# 重要的东西放前面

## perf性能监控: perf Examples

## ftrace脚本实现系统监控: perf-tools

==========以下可以略过的分割线==========

# Perf_events(Perf命令及其扩展)

## 概述

Perf Event是面向事件的观察工具。简要来说，Perf_Events可以用于解决下面的问题：

- 为什么内核占用这么多CPU时间？具体是哪一个代码段耗时？
- 哪部分代码导致CPU会导致L2的缓存失效？
- 是否CPU被memory I/O所害？
- 哪段代码在疯狂分配内存？
- 到底谁导致了TCP的重传？
- 是否内核中的某一个方法被调用了，有多频繁？

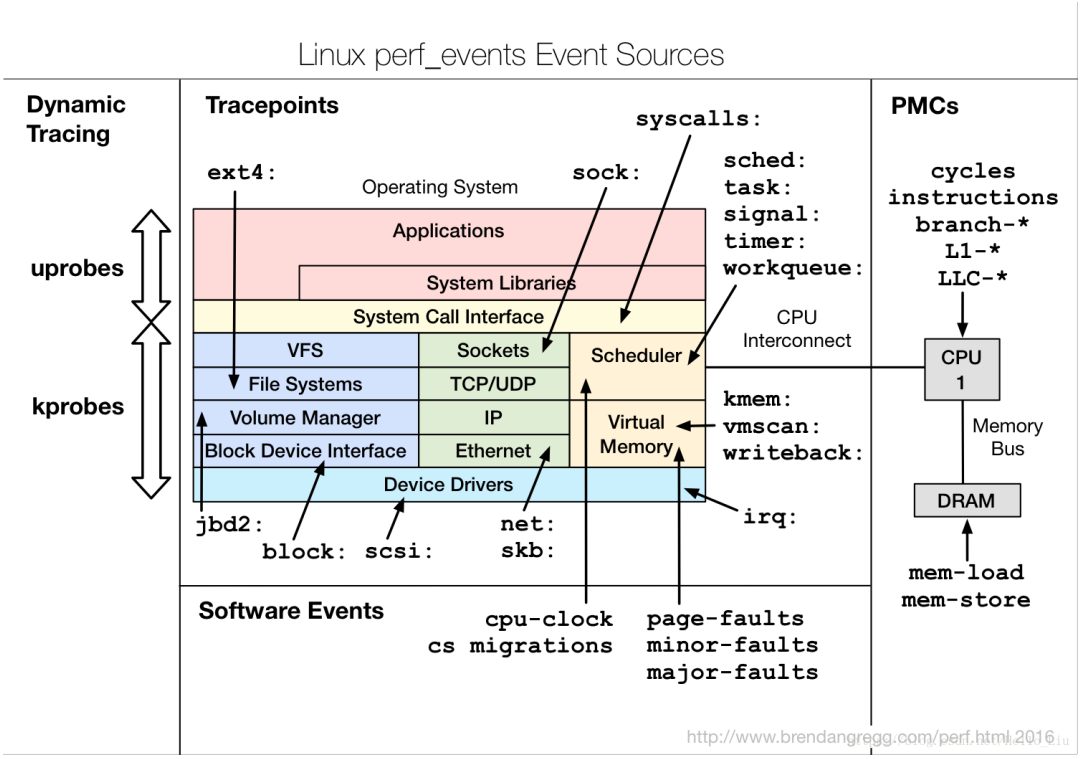
一共有两种Perf_Events：第一种是直接利用现有提供的Events，其覆盖了大部分可能会用到的事件。而如果刚好里面没有想要的，则需要利用第二种方式，自己写新的Perf_Events。

而针对如何如何利用Perf_Events进行测量，一共有三种方法：

1. 直接利用Counting Event计数，可以利用perf工具直接对发生的次数进行计数。这种方法不会生成perf.data文件，直接利用perf stat命令即可。
2. 在指定的时间进行取样，使用这种方法会将Perf_Events数据写到内核缓存里面，然后再由Perf隔一段时间写入perf.data文件中。最后利用perf report或者perf script读取。但是利用这种方法进行采样，report文件的大小\overhead比较高（文件很大）。
3. 最后一种方法是利用BPF触发用户自己写的程序，这种方法最灵活。但是同时这种方法也比较复杂，需要自己写触发程序，在后面的eBPF章节中进

