Notes on Analysing Behaviour Using Events and Tracepoints

Author: Mel Gorman (PCL information heavily based on email from Ingo Molnar)

1. Introduction

Tracepoints (see Documentation/trace/tracepoints.rst) can be used without creating custom kernel modules to register probe functions using the event tracing infrastructure.

Simplistically, tracepoints represent important events that can be taken in conjunction with other tracepoints to build a "Big Picture" of what is going on within the system. There are a large number of methods for gathering and interpreting these events. Lacking any current Best Practises, this document describes some of the methods that can be used.

This document assumes that debugfs is mounted on /sys/kernel/debug and that the appropriate tracing options have been configured into the kernel. It is assumed that the PCL tool tools/perf has been installed and is in your path.

2. Listing Available Events

2.1 Standard Utilities

All possible events are visible from /sys/kernel/debug/tracing/events. Simply calling:

```
$ find /sys/kernel/debug/tracing/events -type d
```

will give a fair indication of the number of events available.

2.2 PCL (Performance Counters for Linux)

Discovery and enumeration of all counters and events, including tracepoints, are available with the perf tool. Getting a list of available events is a simple case of:

```
$ perf list 2>&1 | grep Tracepoint
ext4:ext4_free_inode                     [Tracepoint event]
ext4:ext4_request_inode                  [Tracepoint event]
ext4:ext4_allocate_inode                 [Tracepoint event]
ext4:ext4_write_begin                    [Tracepoint event]
ext4:ext4_ordered_write_end              [Tracepoint event]
[ .... remaining output snipped .... ]
```

3. Enabling Events

3.1 System-Wide Event Enabling

See Documentation/trace/events.rst for a proper description on how events can be enabled system-wide. A short example of enabling all events related to page allocation would look something like:

```
$ for i in `find /sys/kernel/debug/tracing/events -name "enable" | grep mm_`; do echo 1 > $i; done
```

3.2 System-Wide Event Enabling with SystemTap

In SystemTap, tracepoints are accessible using the kernel.trace() function call. The following is an example that reports every 5 seconds what processes were allocating the pages.

```
global page_allocs

probe kernel.trace("mm_page_alloc") {
      page_allocs[execname()]++
}

function print_count() {
      printf ("%-25s %-s\n", "#Pages Allocated", "Process Name")
      foreach (proc in page_allocs-)
              printf("%-25d %s\n", page_allocs[proc], proc)
      printf ("\n")
      delete page_allocs
}

probe timer.s(5) {
        print_count()
}
```

3.3 System-Wide Event Enabling with PCL

By specifying the -a switch and analysing sleep, the system-wide events for a duration of time can be examined.

```
$ perf stat -a \
       -e kmem:mm_page_alloc -e kmem:mm_page_free \
       -e kmem:mm_page_free_batched \
       sleep 10
Performance counter stats for 'sleep 10':

          9630  kmem:mm_page_alloc
          2143  kmem:mm_page_free
          7424  kmem:mm_page_free_batched

  10.002577764  seconds time elapsed
```

Similarly, one could execute a shell and exit it as desired to get a report at that point.

3.4 Local Event Enabling

Documentation/trace/ftrace.rst describes how to enable events on a per-thread basis using set_ftrace_pid.

3.5 Local Event Enablement with PCL

Events can be activated and tracked for the duration of a process on a local basis using PCL such as follows.

```
$ perf stat -e kmem:mm_page_alloc -e kmem:mm_page_free \
               -e kmem:mm_page_free_batched ./hackbench 10
Time: 0.909

  Performance counter stats for './hackbench 10':

        17803  kmem:mm_page_alloc
        12398  kmem:mm_page_free
         4827  kmem:mm_page_free_batched

  0.973913387  seconds time elapsed
```

4. Event Filtering

Documentation/trace/ftrace.rst covers in-depth how to filter events in ftrace. Obviously using grep and awk of trace_pipe is an option as well as any script reading trace_pipe.

