, REC->prio, REC->success, REC->target_cpu
```

事件的帮助，输出分 2 部分：

前面的部分，是所有可以使用该事件的 profiler，以及可以使用的选项参数，事件属性等。

后面的部分，是事件的字段定义，用于属性指定，或者 [trace events 过滤器](#)。

trace -e event help 显示特定 profiler 的事件帮助。

```
[root@kvm ~]# perf-prof trace -e sched:sched_wakeup help

perf-prof trace -e "sched:sched_wakeup/./[stack/]" [-g] [--flame-graph .] [-C .] [-p .] [-i .] [--order] [-m .]

sched:sched_wakeup
name: sched_wakeup
ID: 340
format:
  field:unsigned short common_type;      offset:0;      size:2; signed:0;
  field:unsigned char common_flags;      offset:2;      size:1; signed:0;
  field:unsigned char common_preempt_count; offset:3;      size:1; signed:0;
  field:int common_pid; offset:4;      size:4; signed:1;

  field:char comm[16]; offset:8;      size:16; signed:1;
  field:pid_t pid; offset:24;      size:4; signed:1;
  field:int prio; offset:28;      size:4; signed:1;
  field:int success; offset:32;      size:4; signed:1;
  field:int target_cpu; offset:36;      size:4; signed:1;

print fmt: "comm=%s pid=%d prio=%d success=%d target_cpu=%03d", REC->comm, REC->pid, REC->prio, REC->success, REC->target_cpu
```

3.1.4. bash-completion

利用 bash-comp 可以在书写命令时自动补齐。按[TAB]键自动补齐，按[TAB][TAB]键列出所有可用选项。

```
[root@kvm ~]# perf-prof t[TAB][TAB]
task-state tlstat top trace
[root@kvm ~]# perf-prof trace --t[TAB][TAB]
--tids <tid,...> --tsc
```

在 Host 上执行 yum install bash-completion 之后可以使用。

3.2. 事件指定

```
-e, --event <EVENT,...>      Event selector. use 'perf-prof list' to list available tp events.
                                EVENT,EVENT,...
                                EVENT: sys:name[/filter/ATTR/ATTR/.../]
                                      profiler[/option/ATTR/ATTR/.../]
                                      kprobe:func[/filter/ATTR/ATTR/.../]
                                      uprobe:func@"file"[/filter/ATTR/ATTR/.../]
                                filter: trace events filter
```

通过-e 选项指定事件，事件之间通过逗号分隔，sys:name 指定 tracepoint 事件，使用 perf-prof list 列出所有的事件。事件后紧跟的/filter/是过滤器，用于筛选事件。之后紧跟的/ATTR/是属性。

事件支持通配符模式。通配符会展开为具体事件名。如果指定了过滤器和属性，它们会被保留并应用到所有匹配的事件上。

```
通配符支持:
* : 匹配任意字符序列
? : 匹配任意单个字符
[] : 匹配字符集合中的任意字符

perf-prof trace -e "sched:*"
perf-prof trace -e "sched:sched_wake*//stack/"
perf-prof trace -e "sched:*,kmem:kmallocc"
perf-prof trace -e "sched:sched_switch*,timer:*"

注意: 使用 -v 选项可以查看通配符展开的详细信息，显示哪些事件匹配了通配符模式。
```

kprobe:function 和 uprobe:function@"file"指定 [kprobe、uprobe 事件](#)。

3.3. 事件的输出格式

```
[root@kvm ~]# perf-prof trace -e sched:sched_wakeup -N 1
2024-03-01 16:57:17.8210141 swapper2 03 dNh.4 [000]5 5523186.4998296: sched:sched_wakeup7: hcbs_agent_mcd:108955 [100] success=1 CPU:0008
```

¹事件发生的真实时间。²线程名。³线程 id。⁴中断和软中断情况。⁵事件发生的 cpu。⁶事件内核时间戳。⁷事件的 sys:name。⁸事件各字段的值。

中断和软中断情况：*d* 表示关中断。*N* 表示 NEED_RESCHED。*H* 表示存在硬中断和软中断。*h* 表示存在硬中断。*s* 表示存在软中断。*.* 表示没有软中断，也没有硬中断。

3.4. Attach

perf-prof 使用一些公共参数来控制 perf_event 附加到 CPU、线程、cgroup 上。每条 perf-prof 命令都必须选择一个 Attach 选项。

```
OPTION:
-C, --cpus <cpu[-cpu],...> Monitor the specified CPU, Dflt: all cpu
-p, --pids <pid,...> Attach to processes
-t, --tids <tid,...> Attach to threads
--cgroups <cgroup,...> Attach to cgroups, support regular expression.
```

可以附加到 cpu、pid、tid、cgroup。都不指定，默认是附加到所有 cpu 上，等同 -C 0-N，N 是最大 cpu 号。

3.4.1. Attach to CPU

附加到 CPU，只监控指定的 CPU 上发生的事件。perf_event 会跟随 cpu 的 online 和 offline 启用和停止。只要 cpu 是 **online** 的，这个 cpu 上的 perf_event 就会持续启用，持续采样事件。

```
[root@kvm ~]# perf-prof trace -e sched:sched_wakeup -N 1
2024-03-01 16:57:17.821014 swapper 0 dNh. [000] 5523186.499829: sched:sched_wakeup: hcbs_agent_mcd:108955 [100] success=1 CPU:000
[root@kvm ~]# perf-prof trace -e sched:sched_wakeup -C 1-3,5 -N 2
2024-03-01 16:57:39.206367 swapper 0 dNs. [002] 5523207.885203: sched:sched_wakeup: kworker/2:1H:227245 [100] success=1 CPU:002
2024-03-01 16:57:39.208370 swapper 0 dNs. [005] 5523207.887205: sched:sched_wakeup: kworker/5:2H:26095 [100] success=1 CPU:005
```

3.4.2. Attach to PID/TID

附加到 PID/TID，只监控目标线程上发生的事件。perf_event 会跟随目标线程的**调度**启用和停止。目标线程切换到 cpu 上执行时，其线程上附加的 perf_event 会启用，开始采样事件。目标线程睡眠，放弃 cpu 时，其 perf_event 会停止采样事件。

由 perf-prof 打开 perf_event，并跟随目标线程来启用和停止，每个线程的每个事件都会打开一个 perf_event。目标线程终止，其所有 perf_event 会轮询到 EPOLLHUP 事件。perf-prof 跟踪的所有目标线程都退出，perf-prof 会自动终止。

附加到 PID，会读取该 pid 下的所有线程，转换成附加到 tid。

```
[root@kvm ~]# perf-prof trace -e sched:sched_stat_runtime -p 205660 -N 2
2024-03-01 17:15:13.315910 CPU 0/KVM 206643 d... [049] 5524261.995755: sched:sched_stat_runtime: comm=CPU 0/KVM pid=206643 runtime=56309 [ns] vruntime=976637489 [ns]
2024-03-01 17:15:13.320899 CPU 1/KVM 206644 d... [001] 5524262.000744: sched:sched_stat_runtime: comm=CPU 1/KVM pid=206644 runtime=45940 [ns] vruntime=402766090 [ns]
[root@kvm ~]# perf-prof trace -e sched:sched_stat_runtime -t 691,53233 -N 2
2024-03-01 17:12:42.055376 ksm d 691 d... [024] 5524110.735077: sched:sched_stat_runtime: comm=ksmd pid=691 runtime=9513 [ns] vruntime=2146799438298794 [ns]
2024-03-01 17:12:42.055404 ksm d 691 d.h. [024] 5524110.735105: sched:sched_stat_runtime: comm=ksmd pid=691 runtime=17765 [ns] vruntime=2146799438353096 [ns]
```

3.4.3. Attach to workload

附加到 workload，监控 workload 执行过程中的事件。perf-prof 会通过 fork、execvp 来执行 workload，并得到 workload 的 pid，转换成附加到 pid。可以使用--强制分隔 perf-prof 的参数和 workload 的参数。

```
[root@kvm ~]# perf-prof trace -e sched:sched_switch -- sleep 0.1
2024-03-01 17:17:45.198376 perf-exec 26781 d... [000] 5524413.878368: sched:sched_switch: perf-exec:26781 [120] R ==> migration/0:9 [0]
2024-03-01 17:17:45.198993 sleep 26781 d... [072] 5524413.878984: sched:sched_switch: sleep:26781 [120] S ==> sap1016:152697 [120]
```


3.4.4. Attach to cgroups

附加到 cgroups，监控 cgroup 内所有进程发生的事件。如果附加的 pid 太多，可以把这些 pid 放到 perf_event cgroup 内，附加到该 cgroup，就能够监控到所有这些进程的事件。

perf_event cgroup 需要手动把需要观察的进程放进去。

```
# Example 1:
mkdir /sys/fs/cgroup/perf_event/prof
echo 205835 > /sys/fs/cgroup/perf_event/prof/cgroup.procs
cat /proc/205835/cgroup | grep perf_event
5:perf_event:/prof
perf-prof trace -e sched:sched_stat_runtime --cgroups 'prof' # prof
perf-prof trace -e sched:sched_stat_runtime --cgroups 'pro.*' # 正则表达式
```

```
# Example 2:
mkdir /sys/fs/cgroup/perf_event/prof1
echo 205845 > /sys/fs/cgroup/perf_event/prof1/cgroup.procs
perf-prof trace -e sched:sched_stat_runtime --cgroups 'prof,prof1'
perf-prof trace -e sched:sched_stat_runtime --cgroups 'prof*' # prof, prof1
```

cgroup 的指定相对于 **/sys/fs/cgroup/perf_event/**目录，同时可以使用正则表达式，匹配多个 perf_event cgroup。

3.5. 栈与火焰图

栈的处理方式各种各样，如 perf top 风格的栈处理，火焰图风格的栈处理。

perf-prof 目前支持的栈处理。

- 1) 栈及符号打印。用 **callchain_ctx** 表示，定义了栈的打印风格，可控制内核态、用户态、地址、符号、偏移量、dso、正向栈、反向栈。每个栈帧的分隔符、栈的分隔符。
- 2) key-value 栈。以栈做为 key，可以过滤重复栈，并能唯一寻址 value。用 **key_value_paires** 结构表示，一般相同的栈都有类似的作用，如内存分配栈，可以分析相同的栈分配的总内存量。类似于 gperftools 提供的 HEAPCHECKE 功能，最后报告的内存泄露是以栈为基准的。
- 3) 火焰图。把相同的栈聚合到一起。用 **flame_graph** 结构表示，能够生成折叠栈格式：反向栈、每个函数以";"分隔、末尾是栈的数量。使用 [flamegraph.pl](#) 生成火焰图。例子：swapper;start_kernel;rest_init;cpu_idle;default_idle;native_safe_halt 1。

选项参数

```
--user-callchain      include user callchains, no- prefix to exclude
--kernel-callchain    include kernel callchains, no- prefix to exclude
-g, --call-graph       Enable call-graph recording
--flame-graph <file>  Specify the folded stack file.
```

--user-callchain 打开用户态堆栈，--no-user-callchain 关闭用户态堆栈。--kernel-callchain 打开内核态堆栈，--no-kernel-callchain 关闭内核态堆栈。

```
[root@kvm ~]# perf-prof trace -e sched:sched_switch//stack/ -g -N 1 -t 205660
2024-03-01 17:23:09.556559  qemu-system-x86_205660 d... [001] 5524738.236860: sched:sched_switch: qemu-system-x86:205660 [120] S ==> swapper/1:0 [120]
ffffffff816c51a3 __schedule+0x4f3 ([kernel.kallsyms])
ffffffff816c5609 schedule+0x29 ([kernel.kallsyms])
ffffffff816c4722 schedule_hrttimeout_range_clock+0xb2 ([kernel.kallsyms])
ffffffff816c47d3 schedule_hrttimeout_range+0x13 ([kernel.kallsyms])
ffffffff81228dc5 poll_schedule_timeout+0x55 ([kernel.kallsyms])
ffffffff8122a34d do_sys_poll+0x4cd ([kernel.kallsyms])
ffffffff8122a753 sys_ppoll+0x1b3 ([kernel.kallsyms])
ffffffff816d13fe tracesys+0xe3 ([kernel.kallsyms])
00007febfa5d5d8f ppoll+0x8f (/usr/lib64/libc-2.17.so)
000000000768872 os_host_main_loop_wait+0xb2 (/usr/local/bin/qemu-system-x86_64)
000000000768af5 main_loop_wait+0x85 (/usr/local/bin/qemu-system-x86_64)
0000000005e2483 main_loop+0x33 (/usr/local/bin/qemu-system-x86_64)
00000000047bfff main+0x12ff (/usr/local/bin/qemu-system-x86_64)
00007febfa504555 __libc_start_main+0xf5 (/usr/lib64/libc-2.17.so)
```

使用-g 选项或者 stack 属性，都可以打开堆栈。能输出内核态堆栈和用户态堆栈，但不是所有的 profiler 都默认输出用户态堆栈。

```
[root@kvm ~]# perf-prof trace -e sched:sched_wakeup -g --flame-graph sched -N 10
2024-02-20 17:01:05.229961 perf-prof 116837 dNh. [024] 4659413.146525: sched:sched_wakeup: pal_session:55490 [100] success=1 CPU:024
2024-02-20 17:01:05.234539 perf-prof 116837 dNh. [024] 4659413.146527: sched:sched_wakeup: vpc_agent:69049 [120] success=1 CPU:072
2024-02-20 17:01:05.234609 perf-prof 116837 dNh. [024] 4659413.146598: sched:sched_wakeup: pal_session:55490 [100] success=1 CPU:024
2024-02-20 17:01:05.235050 perf-prof 116837 dNh. [024] 4659413.146669: sched:sched_wakeup: pal_session:55490 [100] success=1 CPU:024
2024-02-20 17:01:05.235111 perf-prof 116837 dNh. [024] 4659413.146739: sched:sched_wakeup: pal_session:55490 [100] success=1 CPU:024
2024-02-20 17:01:05.235157 perf-prof 116837 dNh. [024] 4659413.146810: sched:sched_wakeup: pal_session:55490 [100] success=1 CPU:024
2024-02-20 17:01:05.235223 perf-prof 116837 dNh. [024] 4659413.146813: sched:sched_wakeup: vpc_agent:58181 [120] success=1 CPU:024
2024-02-20 17:01:05.235271 perf-prof 116837 dNh. [024] 4659413.146883: sched:sched_wakeup: pal_session:55490 [100] success=1 CPU:024
2024-02-20 17:01:05.235325 perf-prof 116837 dNh. [024] 4659413.146957: sched:sched_wakeup: pal_session:55490 [100] success=1 CPU:024
2024-02-20 17:01:05.235852 perf-prof 116837 dNh. [024] 4659413.146959: sched:sched_wakeup: pal_ctrl:55492 [100] success=1 CPU:024
To generate the flame graph, running THIS shell command:

flamegraph.pl sched.folded > sched.svg
```

使用-g --flame-graph file 选项生成火焰图。原先在 stdout 直接输出栈，切换成火焰图之后，不会再输出栈，而是会在命令结束时输出火焰图折叠栈文件，文件以 **file.folded** 命名。使用 flamegraph.pl 最终生成 svg 火焰图。

在命令结束时，会提示生成火焰图的命令。

```
[root@kvm ~]# flamegraph.pl sched.folded > sched.svg
[root@kvm ~]# ls sched.svg
sched.svg
```

3.5.1. 网络丢包火焰图

```
perf-prof trace -e skb:kfree_skb -g --flame-graph kfree_skb -m 128 #监控丢包

#每 600 秒间隔输出火焰图
perf-prof trace -e skb:kfree_skb -g --flame-graph kfree_skb -i 600000 -m 128

flamegraph.pl --reverse kfree_skb.folded > kfree_skb.svg
```

3.5.2. CPU 性能火焰图

```
#采样内核态 CPU 利用率的火焰图
perf-prof profile -F 1000 -C 0,1 --exclude-user -g --flame-graph profile

#每 600 秒间隔输出火焰图
perf-prof profile -F 1000 -C 0,1 --exclude-user -g --flame-graph profile -i 600000

#生成 15:46:33 秒开始 600 秒的火焰图
grep "15:46:33" profile.folded | flamegraph.pl > profile.svg
```

3.6. Filter 过滤器

目前支持 3 类过滤器：ebpf 过滤器、pmu 过滤器、ftrace 过滤器。

这三个过滤器都是内核态过滤器，只有过滤出的事件才会存到 ringbuffer。合理使用过滤器，可以减少用户态处理的事件量，降低 perf-prof 自身的 cpu 消耗。

3.6.1. ebpf 过滤器

内核 perf_event 可以通过 **ioctl**(PERF_EVENT_IOC_SET_BPF)来设置 bpf 程序。bpf 程序返回 1，可以继续采样；bpf 程序返回 0，终止采样。依据这样的策略，给每个 perf_event 增加一个过滤器。过滤掉不满足条件的事件。

当前支持 5 个 ebpf 过滤器。

- 1) --irqs_disabled，判断中断是否关闭。--irqs_disabled，--irqs_disabled=1 中断关闭继续采样，中断打开终止采样。--irqs_disabled=0 中断打开继续采样，中断关闭终止采样。

- 2) --tif_need_resched, 判断 TIF_NEED_RESCHED 标记是否设置。--tif_need_resched, --tif_need_resched=1 标记设置继续采样, 标记未设置终止采样。--tif_need_resched=0 标记未设置继续采样, 标记设置终止采样。
- 3) --nr_running_min, --nr_running_max, 判断 runqueue 中 nr_running 进程的数量。满足 nr_running_min <= nr_running <= nr_running_max 条件继续采样, 否则终止采样。
- 4) --exclude_pid, 过滤掉进程 pid。当前进程等于 PID 终止采样, 否则继续采样。

3.6.2. pmu 过滤器

内核 perf 框架默认会带一些简单的过滤器, 主要是基于 perf_event_attr 属性来设置。

当前支持 4 个 pmu 过滤器。

- 1) --exclude-guest, 过滤掉 guest 模式。
- 2) --exclude-host, 过滤掉 host, 只采样 guest。一般用于硬件 PMU。
- 3) --exclude-kernel, 过滤掉内核态。
- 4) --exclude-user, 过滤掉用户态。

3.6.3. trace events 过滤器

每个 tracepoint 事件都可以设置 trace events 过滤器, 这个在内核执行, 使用事件字段作为变量。

```
[root@kvm ~]# perf-prof trace -e 'sched:sched_wakeup/pid<10/' -N 5
2024-02-20 16:37:52.803432      swapper      0 dNs. [000] 4658020.718621: sched:sched_wakeup: ksoftirqd/0:3 [120] success=1 CPU:000
2024-02-20 16:37:52.815405      swapper      0 dNs. [000] 4658020.730621: sched:sched_wakeup: ksoftirqd/0:3 [120] success=1 CPU:000
2024-02-20 16:37:52.826069      swapper      0 dNs. [000] 4658020.741301: sched:sched_wakeup: ksoftirqd/0:3 [120] success=1 CPU:000
2024-02-20 16:37:53.199170      swapper      0 dNs. [000] 4658021.114245: sched:sched_wakeup: ksoftirqd/0:3 [120] success=1 CPU:000
2024-02-20 16:37:53.239853      vstationd 105401 dN.. [000] 4658021.155079: sched:sched_wakeup: migration/0:9 [0] success=1 CPU:000
```

pid<10 为过滤器, 过滤所有 pid 小于 10 的事件。pid 字段来自事件帮助。参考 3.1.3 见上方

过滤器可以使用的运算符: &&, ||, ==, !=, <, <=, >, >=, &, ~。详细参考内核文档: [5. Event filtering](#)

除了事件的字段外, 内核还默认支持 2 个字段: CPU, COMM, cpu, comm。当事件存在 cpu、comm 字段时, 使用大写的 CPU、COMM。

```
[root@kvm ~]# perf-prof trace -e sched:sched_wakeup/CPU==0/ -N 2
2024-12-06 09:58:48.080745      ps      57415 dNs. [000] 3326918.181235: sched:sched_wakeup: ksoftirqd/0:3 [120] success=1 CPU:000
2024-12-06 09:58:48.080851      ps      57415 dNh. [000] 3326918.181341: sched:sched_wakeup: sshd:55809 [120] success=1 CPU:000
[root@kvm ~]# perf-prof trace -e sched:sched_wakeup/COMM==atop/ -N 2
2024-12-06 10:02:03.353338      atop     9666 d.h. [072] 3327112.996179: sched:sched_wakeup: pal_session:134512 [100] success=1 CPU:072
2024-12-06 10:02:03.353732      atop     9666 d.s. [072] 3327112.996571: sched:sched_wakeup: rcu_sched:11 [120] success=1 CPU:072
```

部分过滤器无法在内核执行时, 会由 perf-prof 在用户态执行, 此时会增加 2 个有用字段: __cpu, __pid, 用来扩充事件字段。__pid 指定进程 id, 并非线程 id。

```
[root@kvm ~]# perf-prof trace -e sched:sched_wakeup/__cpu==3/ -N 2
sched:sched_wakeup filters ' __cpu==3' in userspace
2025-11-14 15:25:03.968027      swapper      0 dNh. [003] 1557646.373729: sched:sched_wakeup: perf-prof:298735 [120] success=1 CPU:003
2025-11-14 15:25:03.968096      perf-prof 298735 d... [003] 1557646.373798: sched:sched_wakeup: kworker/u16:0:288151 [120] success=1 CPU:000
```

__cpu==3 会提示在用户态执行。使用这 2 个字段, 必定会在用户态执行过滤器, 会影响性能, 尽量避免这么用。

3.7. Order

```
--order          Order events by timestamp.
```

默认 profiler 是不启用排序的，输出的事件就不是按时间排序的。这主要是由于多个 ringbuffer 导致的，单个 ringbuffer 内的事件都是有序的，多个 ringbuffer 的事件不是有序的。在处理时，会一次处理完一个 ringbuffer，再处理下一个，这样的处理顺序，导致事件输入 profiler 时是乱序。

```
[root@kvm /data]# perf-prof trace -e sched:sched_wakeup -N 1000
2024-02-20 19:24:21.035817      virsh 84247 dNh. [024] 4668008.960201: sched:sched_wakeup: pal_session:55490 [100] success=1 CPU:024
2024-02-20 19:24:21.035834      virsh 84247 dNh2 [024] 4668008.960270: sched:sched_wakeup: pal_session:55490 [100] success=1 CPU:024
2024-02-20 19:24:21.035852      virsh 84247 dNh2 [024] 4668008.960342: sched:sched_wakeup: pal_session:55490 [100] success=1 CPU:024
2024-02-20 19:24:21.035864      fping 84249 d.h. [048] 4668008.959811: sched:sched_wakeup: perf-prof:84227 [120] success=1 CPU:048
2024-02-20 19:24:21.035877      fping 84249 dNh. [048] 4668008.959904: sched:sched_wakeup: hcbs_cli_acce0:58199 [100] success=1 CPU:048
2024-02-20 19:24:21.035892      fping 84249 dNs1 [048] 4668008.959927: sched:sched_wakeup: ksoftirqd/48:344 [120] success=1 CPU:048
...
2024-02-20 19:24:21.042673      perf-prof 84227 d... [048] 4668008.967215: sched:sched_wakeup: perf-prof:84227 [120] success=1 CPU:048
2024-02-20 19:24:21.042686      perf-prof 84227 d... [048] 4668008.967229: sched:sched_wakeup: perf-prof:84227 [120] success=1 CPU:048
2024-02-20 19:24:21.042698      perf-prof 84227 d... [048] 4668008.967240: sched:sched_wakeup: perf-prof:84227 [120] success=1 CPU:048
2024-02-20 19:24:21.042714      swapper 0 dNh. [072] 4668008.959922: sched:sched_wakeup: sap1016:152682 [120] success=1 CPU:072
2024-02-20 19:24:21.042731      swapper 0 dNh. [072] 4668008.960253: sched:sched_wakeup: exec_agent:57312 [120] success=1 CPU:072
2024-02-20 19:24:21.042745      swapper 0 dNh. [072] 4668008.960256: sched:sched_wakeup: time_ticker:70562 [139] success=1 CPU:072
```

从[024]切换到[048]时，时间从 4668008.960342 切换到 4668008.959811，不是顺序的。从[048]切换到[072]时，也不是顺序的。

指定--order 参数，可以使事件按时间排序。

```
[root@kvm /data]# perf-prof trace -e sched:sched_wakeup -N 1000 --order
2024-02-20 19:29:26.035667      swapper 0 dNh. [024] 4668313.938878: sched:sched_wakeup: pal_ctrl:55492 [100] success=1 CPU:024
2024-02-20 19:29:26.035757      swapper 0 dNh. [024] 4668313.938881: sched:sched_wakeup: pal_session:55490 [100] success=1 CPU:024
2024-02-20 19:29:26.035797      swapper 0 dNh. [048] 4668313.938888: sched:sched_wakeup: perf-prof:117527 [120] success=1 CPU:048
2024-02-20 19:29:26.035813      swapper 0 dNh. [024] 4668313.938958: sched:sched_wakeup: pal_session:55490 [100] success=1 CPU:024
2024-02-20 19:29:26.035828      swapper 0 dNh. [048] 4668313.938966: sched:sched_wakeup: perf-prof:117527 [120] success=1 CPU:048
2024-02-20 19:29:26.035903      swapper 0 dNh. [024] 4668313.939019: sched:sched_wakeup: pal_main:55444 [100] success=1 CPU:024
2024-02-20 19:29:26.035920      swapper 0 dNh. [024] 4668313.939022: sched:sched_wakeup: iscsid:50190 [110] success=1 CPU:024
2024-02-20 19:29:26.035933      swapper 0 dNh. [024] 4668313.939023: sched:sched_wakeup: pal_session:55490 [100] success=1 CPU:024
```

3.7.1. 排序原理

单个 ringbuffer 内的事件默认是有序的。利于单个 ringbuffer 的有序性，排序只需要取多个 ringbuffer 的首个事件，执行最小堆排序。弹出最小的事件后，再补充对应 ringbuffer 的下一个事件，堆化后再次弹出最小事件。直到所有 ringbuffer 变空。

最小堆排序，是直接对 ringbuffer 的事件执行排序，可以避免事件拷贝；同时事件在所有 ringbuffer 上是均匀消耗，可以有效减少事件 lost。

3.8. Convert

```
--tsc          Convert perf clock to tsc.
--kvmclock <uuid> Convert perf clock to Guest's kvmclock.
--clock-offset <n> Sum with clock-offset to get the final clock.
```

默认事件的时间戳是 ns 为单位的，使用的是内核的 local_clock()，这个时间戳默认是由 tsc 转换得到的。因此，可以在 perf-prof 内再转换回 tsc 时钟。使用--tsc 选项可以把事件的时间戳调整为 tsc 时钟。使用--kvmclock 可以把事件的时间戳转换为 Guest 时钟。使用--clock-offset <n> 选项，可以给转换后的时钟增加偏移。

时钟转换，主要用于虚拟化场景，参考 6.2.1 见下方。

3.9. Expr

表达式编译器。perf-prof 内部经常会需要利用表达式来做一些简单的计算。

```
perf-prof top -e 'sched:sched_stat_runtime//top-by="runtime/1000"/alias=run(us)/' -C 0 -i 1000
perf-prof top -e 'skb:kfree_skb//key=protocol/comm=k$ksymbol(Location)/' -m 32
perf-prof top -e 'sched:sched_process_exec//comm="(char *)&common_type+filename_offset/' --only-comm
perf-prof kmemleak --alloc 'kmem:mm_page_alloc//ptr=page/size=4096<order>/stack/' --free kmem:mm_page_free//ptr=page/stack/ -m 256 --order
```

在这些简单计算中，有些是为了转换单位 (ns->us, order->bytes)，有些把地址转换为符号 (ip->ksymbol)，还可以获取__data_loc 动态变量对应的字符串。经过计算之后的返回值，再进一步用于 profiler 的内部处理。这些表达式是在用户态计算的，先编译成汇编指令，再模拟执行汇编

指令，得到结果。汇编指令集是个最简集合，非 x86、arm 指令集。

表达式可以使用事件字段、__cpu、__pid 作为全局变量：perf-prof -e event help 查看事件字段；__cpu 指定事件的当前 cpu；__pid 指定事件的当前进程 id，而事件字段 common_pid 指定的是线程 id。

trace 可以利用 **exec** 属性来执行表达式。

```
[root@kvm ~]# perf-prof trace -e 'sched:sched_wakeup//exec="printf("PID=%d %s\n", pid*10, prio>120?"LO PRIO":"HI PRIO")"/' -N 2
2024-02-21 10:00:12.590996          swapper      0 dNh. [000] 4720560.564667: sched:sched_wakeup: vpc_sflow_agent:70632 [139] success=1 CPU:000
PID=706320 LO PRIO
2024-02-21 10:00:12.591086          swapper      0 dN.. [048] 4720560.564669: sched:sched_wakeup: perf-prof:28183 [120] success=1 CPU:048
PID=281830 HI PRIO
```

打印 pid*10，并判断优先级。

执行 **perf-prof expr -h** 能够显示表达式支持的运算符和内建函数。目前所有的 C 运算符都支持，另外支持多个内建函数：printf、ksymbol、ntohl、ntohs、strncmp、comm_get、syscall_name、exit_reason_str、system。同时支持扩展运算符：~ 执行通配符匹配，== != 支持字符串的判断。

```
[root@kvm ~]# perf-prof expr -h
...
Operators
Precedence  Operator      Description
1           ++ --          Suffix/postfix increment and decrement
            ()          Function call
            []          Array subscripting
2           ++ --          Prefix increment and decrement
            + -          Unary plus and minus
            ! ~          Logical NOT and bitwise NOT
            (type)       Cast
            *            Indirection (dereference)
            &            Address-of
            sizeof       Size-of
3           * / %        Multiplication, division, and remainder
4           + -          Addition and subtraction
5           << >>         Bitwise left shift and right shift
6           < <=         For relational operators < and <= respectively
            > >=         For relational operators > and >= respectively
            ~            Extended Operator: Wildcard pattern matching
7           == !=        For relational = and != respectively
            Supports both numeric and string comparison
8           &            Bitwise AND
9           ^            Bitwise XOR (exclusive or)
10          |            Bitwise OR (inclusive or)
11          &&           Logical AND
12          ||           Logical OR
13          ?:           Ternary conditional
14          =            Simple assignment
15          ,            Comma

Built-in Functions
int printf(char *fmt, args...)
    Prints args according to fmt, and return the number of characters
    printed, args can take up to 6 variable parameters.

char *ksymbol(long addr)
    Get the kernel symbol name according to addr, and return a string.

int ntohl(int netlong)
short ntohs(short netshort)
    These functions convert network byte order to host byte order.

int strncmp(const char *s1, const char *s2, long n)
    Compare two strings.

char *comm_get(int pid)
    Get the comm string of pid

char *syscall_name(int id)
    Get the syscall name by id, and return a string.

char *exit_reason_str(int isa, int val)
    Get the KVM exit reason string according to ISA and exit reason value.
    ISA: 1=VMX, 2=SVM, 3=ARM. Returns a string describing the exit reason.

int system(const char *format, ...)
    Format a command string using printf-style formatting and execute it via system().
    Returns the command's exit status. Supports up to 6 additional parameters.
    WARNING: Be cautious with untrusted input to avoid command injection.