行描述。

# 种类与使用方法




如上图所示，Linux Perf_Events可以分为以下几类：

- **硬件事件 Hardware Events: CPU 性能检测计数器**

什么是Hardware Events

硬件事件是利用处理器的performance Monitoring Unit(PMU)实现。读取其中的Performace Monitoring Counters(PMCs)或者称为Performance instrumentation counters(PICs)。这些Counter可以跟踪一些底层的动作，如CPU cycles，instructions retired, memory stall cycles, level 2 cache misses等等。

这些硬件事件的特点是只有其中少数几个事件可以同时被记录。 这时因为硬件资源有限，需要手动指定它们记录哪些event。

如何使用Hardware Events

使用硬件Raw Counter的格式是rUUEE，其中UU是umask，EE是event number。而大部分好用的直接加到了perf list中，直接用对应的事件就行。

如果需要做stack tracing，避免记录的overhead太大，可以指定没n次做一个trace，直接设定-c n即可。如：perf record -e L1-dcache-load-misses -c 10000 -ag -- sleep 5

- **软件事件 Software Events: 利用kernel counters低层次的events，例如CPU迁移, minor faults, major faults.**

- **内核追踪事件 Kernel Tracepoint Events: 有些内核态的tracepoint被硬件编码到内核中的一些地方。**

内核的Tracepoints是在内核代码中有意思的地方或者逻辑上分割的地方编写插入，因此用到这些的高层的事件能够被追踪。比如 system calls、TCP events、file system I/O和disk I/O等等。它们被划分很多组，如sock:代表着socket events，而sched:代表着CPU scheduler events。

**include/trace/events/block.h:**

```
TRACE_EVENT(block_rq_complete,
[...]
        TP_printk("%d,%d %s (%s) %llu + %u [%d]",
                  MAJOR(__entry->dev), MINOR(__entry->dev),
                   __entry->rwbs, __get_str(cmd),
                   (unsigned long long)__entry->sector,
                   __entry->nr_sector, __entry->errors)
```

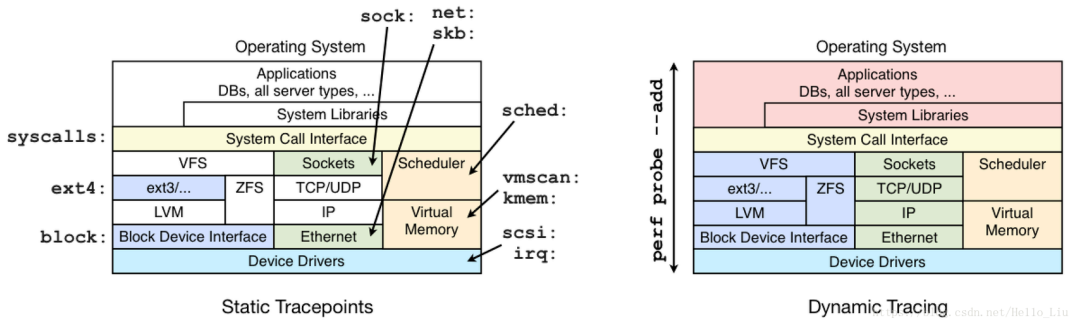
查看输出格式

```
# sudo cat
/sys/kernel/debug/tracing/events/block/block_rq_complete/format
name: block_rq_complete
ID: 942
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:dev_t dev;    offset:8;   size:4; signed:0;
    field:sector_t sector;  offset:16;  size:8; signed:0;
    field:unsigned int nr_sector;   offset:24;  size:4; signed:0;
```

```
    field:int errors;   offset:28;  size:4; signed:1;
    field:char rwbs[8]; offset:32;  size:8; signed:1;
    field:__data_loc char[] cmd;    offset:40;  size:4; signed:1;

print fmt: "%d,%d %s (%s) %llu + %u [%d]", ((unsigned int) ((REC->dev) >> 20)), ((unsigned int) ((REC->dev) & ((1U << 20) - 1))), REC->rwbs, __get_str(cmd), (unsigned long long)REC->sector, REC->nr_sector, REC->errors
```