5. Analysing Event Variances with PCL

Any workload can exhibit variances between runs and it can be important to know what the standard deviation is. By and large, this is left to the performance analyst to do it by hand. In the event that the discrete event occurrences are useful to the performance analyst, then perf can be used.

```
$ perf stat --repeat 5 -e kmem:mm_page_alloc -e kmem:mm_page_free
                      -e kmem:mm_page_free_batched ./hackbench 10
Time: 0.890
Time: 0.895
Time: 0.915
Time: 1.001
Time: 0.899

 Performance counter stats for './hackbench 10' (5 runs):

        16630  kmem:mm_page_alloc         ( +-   3.542% )
        11486  kmem:mm_page_free          ( +-   4.771% )
         4730  kmem:mm_page_free_batched  ( +-   2.325% )

  0.982653002  seconds time elapsed   ( +-   1.448% )
```

In the event that some higher-level event is required that depends on some aggregation of discrete events, then a script would need to be developed.

Using --repeat, it is also possible to view how events are fluctuating over time on a system-wide basis using -a and sleep.

```
$ perf stat -e kmem:mm_page_alloc -e kmem:mm_page_free \
              -e kmem:mm_page_free_batched \
              -a --repeat 10 \
              sleep 1
Performance counter stats for 'sleep 1' (10 runs):

         1066  kmem:mm_page_alloc         ( +-  26.148% )
          182  kmem:mm_page_free          ( +-   5.464% )
```

```
          890  kmem:mm_page_free_batched  ( +-  30.079% )

   1.002251757  seconds time elapsed   ( +-   0.005% )
```

6. Higher-Level Analysis with Helper Scripts

When events are enabled the events that are triggering can be read from /sys/kernel/debug/tracing/trace_pipe in human-readable format although binary options exist as well. By post-processing the output, further information can be gathered on-line as appropriate. Examples of post-processing might include

- Reading information from /proc for the PID that triggered the event
- Deriving a higher-level event from a series of lower-level events.
- Calculating latencies between two events

Documentation/trace/postprocess/trace-pagealloc-postprocess.pl is an example script that can read trace_pipe from STDIN or a copy of a trace. When used on-line, it can be interrupted once to generate a report without exiting and twice to exit.

Simplistically, the script just reads STDIN and counts up events but it also can do more such as

- Derive high-level events from many low-level events. If a number of pages are freed to the main allocator from the per-CPU lists, it recognises that as one per-CPU drain even though there is no specific tracepoint for that event
- It can aggregate based on PID or individual process number
- In the event memory is getting externally fragmented, it reports on whether the fragmentation event was severe or moderate.
- When receiving an event about a PID, it can record who the parent was so that if large numbers of events are coming from very short-lived processes, the parent process responsible for creating all the helpers can be identified

7. Lower-Level Analysis with PCL

There may also be a requirement to identify what functions within a program were generating events within the kernel. To begin this sort of analysis, the data must be recorded. At the time of writing, this required root:

```
$ perf record -c 1 \
      -e kmem:mm_page_alloc -e kmem:mm_page_free \
      -e kmem:mm_page_free_batched \
      ./hackbench 10
Time: 0.894
[ perf record: Captured and wrote 0.733 MB perf.data (~32010 samples) ]
```

Note the use of '-c 1' to set the event period to sample. The default sample period is quite high to minimise overhead but the information collected can be very coarse as a result.