Extended Operator (C++-style operator overloading)
```

```

~ (const char *str, const char *pattern)
    This operator overloads the bitwise NOT operator to provide trace event filter's ~ functionality.
    Supports *, ?, and character classes [abc], [a-z], [^abc], [!abc].
    Returns 1 if str matches pattern, 0 otherwise.

==(const char *s1, const char *s2)
!=(const char *s1, const char *s2)
    When both operands are string pointers (char *), == and != operators
    perform string comparison instead of pointer comparison.
    Returns 1 for true, 0 for false.
    Examples:
        comm == "systemd"          - Check if process name equals systemd
        prev_comm == next_comm     - Compare two string fields

```

```

PROFILER OPTION:
-e, --event <EVENT,...>      Event selector. use 'perf list tracepoint' to list available tp events.
                                EVENT,EVENT,...
                                EVENT: sys:name[/filter/ATTR/ATTR/.../]
                                    profiler[/option/ATTR/ATTR/.../]
                                filter: trace events filter
                                ATTR:
                                    exec=EXPR: a public expression executed by any profiler
                                    top-by=EXPR: add to top, sort by this field
                                    top-add=EXPR: add to top
                                    comm=EXPR: top, show COMM
                                    ptr=EXPR: kmemleak, ptr field, Dflt: ptr=ptr
                                    size=EXPR: kmemleak, size field, Dflt: size=bytes_alloc
                                    num=EXPR: num-dist, num field
                                    key=EXPR: key for multiple events: top, multi-trace
                                EXPR:
                                    C expression. See `perf-prof expr -h` for more information.

```

3.10. Event-spread

事件传播，可以把事件通过网络传递到另外一个系统上去处理。目前主要用于虚拟化场景。用于把 Guest 事件传递到 Host 来处理。

```

-e, --event <EVENT,...>      Event selector. use 'perf list tracepoint' to list available tp events.
                                EVENT,EVENT,...
                                EVENT: sys:name[/filter/ATTR/ATTR/.../]
                                    profiler[/option/ATTR/ATTR/.../]
                                filter: trace events filter
                                ATTR:
                                    push=[IP]:PORT: push events to the local broadcast server IP:PORT
                                    push=chardev: push events to chardev, e.g., /dev/virtio-ports/*
                                    push=file: push events to file
                                    pull=[IP]:PORT: pull events from server IP:PORT
                                    pull=chardev: pull events from chardev
                                    pull=file: pull events from file

```

主要通过 push 和 pull 属性来传播事件。目前支持：[IP:]PORT, chardev, file。

```

[root@kvm ~]# perf-prof trace -e sched:sched_wakeup//push=9900/ -m 128
Listen at 0.0.0.0:9900
Accept client 127.0.0.1:35258
Client hangs up 127.0.0.1:35258

[root@kvm ~]# perf-prof trace -e sched:sched_wakeup//pull=9900/ -N 5
Connected to 127.0.0.1:9900
2024-02-21 10:17:10.830698 G swapper 0 dNh. [000] 4721578.471508: sched:sched_wakeup: hcbs_cli_iscs0:58198 [100] success=1 CPU:000
2024-02-21 10:17:10.830808 G swapper 0 dNh. [000] 4721578.471627: sched:sched_wakeup: time_ticker:70562 [139] success=1 CPU:000
2024-02-21 10:17:10.830887 G swapper 0 dNh. [000] 4721578.471912: sched:sched_wakeup: sap1016:152682 [120] success=1 CPU:000
2024-02-21 10:17:10.830963 G swapper 0 dNh. [000] 4721578.472124: sched:sched_wakeup: iscsid:50190 [110] success=1 CPU:000
2024-02-21 10:17:10.831039 G swapper 0 dNh. [000] 4721578.472563: sched:sched_wakeup: hcbs_cli_iscs0:58198 [100] success=1 CPU:000
Disconnect from 127.0.0.1:9900

```

push=9900, 会监听 9900 端口，并把 sched:sched_wakeup 事件发送给所有连接到 9900 端口的客户端。pull=9900, 连接到 9900 端口，从服务端获取事件。G 标记，表示事件来自 Guest。

push=chardev 属性目前只能支持 virtio-serial 字符设备，主要用于 Guest 和 Host 通信。

3.11. Perf kprobe、uprobe

通过 perf kprobe、uprobe 可以快速跟踪内核态和用户态函数。同时可以避免残留 ftrace kprobe、uprobe 事件。

```

[root@VM ~]# perf-prof trace -e kprobe:try_to_wake_up/CPU==1/ -N 2
2024-12-06 10:29:40.872085 swapper 0 d.h. [001] 16590546.108897: kprobe:try_to_wake_up: (0xfffffffff810ba2a0)
2024-12-06 10:29:40.872106 YDService 2026448 ... [001] 16590546.108918: kprobe:try_to_wake_up: (0xfffffffff810ba2a0)
[root@VM ~]# perf-prof trace -e 'kretprobe:try_to_wake_up/common_pid>0/' -N 2

```



```
2024-12-06 10:30:18.612689      ps 3532435 d... [015] 16590583.849505: kretprobe:try_to_wake_up: (0xffffffff810ba942 <- 0xffffffff810ba2a0)
2024-12-06 10:30:18.613277      ps 3532435 d.s. [015] 16590583.850093: kretprobe:try_to_wake_up: (0xffffffff810ba8a5 <- 0xffffffff810ba2a0)
[root@VM ~]# perf-prof trace -e 'uprobe:printf@"/lib64/libc.so.6"' -N 2
2024-12-06 10:26:48.717332      perf-prof 3528770 .... [006] 16590373.954121: uprobe:printf: (0x7f110de5d8b0)
2024-12-06 10:26:48.717361      perf-prof 3528770 .... [006] 16590373.954151: uprobe:printf: (0x7f110de5d8b0)
[root@VM ~]# perf-prof trace -e 'uretprobe:printf@"/lib64/libc.so.6"' -N 2
2024-12-06 10:27:04.863339      perf-prof 3528859 .... [005] 16590390.100132: uretprobe:printf: (0x42711f <- 0x7efe4fe518b0)
2024-12-06 10:27:04.863362      perf-prof 3528859 .... [005] 16590390.100155: uretprobe:printf: (0x41a98d <- 0x7efe4fe518b0)
```

只有新版本内核支持。同时，perf kprobe、uprobe 事件也可以使用 trace events 过滤器，支持 help 帮助。

3.12. USDT

usdt 是用户态进程静态导出的 trace 点，编译之后存放在 *.note.stapsdt* section 中。解析该 section，创建出 uprobe 就可以跟踪用户态执行。

目前提供 3 个功能：

- 1) list，列出 elf 文件中的 usdt。
- 2) add，利用 usdt 添加 uprobe 点，通过 profinder:name 方式来使用。
- 3) del，删除已添加的 uprobe 点。

```
# Example:
perf-prof usdt add libc:memory_malloc_retry@/usr/lib64/libc.so.6 -v
perf-prof trace -e libc:memory_malloc_retry
```

当前已支持 x86 和 arm64 平台。

[Exploring USDT Probes on Linux](#)

4. 分析单元

分析单元大致分成 2 类：

- 内建事件分析单元：不需要使用 -e 指定事件，一般使用 ebpf、硬件 pmu 作为内建事件源，也可以使用 tracepoint 作为事件源。
- 指定事件分析单元：需要使用 -e 等选项指定事件。

通过 -h 帮助能够区分，没有 -e 选项的使用内建事件。

4.1. 计数分析

4.1.1. Stat

Stat 用于统计事件的发生次数。需要指定 tracepoint、kprobe、uprobe 事件源。

选项参数

```
-e, --event <EVENT,...>      Event selector. use 'perf list tracepoint' to list available tp events.
                                EVENT,EVENT,...
                                EVENT: sys:name[/filter/ATTR/ATTR/.../]
                                profiler[/option/ATTR/ATTR/.../]
                                filter: trace events filter
--period <ns>                 Sample period, Unit: s/ms/us/*ns
--perins                       Print per instance stat
```

```
[root@kvm ~]# perf-prof stat -e sched:sched_wakeup,sched:sched_switch,raw_syscalls:sys_enter
2024-02-21 14:57:43.999022
|sched_wakeup|sched_switch|sys_enter|
|50071      |92755      |319521     |
2024-02-21 14:57:44.999142
|sched_wakeup|sched_switch|sys_enter|
|54114      |101337     |213022     |
```

统计 sched_wakeup, sched_switch, sys_enter 事件发生次数。默认输出间隔：1 秒。默认附加到所有 CPU 上。

```
[root@kvm ~]# perf-prof stat -e sched:sched_wakeup,sched:sched_switch,raw_syscalls:sys_enter --period 200ms
2024-02-21 15:01:53.420094
|sched_wakeup |sched_switch |sys_enter   |
|9560  10397 9488  9556  11748|18244 19403 17912 17567 22612|18699 29814 24491 76547 40879|
2024-02-21 15:01:54.420150
|sched_wakeup |sched_switch |sys_enter   |
|9552  10111 9797  9503  10435|18399 19586 18531 18236 20161|17211 16476 30952 13824 29270|
```

--period 选项指定计数周期。200ms 读取一次计数器值，输出间隔：1 秒。每个输出间隔内，计数小于等于 5 次，会压缩显示。

```
[root@kvm ~]# perf-prof stat -e sched:sched_wakeup,sched:sched_switch,raw_syscalls:sys_enter --period 100ms -i 1000
2024-02-21 15:08:19.576123
  sched:sched_wakeup  5048| 4911| 5730| 5677| 5600| 5606| 5218| 5750| 7830| 6062 | total 57432
  sched:sched_switch  9717| 9348|10647|10457| 9832| 9831| 9339|10524|14335|11287 | total 105317
raw_syscalls:sys_enter 7125|10153|15032|12449|29675|34296|33333|22198|46557|13945 | total 224763
```

--interval 设置计数间隔：1000ms。计数周期：100ms。一个计数间隔内会输出 10 次计数，会直接显示。

```
[root@kvm ~]# perf-prof stat -e sched:sched_wakeup,sched:sched_switch,raw_syscalls:sys_enter --period 100ms -i 1000 -C 0-1 --perins
2024-02-21 15:12:22.384888
[000]  sched:sched_wakeup  624|  505|  480|  514|  466|  493|  433|  371| 1288|  786 | total 5960
[000]  sched:sched_switch  884|  858|  852|  909|  829|  754|  640|  748| 1647| 1480 | total 9601
[000] raw_syscalls:sys_enter 17890| 1270| 3198| 8467| 1650| 3111| 1259| 683|10205| 2158 | total 49891
```



```
[001] sched:sched_wakeup 17| 17| 16| 18| 15| 20| 22| 18| 17| 18| total 178
[001] sched:sched_switch 28| 28| 28| 29| 25| 30| 35| 28| 28| 28| total 287
[001] raw_syscalls:sys_enter 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| total 0
```

-C 设置附加到 CPU。-p 设置附加到进程。--perins 输出每个实例的计数情况，附加到 CPU，实例就是 CPU，附加到进程，实例就是每个线程。

```
[root@kvm ~]# perf-prof stat -e 'sched:sched_wakeup/target_cpu==0/cpus="0,2"/,sched:sched_switch//cpus=1/' --period 200ms -i 1000 -C 0-3 --perins
2024-02-21 16:55:58.578773
[CPU] | sched_wakeup | sched_switch |
[000] | 1087 926 1128 945 615 | |
[001] | | | | | 84 74 187 91 80 |
[002] | 0 0 0 0 0 | |
```

设置 trace events 过滤器和 cpus 属性。sched_wakeup 只过滤 target_cpu==0 的事件，且只附加到 cpus=0,2 上。sched_switch 只附加到 cpus=1 上。

◆ 原理

--period 100ms -i 1000，以 100ms 为周期性读取计数器，读取 10 次之后输出一次显示。如果--period 设置的非常低，会导致性能问题，perf-prof 自身的 cpu 消耗会升高。

◆ 应用场景

统计事件的发生次数，需要了解有哪些事件。如：虚拟化场景，统计 kvm:kvm_exit 的发生次数，统计特定退出原因的发生次数。

4.1.2. Percpu-stat

percpu-stat 也显示事件的计数，但不需要指定事件，内部精选了一些事件。

◆ 内建事件

```
PERF_COUNT_SW_CONTEXT_SWITCHES PERF_COUNT_SW_CPU_MIGRATIONS
PERF_COUNT_SW_PAGE_FAULTS_MIN PERF_COUNT_SW_PAGE_FAULTS_MAJ
irq:irq_handler_entry irq:softirq_entry
timer:hrtimer_expire_entry workqueue:workqueue_execute_start
kvm:kvm_exit net:netif_receive_skb
net:net_dev_xmit net:napi_gro_receive_entry
kmem:mm_page_alloc compaction:mm_compaction_migratepages
migrate:mm_migrate_pages vmscan:mm_vmscan_direct_reclaim_begin
writeback:wbc_writepage filemap:mm_filemap_add_to_page_cache
raw_syscalls:sys_enter power:cpu_idle
```

```
[root@kvm ~]# perf-prof percpu-stat -C 0-1
2024-02-21 17:08:22.683186
[CPU] SOFT csw cpu-mig minflt majflt hardirq softirq hrtimer workqueue KVM exit NET recv xmit gro MM alloc compact reclaim migrate PAGE cache WB pages idle
[000] 10147 401 9728 0 102 1683 5016 127 0 0 8 65 3691 0 0 0 2 0 5204
[001] 249 3 0 0 0 340 110 103 67 1 1 0 0 0 0 0 0 0 250
[ALL] 10396 404 9728 0 102 2023 5126 230 67 1 9 65 3691 0 0 0 2 0 5454
```

--syscalls 选项可以看到系统调用次数。

4.1.3. Top

Top，对事件字段进行计数分析。把字段的所有可能值拆分成独立的计数器，并按大到小排列显示。

选项参数

```
-e, --event <EVENT,...> Event selector. use 'perf list tracepoint' to list available tp events.
                        EVENT,EVENT,...
                        EVENT: sys:name[/filter/ATTR/ATTR/.../]
                        profiler[/option/ATTR/ATTR/.../]
```

```

filter: trace events filter
ATTR:
  top-by=EXPR: add to top, sort by this field
  top-add=EXPR: add to top
  comm=EXPR: top, show COMM
  key=EXPR: key for multiple events: top, multi-trace
EXPR:
  C expression. See `perf-prof expr -h` for more information.
-k, --key <str>      Key for series events
--only-comm          top: only show comm but not key

```

使用 tracepoint、kprobe、uprobe 作为事件源。事件源本身的计数可以使用 stat 来统计。假设 sched:sched_wakeup 事件，通过 stat 统计每秒发生 1000 次。

```

name: sched_wakeup
ID: 340
format:
  field:unsigned short common_type;      offset:0;      size:2; signed:0;
  field:unsigned char common_flags;      offset:2;      size:1; signed:0;
  field:unsigned char common_preempt_count; offset:3;      size:1; signed:0;
  field:int common_pid; offset:4;      size:4; signed:1;

  field:char comm[16]; offset:8;      size:16;      signed:1;
  field:pid_t pid; offset:24;      size:4; signed:1;
  field:int prio; offset:28;      size:4; signed:1;
  field:int success; offset:32;      size:4; signed:1;
  field:int target_cpu; offset:36;      size:4; signed:1;

```

- ✓ 可以说 1 秒内 sched_wakeup 唤醒的不同 pid 进程总计 1000 次，其中唤醒 pid 234 共 500 次，唤醒 pid 235 共 300 次，唤醒 pid 236 共 200 次。
- ✓ 也可以说 1 秒内 sched_wakeup 唤醒进程的不同优先级共计 1000 次。其中唤醒 prio=120 共 900 次，唤醒 prio=125 共 80 次，唤醒 prio=139 共 20 次。
- ✓ 也可以说 1 秒内 sched_wakeup 唤醒进程到不同的目标 CPU 共计 1000 次。其中唤醒到 target_cpu=0 共 400 次，唤醒到 target_cpu=2 共计 300 次，唤醒到 target_cpu=3 共计 300 次。

可以看到 sched_wakeup 事件根据不同的字段，被拆分成了 3 类不同的计数器。这个便可以使用 top 来分析。

```

[root@kvm ~]# perf-prof top -e sched:sched_wakeup//key=pid/ -N 10 -vv | tee
top: enabled after 4751047.460671
2024-02-21 18:28:19.465189      ps 41458 dNh. [024] 4751047.460718: sched:sched_wakeup: pal_session:55490 [100] success=1 CPU:024
2024-02-21 18:28:19.465268      ps 41458 dNh. [024] 4751047.460784: sched:sched_wakeup: pal_session:55490 [100] success=1 CPU:024
2024-02-21 18:28:19.465281      ps 41458 dNh. [024] 4751047.460873: sched:sched_wakeup: pal_session:55490 [100] success=1 CPU:024
2024-02-21 18:28:19.465292      ps 41458 dNh. [024] 4751047.460939: sched:sched_wakeup: pal_session:55490 [100] success=1 CPU:024
2024-02-21 18:28:19.465301      ps 41458 dNh. [024] 4751047.461020: sched:sched_wakeup: pal_session:55490 [100] success=1 CPU:024
2024-02-21 18:28:19.465309      ps 41458 dNh. [024] 4751047.461085: sched:sched_wakeup: pal_session:55490 [100] success=1 CPU:024
2024-02-21 18:28:19.465316      ps 41458 dNh. [024] 4751047.461155: sched:sched_wakeup: pal_session:55490 [100] success=1 CPU:024
2024-02-21 18:28:19.465323      ps 41458 dNh. [024] 4751047.461212: sched:sched_wakeup: pal_ctrl:55492 [100] success=1 CPU:024
2024-02-21 18:28:19.465330      ps 41458 dNh. [024] 4751047.461213: sched:sched_wakeup: pal_session:55490 [100] success=1 CPU:024
2024-02-21 18:28:19.465338      ps 41458 dNh. [024] 4751047.461287: sched:sched_wakeup: pal_session:55490 [100] success=1 CPU:024
2024-02-21 18:28:19.466281 perf-prof - 18:28:19 sample 10 events
PID SCHED_WAKEUP
55490      9
55492      1

```

指定 key=pid 即是根据 pid 的不同值，来独立计数。总共采样 10 个事件，其中 pid=55490 被唤醒 9 次，55492 被唤醒 1 次。

```

[root@kvm ~]# perf-prof top -e sched:sched_wakeup//key=prio/ -N 1000 | tee
2024-02-21 18:46:47.127502 perf-prof - 18:46:47 sample 1000 events
PRIO SCHED_WAKEUP
100      801
120      132
139      41
110      24
125      2
[root@kvm ~]# perf-prof top -e sched:sched_wakeup//key=target_cpu/ -N 1000 | tee
2024-02-21 18:47:13.474917 perf-prof - 18:47:13 sample 1000 events
TARGET_CPU SCHED_WAKEUP
24      623
48      157
0      129
72      91

```

Key=prio，计数不同优先级进程被唤醒的次数。优先级 100 被唤醒 801 次，等等。共计 1000 次，从大到小排列。Key=target_cpu，计数进程被唤醒到不同目标 cpu 的次数。唤醒到 CPU24 共 623 次，等等。共计 1000 次，从大到小排列。

排序是最常用的分析方法。

显示 comm。任意事件上添加 comm=EXPR 属性可以显示 comm，此时 key 具有 PID 含义。

```
[root@kvm ~]# perf-prof top -e sched:sched_wakeup//key=pid/comm=comm/ -N 10 -vv | tee
top: enabled after 4811521.668407
2024-02-22 11:16:13.625316      swapper      0 dNh. [024] 4811521.668449: sched:sched_wakeup: pal_session:55490 [100] success=1 CPU:024
2024-02-22 11:16:13.625353      swapper      0 dNh. [024] 4811521.668515: sched:sched_wakeup: pal_session:55490 [100] success=1 CPU:024
2024-02-22 11:16:13.625368      swapper      0 dNh. [024] 4811521.668581: sched:sched_wakeup: pal_session:55490 [100] success=1 CPU:024
2024-02-22 11:16:13.625375      swapper      0 dNh. [024] 4811521.668632: sched:sched_wakeup: pal_session:55490 [100] success=1 CPU:024
2024-02-22 11:16:13.625381      swapper      0 dNh. [024] 4811521.668713: sched:sched_wakeup: pal_session:55490 [100] success=1 CPU:024
2024-02-22 11:16:13.625387      swapper      0 dNh. [024] 4811521.668778: sched:sched_wakeup: pal_session:55490 [100] success=1 CPU:024
2024-02-22 11:16:13.625394      swapper      0 dNh. [024] 4811521.668843: sched:sched_wakeup: pal_session:55490 [100] success=1 CPU:024
2024-02-22 11:16:13.625400      swapper      0 dNh. [024] 4811521.668909: sched:sched_wakeup: pal_session:55490 [100] success=1 CPU:024
2024-02-22 11:16:13.625404      swapper      0 dNh. [024] 4811521.668973: sched:sched_wakeup: pal_session:55490 [100] success=1 CPU:024
2024-02-22 11:16:13.625409      swapper      0 dNh. [024] 4811521.668973: sched:sched_wakeup: pal_session:55490 [100] success=1 CPU:024
2024-02-22 11:16:13.626221 perf-prof - 11:16:13 sample 10 events
PID SCHED_WAKEUP COMM
55490      9 pal_session
55492      1 pal_ctrl
```

pal_session 线程被唤醒 9 次， pal_ctrl 线程被唤醒 1 次。

如果 key 不具备 PID 含义，可以启用--only-comm 选项，只显示 comm。如果 key 和 comm 都取自同一个事件，key 和 comm 可以都不具备 PID 含义。

```
[root@kvm ~]# perf-prof top -e 'irq:irq_handler_entry//key=irq/comm="(char *)&common_type+name_offset"/' -N 10 --only-comm -vv | tee
2024-02-22 11:22:28.459274      swapper      0 d.h. [000] 4811896.186286: irq:irq_handler_entry: irq=38 name=eth0-0
2024-02-22 11:22:28.459344      swapper      0 d.h. [000] 4811896.194583: irq:irq_handler_entry: irq=38 name=eth0-0
2024-02-22 11:22:28.459351      swapper      0 d.h. [000] 4811896.201713: irq:irq_handler_entry: irq=38 name=eth0-0
2024-02-22 11:22:28.459356      swapper      0 d.h. [000] 4811896.204072: irq:irq_handler_entry: irq=100 name=mlx5_cmd_eq@pci:0000:5e:00.1
2024-02-22 11:22:28.459377      swapper      0 d.h. [000] 4811896.204223: irq:irq_handler_entry: irq=100 name=mlx5_cmd_eq@pci:0000:5e:00.1
2024-02-22 11:22:28.459384      swapper      0 d.h. [000] 4811896.204310: irq:irq_handler_entry: irq=100 name=mlx5_cmd_eq@pci:0000:5e:00.1
2024-02-22 11:22:28.459391      swapper      0 d.h. [000] 4811896.206461: irq:irq_handler_entry: irq=38 name=eth0-0
2024-02-22 11:22:28.459395      swapper      0 d.h. [000] 4811896.219301: irq:irq_handler_entry: irq=38 name=eth0-0
2024-02-22 11:22:28.459399      qos_agent    53713 d.h. [000] 4811896.219705: irq:irq_handler_entry: irq=38 name=eth0-0
2024-02-22 11:22:28.459404      swapper      0 d.h. [000] 4811896.226593: irq:irq_handler_entry: irq=38 name=eth0-0
2024-02-22 11:22:28.460250 perf-prof - 11:22:28 sample 10 events
IRQ_HANDLER_ENTRY (CHAR *)&COMMON_TYPE+NAME_OFFSET
7 eth0-0
3 mlx5_cmd_eq@pci:0000:5e:00.1
[root@kvm ~]# perf-prof top -e 'irq:irq_handler_entry//key=irq/comm="(char *)&common_type+name_offset"/' -N 10 | tee
2024-02-22 11:21:57.498967 perf-prof - 11:21:57 sample 10 events
IRQ IRQ_HANDLER_ENTRY (CHAR *)&COMMON_TYPE+NAME_OFFSET
38      7 eth0-0
35      3 mlx5_cmd_eq@pci:0000:5e:00.0
```

这里的 comm 含义就是中断名。key 的含义就是中断号。

Key 或者 comm 可以都不指定。则默认 key 表示 PID，comm 表示进程名。

```
[root@kvm ~]# perf-prof top -e irq:irq_handler_entry -N 10 -vv | tee
2024-02-22 11:36:29.487348      swapper      0 d.h. [000] 4812736.804842: irq:irq_handler_entry: irq=38 name=eth0-0
2024-02-22 11:36:29.487420      exec_agent  56988 d.h. [000] 4812736.810702: irq:irq_handler_entry: irq=100 name=mlx5_cmd_eq@pci:0000:5e:00.1
2024-02-22 11:36:29.487429      exec_agent  56988 d.h. [000] 4812736.810848: irq:irq_handler_entry: irq=100 name=mlx5_cmd_eq@pci:0000:5e:00.1
2024-02-22 11:36:29.487434      exec_agent  56988 d.h. [000] 4812736.810936: irq:irq_handler_entry: irq=100 name=mlx5_cmd_eq@pci:0000:5e:00.1
2024-02-22 11:36:29.487439      swapper      0 d.h. [000] 4812736.857636: irq:irq_handler_entry: irq=38 name=eth0-0
2024-02-22 11:36:29.487443      net_account_age 61310 d.h. [000] 4812736.864840: irq:irq_handler_entry: irq=38 name=eth0-0
2024-02-22 11:36:29.487448      swapper      0 d.h. [000] 4812736.866954: irq:irq_handler_entry: irq=100 name=mlx5_cmd_eq@pci:0000:5e:00.1
2024-02-22 11:36:29.487453      swapper      0 d.h. [000] 4812736.867635: irq:irq_handler_entry: irq=38 name=eth0-0
2024-02-22 11:36:29.487457      swapper      0 d.h. [000] 4812736.868148: irq:irq_handler_entry: irq=38 name=eth0-0
2024-02-22 11:36:29.487461      swapper      0 d.h. [000] 4812736.877639: irq:irq_handler_entry: irq=38 name=eth0-0
2024-02-22 11:36:29.488285 perf-prof - 11:36:29 sample 10 events
PID IRQ_HANDLER_ENTRY COMM
0      6 swapper
56988      3 exec_agent
61310      1 net_account_age
```

统计中断在特定进程上下文发生的次数。

4.1.3.1. Key 和 comm 的所有组合

Key 是否指定	Comm 是否指定	含义
否	否	Key 表示 PID，comm 表示进程名。统计事件在进程上下文发生的次数
否	是	Key 表示 PID，comm 来自事件。Comm 不表示进程名，使用--only-comm。 使用--only-comm：统计不同 comm 的发生次数。 不使用--only-comm：统计事件在进程上下文发生次数。
是	否	Key 来自事件。统计不同 key 的发生次数。
是	是	Key 来自事件，comm 来自事件。统计不同 key 发生次数。Key 和 comm 含义相同。

4.1.3.2. 多个事件场景

多个事件场景，多个事件的 key 含义必须相同。输出结果 key 字段只显示一个。