- **用户态静态追踪事件 User Statically-Defined Tracing (USDT): 用户态程序中的静态tracepoint。**

- **[动态追踪 Dynamic Tracing]: 利用kprobe和uprobe在任意位置创建event。**




For kernel analysis, I'm using CONFIG_KPROBES=y and CONFIG_KPROBE_EVENTS=y, to enable kernel dynamic tracing, and CONFIG_FRAME_POINTER=y, for frame pointer-based kernel stacks. For user-level analysis, CONFIG_UPROBES=y and CONFIG_UPROBE_EVENTS=y, for user-level dynamic tracing.

```
# $ perf probe --add tcp_sendmsg
Added new event:
probe:tcp_sendmsg    (on tcp_sendmsg)
You can now use it in all perf tools, such as:
perf record -e probe:tcp_sendmsg -aR sleep 1
```

- **Timed Profiling: 可以间隔一段时间时间进行快照。**

目前perf支持的事件可以用perf list列举出来，包括上面提到的

Hardware/Software/Kernel Tracepoint event。

```
# sudo perf list 'block:*'
  block:block_touch_buffer                           [Tracepoint
event]
  block:block_dirty_buffer                           [Tracepoint
event]
  block:block_rq_abort                               [Tracepoint
event]
  block:block_rq_requeue                             [Tracepoint
event]
  block:block_rq_complete                            [Tracepoint
event]
  block:block_rq_insert                              [Tracepoint
event]
  block:block_rq_issue                               [Tracepoint
event]
  block:block_bio_bounce                             [Tracepoint
event]
  block:block_bio_complete                           [Tracepoint
event]
  block:block_bio_backmerge                          [Tracepoint
event]
[...]
# sudo perf record -e block:block_rq_complete -a sleep 10
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.428 MB perf.data (~18687 samples)
]
# sudo perf script
        run 30339 [000] 2083345.722767: block:block_rq_complete:
202,1 W () 12984648 + 8 [0]
        run 30339 [000] 2083345.722857: block:block_rq_complete:
202,1 W () 12986336 + 8 [0]
        run 30339 [000] 2083345.723180: block:block_rq_complete:
202,1 W () 12986528 + 8 [0]
    swapper     0 [000] 2083345.723489: block:block_rq_complete:
202,1 W () 12986496 + 8 [0]
    swapper     0 [000] 2083346.745840: block:block_rq_complete:
202,1 WS () 1052984 + 144 [0]
  supervise 30342 [000] 2083346.746571: block:block_rq_complete:
```

```
202,1 WS () 1053128 + 8 [0]
  supervise 30342 [000] 2083346.746663: block:block_rq_complete:
202,1 W () 12986608 + 8 [0]
        run 30342 [000] 2083346.747003: block:block_rq_complete:
202,1 W () 12986832 + 8 [0]
[...]
```

而如果使用动态追踪，可以追踪其他的事件。

# ftrace(文件系统接口)

简而言之，ftrace通过文件系统接口提供给用户上述perf功能。

需要将系统的 debugfs 或者 tracefs 给挂载到某个地方，通常被挂载到 /sys/kernel/debug 上面（debug 目录下面有一个 tracing 目录），而比较新的内核，则是将 tracefs 挂载到 /sys/kernel/tracing

- **README**

文件提供了一个简短的使用说明，展示了 ftrace 的操作命令序列。可以通过 cat 命令查看该文件以了解概要的操作流程。

- **available_tracers**

我们可以通过 `available_tracers` 这个文件知道当前 ftrace 支持哪些插件。

```
cat available_tracers
hwlat blk mmiotrace function_graph wakeup_dl wakeup_rt wakeup
function nop
```

通常用的最多的就是 `function` 和 `function_graph`，当然，如果我们不想 trace 了，可以使用 `nop`。我们首先打开 `function`：

```
echo function > current_tracer
cat current_tracer
function
```