This record outputted a file called perf.data which can be analysed using perf report.

```
$ perf report
# Samples: 30922
#
# Overhead    Command                     Shared Object
# ........  .........  ...............................
#
    87.27%  hackbench  [vdso]
     6.85%  hackbench  /lib/i686/cmov/libc-2.9.so
     2.62%  hackbench  /lib/ld-2.9.so
     1.52%       perf  [vdso]
     1.22%  hackbench  ./hackbench
     0.48%  hackbench  [kernel]
     0.02%       perf  /lib/i686/cmov/libc-2.9.so
     0.01%       perf  /usr/bin/perf
     0.01%       perf  /lib/ld-2.9.so
     0.00%  hackbench  /lib/i686/cmov/libpthread-2.9.so
#
# (For more details, try: perf report --sort comm,dso,symbol)
#
```

According to this, the vast majority of events triggered on events within the VDSO. With simple binaries, this will often be the case so let's take a slightly different example. In the course of writing this, it was noticed that X was generating an insane amount of page allocations so let's look at it:

```
$ perf record -c 1 -f \
              -e kmem:mm_page_alloc -e kmem:mm_page_free \
              -e kmem:mm_page_free_batched \
              -p `pidof X`
```

This was interrupted after a few seconds and

```
$ perf report
```

```
# Samples: 27666
#
# Overhead  Command                            Shared Object
# ........  .......  ...................................
#
    51.95%     Xorg  [vdso]
    47.95%     Xorg  /opt/gfx-test/lib/libpixman-1.so.0.13.1
     0.09%     Xorg  /lib/i686/cmov/libc-2.9.so
     0.01%     Xorg  [kernel]
#
# (For more details, try: perf report --sort comm,dso,symbol)
#
```

So, almost half of the events are occurring in a library. To get an idea which symbol:

```
$ perf report --sort comm,dso,symbol
# Samples: 27666
#
# Overhead  Command                            Shared Object  Symbol
# ........  .......  ...................................  ......
#
    51.95%     Xorg  [vdso]                                   [.] 0x000000ffffe424
    47.93%     Xorg  /opt/gfx-test/lib/libpixman-1.so.0.13.1  [.] pixmanFillsse2
     0.09%     Xorg  /lib/i686/cmov/libc-2.9.so               [.] _int_malloc
     0.01%     Xorg  /opt/gfx-test/lib/libpixman-1.so.0.13.1  [.] pixman_region32_copy_f
     0.01%     Xorg  [kernel]                                 [k] read_hpet
     0.01%     Xorg  /opt/gfx-test/lib/libpixman-1.so.0.13.1  [.] get_fast_path
     0.00%     Xorg  [kernel]                                 [k] ftrace_trace_userstack
```

To see where within the function pixmanFillsse2 things are going wrong:

```
$ perf annotate pixmanFillsse2
[ ... ]
  0.00 :         34eeb:       0f 18 08                prefetcht0 (%eax)
       :      }
       :
       :      extern __inline void __attribute__((__gnu_inline__, __always_inline__, _
       :      _mm_store_si128 (__m128i *__P, __m128i __B) :      {
       :        *__P = __B;
 12.40 :         34eee:       66 0f 7f 80 40 ff ff    movdqa %xmm0,-0xc0(%eax)
  0.00 :         34ef5:       ff
 12.40 :         34ef6:       66 0f 7f 80 50 ff ff    movdqa %xmm0,-0xb0(%eax)
  0.00 :         34efd:       ff
 12.39 :         34efe:       66 0f 7f 80 60 ff ff    movdqa %xmm0,-0xa0(%eax)
  0.00 :         34f05:       ff
 12.67 :         34f06:       66 0f 7f 80 70 ff ff    movdqa %xmm0,-0x90(%eax)
  0.00 :         34f0d:       ff
 12.58 :         34f0e:       66 0f 7f 40 80          movdqa %xmm0,-0x80(%eax)
 12.31 :         34f13:       66 0f 7f 40 90          movdqa %xmm0,-0x70(%eax)
 12.40 :         34f18:       66 0f 7f 40 a0          movdqa %xmm0,-0x60(%eax)
 12.31 :         34f1d:       66 0f 7f 40 b0          movdqa %xmm0,-0x50(%eax)
```

At a glance, it looks like the time is being spent copying pixmaps to the card. Further investigation would be needed to determine why pixmaps are being copied around so much but a starting point would be to take an ancient build of libpixmap out of the library path where it was totally forgotten about from months ago!