```
[root@kvm ~]# perf-prof top -e sched:sched_wakeup//key=pid/,sched:sched_switch//key=next_pid/ -N 10 --order -vv | tee
top: enabled after 4753300.212589
2024-02-21 19:05:52.213404 swapper 0 dNh. [024] 4753300.212643: sched:sched_wakeup: pal_session:55490 [100] success=1 CPU:024
2024-02-21 19:05:52.213451 swapper 0 d.. [024] 4753300.212644: sched:sched_switch: swapper/24:0 [120] R ==> pal_session:55490 [100]
2024-02-21 19:05:52.213460 pal_session 55490 d.. [024] 4753300.212647: sched:sched_switch: pal_session:55490 [100] S ==> swapper/24:0 [120]
2024-02-21 19:05:52.213466 swapper 0 dNh. [024] 4753300.212708: sched:sched_wakeup: pal_session:55490 [100] success=1 CPU:024
2024-02-21 19:05:52.213471 swapper 0 d.. [024] 4753300.212709: sched:sched_switch: swapper/24:0 [120] R ==> pal_session:55490 [100]
2024-02-21 19:05:52.213476 pal_session 55490 d.. [024] 4753300.212713: sched:sched_switch: pal_session:55490 [100] S ==> swapper/24:0 [120]
2024-02-21 19:05:52.213481 swapper 0 dNh. [024] 4753300.212773: sched:sched_wakeup: pal_session:55490 [100] success=1 CPU:024
2024-02-21 19:05:52.213485 swapper 0 d.. [024] 4753300.212774: sched:sched_switch: swapper/24:0 [120] R ==> pal_session:55490 [100]
2024-02-21 19:05:52.213490 pal_session 55490 d.. [024] 4753300.212778: sched:sched_switch: pal_session:55490 [100] S ==> swapper/24:0 [120]
2024-02-21 19:05:52.213494 swapper 0 dNh. [024] 4753300.212839: sched:sched_wakeup: pal_session:55490 [100] success=1 CPU:024
2024-02-21 19:05:52.213499 perf-prof - 19:05:52 sample 10 events
PID SCHED_WAKEUP SCHED_SWITCH
55490 4 3
0 0 3
```

key=pid, key=next_pid, 含义一致，都表示进程 id。对于 sched_wakeup 来说，统计不同 pid 被唤醒的次数。对于 sched_switch 来说，统计不同的 next_pid 切换到 cpu 上执行的次数。释义，pid 55490 被唤醒 4 次，切换到 cpu 上执行 3 次，pid 0 被唤醒 0 次，切换到 cpu 上执行 3 次，共计 10 次。

```
[root@kvm ~]# perf-prof top -e irq:irq_handler_entry,sched:sched_wakeup//key=pid/ -N 10 -C 0 -vv | tee
2024-02-22 11:59:13.691397 ethtool 130986 dNh. [000] 4814101.737236: sched:sched_wakeup: hcbs_cli_iscs0:58198 [100] success=1 CPU:000
2024-02-22 11:59:13.691465 swapper 0 dNh. [000] 4814101.737255: sched:sched_wakeup: ethtool:130986 [120] success=1 CPU:000
2024-02-22 11:59:13.691473 swapper 0 d.h. [000] 4814101.737301: irq:irq_handler_entry: irq=100 name=mlx5_cmd_eq@pci:0000:5e:00.1
2024-02-22 11:59:13.691481 swapper 0 dNh. [000] 4814101.737302: sched:sched_wakeup: ethtool:130986 [120] success=1 CPU:000
2024-02-22 11:59:13.691486 swapper 0 dNh. [000] 4814101.737315: sched:sched_wakeup: ethtool:130986 [120] success=1 CPU:000
2024-02-22 11:59:13.691490 swapper 0 d.h. [000] 4814101.737382: irq:irq_handler_entry: irq=100 name=mlx5_cmd_eq@pci:0000:5e:00.1
2024-02-22 11:59:13.691495 swapper 0 dNh. [000] 4814101.737383: sched:sched_wakeup: ethtool:130986 [120] success=1 CPU:000
2024-02-22 11:59:13.691499 swapper 0 dNh. [000] 4814101.737427: sched:sched_wakeup: ethtool:130986 [120] success=1 CPU:000
2024-02-22 11:59:13.691503 swapper 0 d.h. [000] 4814101.737482: irq:irq_handler_entry: irq=100 name=mlx5_cmd_eq@pci:0000:5e:00.1
2024-02-22 11:59:13.691509 swapper 0 dNh. [000] 4814101.737483: sched:sched_wakeup: ethtool:130986 [120] success=1 CPU:000
2024-02-22 11:59:13.691538 perf-prof - 11:59:13 sample 10 events
PID IRQ_HANDLER_ENTRY SCHED_WAKEUP
0 3 0
130986 0 6
58198 0 1
```

irq_handler_entry 的 key 表示 PID，key=pid 表示进程 id，含义一致。对于 irq_handler_entry 统计进程被硬中断中断的次数，对于 sched_wakeup 统计进程被唤醒的次数。

```
perf-prof top -e irq:irq_handler_entry//key=irq/,sched:sched_wakeup//key=pid/ -N 10 -C 0
```

key=irq, key=pid 含义不一致, 输出结果无法解释。

多个事件场景, 多个事件的 comm 含义必须相同。

```
[root@kvm ~]# perf-prof top -e irq:irq_handler_entry,sched:sched_wakeup//key=pid/comm=comm/ -N 10 -C 0 -vv | tee
2024-02-22 12:10:53.233083      exec_agent 56988 d... [000] 4814801.280002: sched:sched_wakeup: exec_agent:122589 [120] success=1 CPU:048
2024-02-22 12:10:53.233153      exec_agent 56988 dNh. [000] 4814801.280084: sched:sched_wakeup: hcbs_cli_iscs0:58198 [100] success=1 CPU:000
2024-02-22 12:10:53.233166      exec_agent 56988 dNh. [000] 4814801.280086: sched:sched_wakeup: time_ticker:70562 [139] success=1 CPU:000
2024-02-22 12:10:53.233174      exec_agent 56988 d.h. [000] 4814801.280220: sched:sched_wakeup: fping:210274 [120] success=1 CPU:000
2024-02-22 12:10:53.233179      fping 210274 d.h. [000] 4814801.280430: irq:irq_handler_entry: irq=38 name=eth0-0
2024-02-22 12:10:53.233187      exec_agent 122589 d... [000] 4814801.280499: sched:sched_wakeup: exec_agent:57011 [120] success=1 CPU:048
2024-02-22 12:10:53.233192      exec_agent 122589 d... [000] 4814801.280501: sched:sched_wakeup: exec_agent:56988 [120] success=1 CPU:000
2024-02-22 12:10:53.233206      exec_agent 56988 d... [000] 4814801.280537: sched:sched_wakeup: exec_agent:122589 [120] success=1 CPU:000
2024-02-22 12:10:53.233211      exec_agent 56988 d... [000] 4814801.280551: sched:sched_wakeup: exec_agent:57312 [120] success=1 CPU:000
2024-02-22 12:10:53.233215      exec_agent 122589 d.h. [000] 4814801.280589: sched:sched_wakeup: fping:210274 [120] success=1 CPU:000
2024-02-22 12:10:53.233248 perf-prof - 12:10:53 sample 10 events
  PID IRQ_HANDLER_ENTRY SCHED_WAKEUP COMM
  210274          1          2 fping
  122589          0          2 exec_agent
  56988           0          1 exec_agent
  57011           0          1 exec_agent
  57312           0          1 exec_agent
  58198           0          1 hcbs_cli_iscs0
  70562           0          1 time_ticker
```

irq_handler_entry 的 key 表示 PID, comm 表示进程名。sched_wakeup, key 表示 pid, comm=comm 表示进程名。含义一致。对于 irq_handler_entry 统计进程被硬中断中断的次数, 对于 sched_wakeup 统计进程被唤醒的次数。fping 进程被硬中断中断 1 次, 被唤醒 2 次。

4.1.3.3. top-by 和 top-add 属性

默认情况下, 计数的是事件发生的次数。使用 top-by 和 top-add 属性可以添加事件的字段作为计数器。

所有 top-by 添加的计数器会被优先排序, 之后 top-add 再被排序。

```
[root@kvm ~]# perf-prof top -e sched:sched_stat_runtime//key=pid/comm=comm/ -N 2 -vv | tee
top: enabled after 4829217.416329
2024-02-22 16:11:09.356930      pal_session 55490 d... [024] 4829217.416371: sched:sched_stat_runtime: comm=pal_session pid=55490 runtime=4155 [ns]
vruntime=1895709246134473 [ns]
2024-02-22 16:11:09.357006      pal_session 55490 d... [024] 4829217.416438: sched:sched_stat_runtime: comm=pal_session pid=55490 runtime=6436 [ns]
vruntime=1895709246134547 [ns]
2024-02-22 16:11:09.360021 perf-prof - 16:11:09 sample 2 events
  PID SCHED_STAT_RUNTIME COMM
  55490          2 pal_session
```

默认统计进程调用 sched_stat_runtime 的次数。但其 runtime 参数表示进程实际运行的 ns 数。

```
[root@kvm ~]# perf-prof top -e sched:sched_stat_runtime//key=pid/comm=comm/top-by=runtime/ -N 5 -vv | tee
top: enabled after 4829502.174506
2024-02-22 16:15:54.115029      pal_session 55490 d... [024] 4829502.174555: sched:sched_stat_runtime: comm=pal_session pid=55490 runtime=4431 [ns]
vruntime=1895814181399770 [ns]
2024-02-22 16:15:54.115068      pal_session 55490 d... [024] 4829502.174620: sched:sched_stat_runtime: comm=pal_session pid=55490 runtime=4560 [ns]
vruntime=1895814181399822 [ns]
2024-02-22 16:15:54.115076      pal_session 55490 d... [024] 4829502.174685: sched:sched_stat_runtime: comm=pal_session pid=55490 runtime=4381 [ns]
vruntime=1895814181399872 [ns]
2024-02-22 16:15:54.115081      pal_session 55490 d... [024] 4829502.174750: sched:sched_stat_runtime: comm=pal_session pid=55490 runtime=4228 [ns]
vruntime=1895814181399920 [ns]
2024-02-22 16:15:54.115086      pal_main 55444 d... [024] 4829502.174815: sched:sched_stat_runtime: comm=pal_main pid=55444 runtime=2908 [ns] vruntime=18958141813998790
[ns]
2024-02-22 16:15:54.116045 perf-prof - 16:15:54 sample 5 events
  PID    RUNTIME COMM
  55490   17600 pal_session
  55444   2908 pal_main
```

Top-by=runtime, 会把 runtime 字段作为计数器。Pal_session 共运行 4 次, runtime 累计运行 17600 ns。Pal_main 共运行 1 次, runtime 累计运行 2908 ns。

4.1.3.4. 更多的例子

```

perf-prof top -e sched:sched_wakeup//comm=comm/ --only-comm -m 64
perf-prof top -e block:block_rq_issue//top-by=nr_sector/comm=comm/ --only-comm -m 32
perf-prof top -e 'block:block_rq_issue/rwbs="W"&&nr_sector<4/top-by=nr_sector/comm=comm/' --only-comm -m 32
perf-prof top -e kvm:kvm_exit//key=exit_reason/ -i 1000
perf-prof top -e sched:sched_stat_runtime//key=pid/comm=comm/top-by=runtime/,sched:sched_switch//key=prev_pid/comm=prev_comm/ -m 64
perf-prof top -e 'sched:sched_process_exec//comm="(char *)&common_type+filename_offset/' --only-comm
perf-prof top -e 'irq:irq_handler_entry//comm="(char *)&common_type+name_offset/' --only-comm
perf-prof top -e 'workqueue:workqueue_execute_start//key=common_pid/alias=NUM/comm=ksymbol(function)/' --only-comm
perf-prof top -e 'kmem:kmem_cache_alloc//top-by=bytes_alloc/comm=ksymbol(call_site)/' --only-comm -m 64
perf-prof top -e 'sched:sched_switch//key=prev_pid/comm=prev_comm/,sched:sched_wakeup//key=pid/comm=comm/,sched:sched_stat_runtime//top-by="runtime/1000"/alias=run(us)/' -m 64

```

alias 属性可以调整显示的字段名。

4.2. 延迟分析

4.2.1. Multi-trace

这是一个多功能的 profiler，基于事件关系，可以分析事件延迟（delay），事件是否成对（pair），内存分配和释放（kmemprof），系统调用延迟（syscalls）。目前主要用于分析延迟和成对，kmemprof、syscalls 已独立成 profiler。

Multi-trace 可以做监控，也可以用于分析延迟的具体原因。基本用法：

```
perf-prof multi-trace -e sys:A/filter/key= ./ -e sys:B/filter/key= ./ --order -i interval 只是基本用法
```

-e 添加事件。要分析延迟，必须知道延迟的起始位置和结束位置。如：系统调用进入到退出，中断进入到退出，vmexit 到 vmentry，加锁到释放锁。等等。

一般添加 2 个事件，对应起始和结束。也可以添加多个事件。-e A,B,C -e D,E -e F 用于分析事件 **A,B,C** 到事件 **D,E** 再到事件 **F** 的延迟。延迟可以分成多个阶段，每个阶段的起始和结束位置，都可以有多种可能性。起始和结束满足**因果关系**，起始点必须先发生，结束点后发生。

sys:A，指定起始点，以 tracepoint、kprobe、uprobe 作为事件源。用户态只能使用 uprobe。sys:B 指定结束点。

filter，trace events 过滤器。合理使用过滤器，可以减少干扰，更聚焦。

key=，指定 key 属性，用于把起始点和结束点关联起来。指定事件的一个字段名。

一个简单的例子，跟踪内核 work 的延迟情况：

workqueue:workqueue_queue_work，这个点是 work 加入队列，等待执行。

workqueue:workqueue_execute_start，这个点是 work 开始执行。

workqueue:workqueue_execute_end，这个点是 work 执行结束。

我们可以分析 work 排队等待的时间，以及 work 的执行时间。这三个点都有一个公共的 work 字段，表示 work_struct 结构体指针，可以唯一标记一个 work，使用 work 作为关联字段。

```

[root@kvm ~]# perf-prof multi-trace -e workqueue:workqueue_queue_work//key=work/ -e workqueue:workqueue_execute_start//key=work/ -e workqueue:workqueue_execute_end//key=work/
-i 1000 -N 10 --order -vv
name workqueue_queue_work id 253 filter (null) stack 0
name workqueue_execute_start id 251 filter (null) stack 0
name workqueue_execute_end id 250 filter (null) stack 0
2024-02-22 18:31:05.222908 swapper 0 dNs. [048] 4837612.300710: workqueue:workqueue_queue_work: work struct=0xffff882f7fc181c8 function=cpuinfo_collect_timer
workqueue=0xffff88017fc09000 req_cpu=48
2024-02-22 18:31:05.222975 kworker/48:1H 172832 ... [048] 4837612.300718: workqueue:workqueue_execute_start: work struct 0xffff882f7fc181c8: function
cpuinfo_collect_timer
2024-02-22 18:31:05.222993 kworker/48:1H 172832 ... [048] 4837612.300721: workqueue:workqueue_execute_end: work struct 0xffff882f7fc181c8
2024-02-22 18:31:05.223009 kworker/48:1 709 ... [048] 4837612.306265: workqueue:workqueue_execute_start: work struct 0xffff882f7fc11940: function
lru_add_drain_per_cpu
2024-02-22 18:31:05.223029 kworker/48:1 709 ... [048] 4837612.306267: workqueue:workqueue_execute_end: work struct 0xffff882f7fc11940
2024-02-22 18:31:05.223044 kworker/48:1 709 ... [048] 4837612.306267: workqueue:workqueue_execute_start: work struct 0xffff882f7db77d18: function wq_barrier_func

```



```

2024-02-22 18:31:05.223057 kworker/48:1 709 .... [048] 4837612.306269: workqueue:workqueue_execute_end: work struct 0xfffff882f7db77d18
2024-02-22 18:31:05.223065 exec_agent 155264 d.s. [048] 4837612.310709: workqueue:workqueue_queue_work: work struct=0xfffff882f7fc181c8 function=cpuinfo_collect_timer
workqueue=0xfffff88017fc09000 req_cpu=48 cpu=48
2024-02-22 18:31:05.223078 kworker/48:1H 172832 .... [048] 4837612.310767: workqueue:workqueue_execute_start: work struct 0xfffff882f7fc181c8: function
cpuinfo_collect_timer
2024-02-22 18:31:05.223091 kworker/48:1H 172832 .... [048] 4837612.310772: workqueue:workqueue_execute_end: work struct 0xfffff882f7fc181c8
2024-02-22 18:31:05.223102
-----
start => end calls total(us) min(us) p50(us) p95(us) p99(us) max(us)
-----
workqueue_queue_work => workqueue_execute_start 2 65.970 7.343 58.627 58.627 58.627 58.627
workqueue_execute_start => workqueue_execute_end 4 11.172 1.362 3.970 4.114 4.114 4.114

```

使用的是-e A -e B -e C 这种形式。分析 A=>B 和 B=>C 的延迟。start => end 指示起始点和结束点。calls 起始到结束统计的次数。total(us) 多次统计到的延迟累加和；min(us) 最小延迟；p50(us) 50%分位的延迟；p95(us) 95%分位的延迟；p99(us) 99%分位的延迟；max(us) 最大延迟。分位延迟，如 p99=58.627，表示 99%的延迟都在 58.627us 以下。

延时是使用事件的内核时间戳来计算的。-N 10 共采样 10 个事件。其中 workqueue_queue_work => workqueue_execute_start，统计的是 work 排队等待的时间，共统计到 2 次，最大延迟 58.627us，在 4837612.310709 到 4837612.310767 阶段发生。workqueue_execute_start => workqueue_execute_end，统计的是 work 的执行时间，共统计到 4 次。

4.2.1.1. 原理

multi-trace 分析多个事件之间的关系，并把多个事件转换成 2 个事件的关系，并最终统计 2 个事件的关联信息。以 tracepoint、kprobe、uprobe 作为事件源。

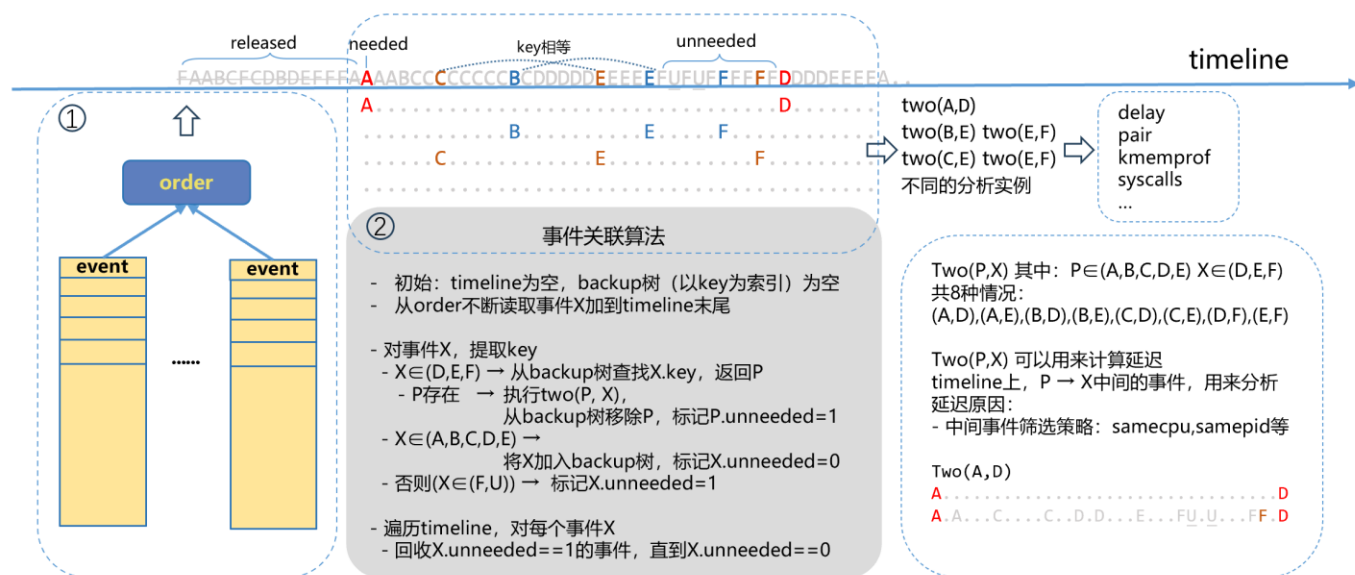
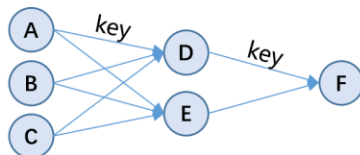
以 `perf-prof multi-trace -e A,B,C -e D,E -e F -k key` 为例，是要分析事件 **A,B,C** 到事件 **D,E**，再到事件 **F** 的关系。转换成 2 个事件的关系：

- A->D, A->E
 - B->D, B->E
 - C->D, C->E
 - D->F
 - E->F

A,B,C 为起点有 3 种可能性，D,E 为中间点有 2 种可能性，F 为终点。A,B,C -> D,E -> F 必须满足以下关系才能测量。

- 因果关系。A,B,C 必须要在 D,E 之前发生，D,E 必须要在 F 之前发生。采样的事件必须具有时间戳。
- 发生关系。A,B,C 只有一个会发生，D,E 也只有一个会发生。A,B,C 一旦发生，D,E 一定会发生。D,E 一旦发生，F 一定会发生。
- 关联关系。通过一个公共的 key 可以关联到所有 A,B,C,D,E,F 事件。不同的 key 值，表示一组不同的事件。每个事件都必须指定 key 属性。所有事件 key 的含义相同，不要求 key 引用的字段名完全一样。

perf-prof multi-trace -e A,B,C -e D,E,U//untraced/ -e F -k key
untraced不参与事件关联



A,B,C,D,E,F 事件的发生顺序非常复杂。其可能会发生在任何 cpu 上, 并缓存在该 cpu 的 ringbuffer 内。每个 ringbuffer 内缓存的事件是按时间排序的。多个 ringbuffer 之间, 是无序的。为了恢复因果关系, 必须要把所有 ringbuffer 内的事件读出来, 按时间排序后, 存放在 timeline 链表上。

timeline 链表, 一颗红黑树, 暂存所有事件, 按事件时间排序。恢复因果关系。丢弃事件发生的 cpu 等信息。

事件的发生关系不需要还原, 在选择事件时, 就要选中满足发生关系的事件。

Timeline 上的每个事件使用 **timeline_node** 结构表示, 要根据其指定的 key 属性, 读取对应的字段值, 存放到 timeline_node::key 上。

```

struct timeline_node {
    struct rb_node timeline_node; // timeline 红黑树
    u64 time; // 事件时间戳
    struct rb_node key_node; // backup 红黑树
    u64 key; // key 属性对应的字段值
    u32 unneeded : 1,
        need_find_prev : 1,
        need_backup : 1,
        need_remove_from_backup : 1;
    union perf_event *event; // 事件本体。
};
  
```

顺序处理 timeline 的每个事件, 事件属于 (A,B,C), 需要存放到 backup 红黑树。事件属于 (D,E), 先根据 timeline_node::key 值在 backup 红黑树查找, 如果找到就转换成 2 个事件来分析 two (A|B|C, D|E), 同时 (D,E) 也需要备份到 backup 红黑树。事件属于 (F), 则根据 timeline_node::key 值在 backup 红黑树查找, 如果找到就转换成 2 个事件来分析 two (D|E, F), F 是最后一级不需要备份到 backup 红黑树。

逐个处理 timeline 的每个事件, 就能还原事件的关联关系。

Backup 红黑树, 临时备份前一级的事件, 按 timeline_node::key 值来索引。

事件的详细处理过程：

- 1) 起点事件 (A,B,C)、中间点事件 (D,E)、终点事件 (F)，全部先存放到 timeline 上，标记为 needed。
- 2) 中间点事件和终点事件 (D,E,F)，需要先获取 key 值，并根据 key 从 backup 红黑树查找前一级的事件。如果找到，就转换成 2 个事件来分析，处理完成后，前一级的事件就不需要了，在 timeline 上标记为 unneeded，等待释放。
- 3) 起点事件和中间点事件 (A,B,C,D,E)，按 key 索引，备份到 backup 红黑树上，等待后一级的处理。
- 4) 终点事件，本身就是不需要的，直接在 timeline 上标记为 unneeded，等待释放。
- 5) 事件回收。如果有事件被标记为 unneeded，按照时间顺序扫描 timeline，释放标记为 unneeded 的事件，直到标记为 needed 的事件为止。

还原出事件的关联关系后，最终会变成 2 个事件的关联关系：two (A|B|C, D|E)、two (D|E, F)。

2 个事件有多种分析方法：

- delay：统计 2 个事件的延迟。分析事件延迟的原因。
- pair：统计事件是否成对。
- syscalls：统计系统调用延迟。
- kmempref：统计内存分配和释放的次数及字节数。
- call：统计事件的调用关系。用于 nested-trace。
- call-delay：调用关系+延迟。

4.2.1.2. 选项参数

4.2.1.2.1. 以实例作为 key

- irq:irq_handler_entry => irq:irq_handler_exit，测量中断执行耗时。中断事件是固定在同一个 cpu 上执行。
- irq:softirq_entry => irq:softirq_exit，测量软中断执行耗时。软中断事件是固定在同一个 cpu 上执行。
- raw_syscalls:sys_enter => raw_syscalls:sys_exit，测量系统调用的耗时。系统调用事件是固定在同一个线程上执行的。以及所有 syscalls:sys_enter_* => syscalls:sys_exit_* 都是固定在同一个线程上执行的。

对于这些事件，不指定任何 key 属性，默认则是以实例作为 key。每个实例使用独立 ringbuffer，其中的事件是有序的。附加到 CPU，则以 CPU 为实例。附加到 PID/TID，则以 TID 为实例。

```
[root@kvm ~]# perf-prof multi-trace -e irq:irq_handler_entry -e irq:irq_handler_exit -i 1000 -N 10 -C 2-3 -vv
name irq_handler_entry id 124 filter (null) stack 0
name irq_handler_exit id 123 filter (null) stack 0
2024-02-23 15:12:00.964650 swapper 0 d.h. [002] 4912068.142656: irq:irq_handler_entry: irq=40 name=eth0-2
2024-02-23 15:12:00.966058 swapper 0 d.h. [002] 4912068.142659: irq:irq_handler_exit: irq=40 ret=handled
2024-02-23 15:12:00.966112
-----
start => end      calls      total(us)  min(us)    p50(us)    p95(us)    p99(us)    max(us)
-----
irq_handler_entry => irq_handler_exit      1           3.028      3.028      3.028      3.028      3.028      3.028
2024-02-23 15:12:02.964617 swapper 0 d.h. [003] 4912070.131248: irq:irq_handler_entry: irq=41 name=eth0-3
2024-02-23 15:12:02.966066 swapper 0 d.h. [003] 4912070.131252: irq:irq_handler_exit: irq=41 ret=handled
```

```
2024-02-23 15:12:02.966130 swapper 0 d.h. [003] 4912070.635193: irq:irq_handler_entry: irq=41 name=eth0-3
2024-02-23 15:12:02.966150 swapper 0 d.h. [003] 4912070.635196: irq:irq_handler_exit: irq=41 ret=handled
2024-02-23 15:12:02.966165
      start => end            calls            total(us)        min(us)        p50(us)        p95(us)        p99(us)        max(us)
-----
irq_handler_entry => irq_handler_exit      2             7.155             3.084             4.071             4.071             4.071             4.071
2024-02-23 15:12:03.964495 swapper 0 d.h. [002] 4912071.166586: irq:irq_handler_entry: irq=40 name=eth0-2
2024-02-23 15:12:03.964555 swapper 0 d.H. [003] 4912071.654494: irq:irq_handler_entry: irq=41 name=eth0-3
2024-02-23 15:12:03.966055 swapper 0 d.h. [002] 4912071.166590: irq:irq_handler_exit: irq=40 ret=handled
2024-02-23 15:12:03.966084 swapper 0 d.H. [003] 4912071.654498: irq:irq_handler_exit: irq=41 ret=handled
2024-02-23 15:12:03.966148
      start => end            calls            total(us)        min(us)        p50(us)        p95(us)        p99(us)        max(us)
-----
irq_handler_entry => irq_handler_exit      2             7.175             3.371             3.804             3.804             3.804             3.804
```

统计 cpu 2-3 上的中断执行耗时，附加到 cpu 上。默认未指定 key 属性。则是以 CPU 作为 key。能够正确统计到中断耗时。其中 4912071.654494 和 4912071.166590 事件来自不同 cpu，即使不是有序的，仍然得到正确结果。

```
[root@kvm ~]# perf-prof multi-trace -e irq:softirq_entry -e irq:softirq_exit -i 1000
2024-02-23 15:21:42.406179
      start => end            calls            total(us)        min(us)        p50(us)        p95(us)        p99(us)        max(us)
-----
softirq_entry => softirq_exit      29706      87496.616             0.281             2.319             6.198            11.373            81.867
```

统计软中断执行耗时。未指定 -C，默认附加到所有 cpu 上。未指定 key 属性，则是以 CPU 作为 key。

```
[root@kvm ~]# perf-prof multi-trace -e raw_syscalls:sys_enter -e raw_syscalls:sys_exit -i 1000 -p 57763 -N 10 -vv
name sys_enter id 78 filter (null) stack 0
name sys_exit id 77 filter (null) stack 0
2024-02-23 15:25:13.887977 qemu-system-x86 57763 .... [049] 4912861.684601: raw_syscalls:sys_exit: NR 271 = 0
2024-02-23 15:25:13.888041 CPU 0/KVM 58966 .... [049] 4912861.681671: raw_syscalls:sys_exit: NR 16 = 0
2024-02-23 15:25:13.888057 CPU 1/KVM 58968 .... [001] 4912861.681518: raw_syscalls:sys_exit: NR 16 = 0
2024-02-23 15:25:13.890149 qemu-system-x86 57763 .... [049] 4912861.684605: raw_syscalls:sys_enter: NR 228 (4, 7ffe2e266e20, 7ffe2e266da0, 0, 8, 0)
2024-02-23 15:25:13.890175 qemu-system-x86 57763 .... [049] 4912861.684605: raw_syscalls:sys_exit: NR 228 = 0
2024-02-23 15:25:13.890187 qemu-system-x86 57763 .... [049] 4912861.684615: raw_syscalls:sys_enter: NR 7 (7ffe2e266ce0, 1, 0, 0, 8747a0, 4af6d8)
2024-02-23 15:25:13.890196 qemu-system-x86 57763 .... [049] 4912861.684618: raw_syscalls:sys_exit: NR 7 = 1
2024-02-23 15:25:13.890204 qemu-system-x86 57763 .... [049] 4912861.684632: raw_syscalls:sys_enter: NR 228 (4, 7ffe2e266e10, 1095100, 23ef860, 0, 0)
2024-02-23 15:25:13.890212 qemu-system-x86 57763 .... [049] 4912861.684632: raw_syscalls:sys_exit: NR 228 = 0
2024-02-23 15:25:13.890219 qemu-system-x86 57763 .... [049] 4912861.684633: raw_syscalls:sys_enter: NR 271 (23f1300, 7, 7ffe2e266da0, 0, 8, 0)
2024-02-23 15:25:13.890314
      start => end            calls            total(us)        min(us)        p50(us)        p95(us)        p99(us)        max(us)
-----
sys_enter => sys_exit              3             4.469             0.409             0.694             3.366             3.366             3.366
```

统计进程 57763 的系统调用耗时。附加到进程，会转换成附加到进程的所有线程上。未指定 key 属性，默认以 tid 作为 key。每个线程使用独立的 ringbuffer，其事件是有序的。如：线程 57763 产生的事件是有序的。

4.2.1.2.2. 间隔输出

-i, --interval <ms> 选项启用间隔输出。