- **current_tracer**

用于设置或显示当前使用的跟踪器；使用 echo 将跟踪器名字写入该文件可以切换到不同的跟踪器。系统启动后，其缺省值为 nop ，即不做任何跟踪操作。在执行完一段跟踪任务后，可以通过向该文件写入 nop 来重置跟踪器。

- **available_filter_functions**

记录了当前可以跟踪的内核函数。对于不在该文件中列出的函数，无法跟踪其活动。**这里的追踪函数不属于events，应该是内核给某些函数添加的入口监控。**

- **function_profile_enabled**

用来配置是否统计每个函数被执行的时间已经被执行的次数。

- **trace**

文件提供了查看获取到的跟踪信息的接口。可以通过 cat 等命令查看该文件以查看跟踪到的内核活动记录，也可以将其内容保存为记录文件以备后续查看。

- **tracing_enabled**

用于控制 current_tracer 中的跟踪器是否可以跟踪内核函数的调用情况。写入 0 会关闭跟踪活动，写入 1 则激活跟踪功能；其缺省值为 1 。

- **set_graph_function**

设置要清晰显示调用关系的函数，显示的信息结构类似于 C 语言代码，这样在分析内核运作流程时会更加直观一些。在使用 function_graph 跟踪器时使用；缺省为对所有函数都生成调用关系序列，可以通过写该文件来指定需要特别关注的函数。

- **buffer_size_kb**

用于设置单个 CPU 所使用的跟踪缓存的大小。跟踪器会将跟踪到的信息写入缓存，每个 CPU 的跟踪缓存是一样大的。跟踪缓存实现为环形缓冲区的形式，如果跟踪到的信息太多，则旧的信息会被新的跟踪信息覆盖掉。注意，要更改该文件的值需要先将 current_tracer 设置为 nop 才可以。

- **tracing_on**

用于控制跟踪的暂停。有时候在观察到某些事件时想暂时关闭跟踪，可以将 0 写入该文件以停止跟踪，这样跟踪缓冲区中比较新的部分是与所关注的事件相关的；写入 1 可以继续跟踪。

- **events**

**即是Perf events的内核追踪事件 Kernel Tracepoint Events，通过对各个接口内对应的文件进行操作，达到监控的目的。**

- **set_ftrace_filter/set_ftrace_notrace**

在编译内核时配置了动态 ftrace （选中 CONFIG_DYNAMIC_FTRACE 选项）后使用。前者用于显示指定要跟踪的函数，后者则作用相反，用于指定不跟踪的函数。如果一个函数名同时出现在这两个文件中，则这个函数的执行状况不会被跟踪。这些文件还支持简单形式的含有通配符的表达式，这样可以用一个表达式一次指定多个目标函数；具体使用在后续文章中会有描述。注意，要写入这两个文件的函数名必须可以在文件 available_filter_functions 中看到。缺省为可以跟踪所有内核函数，文件 set_ftrace_notrace 的值则为空。

- **CONFIG_KPROBE_EVENT=y**

- **CONFIG_DYNAMIC_FTRACE=y**

| Parameter | Definition |
| --- | --- |
| p[:[GRP/]EVENT] SYMBOL[+offs]\|MEMADDR [FETCHARGS] | Set a probe |
| r[:[GRP/]EVENT] SYMBOL[+0] [FETCHARGS] | Set a return probe |
| -:[GRP/]EVENT | Clear a probe |
| **GRP** | Group name. If omitted, "kprobe |
| **EVENT** | Event name. If omitted, the even **SYMBOL[+offs]** or **MEMADDR** |
| **SYMBOL[+offs]** | Symbol+offset where the probe |
| **MEMADDR** | Address where the probe is inse |
| **FETCHARGS** | Arguments. Each probe can hav |
| **%REG** | Fetch register **REG**. |
| **@ADDR** | Fetch memory at **ADDR** (in kerr |
| **@SYM[+\|-offs]** | Fetch memory at **SYM** +\|- offs ( |
| **$stackN** | Fetch *N*th entry of stack (*N* >= ( |
| **$stack** | Fetch stack address |
| **$retval** | Fetch return value 1 |
| **+\|-offs(FETCHARG)** | Fetch memory at **FETCHARG** + |
| **NAME=FETCHARG** | Set **NAME** as the argument nam |