```
[root@kvm ~]# perf-prof multi-trace -e irq:softirq_entry -e irq:softirq_exit
^C2024-02-23 15:39:34.110529
      start => end            calls            total(us)        min(us)        p50(us)        p95(us)        p99(us)        max(us)
-----
softirq_entry => softirq_exit      88009      236684.103             0.280             2.131             5.020            11.920            289.315
```

未指定 -i 选项，则默认结束时统一输出。一般应用于统计，从业务压测开始到结束，整体的耗时情况。

4.2.1.2.3. 启用排序

```
--order          Order events by timestamp.
```

--order 选择启用排序。

默认情况下，multi-trace 可以全都启用排序，而不会对结果产生影响。差异在于，启用排序会导致性能下降，perf-prof 工具自身的 cpu 消耗会增加。以实例作为 key 时，可以不启用排序。参考 4.2.1.2.1 见上方

4.2.1.2.4. Key 选项

`-k, --key <str>`

Key for series events

所有事件公用的 key 属性。事件未指定 key 属性时，会使用--key 选项指定的字段作为 key 属性。

```
perf-prof multi-trace -e workqueue:workqueue_execute_start//key=work/ -e workqueue:workqueue_execute_end//key=work/ -i 1000 -order
perf-prof multi-trace -e workqueue:workqueue_execute_start -e workqueue:workqueue_execute_end -i 1000 --order -k work
```

这两个命令是相同的。

4.2.1.2.5. impl

`--impl <impl>`

Implementation of two-event analysis class. Dflt: delay.
delay: latency distribution between two events
pair: determine if two events are paired
kmemprof: profile memory allocated and freed bytes
syscalls: syscall delay
call: analyze function calls, only for nested-trace.
call-delay: call + delay, only for nested-trace.

Impl 选项默认选择的是 2 事件分析类。不指定--impl 选项，默认启用延迟分析。

目前可以使用--impl pair。Kmemprof、syscalls 已经独立成单独的 profiler。Call、call-delay 用于 nested-trace profiler。

```
[root@kvm ~]# perf-prof multi-trace -e irq:softirq_entry/vec==1/ -e irq:softirq_exit/vec==1/ -i 1000 --impl pair
2024-02-23 16:24:43.666458 irq:softirq_entry unpaired 0
2024-02-23 16:24:43.666517 irq:softirq_entry irq:softirq_exit paired 13748
```

pair，用于分析事件是否成对，不成对则是资源泄露。如：内存分配与释放必须成对，文件描述符打开和关闭必须成对。

4.2.1.2.6. 每实例输出

`--perins`

Print per instance stat

实例就是 key 的各种不同值。

```
[root@kvm ~]# perf-prof multi-trace -e raw_syscalls:sys_enter -e raw_syscalls:sys_exit -i 2000 -p 57763 --perins
2024-02-23 16:38:25.247293
THREAD comm      start => end      calls      total(us)    min(us)      p50(us)      p95(us)      p99(us)      max(us)
-----
57763 qemu-system-x86  sys_enter => sys_exit  13      998016.186    0.396        1.085      638672.189    638672.189    638672.189
57819 IO iothread1     sys_enter => sys_exit  39      14728.654     1.484        2.583      2960.516     3159.172     3159.172
58966 CPU 0/KVM        sys_enter => sys_exit  21      196.970       1.892        10.192     17.456       21.071       21.071
58968 CPU 1/KVM        sys_enter => sys_exit  22      226.289       2.068        8.340      21.289       30.119       30.119
61726 worker          sys_enter => sys_exit   5      8944.014      7.284       1946.865     5024.904     5024.904     5024.904
```

以 tid 作为 key，--perins 打印每个线程的统计信息。

```
[root@kvm ~]# perf-prof multi-trace -e irq:irq_handler_entry -e irq:irq_handler_exit -i 1000 --order -k irq --perins
2024-02-23 16:42:47.266812
irq      start => end      calls      total(us)    min(us)      p50(us)      p95(us)      p99(us)      max(us)
-----
9        irq_handler_entry => irq_handler_exit  2        23.696       9.397        14.299      14.299      14.299      14.299
31       irq_handler_entry => irq_handler_exit  79       340.193      2.769        4.213       5.396       7.553       7.553
35       irq_handler_entry => irq_handler_exit  50       215.352      2.659        4.314       6.083       6.672       6.672
38       irq_handler_entry => irq_handler_exit 169      242.685      0.737        1.533       2.160       2.306       2.721
43       irq_handler_entry => irq_handler_exit  2        5.082        1.737        3.345       3.345       3.345       3.345
46       irq_handler_entry => irq_handler_exit  1        3.076        3.076        3.076       3.076       3.076       3.076
47       irq_handler_entry => irq_handler_exit  2        5.400        2.516        2.884       2.884       2.884       2.884
56       irq_handler_entry => irq_handler_exit  1        2.745        2.745        2.745       2.745       2.745       2.745
59       irq_handler_entry => irq_handler_exit  1        3.012        3.012        3.012       3.012       3.012       3.012
62       irq_handler_entry => irq_handler_exit 10       20.914       1.684        1.981       3.060       3.060       3.060
67       irq_handler_entry => irq_handler_exit  5        12.624       1.829        2.476       3.319       3.319       3.319
```

以 irq 作为 key，统计中断执行耗时。--perins 打印每个 irq 的统计信息。

4.2.1.2.7. 打印高延迟事件

```
--than <ns>          Greater than specified time, Unit: s/ms/us/*ns/percent
--only-than <ns>     Only print those that are greater than the specified time.
--lower <ns>         Lower than specified time, Unit: s/ms/us/*ns
```

--than 选项，打印延迟超过指定值的 2 个事件。--only-than 只有在延迟超过指定值时才输出统计信息。

--lower 选项，打印延迟低于指定值的 2 个事件。

用于分析延迟的原因，需要使用这些选项筛选出关注的事件。

```
[root@kvm ~]# perf-prof multi-trace -e raw_syscalls:sys_enter -e raw_syscalls:sys_exit -i 2000 -p 57763 --perins --than 10ms
2024-02-23 16:47:55.975954 qemu-system-x86 57763 .... [049] 4917822.688618: raw_syscalls:sys_enter: NR 271 (23f1300, 7, 7ffe2e266da0, 0, 8, 0)
2024-02-23 16:47:55.976040 qemu-system-x86 57763 .... [049] 4917823.688652: raw_syscalls:sys_exit: NR 271 = 0
2024-02-23 16:47:55.976088
THREAD comm start => end calls total(us) min(us) p50(us) p95(us) p99(us) max(us) than(reqs)
-----
57763 qemu-system-x86 sys_enter => sys_exit 7 1000047.042 0.480 1.585 1000033.999 1000033.999 1000033.999 1 ( 14%)
58966 CPU 0/KVM sys_enter => sys_exit 21 198.203 1.948 10.260 17.892 21.001 21.001 0 ( 0%)
58968 CPU 1/KVM sys_enter => sys_exit 22 227.130 1.967 8.304 21.212 30.089 30.089 0 ( 0%)
```

统计系统调用耗时。--than 10ms，打印超过 10ms 的事件。

```
[root@kvm ~]# perf-prof multi-trace -e irq:irq_handler_entry -e irq:irq_handler_exit -i 1000 --Lower 800ns
2024-02-23 16:52:20.853617 swapper 0 d.h. [000] 4918088.154088: irq:irq_handler_entry: irq=38 name=eth0-0
2024-02-23 16:52:20.853688 swapper 0 d.h. [000] 4918088.154089: irq:irq_handler_exit: irq=38 ret=handled
2024-02-23 16:52:20.853702 hcbs_cli_iscs0 58198 d.h. [000] 4918088.234517: irq:irq_handler_entry: irq=38 name=eth0-0
2024-02-23 16:52:20.853712 hcbs_cli_iscs0 58198 d.h. [000] 4918088.234518: irq:irq_handler_exit: irq=38 ret=handled
2024-02-23 16:52:20.853763 swapper 0 d.h. [000] 4918088.922731: irq:irq_handler_entry: irq=38 name=eth0-0
2024-02-23 16:52:20.853773 swapper 0 d.h. [000] 4918088.922732: irq:irq_handler_exit: irq=38 ret=handled
2024-02-23 16:52:20.853782 swapper 0 d.h. [000] 4918088.984730: irq:irq_handler_entry: irq=38 name=eth0-0
2024-02-23 16:52:20.853790 swapper 0 d.h. [000] 4918088.984731: irq:irq_handler_exit: irq=38 ret=handled
2024-02-23 16:52:20.853854
start => end calls total(us) min(us) p50(us) p95(us) p99(us) max(us)
-----
irq_handler_entry => irq_handler_exit 287 689.974 0.707 2.118 4.533 8.588 15.420
```

--lower 800ns，打印小于 800ns 的事件。用于分析事件延迟太低的情况。如：睡眠 10ms，实际只睡眠了 5ms 就退出了。

4.2.1.2.8. 打印中间细节

```
--detail[=<-N,+N,samecpu,samepid>]
More detailed information output.
For multi-trace profiler:
-N: Before event1, print events within N nanoseconds.
+N: After event2, print events within N nanoseconds.
hide<N: Hide event intervals less than N nanoseconds.
samecpu: Only show events with the same cpu as event1 or event2.
samepid: Only show events with the same pid as event1 or event2.
sametid: Only show events with the same tid as event1 or event2.
samekey: Only show events with the same key as event1 or event2.
```

要分析延迟的原因，重点是输出延迟的中间细节。2 个事件中间可以添加更多 untraced 事件，来逐步还原延迟的中间细节。参考 4.2.1.3.2 见下方。参考 5.1 见下方。

在 timeline 上，2 个事件中间可能会发生非常多的其他事件，来自不关心的 CPU，来自不关心的线程等等。detail 选项，主要用于过滤出 2 个事件中间真正需要关注的事件。

-N，起始点之前 Nns 内的事件。

+N，结束点之后 Nns 内的事件。

hide<N，隐藏前后事件间隔小于 Nns 的所有事件。中间事件非常多时，用于筛选间隔大的事件。跟比较大--than 选项配合使用。

samecpu, samepid, sametid, samekey, 跟起始点和结束点相同 cpu、pid、tid、key 的所有事件。

```
[root@kvm ~]# perf-prof multi-trace -e sched:sched_switch/prev_state==2/key=prev_pid/ -e sched:sched_wakeup//key=pid/ -m 512 -i 1000 --order --than 3ms --detail=-1ms,sametid,+1ms
-Previous 1 -989.942 us
|
| moagent 227584 d.h. [000] 4982349.450730: sched:sched_wakeup: outlier_detect:70570 [139] success=1 CPU:000
| moagent 227584 dNh. [000] 4982349.450848: sched:sched_wakeup: hcbs_cli_iscs0:58198 [100] success=1 CPU:000
| moagent 227584 dNh. [000] 4982349.450850: sched:sched_wakeup: exec_agent:218829 [120] success=1 CPU:000
| moagent 227584 dNh. [000] 4982349.451164: sched:sched_wakeup: moagent:225319 [120] success=1 CPU:000
2024-02-24 10:43:22.869109 2
| 4943.193 us 4
| 5
| moagent 225319 d... [000] 4982349.451171: sched:sched_switch: moagent:225319 [120] D ==> moagent:227584 [120]
| moagent 227584 dNh. [000] 4982349.451192: sched:sched_wakeup: net_account_age:61311 [120] success=1 CPU:000
| moagent 227584 dNh. [000] 4982349.451911: sched:sched_wakeup: hcbs_cli_iscs0:58198 [100] success=1 CPU:000
| moagent 227584 dNh. [000] 4982349.452252: sched:sched_wakeup: net_account_age:61311 [120] success=1 CPU:000
| moagent 227584 dNh. [000] 4982349.452971: sched:sched_wakeup: hcbs_cli_iscs0:58198 [100] success=1 CPU:000
| moagent 227584 dNh. [000] 4982349.455093: sched:sched_wakeup: hcbs_cli_iscs0:58198 [100] success=1 CPU:000
2024-02-24 10:43:22.874052 6
| moagent 227584 dNh. [000] 4982349.456115: sched:sched_wakeup: moagent:225319 [120] success=1 CPU:000
| moagent 225319 d.h2 [000] 4982349.456153: sched:sched_wakeup: hcbs_cli_iscs0:58198 [100] success=1 CPU:000
| moagent 227584 dNh. [000] 4982349.456249: sched:sched_wakeup: sap1015:152678 [120] success=1 CPU:000
After 1011.108 us 7
2024-02-24 10:43:23.884860 8
start => end calls total(us) min(us) p50(us) p95(us) p99(us) max(us) than(reqs)
-----
sched_switch => sched_wakeup 739 1023575.938 1.589 1991.146 1994.130 1995.676 4943.987 4 ( 0%)
```

统计进程 D 的耗时，prev_state==2 表示从进程 D 而切换出去（key=prev_pid），到进程被唤醒（key=pid），这中间的延迟，打印超过 3ms 的事件。--detail=-1ms,sametid,+1ms，显示前后 1ms 内的事件，过滤相同 tid 的事件。具体输出格式：

- 1Previous，显示起始点之前一定时间的事件。
- 2延迟起始点，事件发生的时间，有轻微误差。即使事件在 ringbuffer、order buffer 内部缓存很久，也能获得相对准确的时间。
- 3事件时间戳，local_clock()，精确的事件发生时刻。
- 4起始点到结束点的具体延迟。使用事件时间戳计算 6-2。4982349.456115 - 4982349.451171。
- 5中间事件，起始点到结束点中间发生的事件，经过--detail=sametid 的过滤。'|'开始的全部事件，包含起始点之前和结束点之后的事件。
- 6延迟结束点，显示事件发生的时间。
- 7 After，显示结束点之后一定时间的事件。
- 8当前时间。事件会在 order buffer 内部缓存一个刷新周期，这个时间显示的统计信息是上一周期发生的事件。

先要定位起始点 2 和结束点 6，moagent:225319 在 4982349.451171 时刻因为 D 而切换出去，在 4982349.456115 时刻被 moagent 227584 线程唤醒。

sametid 过滤跟起始点和结束点相同 tid 的事件。

```
[root@kvm ~]# perf-prof multi-trace -e sched:sched_wakeup//key=pid/ -e sched:sched_switch//key=next_pid/,sched:sched_migrate_task//key=pid/untraced/ -m 512 -i 1000 --order --than 14ms --detail=samekey,+1ms
2024-02-24 14:39:32.061933
start => end calls total(us) min(us) p50(us) p95(us) p99(us) max(us) than(reqs)
-----
sched_wakeup => sched_switch 40330 309931.650 0.609 1.918 13.103 39.377 5787.718 0 ( 0%)
2024-02-24 14:39:33.004237
| 17151.219 us
| 2024-02-24 14:39:33.004824
| exec_agent 218828 d.h. [048] 4996519.618354: sched:sched_wakeup: vpc_sflow_agent:70627 [139] success=1 CPU:048
| | 17151.219 us
| | 0 dNs. [000] 4996519.624515: sched:sched_migrate_task: comm=vpc_sflow_agent pid=70627 prio=139 orig_cpu=48 dest_cpu=0
| | 2024-02-24 14:39:33.004824
| | exec_agent 57011 d... [000] 4996519.635505: sched:sched_switch: exec_agent:57011 [120] S ==> vpc_sflow_agent:70627 [139]
| | | check_adamPlugi 106904 d.h. [000] 4996519.635561: sched:sched_wakeup: vpc_sflow_agent:70627 [139] success=1 CPU:000
| | | exec_agent 57011 d... [000] 4996519.636109: sched:sched_switch: exec_agent:57011 [120] S ==> vpc_sflow_agent:70627 [139]
| | | swapper 0 dNh. [000] 4996519.636166: sched:sched_wakeup: vpc_sflow_agent:70627 [139] success=1 CPU:000
| | | exec_agent 57011 d... [000] 4996519.636422: sched:sched_switch: exec_agent:57011 [120] S ==> vpc_sflow_agent:70627 [139]
| | | swapper 0 dNh. [000] 4996519.636477: sched:sched_wakeup: vpc_sflow_agent:70627 [139] success=1 CPU:000
| | | swapper 0 d... [000] 4996519.636479: sched:sched_switch: swapper/0:0 [120] R ==> vpc_sflow_agent:70627 [139]
| | | swapper 0 dNh. [000] 4996519.636490: sched:sched_wakeup: vpc_sflow_agent:70627 [139] success=1 CPU:000
After 1004.979 us
2024-02-24 14:39:33.077398
start => end calls total(us) min(us) p50(us) p95(us) p99(us) max(us) than(reqs)
-----
sched_wakeup => sched_switch 48289 744334.860 0.617 2.536 20.739 201.582 17151.219 1 ( 0%)
```

测量调度延迟，统计进程被唤醒（key=pid）到进程开始执行（key=next_pid）之间的延迟。添加了一个 untraced 中间事件，观察调度延迟中间发生的事件。vpc_sflow_agent:70627 的调度延迟是 17151.219us。

--detail=samekey,+1ms，只过滤相同 key 的中间事件。可以看到 sched_wakeup、sched_switch、sched_migrate_task，这三个事件只要 key 相同都会被打印出来。

4.2.1.2.8.1. 特殊事件

sched:sched_wakeup	samecpu：也判断 target_cpu 字段。samepid/sametid：也判断 pid 字段。
sched:sched_waking	samecpu：也判断 target_cpu 字段。samepid/sametid：也判断 pid 字段。
sched:sched_wakeup_new	samecpu：也判断 target_cpu 字段。samepid/sametid：也判断 pid 字段。
sched:sched_switch	samepid/sametid：也判断 prev_pid、next_pid 字段。
sched:sched_migrate_task	samecpu：也判断 prev_cpu、next_cpu 字段。samepid/sametid：也判断 pid 字段。
sched:sched_stat_runtime	samepid/sametid：也判断 pid 字段。
sched:sched_process_free	samepid/sametid：也判断 pid 字段。
sched:sched_process_exit	samepid/sametid：也判断 pid 字段。
sched:sched_process_fork	samepid/sametid：也判断 parent_pid、child_pid 字段。
sched:sched_process_exec	samepid/sametid：也判断 pid 字段。

samecpu、samepid、sametid 在判断时，从事件取对应的字段，并判断是否与起始点和结束点的 cpu、pid 相同。

4.2.1.2.8.2. 新增事件

新增的 kprobe 事件，如果含有 “cpu” 、 “pid” 参数，会自动取出对应的字段，并判断是否与起始点和结束点的 cpu、pid 相同。

```
echo 'p:unthrottle_cfs_rq unthrottle_cfs_rq cpu=+0xaa0(+0xaa0(%di)):s32' >> /sys/kernel/debug/tracing/kprobe_events
```

增加一个 kprobes:unthrottle_cfs_rq 事件，并获取 cpu 参数。samecpu 会自动取出 kprobes:unthrottle_cfs_rq 事件的 cpu 字段，并与起始点和结束点的 cpu 做判断，相同则输出 kprobes:unthrottle_cfs_rq 事件。

添加 kprobe 事件并获取 pid 参数，samepid/sametid 会自动取出时间的 pid 字段，并与起始点和结束点的 pid/tid 做判断，相同则输出事件。

```
[root@kvm ~]# perf-prof rundelay -e sched:sched_wakeup,sched:sched_wakeup_new,sched:sched_switch//key=prev_pid/ -e
sched:sched_switch//key=next_pid/,sched:sched_switch//untraced/,kprobes:unthrottle_cfs_rq//untraced/stack/ -k pid --order -i 1000 -p 12633 --than 30ms --detail=samecpu
2024-10-29 10:51:14.157823 swapper 0 dNh. [014] 63063828.027737: sched:sched_wakeup: net_account_age:12634 [120] success=1 CPU:014
|+ 73919.448 us swapper 0 d... [014] 63063828.030715: sched:sched_switch: swapper/14:0 [120] R ==> CPU 1/KVM:384541 [120]
| ...
| 28.743 CPU 7/KVM swapper 0 d... [014] 63063828.099909: sched:sched_switch: swapper/14:0 [120] R ==> CPU 7/KVM:384547 [120]
| CPU 7/KVM 384547 d... [014] 63063828.099938: sched:sched_switch: CPU 7/KVM:384547-[120] S ==> swapper/14:0 [120]
| swapper 0 d.h. 0010 63063828.101653: kprobes:unthrottle_cfs_rq: (ffffff810d8d10) cpu=14
| 73918.513 [running] swapper 0 d... [014] 63063828.101656: sched:sched_switch+swapper/14:0 [120] R ==> net_account_age:12634 [120]
2024-10-29 10:51:14.231916 swapper 0 d... [014] 63063828.101657: sched:sched_switch: swapper/14:0 [120] R ==> net_account_age:12634 [120]
```

起始点和结束点中间的每一个事件，都需要执行多次判断。黄色箭头指示 kprobes:unthrottle_cfs_rq 事件的判断次数。

4.2.1.2.9. 单事件延迟

```
--cycle multi-trace: event cycle, from the last one back to the first.
```

--cycle 选项把所有的事件构成一个环。[-e A,B,C -e D,E -e F --cycle]还会追加测量 F->A,B,C 之间的延迟。[-e A --cycle]只有单个事件时，测量的就是 A->A 之间的延迟。

```
[root@kvm ~]# perf-prof multi-trace -e sched:sched_switch,sched:sched_wakeup//untraced/ --cycle -i 1000 -m 512 --order --than 11ms --detail=samekey
2024-02-24 15:15:19.743725 swapper 0 d... [072] 4998666.770102: sched:sched_switch: swapper/72:0 [120] R ==> perf-prof:115526 [120]
| 11263.501 us perf-prof 115526 d.h. [072] 4998666.770282: sched:sched_wakeup: hCBS_agent_mcd:108955 [100] success=1 CPU:024
| perf-prof 115526 d.h. [072] 4998666.771022: sched:sched_wakeup: time_ticker:70562 [139] success=1 CPU:024
| perf-prof 115526 d.s. [072] 4998666.772301: sched:sched_wakeup: ksmid:691 [125] success=1 CPU:072
```



```
| perf-prof 115526 dNh. [072] 4998666.775106: sched:sched_wakeup: fping:124268 [120] success=1 CPU:072
| perf-prof 115526 dNs. [072] 4998666.775302: sched:sched_wakeup: kworker/72:2H:100436 [100] success=1 CPU:072
2024-02-24 15:15:19.743899 perf-prof 115526 d... [072] 4998666.781365: sched:sched_switch: perf-prof:115526 [120] R ==> kworker/72:2H:100436 [100]
2024-02-24 15:15:19.750355
start => end calls total(us) min(us) p50(us) p95(us) p99(us) max(us) than(reqs)
-----
sched_switch => sched_switch 78156 95880997.836 1.295 19.409 9987.500 9992.242 11263.501 1 ( 0%)
```

测量进程单次执行耗时，未指定 key 属性，是以 CPU 作为 key。perf-prof:115526 从 4998666.770102 切换到 cpu 上开始执行，到 4998666.781365 切换出去。在 cpu 上单次执行 11263.501 us。

samekey 就过滤相同 cpu 上的事件。

4.2.1.3. 属性

```
-e, --event <EVENT,...> Event selector. use 'perf list tracepoint' to list available tp events.
EVENT,EVENT,...
EVENT: sys:name[/filter/ATTR/ATTR/.../]
        profiler[/option/ATTR/ATTR/.../]
filter: trace events filter
ATTR:
    stack: sample_type PERF_SAMPLE_CALLCHAIN
    key=EXPR: key for multiple events: top, multi-trace
    untraced: multi-trace, auxiliary, no two-event analysis
    trigger: multi-trace, use events to trigger interval output
vm=uuid: get the mapping from Guest vcpu to Host tid
push=[IP:]PORT: push events to the local broadcast server IP:PORT
push=chardev: push events to chardev, e.g., /dev/virtio-ports/*
push=file: push events to file
pull=[IP:]PORT: pull events from server IP:PORT
pull=chardev: pull events from chardev
pull=file: pull events from file
EXPR:
    C expression. See `perf-prof expr -h` for more information.
```

Stack，打开事件的堆栈。用于确认事件发生的上下文，定位问题非常有帮助。

Key，用于指定事件的关联关系。

Untraced，添加辅助分析点。Untraced 属性的事件不参与延迟统计，只在打印中间细节时显示。

Trigger，可以不启用间隔输出，直到标记为 trigger 属性的事件发生时，触发输出一次。用于隔离开事件前后的统计信息。

Vm、push、pull，用于虚拟化场景，可以接收 Guest 传递过来的事件，进行延迟分析。

4.2.1.3.1. key

Key 属性指定事件字段名，可以借助 help 帮助快速找到字段名。

4.2.1.3.2. untraced

标记为 untraced 属性的事件，不参与延迟统计，可以加在任何位置。

```
-e A,B,C//untraced/ -e D,E//untraced/
-e A,B                -e D
```

只统计 A->D，B->D 的延迟，C，E 不参与。C，E 是猜测的可能会出现在 A->D，B->D 中间的事件，用于还原延迟的中间细节。更多参考 5.1 见下方

4.2.1.3.3. role

```
[root@VM ~]# perf-prof multi-trace -e sched:sched_switch// --cycle -i 1000 --than 1s
2024-05-07 09:20:01.532107 kworker/5:0-eve 38 d... [005] 620.537688: sched:sched_switch: kworker/5:0:38 [120] I ==> swapper/5:0 [120]
2024-05-07 09:20:02.556150 swapper 0 d... [005] 621.561730: sched:sched_switch: swapper/5:0 [120] R ==> kworker/5:0:38 [120]
2024-05-07 09:20:01.184696 systemd-udevd 1067 d... [034] 620.190277: sched:sched_switch: systemd-udevd:1067 [120] S ==> swapper/34:0 [120]
2024-05-07 09:20:02.236926 swapper 0 d... [034] 621.242506: sched:sched_switch: swapper/34:0 [120] R ==> kworker/34:0:183 [120]
2024-05-07 09:20:02.688341
start => end calls total(us) min(us) p50(us) p95(us) p99(us) max(us) than(reqs)
-----
sched_switch => sched_switch 1792 35469824.596 1.947 49.070 61568.620 567855.070 1052229.371 2 ( 0%)
```

```
[root@VM ~]# perf-prof multi-trace -e sched:sched_switch//role="(next_pid?1:0)|(prev_pid?2:0)"/ --cycle -i 1000 --than 1s
2024-05-07 09:20:50.345957
start => end          calls          total(us)        min(us)          p50(us)          p95(us)          p99(us)          max(us)          than(reqs)
-----
sched_switch => sched_switch    753          19713.338           1.995           4.539           63.149          567.934          2707.333           0 ( 0%)
```

统计 sched_switch => sched_switch 的延迟，其统计的就是进程在 CPU 上的执行时间。不加 role 属性，其统计的进程执行时间包含 swapper 进程，sched_switch 切换到 swapper 进程，并再从 swapper 进程切出去。但 swapper 进程并不是我们需要的，且不能通过添加过滤器解决。如：sched:sched_switch/prev_pid>0|next_pid>0/。

对于 2 个事件分析 two(event1, event2)，一个事件作为 event1，其需要临时备份，等待 event2 发生。一个事件作为 event2，其需要查找 event1，并组成一对分析延迟。对--cycle 选项，所有事件既可以作为 event1，也可以作为 event2。

role 属性的作用就是调控事件角色，使其只能作为 event1，或者 event2。Role 由 2 位组成，role.bit0 置位则事件作为 event1，role.bit1 置位则事件作为 event2。

role="(next_pid?1:0)|(prev_pid?2:0)" 属性。next_pid 不等于 0，表示切换到非 idle 进程上，则作为 event1，临时备份。prev_pid 不等于 0，表示从非 idle 进程切换出去，则作为 event2，需要查找 event1。

一些例子：

```
ls      89825 d... [000] 27230104.110155: sched:sched_switch: ls:89825 [120] R ==> rcu_sched:11 [120] | role = 3 统计 rcu_sched 执行时间
rcu_sched 11 d... [000] 27230104.110169: sched:sched_switch: rcu_sched:11 [120] S ==> ls:89825 [120] | role = 3 统计 ls 执行时间
ls      89825 d... [000] 27230104.110366: sched:sched_switch: ls:89825 [120] S ==> swapper/0:0 [120] | role = 2 不统计 swapper 的执行时间
swapper 0 d... [000] 27230104.110412: sched:sched_switch: swapper/0:0 [120] R ==> python:178754 [120] | role = 1 统计 python 的执行时间
```

4.2.1.3.4. trigger

```
[root@kvm ~]# perf-prof multi-trace -e sched:sched_wakeup//key=pid/ -e sched:sched_switch//key=next_pid/,sched:sched_process_exec/filename=~*touch/trigger/untraced/ -m 512 -i 5000 --order
Trick: Enable userland unnecessary detection of sched:sched_wakeup events.
2024-02-26 16:03:14.413464
start => end          calls          total(us)        min(us)          p50(us)          p95(us)          p99(us)          max(us)
-----
sched_wakeup => sched_switch    211649          1681520.656           0.368           1.543          10.585          30.742          21381.243
2024-02-26 16:03:18.279799
start => end          calls          total(us)        min(us)          p50(us)          p95(us)          p99(us)          max(us)
-----
sched_wakeup => sched_switch    51726          492346.552           0.381           1.853          14.276          94.084          10398.997
2024-02-26 16:03:10.507988          touch 104281 .... [024] 5174338.860228: sched:sched_process_exec: filename=/usr/bin/touch pid=104281 old_pid=104281
2024-02-26 16:03:19.478914
start => end          calls          total(us)        min(us)          p50(us)          p95(us)          p99(us)          max(us)
-----
sched_wakeup => sched_switch    178052          2717023.498           0.365           1.913          28.932          247.705          26647.034
```

统计调度延迟，执行 touch 命令来触发一次间隔输出。

实际的使用场景：

- trigger 事件是周期性的，则会周期性的触发间隔输出。
- trigger 是 breakpoint 剖析器，则可以在读写某个内存地址时，触发间隔输出。

4.2.1.3.5. Stack

可以在任何事件上添加 stack 属性，打开堆栈显示。包括标记 untraced、trigger 属性的事件。

```
[root@kvm ~]# perf-prof multi-trace -e sched:sched_wakeup//key=pid/ -e sched:sched_switch//key=next_pid/,sched:sched_migrate_task//key=pid/untraced/stack/ -m 512 -i 1000 --order --than 14ms --detail=samekey,+1ms
2024-02-26 16:25:47.491432          atop 54312 d.s. [000] 5175695.844862: sched:sched_wakeup: kipi0:1771 [139] success=1 CPU:000
| 26170.022 us          swapper 0 dNs. [048] 5175695.870899: sched:sched_migrate_task: comm=kipi0 pid=1771 prio=139 orig_cpu=0 dest_cpu=48
| ffffffff810c94a3 set_task_cpu+0xf3 ([kernel.kallsyms])
| ffffffff810d147d move_task+0x2d ([kernel.kallsyms])
| ffffffff810da422 load_balance+0x562 ([kernel.kallsyms])
| ffffffff810da9d0 rebalance_domains+0x170 ([kernel.kallsyms])
| ffffffff810dac32 run_rebalance_domains+0x122 ([kernel.kallsyms])
| ffffffff810960c5 __do_softirq+0xf5 ([kernel.kallsyms])
| ffffffff816d2c9c call_softirq+0x1c ([kernel.kallsyms])
| ffffffff8102c475 do_softirq+0x65 ([kernel.kallsyms])
| ffffffff81096445 irq_exit+0x105 ([kernel.kallsyms])
| ffffffff816d3938 smp_apic_timer_interrupt+0x48 ([kernel.kallsyms])
```



```

ffffff816d1e5d apic_timer_interrupt+0x6d ([kernel.kallsyms])
ffffff81541fde cpuidle_idle_call+0xde ([kernel.kallsyms])
ffffff8103412e arch_cpu_idle+0xe ([kernel.kallsyms])
ffffff810f654a cpu_startup_entry+0x14a ([kernel.kallsyms])
ffffff81055042 start_secondary+0x1d2 ([kernel.kallsyms])
2024-02-26 16:25:47.517602          ksm d... [048] 5175695.871032: sched:sched_switch: ksm d:691 [125] D ==> kipmi0:1771 [139]
|                               swapper 0 dNs. [048] 5175695.871866: sched:sched_wakeup: kipmi0:1771 [139] success=1 CPU:048
|                               swapper 0 d... [048] 5175695.871867: sched:sched_switch: swapper/48:0 [120] R ==> kipmi0:1771 [139]
`After 1019.792 us
2024-02-26 16:25:49.148249
start => end          calls          total(us)        min(us)        p50(us)        p95(us)        p99(us)        max(us)        than(reqs)
-----
sched_wakeup => sched_switch  42454          862357.285          0.375          1.619          12.459          147.572          26170.022          17 ( 0%)

```

测量调度延迟，观察 sched_migrate_task 事件的堆栈。可以看到 kipmi0:1771 线程在 CPU:000 上等待了 26ms，之后软中断执行负载均衡，将其迁移到 CPU:048，很快得到了执行。

4.2.1.4. 过滤器

合理使用过滤器，可以减少事件量，更聚焦。实际应用，需要不断调整过滤器，缩小事件范围，逐渐深入到延迟的内部细节。

过滤器，必须保证参与延迟分析的事件，过滤出来的事件是一致的，都要满足因果关系、发生关系、关联关系。

```

[root@kvm ~]# perf-prof multi-trace -e sched:sched_wakeup/comm~python/key=pid/ -e
sched:sched_switch/next_comm~python/key=next_pid/,sched:sched_migrate_task//key=pid/untraced/,sched:sched_switch/prev_comm~python/key=prev_pid/untraced/ -m 512 -i 1000 --
order --than 2ms --detail=samekey,+1ms
2024-02-26 16:57:12.496584          atop 54312 dNh. [000] 5177580.851790: sched:sched_wakeup: python:26821 [120] success=1 CPU:000
|        6881.959 us          swapper 0 dNs. [048] 5177580.858598: sched:sched_migrate_task: comm=python pid=26821 prio=120 orig_cpu=0 dest_cpu=48
2024-02-26 16:57:12.503466          bio_sdk_mng 58168 d... [048] 5177580.858672: sched:sched_switch: bio_sdk_mng:58168 [100] S ==> python:26821 [120]
|          python 26821 d... [048] 5177580.858689: sched:sched_switch: python:26821 [120] S ==> time_ticker:70562 [139]
|          swapper 0 dNh. [048] 5177580.859725: sched:sched_wakeup: python:26821 [120] success=1 CPU:048
`After 1052.635 us
2024-02-26 16:57:14.393759
start => end          calls          total(us)        min(us)        p50(us)        p95(us)        p99(us)        max(us)        than(reqs)
-----
sched_wakeup => sched_switch  1174          15838.145          0.533          1.461          20.448          104.897          6881.959          1 ( 0%)

```

测量 python 进程的调度延迟，comm~python，next_comm~python，只过滤出 python 进程。

4.2.1.5. 各个参数的工作位置



4.2.1.6. 注解

在事件的左侧，会显示事件的注解。

```

[root@kvm ~]# perf-prof rundelay -e 'sched:sched_wakeup,sched:sched_wakeup_new,sched:sched_switch//key=prev_pid/' -e
'sched:sched_switch//key=next_pid/,sched:sched_migrate_task//untraced/' -k pid --order -m 512 -i 1000 --than 20ms --detail=samecpu
2024-09-29 16:01:57.557448          atop 212507 d.h. [048] 2690982.382248: sched:sched_wakeup: main_task:23691 [139] success=1 CPU:048
|+ 20250.508 us          atop 212507 d.h. [048] 2690982.382311: sched:sched_wakeup: net_account_age:218542 [120] success=1 CPU:048
|          atop 212507 d.h. [048] 2690982.394862: sched:sched_wakeup: loglistener:13197 [120] success=1 CPU:048
|          atop 212507 dNs. [048] 2690982.401465: sched:sched_wakeup: kworker/48:1H:12633 [100] success=1 CPU:048
|          atop 212507 d... [048] 2690982.401476: sched:sched_switch: atop:212507 [100] R ==> harp-agent:147403 [120]
|          atop 212507 d... [048] 2690982.401477: sched:sched_switch: atop:212507 [100] R ==> harp-agent:147403 [120]
|          harp-agent 147403 d... [048] 2690982.401484: sched:sched_wakeup: harp-agent:147402 [120] success=1 CPU:048
|          harp-agent 147403 d... [048] 2690982.401495: sched:sched_switch: harp-agent:147403 [120] S ==> ksoftirqd/48:343 [120]
|          29.205 ksoftirqd/4 ksoftirqd/48 343 d... [048] 2690982.401524: sched:sched_switch: ksoftirqd/48:343 [120] S ==> harp-agent:147401 [120]
|          atop 212507 d.h. [048] 2690982.401736: sched:sched_migrate_task: comm=net_account_age pid=218542 prio=120 orig_cpu=48 dest_cpu=0
|          atop 212507 d.h. [048] 2690982.401737: sched:sched_wakeup: net_account_age:218542 [120] success=1 CPU:000
|          20231.624 [migrate]          swapper 0 dNs. [000] 2690982.402479: sched:sched_migrate_task: comm=main_task pid=23691 prio=139 orig_cpu=48 dest_cpu=0
|          swapper 0 dNs. [000] 2690982.402481: sched:sched_migrate_task: comm=circuit_break pid=23694 prio=139 orig_cpu=48 dest_cpu=0

```

```
| swapper 0 d... [000] 2690982.402483: sched:sched_switch: swapper/0:0 [120] R ==> rcu_sched:11 [120]
| 11.449 rcu_sched rcu_sched 11 d... [000] 2690982.402494: sched:sched_switch: rcu_sched:11 [120] S ==> circuit_break:23694 [139]
2024-09-29 16:01:57.577746 circuit_break 23694 d... [000] 2690982.402498: sched:sched_switch: circuit_break:23694 [139] S ==> main_task:23691 [139]
```

事件注解分 2 类：runtime, comm; time_offset, [reason]。

runtime, comm。用于 rundelay，表示进程的运行时间。可以显示调度延迟内，哪个进程运行的时间很长。仅作用于 sched:sched_switch 事件。

time_offset, [reason]。reason 在中括号内，时间是从 e1 开始的偏移。这是对进程关键状态的记录，不同的事件会产生不同的进程状态。

sched:sched_wakeup: [wakeup]，进程被唤醒。sched:sched_switch: [running]，进程正在运行；[preempt]，进程被抢占；[sleeping]，进程睡眠；[waiting]，进程在等待资源。sched:sched_migrate_task: [migrate]，进程迁移到其他 cpu。

4.2.2. Syscalls

统计系统调用延迟，基于 multi-trace。功能和选项参数同 multi-trace。

```
[root@kvm ~]# perf-prof syscalls -e 'raw_syscalls:sys_enter' -e 'raw_syscalls:sys_exit' -p 181860 --perins -i 1000 --than 100ms
2024-02-26 16:41:15.548346
thread comm syscalls calls total(us) min(us) avg(us) max(us) err than(reqs)
-----
181860 qemu-system-x86 poll(7) 1 2.556 2.556 2.556 2.556 0 0 ( 0%)
181860 qemu-system-x86 clock_gettime(228) 2 1.163 0.402 0.581 0.761 0 0 ( 0%)
183118 CPU 0/KVM ioctl(16) 21 65.423 0.613 3.115 7.953 0 0 ( 0%)
183119 CPU 1/KVM ioctl(16) 22 109.568 0.669 4.980 19.091 0 0 ( 0%)
2024-02-26 16:41:14.556469 qemu-system-x86 181860 .... [001] 5176622.910799: raw_syscalls:sys_enter: NR 271 (2c3b300, 7, 7ffdfbe33840, 0, 8, 0)
2024-02-26 16:41:15.556491 qemu-system-x86 181860 .... [001] 5176623.910821: raw_syscalls:sys_exit: NR 271 = 0
2024-02-26 16:41:16.548405
thread comm syscalls calls total(us) min(us) avg(us) max(us) err than(reqs)
-----
181860 qemu-system-x86 ppoll(271) 1 1000022.107 1000022.107 1000022.107 1000022.107 0 1 (100%)
181860 qemu-system-x86 poll(7) 1 4.640 4.640 4.640 4.640 0 0 ( 0%)
181860 qemu-system-x86 clock_gettime(228) 2 2.921 1.285 1.460 1.636 0 0 ( 0%)
```

未指定 key 属性，以 tid 作为 key。--perins 打印每个线程的系统调用延迟。--than 100ms 打印超过 100ms 的事件。err 统计系统调用出错的次数。raw_syscalls:sys_enter 和 raw_syscalls:sys_exit 这两个事件是固定的，不能指定其他事件，可以设置 trace events 过滤器。

```
[root@kvm ~]# perf-prof syscalls -e 'raw_syscalls:sys_enter/!id<10/' -e 'raw_syscalls:sys_exit/!id<10/' -m 512 --order -i 1000
2024-02-26 17:13:20.648768
syscalls calls total(us) min(us) avg(us) max(us) err
-----
poll(7) 1106 1543985.596 0.823 1396.008 500306.730 0
read(0) 26047 507751.544 0.413 19.493 56679.081 15004
open(2) 9709 24566.057 0.858 2.530 37.431 144
close(3) 22580 11307.817 0.330 0.500 66.867 11128
stat(4) 5147 8934.753 0.625 1.735 148.261 271
mmap(9) 3128 5667.843 0.609 1.811 21.746 0
write(1) 341 2497.265 0.551 7.323 224.064 0
fstat(5) 2222 2004.295 0.433 0.902 15.808 0
lstat(6) 355 351.728 0.398 0.990 1.974 2
lseek(8) 296 186.081 0.362 0.628 18.173 32
```

统计系统调用号小于 10 的延迟，默认以 common_pid 作为 key。

exit 和 exit_group 系统调用不会发生 raw_syscalls:sys_exit 事件，不满足发生关系。在内部测量的是 raw_syscalls:sys_enter 到 sched:sched_process_free 的延迟，统计从系统调用进入到进程释放之间的延迟。

Syscalls 不支持--detail 参数。统计到具体的系统调用耗时异常后，可以使用 multi-trace 的延迟分析来确认具体的耗时原因。

◆ 应用场景

在用户态和内核态边界上测量，可以区分延迟是用户态还是内核态导致的。

4.2.3. Nested-trace

统计函数调用关系，以及函数的执行耗时。函数调用可能会嵌套、递归。基于 multi-trace。

用法跟 multi-trace 有差异。

```
perf-prof nested-trace -e A,A_ret -e B,B_ret -e C,C_ret
```

每个-e 选项指定一对事件，用于确定入口和出口。如：函数调用进入和退出；系统调用进入和退出，等等。

```
--impl <impl>      Implementation of two-event analysis class. Dflt: delay.
                    call: analyze function calls, only for nested-trace.
                    call-delay: call + delay, only for nested-trace.
```

Call 分析函数调用关系， call-delay，分析函数调用关系，以及函数的执行耗时。

```
[root@kvm ~]# echo 'p:run_timer_softirq run_timer_softirq' >> /sys/kernel/debug/tracing/kprobe_events
[root@kvm ~]# echo 'r:run_timer_softirq_ret run_timer_softirq' >> /sys/kernel/debug/tracing/kprobe_events
[root@kvm ~]# perf-prof nested-trace -e irq:softirq_entry,irq:softirq_exit -e timer:timer_expire_entry,timer:timer_expire_exit -e
kprobes:run_timer_softirq,kprobes:run_timer_softirq_ret -i 1000 --impl call-delay --order
2024-02-26 17:47:03.218007
function call R      calls      total(us)  min(us)  p50(us)  p95(us)  p99(us)  max(us)
-----
softirq_entry        36103    282100.998    0.277    2.367    33.702    44.643    424.912
|-run_timer_softirq  13933    56943.405    0.369    2.662    12.132    18.168    32.614
|   |-timer_expire_entry 12777    11385.436    0.335    0.731    1.745    2.270    28.318
```

软中断的执行和 timer 的执行有一定的调用关系。左侧显示调用关系，右侧显示执行耗时。R 标记检测是否存在递归调用。

4.2.4. Rundelay

调度延迟分析。功能跟 multi-trace 一样，只是提供了自动化设置过滤器的功能。

测量三类调度延迟：

- 单个进程。-p pid、-t tid、workload
- 一类进程。--filter comm
- 所有进程。不设置-p、-t、--filter

```
[root@kvm ~]# perf-prof rundelay -e 'sched:sched_wakeup*,sched:sched_switch' -e 'sched:sched_switch' -p 181860 -i 1000 --perins -v
Wildcard expansion: 'sched:sched_wakeup*,sched:sched_switch' -> 'sched:sched_wakeup,sched:sched_wakeup_new,sched:sched_switch'
name sched_wakeup id 340 filter (null) stack 0
name sched_wakeup_new id 339 filter (null) stack 0
name sched_switch id 338 filter (null) stack 0
Wildcard expansion: 'sched:sched_switch' -> 'sched:sched_switch'
name sched_switch id 338 filter (null) stack 0
Trick: Enable userland unnecessary detection of sched:sched_wakeup events.
sched:sched_wakeup key "pid"
sched:sched_wakeup filter "pid==181860 || (pid==181883&&pid<=181884) || (pid==183118&&pid<=183119) || (pid==183121&&pid<=183123) || pid==184830"
sched:sched_wakeup_new key "pid"
sched:sched_wakeup_new filter "pid==181860 || (pid==181883&&pid<=181884) || (pid==183118&&pid<=183119) || (pid==183121&&pid<=183123) || pid==184830"
sched:sched_switch key "prev_pid"
sched:sched_switch filter "prev_state==0 && (prev_pid==181860 || (prev_pid==181883&&prev_pid<=181884) || (prev_pid==183118&&prev_pid<=183119) ||
(prev_pid==183121&&prev_pid<=183123) || prev_pid==184830)"
sched:sched_switch key "next_pid"
sched:sched_switch filter "next_pid==181860 || (next_pid==181883&&next_pid<=181884) || (next_pid==183118&&next_pid<=183119) || (next_pid==183121&&next_pid<=183123) ||
next_pid==184830"
Trick: Enable userland unnecessary detection of sched:sched_wakeup events.
2024-02-26 19:06:42.824007
pid  comm      start => end      calls      total(us)  min(us)  p50(us)  p95(us)  p99(us)  max(us)
-----
181860 qemu-system-x86 sched_wakeup => sched_switch      2      14.864      7.324      7.540      7.540      7.540      7.540
183118 CPU 0/KVM      sched_wakeup => sched_switch      6      47.328      6.236      8.372     10.092     10.092     10.092
```

183119	CPU 1/KVM	sched_wakeup => sched_switch	12	88.872	5.992	7.416	9.792	9.792	9.792
--------	-----------	------------------------------	----	--------	-------	-------	-------	-------	-------

测量 181860 进程的调度延迟。其中-e 指定的 4 个事件是固定不变的，自动添加过滤器，自动设定 key 属性，不可以设置为 untraced，可以添加其他属性。过滤器和 key 会自动设置（-v 参数能够显示过滤器字符串和 key）。

除了固定的 4 个事件之外，可以添加更多的 untraced 事件。Untraced 事件可以自己设置过滤器，key 等属性。

```
[root@kvm ~]# perf-prof rundelay -e 'sched:sched_wakeup*,sched:sched_switch' -e 'sched:sched_switch,sched:sched_migrate_task//key=pid/untraced/' --filter pyth*,awk -i 1000 --
than 2ms --detail=samecpu
Trick: Enable userland unnecessary detection of sched:sched_wakeup events.
Trick: Enable userland unnecessary detection of sched:sched_wakeup events.
2024-02-26 19:19:58.487519      awk 182927 d... [072] 5186146.850781: sched:sched_switch: awk:182927 [120] R ==> vstationd:182921 [120]
|      3777.419 us      vstationd 182921 d... [072] 5186146.851043: sched:sched_migrate_task: comm=vstationd pid=182921 prio=120 orig_cpu=72 dest_cpu=72
|      vstationd 182921 d.h. [072] 5186146.852798: sched:sched_migrate_task: comm=sap1015 pid=152678 prio=120 orig_cpu=72 dest_cpu=24
|      vstationd 182921 d... [072] 5186146.854138: sched:sched_migrate_task: comm=vstationd pid=182929 prio=120 orig_cpu=72 dest_cpu=48
|      vstationd 182921 d... [072] 5186146.854554: sched:sched_migrate_task: comm=vstationd pid=182923 prio=120 orig_cpu=48 dest_cpu=0
|      vstationd 182921 d... [072] 5186146.854558: sched:sched_switch: vstationd:182921 [120] S ==> awk:182927 [120]
2024-02-26 19:19:58.491296
2024-02-26 19:19:59.722419
start => end      calls      total(us)      min(us)      p50(us)      p95(us)      p99(us)      max(us)      than(reqs)
-----
sched_switch => sched_switch      320      13362.782      2.611      7.505      87.640      846.318      3777.419      1 ( 0%)
sched_wakeup => sched_switch      1218      11062.819      0.576      1.596      21.809      173.037      843.877      0 ( 0%)
sched_wakeup_new => sched_switch      2      5.355      1.935      3.420      3.420      3.420      3.420      0 ( 0%)
```

测量 pyth*,awk 进程的调度延迟。sched_migrate_task 是新加的 untraced 事件，未设置过滤。

4.2.4.1. 打印中间细节

--detail=samecpu 这个过滤对于 rundelay 有一定的特殊性。

- ◆ sched:sched_wakeup 事件，唤醒一个进程到指定的 **目标cpu** 上。
- ◆ sched:sched_migrate_task 事件，迁移一个进程到指定的 **目标cpu** 上。

对于调度延迟，其延迟原因一定发生在这个 **目标cpu** 上。对于 rundelay，--detail=samecpu 过滤会跟踪 sched:sched_wakeup、sched:sched_migrate_task 事件的目标 cpu，并只筛选这个目标 cpu 上的事件。

```
[root@kvm ~]# perf-prof rundelay -e 'sched:sched_wakeup*,sched:sched_switch' -e 'sched:sched_switch,sched:sched_migrate_task//untraced/' -m 512 -i 1000 --than 20ms --
detail=samecpu
2024-03-02 17:49:51.414151      safe_TsysAgent. 176719 d... [000] 5612740.1812901: sched:sched_wakeup_new: safe_TsysAgent.:176723 [120] success=1 CPU:024
|      24655.132 us      exec_agent 218830 dNh. [024] 5612740.181335: sched:sched_wakeup: pal_session:31989 [100] success=1 CPU:024
|      exec_agent 218830 d... [024] 5612740.181453: sched:sched_switch: exec_agent:218830 [120] R ==> pal_session:31989 [100]
|      exec_agent 218830 d... [024] 5612740.181453: sched:sched_switch: exec_agent:218830 [120] R ==> pal_session:31989 [100]
|      7.349 pal_session4      pal_session 31989 d... [024] 5612740.181460: sched:sched_switch: pal_session:31989 [100] S ==> exec_agent:218830 [120]
|      exec_agent 218830 dNh. [024] 5612740.181511: sched:sched_wakeup: pal_session:31989 [100] success=1 CPU:024
|
...
|      16247.225 [migrate]5      pal_session 31989 d... [024] 5612740.197520: sched:sched_switch: pal_session:31989 [100] S ==> ps:176721 [120]
|      memdump 176693 d.s. [072] 5612740.1975372: sched:sched_migrate_task: comm=safe_TsysAgent. pid=176723 prio=120 orig_cpu=24 dest_cpu=72
|      memdump 176693 dNs. [072] 5612740.198514: sched:sched_wakeup: kworker/72:1H:179485 [100] success=1 CPU:072
|      memdump 176693 d... [072] 5612740.198516: sched:sched_switch: memdump:176693 [120] R ==> kworker/72:1H:179485 [100]
|      memdump 176693 d... [072] 5612740.198517: sched:sched_switch: memdump:176693 [120] R ==> kworker/72:1H:179485 [100]
|
...
|      memdump 176693 d... [072] 5612740.205934: sched:sched_switch: memdump:176693 [120] R ==> grep:176722 [120]
|      memdump 176693 d... [072] 5612740.205935: sched:sched_switch: memdump:176693 [120] R ==> grep:176722 [120]
2024-03-02 17:49:51.438806      grep 176722 d... [072] 5612740.2059453: sched:sched_switch: grep:176722 [120] S ==> safe_TsysAgent.:176723 [120]
2024-03-02 17:49:53.402933
start => end      calls      total(us)      min(us)      p50(us)      p95(us)      p99(us)      max(us)      than(reqs)
-----
sched_wakeup => sched_switch      44520      862016.592      0.379      3.397      84.254      258.592      5996.165      0 ( 0%)
sched_switch => sched_switch      9120      346108.075      1.592      6.800      71.896      773.786      15849.178      0 ( 0%)
sched_wakeup_new => sched_switch      137      285063.957      1.048      40.878      11668.634      23573.548      24655.132      2 ( 1%)
```

¹ 唤醒 176723 线程到 CPU:024 上，之后就只过滤 CPU:024 上的事件。跟踪线程为什么在 CPU:024 上等待。

² 迁移 176723 线程到 CPU:072 上，之后就只过滤 CPU:072 上的事件。跟踪线程为什么在 CPU:072 上等待。

³ 线程开始执行。线程 176723 总调度延迟 24.6ms，在 CPU:024 上等待 16.2ms，在 CPU:072 上等待 8.4ms。

⁴提示进程的执行时间，可以观察到调度延迟内每个进程的执行时间。⁵提示发生一次迁移，16247.225 us 是从 ¹到 ²的时间偏移。

4.3. 进程

4.3.1. Task-state

任务状态分析。可以统计进程 R (running), RD (rundelay), S (sleeping), D (disk sleep), T (stopped), I (idle)等状态的分布情况。

测量三类进程状态：

- 单个进程。-p pid、-t tid
- 一类进程。--filter comm
- 所有进程。不设置-p、-t、--filter

选项参数

```
PROFILER OPTION:
--filter <filter>      Event filter/comm filter
-S, --interruptible    TASK_INTERRUPTIBLE, no- prefix to exclude
-D, --uninterruptible  TASK_UNINTERRUPTIBLE
--than <ns>            Greater than specified time, Unit: s/ms/us/*ns/percent
--perins               Print per instance stat
-g, --call-graph        Enable call-graph recording
--flame-graph <file>   Specify the folded stack file.
--ptrace               Use ptrace to track newly created threads.
```

--filter 过滤进程，只关注这些进程的状态。

-S, -D, 用来过滤进程状态，只关注 S 或 D 状态。

--than, 打印超过指定时间的事件。

--perins, 显示每个线程的状态统计情况。

-g, 打开堆栈。

--ptrace, 跟踪新建的线程，子进程等。

```
[root@kvm ~]# perf-prof task-state -i 1000 -m 128 -p 31958 --perins
2024-02-28 00:09:25.835289
thread comm      St  calls  total(us)  min(us)  p50(us)  p95(us)  p99(us)  max(us)
-----
31958 pal_main    R    967    3206.378    2.792    2.999    4.654    5.196    10.200
31958 pal_main    S    966   991444.431  1000.577  1018.976  1050.420  1051.041  1052.175
31958 pal_main    RD   967    2438.107    0.894    1.469    6.778    14.124    220.987
31974 worker0_0.23 R    99    989630.762  9990.640  9996.379  9998.807  10002.069  10002.069
31974 worker0_0.23 RD   100    374.146    3.403    3.729    3.971    4.125    4.125
31989 pal_session R  14611   89149.882    4.454    5.654    7.860    11.117    130.998
31989 pal_session S  14602   868154.867    4.998    60.202    60.879    61.343    87.849
31989 pal_session RD  14611   40948.154    0.740    1.399    7.898    13.912    1260.104
31990 pal_ctrl    R    965    9290.934    5.111    5.827    10.850    199.232    396.133
31990 pal_ctrl    S    964   985814.011   15.276   1016.296  1050.214  1050.946  1055.409
31990 pal_ctrl    RD   965    2197.995    1.137    1.552    5.334    10.166    18.505
```

显示进程 31958 的状态。

- R 状态统计线程在 CPU 上运行过多少次，以及分布情况。pal_main 线程 1 秒内共运行 967 次，总共执行时间 3206us，最大单次执行 10us。通过总运行时间，可以计算线程的 cpu 利用率。
- S 状态统计线程睡眠状态，从开始睡眠到被唤醒这中间的时间分布。pal_main 线程 1 秒内共睡眠 966 次，总共睡眠时间 991ms，平均睡眠时间 1ms。

- RD 状态统计线程调度延迟，从线程被唤醒到线程在 cpu 开始运行这中间的时间分布。

```
[root@kvm ~]# perf-prof task-state -i 1000 --filter 'java,python*' -S --than 1000ms -g
2024-02-28 00:24:39.272155 task-state: 174910 python WAIT 1000 ms
2024-02-28 00:24:36.884994 python 174910 d... [048] 5290825.350076: sched:sched_switch: python:174910 [120] S ==> swapper/48:0 [120]
ffffff816c51a3 __schedule+0x4f3 ([kernel.kallsyms])
ffffff816c5609 schedule+0x29 ([kernel.kallsyms])
ffffff816c4722 schedule_hrtimeout_range_clock+0xb2 ([kernel.kallsyms])
ffffff816c47d3 schedule_hrtimeout_range+0x13 ([kernel.kallsyms])
ffffff81228dc5 poll_schedule_timeout+0x55 ([kernel.kallsyms])
ffffff81229741 do_select+0x6d1 ([kernel.kallsyms])
ffffff81229a0b core_sys_select+0x1db ([kernel.kallsyms])
ffffff81229bea sys_select+0xba ([kernel.kallsyms])
ffffff816d13fe tracesys+0xe3 ([kernel.kallsyms])
00007ff66636ca33 Unknown+0x0 (/usr/lib64/libc-2.17.so)
000000001837010 Unknown
2024-02-28 00:24:37.885658 swapper 0 dNh. [048] 5290826.350741: sched:sched_wakeup: python:174910 [120] success=1 CPU:048
ffffff810c9295 ttwu_do_wakeup+0xb5 ([kernel.kallsyms])
ffffff810c939d ttwu_do_activate.constprop.115+0x5d ([kernel.kallsyms])
ffffff810cc21d try_to_wake_up+0x18d ([kernel.kallsyms])
ffffff810cc445 wake_up_process+0x15 ([kernel.kallsyms])
ffffff810ba3b2 hrtimer_wakeup+0x22 ([kernel.kallsyms])
ffffff810bab16 __hrtimer_run_queues+0xd6 ([kernel.kallsyms])
ffffff810bb0af hrtimer_interrupt+0xaf ([kernel.kallsyms])
ffffff81056f3b local_apic_timer_interrupt+0x3b ([kernel.kallsyms])
ffffff816d3933 smp_apic_timer_interrupt+0x43 ([kernel.kallsyms])
ffffff816d1e5d apic_timer_interrupt+0x6d ([kernel.kallsyms])
ffffff81541fde cpuidle_idle_call+0xde ([kernel.kallsyms])
ffffff8103412e arch_cpu_idle+0xe ([kernel.kallsyms])
ffffff810f654a cpu_startup_entry+0x14a ([kernel.kallsyms])
ffffff81055042 start_secondary+0x1d2 ([kernel.kallsyms])
2024-02-28 00:24:39.276591
St calls total(us) min(us) p50(us) p95(us) p99(us) max(us)
--
S 1184 8867721.059 2.662 1048.994 10050.700 100100.627 1000664.614
```

统计'java,python*'进程状态，只观察 S 状态。过滤超过 1000ms 的。-g 打开堆栈可以看到线程睡眠、唤醒的原因。python:174910 线程睡眠是因为执行 select 系统调用，唤醒是因为定时器到期。

```
[root@kvm ~]# perf-prof task-state -- ip link show eth0
2: eth0: <BROADCAST,MULTICAST,SLAVE,UP,LOWER_UP> mtu 9100 qdisc mq master bond1 state UP mode DEFAULT qlen 1000
link/ether b8:59:9f:d7:f5:78 brd ff:ff:ff:ff:ff:ff
2024-02-28 00:31:47.893170
St calls total(us) min(us) p50(us) p95(us) p99(us) max(us)
--
R 14 13164.998 202.284 1116.774 1635.135 1635.135 1635.135
S 1 13.700 13.700 13.700 13.700 13.700 13.700
RD 14 104.587 1.746 7.387 12.715 12.715 12.715
[root@kvm ~]# perf-prof task-state -- sleep 1
2024-02-28 00:33:36.316736
St calls total(us) min(us) p50(us) p95(us) p99(us) max(us)
--
R 6 1251.103 57.343 203.382 398.259 398.259 398.259
S 2 1000094.266 42.443 1000051.823 1000051.823 1000051.823 1000051.823
RD 6 64.130 1.530 12.924 28.805 28.805 28.805
```

执行命令，并观察命令执行期间进程的状态。R 状态、S 状态 total(us)、RD 状态之和就是进程实际执行时间。

4.3.1.1. ptrace

Ptrace 用来控制新建的线程。一旦附加到进程、线程后，如果不启用--ptrace 选项，则新建的线程或进程就无法被跟踪。

原理上，--ptrace 选项，利用 ptrace 跟踪进程，并打开 PTRACE_O_TRACEFORK 等功能，这样新建的线程默认处于 ptrace-stop 状态，为新线程克隆一个新的 task-state prof_dev 设备后，再通知新线程继续执行。这样就跟踪到新线程。

如果附加到 workload，则默认启用--ptrace 选项。

```
[root@kvm ~]# cat work.sh
#!/bin/bash
ps -ef|grep awk
[root@kvm ~]# perf-prof task-state --perins -- ./work.sh
2024-12-06 15:11:01.063353
thread comm St calls total(us) min(us) p50(us) p95(us) p99(us) max(us)
```


31910	work.sh	R	8	1455.878	24.566	102.993	697.327	697.327	697.327
31910	work.sh	S	3	25419.469	62.017	1379.939	23977.513	23977.513	23977.513
31910	work.sh	t	3	22932.743	23.444	280.997	22628.302	22628.302	22628.302
31910	work.sh	RD	8	78.621	3.847	5.401	42.852	42.852	42.852
31911	ps	R	7	5994.272	54.271	166.493	3942.374	3942.374	3942.374
31911	ps	S	3	135.677	24.408	24.927	86.342	86.342	86.342
31911	ps	D	2	17.782	4.487	13.295	13.295	13.295	13.295
31911	ps	RD	7	39.705	1.414	5.240	12.112	12.112	12.112
31912	grep	R	4	1232.452	121.679	214.362	714.860	714.860	714.860
31912	grep	S	2	146.825	67.515	79.310	79.310	79.310	79.310
31912	grep	RD	4	29.406	2.244	8.587	13.325	13.325	13.325

启用--ptrace 可能会导致新建的线程有轻微停顿，新建线程越频繁，停顿越多。根据业务实际情况，决定是否启用--ptrace。

4.3.1.2. 原理

其内部默认是跟踪 sched:sched_switch、sched:sched_wakeup、sched:sched_wakeup_new 这样的三个事件。并根据-p、-t、--filter、-SD 等选项自动设置这三个事件的过滤器

```
[root@kvm ~]# perf-prof task-state -vv -- sleep 1
sched:sched_switch filter "prev_pid==153959"
sched:sched_switch filter "next_pid==153959"
sched:sched_wakeup filter "pid==153959"
sched:sched_wakeup_new filter "pid==153959"
2024-02-28 00:39:29.953483 perf-prof 153958 d... [072] 5291718.419377: sched:sched_wakeup1: perf-exec:153959 [120] success=1 CPU:024
2024-02-28 00:39:29.953485 swapper 0 d... [024] 5291718.419379: sched:sched_switch2: swapper/24:0 [120] R ==> perf-exec:153959 [120]
2024-02-28 00:39:29.953684 sleep 153959 d... [024] 5291718.419578: sched:sched_switch3: perf-exec:153959 [120] R ==> migration/24:175 [0]
2024-02-28 00:39:29.953691 swapper 0 d... [048] 5291718.419585: sched:sched_switch4: swapper/48:0 [120] R ==> perf-exec:153959 [120]
2024-02-28 00:39:29.954282 sleep 153959 d... [048] 5291718.420175: sched:sched_switch5: sleep:153959 [120] S ==> swapper/48:0 [120]
2024-02-28 00:39:29.954308 swapper 0 dN... [048] 5291718.420202: sched:sched_wakeup6: sleep:153959 [120] success=1 CPU:048
2024-02-28 00:39:29.954309 swapper 0 d... [048] 5291718.420203: sched:sched_switch: swapper/48:0 [120] R ==> sleep:153959 [120]
2024-02-28 00:39:29.954397 sleep 153959 d... [048] 5291718.420291: sched:sched_switch: sleep:153959 [120] S ==> swapper/48:0 [120]
2024-02-28 00:39:30.954448 swapper 0 dNh... [048] 5291719.420342: sched:sched_wakeup: sleep:153959 [120] success=1 CPU:048
2024-02-28 00:39:30.954451 swapper 0 d... [048] 5291719.420345: sched:sched_switch: swapper/48:0 [120] R ==> sleep:153959 [120]
2024-02-28 00:39:30.954560 sleep 153959 d... [048] 5291719.420454: sched:sched_switch: sleep:153959 [120] x ==> swapper/48:0 [120]
2024-02-28 00:39:30.957584
St calls total(us) min(us) p50(us) p95(us) p99(us) max(us)
--
R 4 986.504 87.611 199.160 590.704 590.704 590.704
S 2 1000078.167 26.839 1000051.328 1000051.328 1000051.328 1000051.328
RD 4 11.775 0.984 2.472 6.489 6.489 6.489
```

-vv 显示原始事件。filter 显示自动设置的过滤器，由于过滤器会提前设置好，如果进程会新建线程，则没办法支持。

从 1 到 2 统计一次 RD 状态的耗时，从 153959 被唤醒到开始执行。

从 2 到 3 统计一次 R 状态的耗时，从 153959 开始执行到执行结束，R 状态切换出去，会放到 runqueue 队列末尾。

从 3 到 4 统计一次 RD 状态耗时，从 153959 R 切换出去到再切换回来。统计的也是一次调度延迟。

从 4 到 5 统计一次 R 状态耗时，执行大概 590us。

从 5 到 6 统计一次 S 状态耗时，从 153959 睡眠到被唤醒。这个进程切换会有多种状态，s 表示睡眠切换出去，统计的就是 S 状态耗时。D 表示 disk sleep 切换出去，统计的就是 D 状态耗时。以此类推。

◆ 内建事件

sched:sched_switch、sched:sched_wakeup、sched:sched_wakeup_new

◆ 应用场景

- 观察任务的状态。虚拟化场景，可用于观察 qemu vcpu 线程的运行情况。

4.3.2. Oncpu

观察 cpu 上运行的进程。或者，观察进程运行在哪些 cpu 上。


```
[root@kvm ~]# perf-prof oncpu -C 0-2
2024-02-28 01:24:34.618086
CPU1 SUM(ms)2 COMM:TID(ms)3
000 149 ksm:691(29.8ms) sh:6188(8.2ms) vstationd:6172(6.6ms) sap1016:152682(6.1ms) pal-shell:6182(4.8ms) sap1008:152624(4.6ms) safe_TsysAgent.:50411(4.5ms)
hcb:agent_mcd:108955(3.5ms) sap1009:152626(3.3ms)
001 4 kworker/1:0H:215142(2.5ms) CPU 1/KVM:101592(1.3ms) qemu-system-x86:100037(0.1ms) worker:115686(0.1ms) kworker/1:2:204103(0.1ms) ksoftirqd/1:15(0.0ms)
002 1 kworker/2:2H:181419(0.8ms) kworker/u448:1:226834(0.5ms)
```

¹显示 cpu。²显示这个 cpu 进程执行时间的总和。³显示每个进程的执行时间。

```
[root@kvm ~]# perf-prof oncpu -p 691,31958
2024-02-28 01:30:38.522024
THREAD COMM SUM(ms) CPUS(ms)4
691 ksm 48 0(19ms) 24(10ms) 48(7ms) 72(11ms)
31958 pal_main 2 24(2ms)
31974 worker0_0.23 992 23(992ms)
31989 pal_session 76 24(76ms)
31990 pal_ctrl 7 24(7ms)
```

显示进程的每个线程都在哪些 cpu 上执行过。⁴显示在每个 cpu 上的执行时间。

◆ 内建事件

sched:sched_switch、sched:sched_stat_runtime

◆ 应用场景

检测部分 cpu 利用率高的问题。

观察 Host 线程如何跟 vcpu 争抢 cpu。

4.4. 内存

4.4.1. Kmemleak

内存泄露分析。可以分析内核所有的内存分配器，以及用户态内存分配器。内核内存分配器包括：buddy、slab、kmalloc、vmalloc、percpu 等。用户态内存分配器：glibc、tcmalloc 等。

内存泄露检测原理，监控一个指定内存分配器的所有分配点和释放点。只有分配，没有释放，就是泄露。内存分配点会打开堆栈。命令结束时会聚合显示所有未释放内存的堆栈，未释放内存数量。会存在一定误差，只要内存存在持续泄露，跟踪足够长的时间，一定能够跟踪到泄露的位置。这是个 **概率思维**，有一定概率能找到内存泄露的位置。

选项参数

```
-e, --event <EVENT,...>      Event selector. use 'perf list tracepoint' to list available tp events.
                                EVENT,EVENT,...
                                EVENT: sys:name[/filter/ATTR/ATTR/...]
                                profiler[/option/ATTR/ATTR/...]
                                filter: trace events filter
                                ATTR:
                                    stack: sample_type PERF_SAMPLE_CALLCHAIN
                                    ptr=EXPR: kmemleak, ptr field, Dflt: ptr=ptr
                                    size=EXPR: kmemleak, size field, Dflt: size=bytes_alloc
                                EXPR:
                                    C expression. See `perf-prof expr -h` for more information.
--alloc <EVENT>               Memory alloc tracepoint/kprobe/uprobe
--free <EVENT>                 Memory free tracepoint/kprobe/uprobe
-g, --call-graph               Enable call-graph recording
--flame-graph <file>          Specify the folded stack file.
```

--alloc 指定内存分配点。--free 指定内存释放点。-g 打开内存分配点的堆栈。ptr 属性指定内存分配器返回的内存地址，同时也是释放的内存地址。size 属性指定分配的内存大小。

```
[root@kvm ~]# perf-prof trace -e kmem:kmalloc,kmem:kfree -N 2
2024-02-28 14:26:22.091263      ps 102295  .... [000] 5341330.603820: kmem:kmalloc: (seq_buf_alloc+0x17) call_site=ffffffff81238d77 ptr=0xffff8801b095f000
bytes_req=4096 bytes_alloc=4096 gfp_flags=GFP_KERNEL|GFP_NOWARN
2024-02-28 14:26:22.091268      ps 102295  .... [000] 5341330.603825: kmem:kfree: (seq_buf_free+0x35) call_site=ffffffff81238be5 ptr=0xffff8801b095f000

[root@kvm ~]# perf-prof kmemleak --alloc kmem:kmalloc//ptr=ptr/size=bytes_alloc/ --free kmem:kfree//ptr=ptr/ --order -m 128 -g
^C2024-02-28 12:21:44.005026
KMEMLEAK STATS:
TOTAL alloc1 163820 free2 170934

LEAKED BYTES REPORT3:
Leak of 102404 bytes in 55 objects allocated from:
  ffffffff811f0a4d __kmalloc+0x10d ([kernel.kallsyms])
  ffffffff81232660 alloc_fdmem+0x20 ([kernel.kallsyms])
  ffffffff8123272c alloc_fdttable+0x6c ([kernel.kallsyms])
  ffffffff81232cfd dup_fd+0x20d ([kernel.kallsyms])
  ffffffff8108afc9 copy_process+0xc09 ([kernel.kallsyms])
  ffffffff8108bee1 do_fork+0x91 ([kernel.kallsyms])
  ffffffff8108c1f6 sys_clone+0x16 ([kernel.kallsyms])
  ffffffff816d1549 stub_clone+0x69 ([kernel.kallsyms])
Leak of 4608 bytes in 9 objects allocated from:
```

分析 kmalloc 的内存泄露。ptr=ptr，ptr 属性指定 kmem:kmalloc 内存分配器返回的内存地址；指定 kmem:kfree 释放的内存地址。size=bytes_alloc 指定分配的字节数，通过 bytes_alloc 字段获取。

¹和²指定内存分配与释放的状态。³指示报告的内存泄露情况。所有内存泄露的堆栈会去重，相同的堆栈只报告一次。

⁴和⁵指定一个堆栈具体泄露的情况，发生 5 次内存分配，总共泄露 10240 字节。这是个误差，统计的时间足够长，就能够抓到真正的内存泄露。

```
perf-prof kmemleak --alloc kmem:kmalloc//ptr=ptr/size=bytes_alloc/,kmem:kmalloc_node//ptr=ptr/size=bytes_alloc/ \
--free kmem:kfree//ptr=ptr/ --order -m 128 -g
```

--alloc 和--free 指定的内存分配与释放点，可以指定多个。既可以是一个内存分配器的多种可能的入口。也可以是多个内存分配器的混合。

◆ 应用场景

一般用于内存泄露分析。可以借助/proc/meminfo 来判断大概的内存泄露方向，然后再使用 perf-prof kmemleak 来分析。

用户态的内存分配器。需要添加 uprobe 点，并导出内存分配和释放的地址，然后才可以使用 perf-prof kmemleak 来分析。

4.4.2. Kmemprof

内存热点分析。主要观察内存分配和释放的热点。

基于 multi-trace。基本用法：

```
perf-prof kmemprof -e 内存分配点 -e 内存释放点
```

使用-e 来指定内存分配与释放，不使用--alloc 和--free。

```
[root@kvm ~]# perf-prof kmemprof -e 'kmem:mm_page_alloc//size=4096<<order/key=page/stack/' -e kmem:mm_page_free//key=page/stack/ -m 256 --order
^C2024-02-28 15:01:15.793161
kmem:mm_page_alloc => kmem:mm_page_free
kmem:mm_page_alloc total alloc 1562132481 bytes on 366222 objects
Allocate 42074112 (26.9%3) bytes on 10272 (28.0%4) objects:
  ffffffff8119bf79 __alloc_pages_nodemask+0x279 ([kernel.kallsyms])
  ffffffff811e45d8 alloc_pages_vma+0xc8 ([kernel.kallsyms])
  ffffffff811c211c handle_mm_fault+0xc5c ([kernel.kallsyms])
  ffffffff816cc124 __do_page_fault+0x154 ([kernel.kallsyms])
  ffffffff816cc455 do_page_fault+0x35 ([kernel.kallsyms])
  ffffffff816c8648 page_fault+0x28 ([kernel.kallsyms])
...
Skipping alloc numbered 11..775
kmem:mm_page_free total free 156213248 bytes on 36622 objects
```

```
Free 73363456 (47.0%) bytes on 17911 (48.9%) objects:
ffffff81199ed4 free_pages_prepare+0x124 ([kernel.kallsyms])
ffffff8119a994 free_hot_cold_page+0x74 ([kernel.kallsyms])
ffffff8119aac6 free_hot_cold_page_list+0x46 ([kernel.kallsyms])
ffffff8119ff7e release_pages+0x24e ([kernel.kallsyms])
ffffff811d448d free_pages_and_swap_cache+0xad ([kernel.kallsyms])
ffffff811bc87c tlb_flush_mmu.part.66+0x6c ([kernel.kallsyms])
ffffff811bdeb5 tlb_finish_mmu+0x55 ([kernel.kallsyms])
ffffff811c8c9b exit_mmap+0xdb ([kernel.kallsyms])
ffffff81089967 mmpu+0x67 ([kernel.kallsyms])
ffffff81092f15 do_exit+0x285 ([kernel.kallsyms])
ffffff8109374f do_group_exit+0x3f ([kernel.kallsyms])
ffffff810937c4 sys_exit_group+0x14 ([kernel.kallsyms])
ffffff816d13fe tracesys+0xe3 ([kernel.kallsyms])
...
Skipping free numbered 11..84
```

¹和²指定总的内存分配字节数和内存分配次数。³和⁴指定一个内存分配堆栈的占比，占总分配字节数的 26.9%，占总分配次数的 28%。

⁵指定隐藏的部分。默认只输出 10 个堆栈，剩余部分都会被隐藏。

◆ 应用场景

业务性能优化，可以借此发现内存分配热点。优化内存分配方式。

用户态的内存分配器，需要添加 uprobe 点。

4.5. 虚拟化

4.5.1. Kvm-exit

虚拟化退出耗时。测量 kvm:kvm_exit => kvm:kvm_entry 之间的延迟，测量的是每一次退出在 kvm 模块内的处理时长。

可以使用 multi-trace 来统计耗时。但 vmexit 的数量非常大，multi-trace 的性能达不到，kvm-exit 是高性能版本。

选项参数

```
--filter <filter>      Event filter/comm filter
--than <ns>            Greater than specified time, Unit: s/ms/us/*ns/percent
--perins               Print per instance stat
--heatmap <file>       Specify the output latency file.
```

--filter 设置 kvm:kvm_exit 的 trace events 过滤器。

--than 打印高延迟的事件。

--perins 打印每个实例的情况。

```
[root@kvm ~]# perf-prof kvm-exit -p 100037 -i 1000
2024-02-28 15:27:29.648194 kvm-exit latency
exit_reason      calls      total(us)  min(us)  avg(us)  p99(us)  max(us)  %gsys
-----
HLT               17      1414631.765   69.335   83213.633 250286.355 250286.355 100.00
EPT_VIOLATION     80       213.203     0.717     2.665     14.424    14.424 100.00
MSR_WRITE         68       52.335     0.573     0.769     1.271     1.271 100.00
EXTERNAL_INTERRUPT 1         0.729     0.729     0.729     0.729     0.729  0.00
```

统计 QEMU 进程 100037 的虚拟化退出耗时。exit_reason 指定每一种退出原因。当前支持 Intel 和 AMD 平台。

```
[root@kvm ~]# perf-prof kvm-exit -p 100037 -i 1000 --perins --filter 'exit_reason != 12'
2024-02-28 15:32:48.236912 kvm-exit latency
[THREAD] exit_reason      calls      total(us)  min(us)  avg(us)  p99(us)  max(us)  %gsys
-----
[101591] IO_INSTRUCTION     18       78.221     1.276     4.345    10.627    10.627 100.00
[101591] MSR_WRITE          47      35.830     0.565     0.762     1.234     1.234 100.00
```

[101591]	EXTERNAL_INTERRUPT	1	0.741	0.741	0.741	0.741	0.741	100.00
[101592]	EPT_VIOLATION	80	186.778	0.720	2.334	10.408	10.408	100.00
[101592]	IO_INSTRUCTION	19	103.241	1.697	5.433	36.266	36.266	100.00
[101592]	MSR_WRITE	61	44.488	0.561	0.729	1.132	1.132	100.00
[101592]	PAUSE_INSTRUCTION	3	2.185	0.419	0.728	1.340	1.340	100.00

打印每个 vcpu 线程的虚拟化退出耗时。exit_reason 是 kvm:kvm_exit 点的字段。exit_reason != 12 是过滤掉 HLT 退出。

◆ 内建事件

kvm:kvm_exit、kvm:kvm_entry

◆ 应用场景

用于分析虚拟化层的损耗。

- Guest 内性能压测从开始到结束，可以统计出总的虚拟化层耗时，以此判断虚拟化对性能的影响。所有 exit_reason 的 total(us)累加起来就是虚拟化层总的损耗。
- 线上监控 EPT 退出次数及退出耗时。虚拟机热迁移时的 EPT 退出情况。

4.6. 块设备

4.6.1. Blktrace

统计块设备 request 请求在各阶段的耗时。

Q - 即将生成 IO 请求
G - IO 请求生成
I - IO 请求进入 IO Scheduler 队列
D - IO 请求进入 driver
C - IO 请求执行完毕

选项参数

--than <ns> Greater than specified time, Unit: s/ms/us/*ns/percent
-d, --device <device> Block device, /dev/sdx

```
[root@kvm ~]# perf-prof blktrace -d /dev/sda -i 1000 -m 32
2024-02-28 15:53:08.179605
start => end                reqs      total(us)  min(us)   avg(us)   max(us)
-----
block_getrqC => block_rq_insertI    3091    361196.921    2.857    116.854    1166.554
block_rq_insertI => block_rq_issueD    1233     31042.043     0.973     25.176     105.535
block_rq_issueD => block_rq_completeC  1993    273747.745    33.492    137.354     353.104
```

统计/dev/sda 设备 GIDC 各个阶段的耗时。

◆ 内建事件

block:block_getrq、block:block_rq_insert、block:block_rq_issue、block:block_rq_complete

◆ 应用场景

跟踪 IO 的耗时。perf-prof blktrace 单条命令，数据不落盘。比 blktrace、blkparse 易用。

4.7. 硬件与调试

4.7.1.Profile

以指定频率采样分析。使用的是硬件 PMU：cycles。硬件 PMU 在 x86 平台都是发起 NMI 中断，可以采样到关中断的场景。

选项参数

```
FILTER OPTION:
-G, --exclude-host      Monitor GUEST, exclude host
--exclude-guest         exclude guest
--exclude-user          exclude user
--exclude-kernel        exclude kernel
--user-callchain        include user callchains, no- prefix to exclude
--kernel-callchain      include kernel callchains, no- prefix to exclude
--irqs_disabled[=<0|1>] ebpf, irqs disabled or not.
                        (not built-in because NO CONFIG_LIBBPF=y)
--tif_need_resched[=<0|1>]
                        ebpf, TIF_NEED_RESCHED is set or not.
                        (not built-in because NO CONFIG_LIBBPF=y)
--exclude_pid <pid>    ebpf, exclude pid
                        (not built-in because NO CONFIG_LIBBPF=y)
--nr_running_min <n>   ebpf, minimum number of running processes for CPU runqueue.
                        (not built-in because NO CONFIG_LIBBPF=y)
--nr_running_max <n>   ebpf, maximum number of running processes for CPU runqueue.
                        (not built-in because NO CONFIG_LIBBPF=y)

PROFILER OPTION:
-F, --freq <n>         Profile at this frequency, No profile: 0
--than <ns>           Greater than specified time, Unit: s/ms/us/*ns/percent
-g, --call-graph       Enable call-graph recording
--flame-graph <file>  Specify the folded stack file.
```

过滤器：--exclude-*是内核 perf 系统的过滤器，只能过滤用户态、内核态。

ebpf 过滤器：需要编译时带上 CONFIG_LIBBPF=y。

-F 指定采样频率。-g 打开采样堆栈。

```
[root@kvm ~]# perf-prof profile -F 1000 -g -C 0 -N 2
2024-02-28 16:25:54.388121      sh  6881 [000] 5348502.907323: profile: 2504300 cpu-cycles
ffffffffff812190b5  vfs_getattr+0x5 ([kernel.kallsyms])
ffffffffff8121977e  SYSC_newstat+0x2e ([kernel.kallsyms])
ffffffffff81219a5e  sys_newstat+0xe ([kernel.kallsyms])
ffffffffff816d13fe  tracesys+0xe3 ([kernel.kallsyms])
00007fa9d6430355  _xstat+0x15 (/usr/lib64/libc-2.17.so)
00000000048172f  glob_filename+0x31f (/usr/bin/bash)
0000000021000000  Unknown
2024-02-28 16:25:54.389127      sh  6881 [000] 5348502.908329: profile: 2498900 cpu-cycles
ffffffffff8122e1ee  __d_lookup+0x6e ([kernel.kallsyms])
ffffffffff8121f5c8  lookup_fast+0x188 ([kernel.kallsyms])
ffffffffff81221a05  path_lookupat+0x165 ([kernel.kallsyms])
ffffffffff8122208b  filename_lookup+0x2b ([kernel.kallsyms])
ffffffffff81225d37  user_path_at_empty+0x67 ([kernel.kallsyms])
ffffffffff81225da1  user_path_at+0x11 ([kernel.kallsyms])
ffffffffff81219213  vfs_fstatat+0x63 ([kernel.kallsyms])
ffffffffff812197e1  SYSC_newlstat+0x31 ([kernel.kallsyms])
ffffffffff81219a6e  sys_newlstat+0xe ([kernel.kallsyms])
ffffffffff816d13fe  tracesys+0xe3 ([kernel.kallsyms])
00007fa9d64303f5  _lxstat+0x15 (/usr/lib64/libc-2.17.so)
00000000048172f  glob_filename+0x31f (/usr/bin/bash)
0000000021000000  Unknown
```

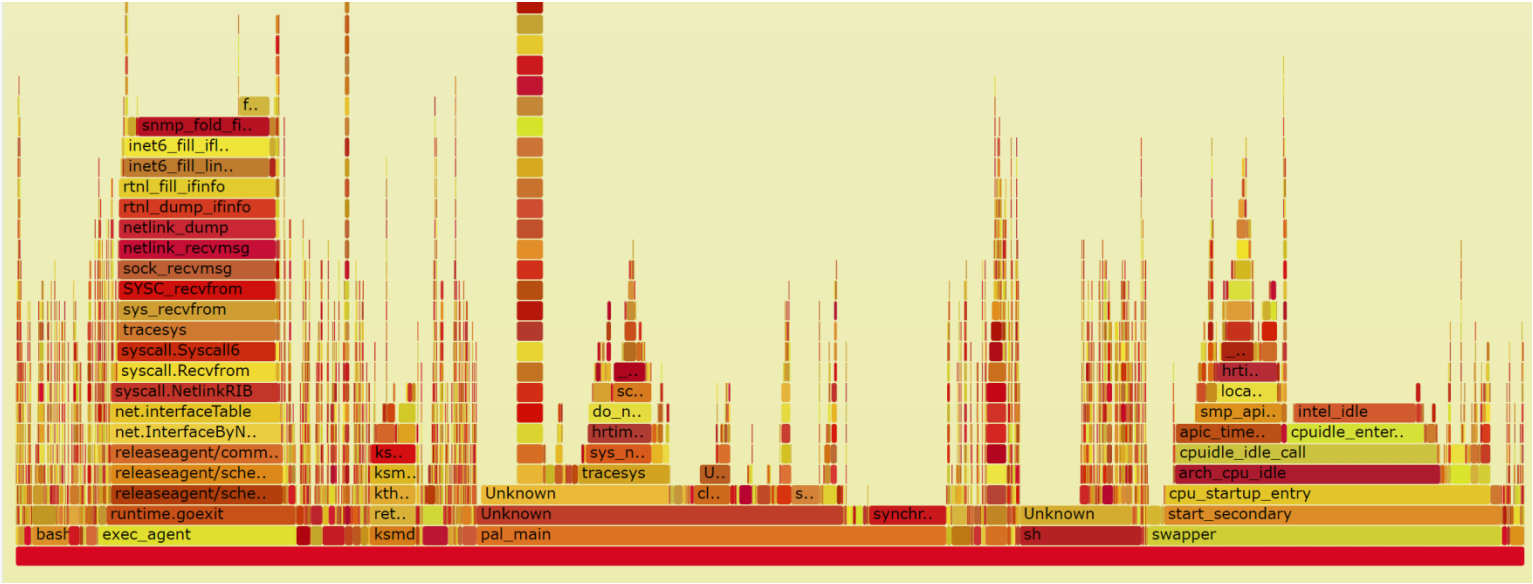
以 1000 频率采样 CPU:0，观察堆栈。

```
[root@kvm ~]# perf-prof profile -F 997 -C 24 -g --flame-graph cpu
...
2024-02-28 16:54:39.559678      swapper  0 [024] 5350228.080499: profile: 2507600 cpu-cycles
^Cto generate the flame graph, running THIS shell command:

    flamegraph.pl cpu.folded > cpu.svg

[root@kvm ~]# flamegraph.pl cpu.folded > cpu.svg
```

采样 CPU:24 的堆栈，并生成火焰图。



从火焰图上能够很清楚的看到哪些堆栈占比比较大。

◆ 内建事件

cycles

◆ 应用场景

一般用于分析 cpu 利用率高的场景。某个 cpu 的利用率高，附加到 CPU 上采样。进程 cpu 利用率高，附加到进程上采样。

通过 mpstat、top 等先观察到哪些 CPU、进程的 cpu 消耗占用比较高，然后再利用 perf-prof profile 采样。

4.7.2. Breakpoint

断点分析。用于捕获虚拟地址的读、写、执行。目前支持 x86 平台，每个 cpu 上可以添加 4 个断点。

基本用法

```
perf-prof breakpoint <addr>[/1/2/4/8][:rwx] ...
```

addr 指定一个虚拟地址。可以是用户态的，也可以是内核态的。

/1/2/4/8 指定虚拟地址对应的数据长度，分别对应 1、2、4、8 字节。

rwx 指定内存访问类型，分别对应读、写、执行。

```
[root@kvm ~]# cat /proc/kallsyms | grep ignore_msrs
ffffffffc2cc004a d ignore_msrs [kvm]
[root@kvm ~]# perf-prof breakpoint 0xffffffffc2cc004a/1:rw -g
2024-02-28 18:55:39.210336 cat 123827 [072] 5357487.737906: breakpoint: 0xffffffffc2cc004a/1:RW
RIP: ffffffff810b3ad7 RSP: ffff8834ec6cbe80 RFLAGS:00000246
RAX: ffffffff810b3ad7 RBX: ffff883491b4b628 RCX: ffff8834ec6cbfd8
RDX: 0000000000000000 RSI: ffffffff81948368 RDI: ffff8834b4b13000
RBP: ffff8834ec6cbe80 R08: ffffffff8193cc98 R09: ffffea00d2d2c500
R10: ffffea00d2d2c500 R11: 0000000000000246 R12: ffff8834b4b13000
R13: ffff8834e1e7e310 R14: ffff8834bcb38240 R15: ffffffff816f6d90
CS: 0010 SS: 0018
ffffffff810b3ad7 param_get_bool+0x17 ([kernel.kallsyms])
ffffffff810b3f81 param_attr_show+0x41 ([kernel.kallsyms])
ffffffff810b36f0 module_attr_show+0x20 ([kernel.kallsyms])
ffffffff812941e9 sysfs_read_file+0x99 ([kernel.kallsyms])
ffffffff8121390f vfs_read+0x9f ([kernel.kallsyms])
ffffffff812147df sys_read+0x7f ([kernel.kallsyms])
ffffffff816d13fe tracesys+0xe3 ([kernel.kallsyms])
00007f3c6d7a7a30 Unknown
```

```
2024-02-28 18:55:46.329124      bash 118391 [024] 5357494.856694: breakpoint: 0xffffffffc2cc004a/1:RW
RIP: ffffffff81351505 RSP: ffff8834f476be60 RFLAGS:00000246
RAX: 0000000000000000 RBX: ffff883491b4b628 RCX: ffff8834f476bfd8
RDX: 0000000000000000 RSI: ffffffff82cc004a RDI: ffff8834a7fdc000
RBP: ffff8834f476be60 R08: ffffffff82cc3210 R09: ffffea00d29ff740
R10: ffffea00d29ff740 R11: 0000000000000246 R12: 0000000000000002
R13: ffff8834a7fdc000 R14: ffff8834e1e7e310 R15: ffffffff816f6d90
CS: 0010 SS: 0018
ffffffff81351505 kstrtobool+0x45 ([kernel.kallsyms])
ffffffff810b3e40 param_set_bool+0x20 ([kernel.kallsyms])
ffffffff810b37d4 param_attr_store+0x44 ([kernel.kallsyms])
ffffffff810b3720 module_attr_store+0x20 ([kernel.kallsyms])
ffffffff812940db sysfs_write_file+0xcb ([kernel.kallsyms])
ffffffff81213aa0 vfs_write+0xc0 ([kernel.kallsyms])
ffffffff812148bf sys_write+0x7f ([kernel.kallsyms])
ffffffff816d13fe tracesys+0xe3 ([kernel.kallsyms])
00007f599a510a90 __write_nocancel+0x7 (/usr/lib64/libc-2.17.so)
315f395f6d766b40 Unknown
```

检测 kvm 模块 ignore_msr 参数的读写。其在内核态的地址是：ffff82cc004a，占用 1 个字节。

执行一次读和写。

```
[root@kvm ~]# cat /sys/module/kvm/parameters/ignore_msrs
Y
[root@kvm ~]# echo 1 > /sys/module/kvm/parameters/ignore_msrs
```

能够检测到读和写的堆栈、以及读写时对应的通用寄存器信息，便于反汇编分析。

◆ **内建事件**

breakpoint pmu

◆ **应用场景**

- ◆ 检测内核全局变量的改写。一般对应着内核参数。
- ◆ 检测内存地址的改写。需要使用 crash、gdb 等工具找到具体的内存地址，函数地址等。
- ◆ 检测内存越界写。需要通过调试可以找到内存的边界。

4.7.3. Page-faults

缺页异常。采集 cpu 或进程发生的缺页异常。

选项参数

```
--exclude-user      exclude user
--exclude-kernel    exclude kernel
-g, --call-graph     Enable call-graph recording
```

使用--exclude-*可以简单的过滤掉用户态还是内核态的缺页异常。

```
[root@kvm ~]# perf-prof page-faults -C 0 -g --exclude-user -N 1
2024-02-28 19:08:51.386618      grep 213587 [000] 5358279.914919: page-fault: addr 0000000001c1f000
ffffffff81346fb0 copy_user_generic_string+0x30 ([kernel.kallsyms])
ffffffff81212ee3 do_sync_read+0x93 ([kernel.kallsyms])
ffffffff8121390f vfs_read+0x9f ([kernel.kallsyms])
ffffffff812147df sys_read+0x7f ([kernel.kallsyms])
ffffffff816d13fe tracesys+0xe3 ([kernel.kallsyms])
RIP: 00007fdef0cea30 RSP: 00007fff6efe6868 RFLAGS:00000246
RAX: ffffffffffffffffda RBX: ffffffff816d13fe RCX: ffffffffffffffff
RDX: 0000000000004000 RSI: 0000000001c1f000 RDI: 0000000000000000
RBP: 0000000000001000 R08: ffffffffffffffff R09: 0000000000000000
R10: 0000000000001000 R11: 0000000000000246 R12: ffffffffffffffff7
R13: 0000000000000000 R14: 0000000001c1f000 R15: 0000000000004000
CS: 0033 SS: 002b
00007fdef0cea30 __read_nocancel+0x7 (/usr/lib64/libc-2.17.so)
```

过滤掉用户态，采集 CPU:0 发生的缺页异常。可以看到在内核态，往用户态内存上拷贝数据时发生缺页异常。访问的内存地址是：**1c1f000**


```
[root@kvm ~]# perf-prof page-faults -p 56988 -g -N 1
2024-02-28 19:11:39.202893      exec_agent 122591 [000] 5358447.731346: page-fault: addr 000000c000474180
    RIP: 0000000000463933 RSP: 000000c000161840 RFLAGS:00010283
    RAX: 0000000000000000 RBX: 00000000000005e80 RCX: 0000000000018180
    RDX: 00000000000005e80 RSI: 00007fa566507140 RDI: 000000c000474180
    RBP: 000000c0001618a0 R08: 0000000000000001 R09: 000000000001e000
    R10: 6ec24cfa1cad8201 R11: 6ec24cfa1caca92 R12: 0000000000000002
    R13: 00000000009afe40 R14: 0000000000000000 R15: 0000000000462900
    CS: 0033 SS: 002b
0000000000463933 runtime.memclrNoHeapPointers+0x113 (/usr/local/services/release_agent-1.0/bin/exec_agent)
00000000004a26d0 syscall.NetlinkRIB+0x670 (/usr/local/services/release_agent-1.0/bin/exec_agent)
000000000050d688 net.interfaceTable+0x48 (/usr/local/services/release_agent-1.0/bin/exec_agent)
000000000050c73e net.Interfaces+0x2e (/usr/local/services/release_agent-1.0/bin/exec_agent)
00000000005c5d95 releaseagent/common.GetIfIps+0x45 (/usr/local/services/release_agent-1.0/bin/exec_agent)
00000000006c44f3 releaseagent/schedule.Heartbeat+0x83 (/usr/local/services/release_agent-1.0/bin/exec_agent)
00000000006c59f3 releaseagent/schedule.addJob.func1+0x93 (/usr/local/services/release_agent-1.0/bin/exec_agent)
0000000000462901 runtime.goexit+0x1 (/usr/local/services/release_agent-1.0/bin/exec_agent)
```

观察 56988 进程发生的缺页异常。打印的通用寄存器信息是用户态的堆栈的，用于反汇编分析。

◆ 内建事件

page-faults

◆ 应用场景

检测进程的缺页异常。

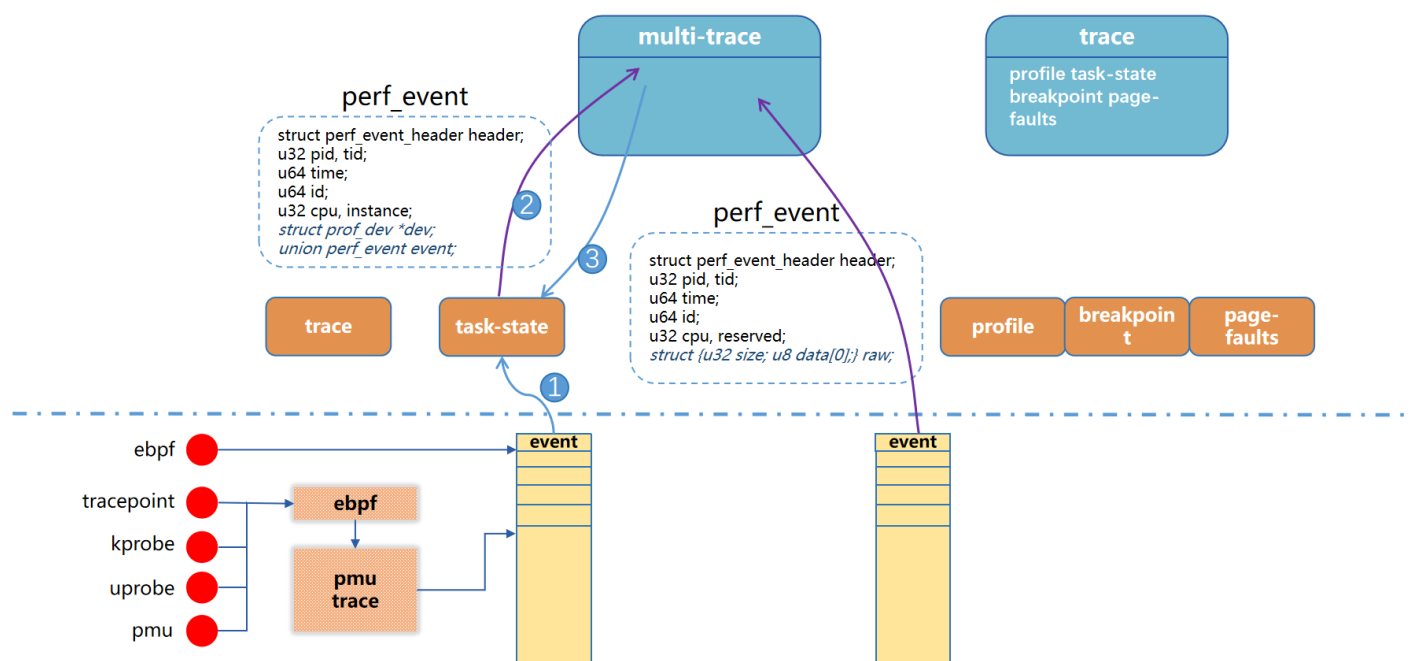
5. 联合分析

[illegible]

对于 tracepoint、kprobe、uprobe 事件源可以使用 sys:name 表示。

对于 ebp、pmu 事件源，没办法使用 sys:name 表示，其是封装在 profiler 内部，属于 profiler 的内建事件，因此这些 profiler 本身可以看做事件源。经过扩展，所有含有内建事件的 profiler，都可以看做事件源。

联合分析，就是把这些 tracepoint、kprobe、uprobe 事件源，profiler 事件源产生的事件联合起来一起分析。



以 task-state 为例，其内部使用 sched:sched_switch、sched:sched_wakeup、sched:sched_wakeup_new 这三个内建事件。Task-state 可以看做一个事件源，使其参与 multi-trace 的延迟分析。详细的处理过程如下：

1. 打开一个 `prof_dev` 设备，为 `task-state` 提供工作环境。从 `prof_dev` 的 `ringbuffer` 接收三个内建事件。`Task-state` 作为事件源，也是以独立的设备工作的。
2. 把接收的事件封装成一个特殊的 `perf_event`，包含一些标准头，源 `prof_dev`，原始事件。封装后的 `perf_event` 转发给 `multi-trace`。`Multi-trace` 既可以接收 `tracepoint` 事件，也可以接收 `profiler` 事件源转发过来的事件。`Multi-trace` 对这些事件的处理方式是一致的。
3. `Multi-trace` 处理事件，如果事件需要输出，则判断事件来源。如果是特殊的 `perf event`，则转发回源 `prof dev` 进行输出或其他处理。

这样， profiler 既可以作为剖析器，也可以作为事件源。使用 `-e profiler` 来打开一个 profiler 作为事件源。

使用包含内建事件的 profiler 作为事件源，有一定的优势。

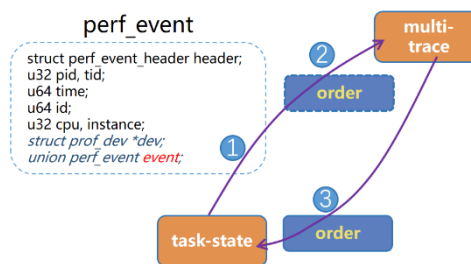
- ◆ 这个 profiler 内部可以使用 ebp、pmu 事件源，且能够支持多种硬件平台，如：arm64，x86(Intel、AMD)等。通过 profiler 来封装差异。
- ◆ Profiler 内包含的事件、过滤器、处理逻辑、输出显示，可以作为一个整体来使用。

◆ 性能问题

使用联合分析，会由于内存拷贝导致性能问题。还是以 task-state 和 multi-trace 作为示例。

给 multi-trace 转发事件，再转发回 task-state，要经历三次内存拷贝。

1. 封装成特殊的 perf_event 时，需要拷贝一次事件。
2. Multi-trace 启用 order 排序，在 order 内部需要拷贝事件。
3. 转发回源 task-state，经过 order 时需要拷贝事件。



为了减少性能问题，使用 profiler 事件源时，应尽可能减少转发的事件量。

5.1. Multi-trace

multi-trace 可以接受 trace、profile、task-state、breakpoint、page-faults 这几个作为事件源。只有这些 profiler 是单独设计过的，其他的 profiler 暂时未涉及。

在 multi-trace 内部，profiler 事件源会被默认加上 untraced 标记。也就是说，profiler 事件源不能参与延迟分析，只能作为还原中间细节来使用。基本用法：

```
perf-prof multi-trace -e A -e B,profiler/option/untraced/
```

测量 A=>B 的延迟，并添加 profiler 事件源。其中 profiler 事件源和 multi-trace 附加到相同的实例。

```
[root@kvm ~]# perf-prof multi-trace -e raw_syscalls:sys_enter -e 'raw_syscalls:sys_exit,task-state//untraced/' --order --than 20ms --detail=sametid -- sleep 0.1
task-state events are forwarded to multi-trace, reset instance.
2024-02-29 19:38:50.094146      sleep 11982 .... [072] 5446478.7021401: raw_syscalls:sys_enter: NR 35 (7fff83e632f0, 0, 5f5e100, 7fff83e626e0, 1, 1)
| 100059.988 us                sleep 11982 d... [072] 5446478.7021412: sched:sched_switch: sleep:11982 [120] S ==> swapper/72:0 [120]
|                               swapper      0 dNh. [072] 5446478.8021933: sched:sched_wakeup: sleep:11982 [120] success=1 CPU:072
|                               swapper      0 d... [072] 5446478.8021964: sched:sched_switch: swapper/72:0 [120] R ==> sleep:11982 [120]
|                               sleep 11982 .... [072] 5446478.8022005: raw_syscalls:sys_exit: NR 35 = 0
2024-02-29 19:38:50.194206
2024-02-29 19:38:50.197859
start => end      calls      total(us)  min(us)  p50(us)  p95(us)  p99(us)  max(us)  than(reqs)
-----
sys_enter => sys_exit  132      100511.704    0.309    1.642    6.841    161.683  100059.988    1 ( 0%)
```

测量 sleep 命令的系统调用耗时，打印耗时超过 20ms 的事件。添加 task-state 事件源，采集耗时中间，任务状态变化情况。

¹执行 nanosleep(NR 35)系统调用。²sleep 进程切换出去。³sleep 进程被唤醒。在 100ms 之后被唤醒的。⁴sleep 进程开始执行。⁵系统调用返回。

Multi-trace 执行 sleep 0.1 命令，附加到 11982 进程，task-state 也附加到 11982 进程。

```
[root@kvm ~]# perf-prof multi-trace -e raw_syscalls:sys_enter -e 'raw_syscalls:sys_exit,task-state/-g/untraced/,profile/-F 200 --watermark 50 -m 32 -g/untraced/' --order --
than 20ms --detail=sametid -- insmod ./delay.ko
task-state events are forwarded to multi-trace, reset instance.
insmod: ERROR: could not insert module ./delay.ko: Operation not permitted
2024-02-29 20:19:56.851690      insmod 67339 .... [072] 5448945.4617991: raw_syscalls:sys_enter: NR 313 (3, 41a15c, 0, 3, 0, 7ffc92f1cf38)
| 61388.232 us                insmod 67339 [072] 5448945.465744: profile: 12506700 cpu-cycles
|                               insmod 67339 [072] 5448945.470745: profile: 12494800 cpu-cycles
|                               insmod 67339 [072] 5448945.475747: profile: 12499700 cpu-cycles
|                               insmod 67339 [072] 5448945.480749: profile: 12499900 cpu-cycles
|                               insmod 67339 [072] 5448945.485752: profile: 12500900 cpu-cycles
|                               insmod 67339 [072] 5448945.490754: profile: 12499300 cpu-cycles
|                               insmod 67339 [072] 5448945.495756: profile: 12499600 cpu-cycles
|                               insmod 67339 [072] 5448945.500757: profile: 12500100 cpu-cycles
|                               insmod 67339 [072] 5448945.505759: profile: 12499700 cpu-cycles
|                               insmod 67339 [072] 5448945.5107602: profile: 12500000 cpu-cycles
|
| ffffffff81347475 delay_tsc+0x45 ([kernel.kallsyms])
| ffffffff813473e1 __udelay+0x31 ([kernel.kallsyms])
| ffffffff800f3027 __key.25126+0x2a13 ([kernel.kallsyms])
| ffffffff810021fa do_one_initcall+0xba ([kernel.kallsyms])
| ffffffff8110f7b4 load_module+0x2424 ([kernel.kallsyms])
```

```

ffffff8110ff56 sys_finit_module+0xa6 ([kernel.kallsyms])
ffffff816d13fe tracesys+0xe3 ([kernel.kallsyms])
00007f95aab9ad19 Unknown
20646564616f6c00 Unknown
|
|           insmod 67339 d... [072] 5448945.514928: sched:sched_switch: insmod:67339 [120] R ==> migration/72:509 [0]
|           swapper 0 d... [024] 5448945.514935: sched:sched_switch: swapper/24:0 [120] R ==> insmod:67339 [120]
|           insmod 67339 d... [024] 5448945.514952: sched:sched_switch: insmod:67339 [120] D ==> migrate_vpc:177428 [98]
ffffff816c51a3 __schedule+0x4f3 ([kernel.kallsyms])
ffffff816c5609 schedule+0x29 ([kernel.kallsyms])
ffffff816c3099 schedule_timeout+0x239 ([kernel.kallsyms])
ffffff816c59bd wait_for_completion+0xfd ([kernel.kallsyms])
ffffff810b338e wait_rcu_gp+0x5e ([kernel.kallsyms])
ffffff81147ddb synchronize_sched+0x3b ([kernel.kallsyms])
ffffff8110f8cb load_module+0x253b ([kernel.kallsyms])
ffffff8110ff56 sys_finit_module+0xa6 ([kernel.kallsyms])
ffffff816d13fe tracesys+0xe3 ([kernel.kallsyms])
00007f95aab9ad19 Unknown
20646564616f6c00 Unknown
|
|           pal_session 31989 d.s. [024] 5448945.522472: sched:sched_wakeup: insmod:67339 [120] success=1 CPU:072
|           ksoftirqd/72 510 d... [072] 5448945.522485: sched:sched_switch: ksoftirqd/72:510 [120] S ==> insmod:67339 [120]
|           insmod 67339 d... [072] 5448945.522865: sched:sched_switch: insmod:67339 [120] R ==> migration/72:509 [0]
|           sap1008 152624 d... [072] 5448945.522919: sched:sched_switch: sap1008:152624 [120] S ==> insmod:67339 [120]
2024-02-29 20:19:56.913078 insmod 67339 .... [072] 5448945.523187: raw_syscalls:sys_exit: NR 313 = -1
2024-02-29 20:19:56.924461
-----
start => end      calls      total(us)    min(us)      p50(us)      p95(us)      p99(us)      max(us)      than(reqs)
-----
sys_enter => sys_exit 154      61885.592    0.308        1.340        5.337        211.935     61388.232    1 ( 0%)

```

执行 insmod 命令，测量系统调用耗时，并打印耗时超过 20ms 的事件。输出耗时中间的任务状态变化情况，并以-F 200 频率采样堆栈。

- ¹ 执行 finit_module 系统调用，耗时 61388us。
- ² 采样的中间细节。每 5ms 采样一个点，大概采样到 10 个点。都在执行 udelay。
- ³ 进程 D 住，在等待 rcu 同步。

综合来看：61ms 的延迟，其中有 50ms 在执行 udelay 延迟，7.5ms 在等待 rcu 同步。

◆ 应用场景

主要用于还原延迟的中间细节。

- ◆ 猜测延迟中间有任务等待。加上 task-state 事件源，采样调度细节。
- ◆ 猜测延迟中间有死循环。加上 profile 事件源，采样堆栈。
- ◆ 猜测延迟中间有缺页异常。加上 page-faults 事件源。
- ◆ 需要捕获延迟中间对某个内存的改写。加上 breakpoint 事件源。

5.2. Trace

Trace 主要用于排序显示事件。基本用法：

```
perf-prof trace -e A,B,profiler/option/
```

```

[root@kvm ~]# perf-prof trace -e 'raw_syscalls:sys_enter/id>200/,task-state/-N 20/,page-faults/-N 10/,profile/-F 5000 -N 10/,breakpoint/"0xffffffffc2cc004a/1:rw -g"/' --order
-- cat /sys/module/kvm/parameters/ignore_msrs
task-state events are forwarded to trace, reset instance.
Y
2024-03-01 10:19:49.196993 swapper 0 dN.. [072] 5499337.853422: sched:sched_wakeup: perf-exec:119154 [120] success=1 CPU:072
2024-03-01 10:19:49.196996 swapper 0 d... [072] 5499337.853425: sched:sched_switch: swapper/72:0 [120] R ==> perf-exec:119154 [120]
2024-03-01 10:19:49.197022 cat 119154 [072] 5499337.853451: page-fault: addr 0000000000404100 ip 0000000000404100
2024-03-01 10:19:49.197034 cat 119154 [072] 5499337.853463: page-fault: addr 00007f73abc988e0 ip 00007f73abc988e0
2024-03-01 10:19:49.197039 cat 119154 [072] 5499337.853468: page-fault: addr 00007f73dd1401c40 ip 00007f73abb94a3f
2024-03-01 10:19:49.197044 cat 119154 [072] 5499337.853473: page-fault: addr 00007f73aaa51aa0 ip 00007f73abb87d6d
2024-03-01 10:19:49.197048 cat 119154 [072] 5499337.853477: page-fault: addr 00007f73aaa5de7c ip 00007f73abb87570
2024-03-01 10:19:49.197050 cat 119154 [072] 5499337.853479: page-fault: addr 00007f73aaa610a0 ip 00007f73abb97dc0
2024-03-01 10:19:49.197053 cat 119154 [072] 5499337.853482: page-fault: addr 00007f73abd9a2c0 ip 00007f73abb8ce29
2024-03-01 10:19:49.197061 cat 119154 [072] 5499337.853490: page-fault: addr 00007f73aab3dae0 ip 00007f73aab3dae0
2024-03-01 10:19:49.197068 cat 119154 [072] 5499337.853497: page-fault: addr 00007f73abc99510 ip strncpy_from_user+0x6c

```

```
2024-03-01 10:19:49.197074 cat 119154 [072] 5499337.853503: page-fault: addr 00007f73aaa55194 ip 00007f73abb87570
2024-03-01 10:19:49.197216 cat 119154 [072] 5499337.853645: profile: 505900 cpu-cycles
2024-03-01 10:19:49.197408 cat 119154 .... [072] 5499337.853838: raw_syscalls:sys_enter: NR 202 (7f73ab75e0d0, 81, 7fffffff, 7fdd1402960, 1, 28)
2024-03-01 10:19:49.197417 cat 119154 [072] 5499337.853846: profile: 495500 cpu-cycles
2024-03-01 10:19:49.197469 cat 119154 d... [072] 5499337.853899: sched:sched_switch: perf-exec:119154 [120] R ==> migration/72:509 [0]
2024-03-01 10:19:49.197479 swapper 0 d... [048] 5499337.853909: sched:sched_switch: swapper/48:0 [120] R ==> perf-exec:119154 [120]
2024-03-01 10:19:49.197644 cat 119154 [048] 5499337.854073: profile: 502800 cpu-cycles
2024-03-01 10:19:49.197673 cat 119154 d... [048] 5499337.854102: sched:sched_switch: cat:119154 [120] R ==> hcbs_cli_acce0:58199 [100]
2024-03-01 10:19:49.197683 hcbs_cli_acce0 58199 d... [048] 5499337.854113: sched:sched_switch: hcbs_cli_acce0:58199 [100] S ==> cat:119154 [120]
2024-03-01 10:19:49.197863 cat 119154 [048] 5499337.854292: profile: 497500 cpu-cycles
2024-03-01 10:19:49.198064 cat 119154 [048] 5499337.854493: profile: 499300 cpu-cycles
2024-03-01 10:19:49.198265 cat 119154 [048] 5499337.854694: profile: 500500 cpu-cycles
2024-03-01 10:19:49.198466 cat 119154 [048] 5499337.854895: profile: 499600 cpu-cycles
2024-03-01 10:19:49.198598 cat 119154 d... [048] 5499337.855028: sched:sched_switch: cat:119154 [120] S ==> swapper/48:0 [120]
2024-03-01 10:19:49.198612 sap1003 152612 d... [000] 5499337.855042: sched:sched_wakeup: cat:119154 [120] success=1 CPU:048
2024-03-01 10:19:49.198614 swapper 0 d... [048] 5499337.855044: sched:sched_switch: swapper/48:0 [120] R ==> cat:119154 [120]
2024-03-01 10:19:49.198693 cat 119154 [048] 5499337.855122: profile: 500200 cpu-cycles
2024-03-01 10:19:49.198737 cat 119154 d... [048] 5499337.855166: sched:sched_switch: cat:119154 [120] R ==> hcbs_cli_acce0:58199 [100]
2024-03-01 10:19:49.198748 hcbs_cli_acce0 58199 d... [048] 5499337.855178: sched:sched_switch: hcbs_cli_acce0:58199 [100] S ==> cat:119154 [120]
2024-03-01 10:19:49.198803 cat 119154 .... [048] 5499337.855233: raw_syscalls:sys_enter: NR 221 (3, 0, 0, 2, 0, 0)
2024-03-01 10:19:49.198820 cat 119154 [048] 5499337.855249: breakpoint: 0xffffffffc2cc004a/1:RW
RIP: ffffffff810b3ad7 RSP: ffff882f7c9f3e80 RFLAGS:00000246
RAX: ffffffff8c2cc004a RBX: ffff883491b4b628 RCX: ffff882f7c9f3fd8
RDX: 0000000000000000 RSI: ffffffff81948368 RDI: ffff88042ce7b000
RBP: ffff882f7c9f3e80 R08: ffffffff8193cc98 R09: ffffea0010b39f00
R10: ffffea0010b39f00 R11: 0000000000000246 R12: ffff88042ce7b000
R13: ffff8834e1e7e310 R14: ffff885f41fa3180 R15: ffffffff816f6d90
CS: 0010 SS: 0018
ffffffff810b3ad7 param_get_bool+0x17 ([kernel.kallsyms])
ffffffff810b3f81 param_attr_show+0x41 ([kernel.kallsyms])
ffffffff810b36f0 module_attr_show+0x20 ([kernel.kallsyms])
ffffffff812941e9 sysfs_read_file+0x99 ([kernel.kallsyms])
ffffffff8121390f vfs_read+0x9f ([kernel.kallsyms])
ffffffff812147df sys_read+0x7f ([kernel.kallsyms])
ffffffff816d13fe tracesys+0xe3 ([kernel.kallsyms])
00007fd912a20a30 Unknown
2024-03-01 10:19:49.198835 cat 119154 d... [048] 5499337.855264: sched:sched_wakeup: cat:119154 [120] success=1 CPU:048
2024-03-01 10:19:49.198861 cat 119154 .... [048] 5499337.855291: raw_syscalls:sys_enter: NR 231 (0, 0, 0, ffffffff70, 3c, e7)
2024-03-01 10:19:49.198911 cat 119154 [048] 5499337.855340: profile: 499800 cpu-cycles
2024-03-01 10:19:49.198938 cat 119154 d... [048] 5499337.855367: sched:sched_switch: cat:119154 [120] x ==> swapper/48:0 [120]
```

按时间顺序显示 cat 命令的系统调用、任务状态、page-faults、采样事件、断点。命令执行过程中，所有关键事件都能显示出来。

◆ 应用场景

主要用于验证 profiler 作为事件源时，能否正常工作。也可以配合脚本工作。

5.3.更多的可能性

- ◆ Profiler 事件源，目前仅指定了包含内建事件的 profiler。可以扩展到所有 profiler。
- ◆ 添加更多的 profiler 作为事件源使用。如：硬件 pmu 的很多事件。

6. 其他主题

6.1. 事件丢失

因为 ringbuffer 的大小是固定的，在用户态未及时读取时，会导致 ringbuffer 满而丢失事件。在高负载的环境里，事件丢失是常态。事件丢失后，意味这各个 profiler 的输出结果是不正确的。

事件丢失的 2 种应对方式：

- 1. 使用-m 选项调大 ringbuffer。直到不存在事件丢失，保证结果的正确性。其弊端是会占用较大内存。
- 2. 容忍事件丢失，只保证未丢失部分输出结果是正确的。对比未丢失时，其输出结果不是完整的，但是正确的。Ringbuffer 越小，事件丢失的越多，结果越不完整。反之，ringbuffer 越大，事件丢失的越少，结果越完整。

实际使用时，根据情况选择。可用内存比较多，调大 ringbuffer，保证结果完整。

```
[root@kvm ~]# perf-prof rundelay -e 'sched:sched_wakeup,sched:sched_wakeup_new,sched:sched_switch//key=prev_pid/' -e 'sched:sched_switch//key=next_pid/' -k pid --order -i 1000 -m 32 --than 16ms
Trick: Enable userland unnecessary detection of sched:sched_wakeup events.
Trick: Enable userland unnecessary detection of sched:sched_wakeup events.
2024-03-03 17:07:39.025165 rundelay: Lost 1804 events on CPU #24 (5696605.869896, 5696605.898613)
2024-03-03 17:07:39.026912 rundelay: Lost 1007 events on CPU #0 (5696605.874345, 5696605.899035)
2024-03-03 17:07:37.247235          sh 33419 d... [000] 5696606.098766: sched:sched_wakeup_new: sh:33420 [120] success=1 CPU:000
2024-03-03 17:07:37.277179    net_account_age 61311 d... [000] 5696606.128711: sched:sched_switch: net_account_age:61311 [120] S ==> sh:33420 [120]
2024-03-03 17:07:37.252374    exec_agent 218828 dN.. [024] 5696606.103905: sched:sched_wakeup_new: exec_agent:33421 [120] success=1 CPU:000
2024-03-03 17:07:37.282702    hcbs_cli_acce0 58199 d... [048] 5696606.134233: sched:sched_switch: hcbs_cli_acce0:58199 [100] S ==> exec_agent:33421 [120]
2024-03-03 17:07:39.103190
-----
start => end          calls      total(us)    min(us)      p50(us)      p95(us)      p99(us)      max(us)      than(reqs)
-----
sched_wakeup => sched_switch 46119      791953.658      0.382        2.170        38.587        275.570      12137.552      0 ( 0%)
sched_switch => sched_switch 4510       137699.259      1.730        5.748        33.537        295.021      14008.336      0 ( 0%)
sched_wakeup_new => sched_switch 91         101656.906      1.049        3.368        7634.338      30328.285      30328.285      2 ( 2%)
```

调小了 ringbuffer，其输出的结果是正确的。输出的 2 组超过 16ms 的事件是正确的。输出的 calls、total、min、p50、p95、p99、max 结果也是正确的，但结果不完整，实际的 sched_wakeup => sched_switch 应该有 5 万多次。

6.2. 虚拟化场景

6.2.1. 时间戳对齐

在虚拟化场景，经常需要同时跟踪，host 发生的事件和 guest 发生的事件。但由于 kvm 模块存在 tsc-offset、tsc-scaling，host 事件和 guest 事件的时间戳是不对应的。会对分析造成一定困扰。如：

- Host 注入中断，到 Guest 收到中断。时间戳不一致，无法分析中断注入是否存在延迟。
- Host 上 vcpu 线程执行一次，Guest 内执行了哪些进程。监控 Guest 和 Host 的 sched:sched_switch 事件，因为事件戳不一致，无法分析 vcpu 线程切换一次，Guest 内切换过哪些线程。

所以，时间戳对齐，就是把 host 事件的时间戳转换为 guest 时间戳。时间戳在同一尺度上，就可以跟 guest 事件做比较。

详细参考：[perf clock to guest clock](#)。

6.2.2. 事件传播

时间戳对齐后，可以把 Guest 的事件传递到 Host，参与延迟分析。Host 和 Guest 属于 2 个系统，其 pid->comm 映射不一致，其堆栈符号解析也

不一致。因此，目前参与延迟分析，只能做最简单的分析。

- 事件传播途径。Guest 和 Host 基于 virtio-ports 传递事件。
- 延迟分析。利用 multi-trace 对 Guest 和 Host 的事件进行延迟分析。

详细参考：[virtio-ports](#)

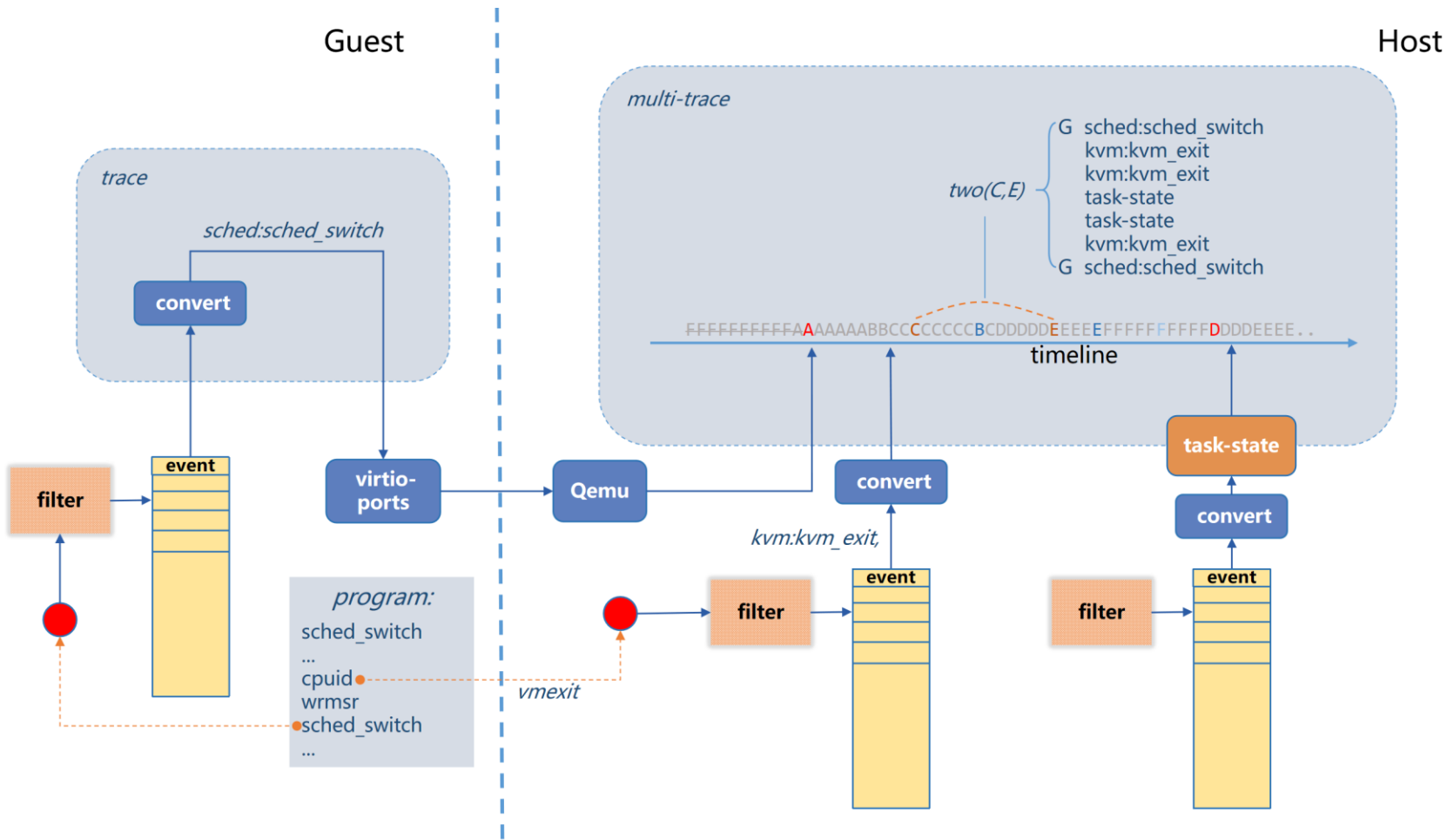
主要用途：

- 中断延迟。从 dpdk 注入中断到 Guest 内开始处理中断，这之间的延迟。
- 调度分析。Host vcpu 调度延迟，Guest 内哪些进程受到影响。
- IO 分析。Guest 发起 IO 到 Host 接收到 IO，这之间的延迟。

6.3.Soft lockup 分析

Soft lockup 是有进程占用 cpu 太长时间不发生调度导致的。一般的可以把 watchdog_thresh 调大一些，确保不发生 softlockup 时，再跟踪进程执行时间。

整体上分为 Guest 跟踪和 Host 跟踪。



6.3.1. Guest

Guest 需要跟踪 `sched:sched_switch` 事件，并转换为 `kvmclock` 时钟，再通过 `virtio-ports` 通道传递到 Host。

转换为 kvmclock 时钟。参考 6.2.1。

```
[root@VM ~]# echo 'p:try_to_wake_up try_to_wake_up kvm_sched_clock_offset=@kvm_sched_clock_offset __sched_clock_offset=@__sched_clock_offset' > /sys/kernel/debug/tracing/kprobe_events
[root@VM ~]# perf-prof trace -e kprobes:try_to_wake_up -N 1
2024-05-07 09:56:55.084528 swapper 0 d.s. [064] 48.326549: kprobes:try_to_wake_up: (ffffffff810ba2a0) kvm_sched_clock_offset=0x1ad65fcea
__sched_clock_offset=0xfffffffff2f91b58
[root@VM ~]# printf "0x%x\n" $((0x1ad65fcea-0xfffffffff2f91b58))
0x1ba6ce192
[root@VM ~]# perf-prof trace -e sched:sched_switch --clock-offset 0x1ba6ce192 -N 2
2024-05-07 09:57:31.034405 perf-prof 3469 d... [060] 91.699180: sched:sched_switch: perf-prof:3469 [120] S ==> swapper/60:0 [120]
2024-05-07 09:57:31.034409 swapper 0 d... [060] 91.699184: sched:sched_switch: swapper/60:0 [120] R ==> perf-prof:3469 [120]
```

通过 virtio-ports 通道传递。参考 6.2.2。

```
[root@VM ~]# perf-prof trace -e 'sched:sched_switch//push="/dev/virtio-ports/org.qemu.perf0/' --clock-offset 0x1ba6ce192 -i 1000
Open cdev /dev/virtio-ports/org.qemu.perf0
Cdev /dev/virtio-ports/org.qemu.perf0 has hang up
Cdev /dev/virtio-ports/org.qemu.perf0 is connected
```

采样 sched:sched_switch 事件，先转换为 kvmclock，再 push 到 org.qemu.perf0 通道。-i 1000 每 1000ms 刷新一次 ringbuffer，避免事件堆积。

获取 watchdog_thresh 时间。

```
[root@VM ~]# cat /proc/sys/kernel/watchdog_thresh
10
```

6.3.2. Host

Host 需要从 virtio-ports 通道读取 sched:sched_switch 事件，并利用 multi-trace 分析进程执行时间。参考 4.2.1.3.3。

首先，做最简单的测试，判断 virtio-ports 通道是否正常。

```
[root@kvm ~]# perf-prof multi-trace -e 'sched:sched_switch//pull=9900/vm=a131f078-68c7-4fd1-a784-d8cfea2b5b25/role="(next_pid?1:0)|(prev_pid?2:0)"/' --cycle --kvmclock a131f078-68c7-4fd1-a784-d8cfea2b5b25 -i 1000
2024-05-07 10:05:33.649231 multi-trace: wait pvclock update
Connected to 127.0.0.1:9900
2024-05-07 10:05:58.542022 multi-trace: pvclock updated.
2024-05-07 10:05:59.542421
start => end      calls      total(us)  min(us)    p50(us)    p95(us)    p99(us)    max(us)
-----
sched_switch => sched_switch      773      13820.070      1.809      4.887      54.291      200.404      1406.171
2024-05-07 10:06:00.542424
start => end      calls      total(us)  min(us)    p50(us)    p95(us)    p99(us)    max(us)
-----
sched_switch => sched_switch      571      14523.355      1.966      4.778      59.389      402.694      3223.419
```

virtio-ports 通道连接到 qemu 的 9900 端口。pull=9900 属性就是从 Guest 拉取事件。vm=a131f078-68c7-4fd1-a784-d8cfea2b5b25 属性把 Guest 事件做进一步转换。role="(next_pid?1:0)|(prev_pid?2:0)"属性用于调节 Guest 事件的角色，不统计 idle 进程的执行时间。--kvmclock a131f078-68c7-4fd1-a784-d8cfea2b5b25 选项用于把 Host 事件转换为 Guest kvmclock 时钟。

进程的执行时间超过 watchdog_thresh 就会发生 soft lockup。Guest 内 watchdog_thresh 为 10 秒。

6.3.2.1. 添加更多的 Host 事件

```
[root@kvm ~]# perf-prof multi-trace -e 'sched:sched_switch//pull=9900/vm=a131f078-68c7-4fd1-a784-d8cfea2b5b25/role="(next_pid?1:0)|(prev_pid?2:0)"/,kvm:kvm_exit//untraced/,kvm:kvm_entry//untraced/,task-state//untraced/' --cycle --kvmclock a131f078-68c7-4fd1-a784-d8cfea2b5b25 -i 1000 -p 192486 --order --than 100ms '--detail=sametid,hide<1ms'
2024-05-07 15:00:04.629622 task-state: wait pvclock update
task-state events are forwarded to multi-trace, reset instance.
2024-05-07 15:00:04.655108 multi-trace: wait pvclock update
Connected to 127.0.0.1:9900
2024-05-07 15:02:00.845489 multi-trace: pvclock updated.
2024-05-07 15:02:00.850787 task-state: pvclock updated.
2024-05-07 15:02:02.856581
start => end      calls      total(us)  min(us)    p50(us)    p95(us)    p99(us)    max(us)    than(reqs)
-----
sched_switch => sched_switch      798      48384.333      1.993      6.274      119.284      848.657      18452.685      0 ( 0%)
2024-05-07 15:02:03.861440
start => end      calls      total(us)  min(us)    p50(us)    p95(us)    p99(us)    max(us)    than(reqs)
-----
```

```

sched_switch => sched_switch      1148      34504.762      2.072      6.001      77.816      369.491      3128.725      0 ( 0%)
...
2024-05-07 15:03:18.787152 G      CPU 51/KVM 193283 d... [077] 18439.494546: sched:sched_switch: swapper/51:0 [120] R ==> bash:114768 [120]
| 5420355.304 us      CPU 51/KVM 193283 .... [077] 18439.494697: kvm:kvm_exit: reason EXTERNAL_INTERRUPT rip 0xffffffff813ee57b info 0 800000ef
|      27746 hidden
|      CPU 51/KVM 193283 d... [077] 18444.904710: kvm:kvm_entry: vcpu 73
|      CPU 51/KVM 193283 .... [077] 18444.914496: kvm:kvm_exit: reason MSR_WRITE rip 0xffffffff81059608 info 0 0
|      91 hidden
2024-05-07 15:03:24.207507 G      CPU 51/KVM 193283 d... [077] 18444.914901: sched:sched_switch: insmod:114768 [120] W ==> migration/51:267 [0]
2024-05-07 15:03:25.931261
start => end      calls      total(us)      min(us)      p50(us)      p95(us)      p99(us)      max(us)      than(reqs)
-----
sched_switch => sched_switch      1233      5508945.518      2.048      10.181      134.430      971.445      5420355.304      1 ( 0%)

```

添加 `kvm:kvm_exit`, `kvm:kvm_entry`, `task-state` 事件, Guest soft lockup 可能是因为 Guest 的进程在执行时, 退出到 Host 上, 并在 Host 上发生 `vcpu` 线程的阻塞导致。添加 `kvm`:事件, 可以观察 `vmexit` 退出的情况。添加 `task-state` 事件, 可以观察 `vcpu` 线程状态, 判断是否有 `vcpu` 线程阻塞。--detail=sametid,hide<1ms, 过滤相同 tid 的事件, 并隐藏事件间隔小于 1ms 的事件。

在 Guest 内 `insmod` 插入一个耗时 5s 的内核模块, 5s 内不会发生任何调度。在 Host 上观察 multi-trace 统计结果。

可以看到, 跟踪到一次 5.4s 的延迟, **G** 标识两个事件都来自 Guest, 延时是 `insmod:114768` 进程造成的。中间的 27746 个事件间隔<1ms, 被隐藏, 说明在 Host 上并无阻塞点, 没有事件间隔大于 1ms 的情况。问题来自 Guest。

6.3.2.2. Host 阻塞场景

```

[root@kvm ~]# perf-prof multi-trace -e 'sched:sched_switch//pull=9900/vm=a131f078-68c7-4fd1-a784-d8cfea2b5b25/role="(next_pid?1:0)|(prev_pid?2:0)"/,kvm:kvm_exit//untraced/,kvm:kvm_entry//untraced/,task-state/-g/untraced/' --cycle --kvmclock a131f078-68c7-4fd1-a784-d8cfea2b5b25 -i 1000 -p 225290 --order --than 100ms '--detail=sametid,hide<1ms'
2024-05-07 16:19:02.528799 task-state: wait pvclock update
task-state events are forwarded to multi-trace, reset instance.
2024-05-07 16:19:02.591962 multi-trace: wait pvclock update
Connected to 127.0.0.1:9900
2024-05-07 16:20:30.221832 multi-trace: pvclock updated.
2024-05-07 16:20:30.226635 task-state: pvclock updated.
2024-05-07 16:20:32.233987
start => end      calls      total(us)      min(us)      p50(us)      p95(us)      p99(us)      max(us)      than(reqs)
-----
sched_switch => sched_switch      614      5879.164      1.708      4.646      33.996      65.440      368.319      0 ( 0%)
...
2024-05-07 16:31:39.395683 G      CPU 47/KVM 226562 d... [027] 968.8026121: sched:sched_switch: swapper/47:0 [120] R ==> bash:8937 [120]
| 640555.291 us      CPU 47/KVM 226562 .... [027] 968.802951: kvm:kvm_exit: reason CUID rip 0x7f64bc1ae80f info 0 0
|      88 hidden
|      CPU 47/KVM 226562 d... [027] 968.8049522: sched:sched_switch: CPU 47/KVM:226562 [120] S ==> swapper/27:0 [120]
|      ffffffff816c51a3 __schedule+0x4f3 ([kernel.kallsyms])
|      ffffffff816c5609 schedule+0x29 ([kernel.kallsyms])
|      ffffffff81103c46 futex_wait_queue_me+0xc6 ([kernel.kallsyms])
|      ffffffff8110492b futex_wait+0x17b ([kernel.kallsyms])
|      ffffffff81106666 do_futex+0x106 ([kernel.kallsyms])
|      ffffffff81106b80 sys_futex+0x80 ([kernel.kallsyms])
|      ffffffff816d13fe tracesys+0xe3 ([kernel.kallsyms])
|      00007f77bc17342d __l1l_lock_wait+0x1d (/usr/lib64/libpthread-2.17.so)
|      0000000004ad4b2 qemu_mutex_lock_iothread+0x42 (/usr/local/bin/qemu-system-x86_64)
|      00000000047f675 prepare_mmio_access.isra.22+0x95 (/usr/local/bin/qemu-system-x86_64)
|      0000000004844c4 flatview_read_continue+0xd4 (/usr/local/bin/qemu-system-x86_64)
|      0000000004846a0 flatview_read+0x70 (/usr/local/bin/qemu-system-x86_64)
|      00000000048480a address_space_read_full+0x5a (/usr/local/bin/qemu-system-x86_64)
|      0000000004c2212 kvm_cpu_exec+0x3c2 (/usr/local/bin/qemu-system-x86_64)
|      0000000004adc2 qemu_kvm_cpu_thread_fn+0x3c2 (/usr/local/bin/qemu-system-x86_64)
|      00007f77bc16ce25 start_thread+0xc5 (/usr/lib64/libpthread-2.17.so)
|      qemu-system-x86 225290 d... [029] 968.8512213: sched:sched_wakeup: CPU 47/KVM:226562 [120] success=1 CPU:027
|      ffffffff810c9295 ttwu_do_wakeup+0xb5 ([kernel.kallsyms])
|      ffffffff810c939d ttwu_do_activate.constprop.115+0x5d ([kernel.kallsyms])
|      ffffffff810cc21d try_to_wake_up+0x18d ([kernel.kallsyms])
|      ffffffff810cc4ab wake_up_q+0x5b ([kernel.kallsyms])
|      ffffffff81103b6b futex_wake+0x16b ([kernel.kallsyms])
|      ffffffff8110668a do_futex+0x12a ([kernel.kallsyms])
|      ffffffff81106b80 sys_futex+0x80 ([kernel.kallsyms])
|      ffffffff816d13fe tracesys+0xe3 ([kernel.kallsyms])
|      00007f77bc1734ca __l1l_unlock_wake+0x1a (/usr/lib64/libpthread-2.17.so)
|      0000000007569b7 os_host_main_loop_wait+0x77 (/usr/local/bin/qemu-system-x86_64)
|      000000000756c75 main_loop_wait+0x85 (/usr/local/bin/qemu-system-x86_64)
|      0000000005d12e3 main_loop+0x33 (/usr/local/bin/qemu-system-x86_64)
|      00000000047b1df main+0x12ff (/usr/local/bin/qemu-system-x86_64)
|      00007f77b7d56c05 __libc_start_main+0xf5 (/usr/lib64/libc-2.17.so)
|      2993 hidden4
|      CPU 47/KVM 226562 d... [027] 969.394448: kvm:kvm_entry: vcpu 69
|      CPU 47/KVM 226562 .... [027] 969.404559: kvm:kvm_exit: reason MSR_WRITE rip 0xffffffff8105ffd9 info 0 0
|      4 hidden
|      CPU 47/KVM 226562 d... [027] 969.4046065: sched:sched_switch: CPU 47/KVM:226562 [120] S ==> swapper/27:0 [120]
|      ffffffff816c51a3 __schedule+0x4f3 ([kernel.kallsyms])
|      ffffffff816c5609 schedule+0x29 ([kernel.kallsyms])

```

```
ffffff81103c46 futex_wait_queue_me+0xc6 ([kernel.kallsyms])
ffffff8110492b futex_wait+0x17b ([kernel.kallsyms])
ffffff81106666 do_futex+0x106 ([kernel.kallsyms])
ffffff81106b80 sys_futex+0x80 ([kernel.kallsyms])
ffffff816d13fe tracesys+0xe3 ([kernel.kallsyms])
00007f77bc17342d __lll_lock_wait+0x1d (/usr/lib64/libpthread-2.17.so)
00000000004ad4b2 qemu_mutex_lock_iothread+0x42 (/usr/local/bin/qemu-system-x86_64)
000000000047f675 prepare_mmio_access.isra.22+0x95 (/usr/local/bin/qemu-system-x86_64)
00000000004844c4 flatview_read_continue+0xd4 (/usr/local/bin/qemu-system-x86_64)
00000000004846a0 flatview_read+0x70 (/usr/local/bin/qemu-system-x86_64)
000000000048480a address_space_read_full+0x5a (/usr/local/bin/qemu-system-x86_64)
00000000004c2212 kvm_cpu_exec+0x3c2 (/usr/local/bin/qemu-system-x86_64)
00000000004addc2 qemu_kvm_cpu_thread_fn+0x3c2 (/usr/local/bin/qemu-system-x86_64)
00007f77bc16ce25 start_thread+0xc5 (/usr/lib64/libpthread-2.17.so)
|      softlockup_test 23504 d... [027] 969.4427046: sched:sched_wakeup: CPU 47/KVM:226562 [120] success=1 CPU:027
ffffff810c9295 ttwu_do_wakeup+0xb5 ([kernel.kallsyms])
ffffff810c939d ttwu_do_activate.constprop.115+0x5d ([kernel.kallsyms])
ffffff810cc21d try_to_wake_up+0x18d ([kernel.kallsyms])
ffffff810cc4ab wake_up_q+0x5b ([kernel.kallsyms])
ffffff81103b6b futex_wake+0x16b ([kernel.kallsyms])
ffffff8110668a do_futex+0x12a ([kernel.kallsyms])
ffffff81106b80 sys_futex+0x80 ([kernel.kallsyms])
ffffff816d13fe tracesys+0xe3 ([kernel.kallsyms])
00007f77bc1734ca __lll_unlock_wake+0x1a (/usr/lib64/libpthread-2.17.so)
00000000006e769a softlockup_test_thread+0x1aa (/usr/local/bin/qemu-system-x86_64)
00007f77bc16ce25 start_thread+0xc5 (/usr/lib64/libpthread-2.17.so)
|      98 hidden
2024-05-07 16:31:40.036238 G      CPU 47/KVM 226562 d... [027] 969.4431677: sched:sched_switch: insmod:8937 [120] W ==> migration/47:247 [0]
2024-05-07 16:31:41.336637
start => end      calls      total(us)  min(us)  p50(us)  p95(us)  p99(us)  max(us)  than(reqs)
-----
sched_switch => sched_switch  1169      671255.767    1.963    12.255    98.732    256.468    640555.291    1 ( 0%)
```

在 Guest 内 insmod 插入一个耗时 500ms 的内核模块。在 Host 上观察 multi-trace 统计结果。Task-state 事件打开堆栈。

有一次 640ms 的延迟，按时间顺序看：

- 1 Guest 事件，切换到 insmod:8937 进程。
- 2 Host 事件，vcpu 线程睡眠，在 qemu_mutex_lock_iothread()函数内等待 futex 锁。
- 3 Host 事件，vcpu 线程等到 futex 锁，被唤醒。等锁时间 46ms。在 os_host_main_loop_wait()内释放锁。
- 4 隐藏 2993 个事件，这些事件间隔都<1ms。
- 5 Host 事件，vcpu 线程再次等待 futes 锁。
- 6 Host 事件，vcpu 线程等待 futex 锁。等锁时间 38ms。
- 7 Guest 事件，insmod:8937 进程切出。insmod 单次执行 640ms。

可以观察到 Guest insmod 命令执行过程中，Host 上有 2 次等待，共等待 84ms。Host 阻塞次数多，阻塞时间长，就可能会导致 Guest 进程执行时间很长，会发生 Soft lockup。

6.3.3. 过滤

如果只需要关注某一个业务进程受到 Host 的影响，则需要设置过滤器。

Guest

```
[root@VM ~]# perf-prof trace -e 'sched:sched_switch/prev_comm~"stress*")/next_comm~"stress*"/push="/dev/virtio-ports/org.qemu.perf0"/' --clock-offset 0x1ba6ce192 -i 1000
Open cdev /dev/virtio-ports/org.qemu.perf0
Cdev /dev/virtio-ports/org.qemu.perf0 has hang up
```

过滤 stress*进程，只考虑 stress*进程受到的影响。

```
[root@kvm ~]# perf-prof multi-trace -e 'sched:sched_switch//pull=9900/vm=a131f078-68c7-4fd1-a784-d8cfea2b5b25/role="(strncmp(next_comm,"stress",6)==0?1:0)/(strncmp(prev_comm,"stress",6)==0?2:0)"/,kvm:kvm_exit//untraced/,kvm:kvm_entry//untraced/,task-state/-g/untraced/' --cycle --kvmclock a131f078-68c7-4fd1-a784-d8cfea2b5b25 -i 1000 -p 225290 --order --than 1000ms '--detail=sametid,hide<1ms'
2024-05-08 15:30:57.615873 task-state: wait pvclock update
task-state events are forwarded to multi-trace, reset instance.
2024-05-08 15:30:57.679168 multi-trace: wait pvclock update
Connected to 127.0.0.1:9900
2024-05-08 15:35:22.767083 multi-trace: pvclock updated.
2024-05-08 15:35:22.772195 task-state: pvclock updated.
2024-05-08 15:35:24.785799
-----
start => end          calls      total(us)  min(us)  p50(us)  p95(us)  p99(us)  max(us)  than(reqs)
-----
sched_switch => sched_switch      91      1976690.921      1.281      6.928  178122.228  552997.160  552997.160      0 ( 0%)
```

使用 role 属性只测量 stress*进程的执行时间。并分析在 stress*进程执行过程中，Host 上 vcpu 的运行状态，vmexit 的情况。

Guest 的 sched:sched_switch 事件是经过过滤的，必须要配合 role 属性，才能只测量 stress*进程的执行时间，避免测量到其他进程。

6.3.4. 整体逻辑

Guest softlockup 可能和 Host 有关系，需要把 Guest 事件传递到 Host 进行联合分析。

Guest 跟踪 sched:sched_switch 事件，可以过滤指定业务进程的事件，再转换为 kvmclock 时钟，写入到 Host。

Host 接收 Guest 事件，跟踪 Guest 进程的执行时间。并追加 kvm:kvm_exit, kvm:kvm_entry, task-state 等作为中间事件，标记为 untraced，中间事件也都转换为 kvmclock 时钟。同 Guest 事件一起加入 timeline，并分析 Guest 进程执行时间长的原因，判断是否受到 Host 的影响。