How to Use Perf to Debug Perf

Leo Yan <*leo.yan@linaro.org*>
January 28, 2024

Abstract

Debugging is for inspecting a program, for both user space and kernel space. The perf tool contains complex logic for exchanging data through system calls, making debugging perf a challenging practice.

This article explores various debugging techniques, organized from simple to complex, with using the perf tool as the target program. As it explores these debugging methods, the attention is directed towards the perf as a debugging tool. At last, the article explains how to use perf to debug perf.

Keywords: Linux, debug, ftrace, kprobe, uprobe, perf

Introduction

First of all, this article has no plan to address traditional debuggers, e.g. GDB or JTAG based debuggers. These tools use stop-the-world method to debug programs: it firstly stops a target program, then it takes chance to check the context for the program. The *context* can be a software concept, e.g. a thread or task context, a debugger can read variables from the task's stack or heap. The *context* can be a hardware context as well - when a developer is using JTAG debugger to connect hardware and stop CPU, the hardware registers and memory can be reviewed.

The purpose of this documentation is to discuss tools which can provide tracing capability in runtime. The trace data, including events and variable values, are gathered in a certain context without pausing the program for debugging. The content will be divided into four sections:

- Printing;
- Debugging with ftrace;
- Dynamic tracing;
- Using perf to debug perf.

Printing

Usually, a beginner studying C programming learns the first code piece is for printing the string 'hello world!'.

```
#include <stdio.h>
int main(void)
{
        printf("Hello world!\n");
        return 0;
}
```

The above code implicitly introduces a handy tracing tool: libc's `printf()`. Then, when starting to access the Linux kernel, its equivalent function `printk()` will no longer be strange.

A printing log means an event has occurred, alongside variables can be printed out - this is perfect to act as tracing.

The printing is reliable in most cases. `printk()` is a context safe API - even if a developer has no knowledge for interrupt context, thread context and bottom-half context (e.g. in softirq or tasklet), printing still can work as a main debugging method.

On the other hand, developers need to tolerate cons introduced by printing. If logs are output into the UART console, developers suffer performance penalty caused by the low speed of the UART port. In a worse case, the printing can alter program flow and might lead to the timing issues hardly to be reproduced.

It becomes challenging for using printing to debug a program which crosses both user space and kernel space. The reason is `printf()` and `printk()` store logs in separate buffers, resulting in logs that are not easily readable due to out-of-order output.

To resolve this issue, syslog is suggested. A program needs to use `syslog()` to replace `printf()` for routing logs to syslog service, all modules supporting syslog in system can output logs into a central place. But syslog is not necessarily deployed in a system, and many programs, including the perf, don't support syslog at all. As syslog may not be pluasible in some situations, we need to explore other debugging measures.

Debugging with ftrace

If we are looking for a debugging tool with low performance penalties and support for tracing in both user space and kernel space, ftrace appears as a promising candidate:

- Ftrace uses a ring buffer to store trace data, allowing users to save the trace data into a file for post-analysis. This approach helps avoid the tracing latency caused by console.
- Ftrace supports both kernel and user space tracing. The entire trace data is recorded in a single file and displayed in a time-ordered format, thus the output result is friendly for review.

Printing in ftrace

Printing is supported in ftrace.

In the kernel, trace_printk() is used to output logs into the ftrace ring buffer. Once you use it, you will find this API quite useful: logs saved in ftrace buffer will not be bothered by console's latency, and by enlarging buffer size, you will have sufficient capacity to store extensive logs. These logs will not flood the console immediately, you can extract logs into file whenever you want to parse them.

A sysfs node called trace_marker that allows the user space to place logs into ftrace buffer. Through this interface, user space events and kernel events can be synchronized together, thus developers can understand a flow spanning the two spaces. The kernel documentation Documentation/trace/ftrace.rst gives an example for how to write a log into the trace_marker node in C code. The code below is slightly tweaked for easier calling.

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

static void trace_write(const char *fmt, ...)
{
	va_list ap;
	char buf[256];
	int trace_fd, n;

	trace_fd = open("/sys/kernel/debug/tracing/trace_marker",
			O_WRONLY);
	if (trace_fd < 0)
		return;

	va_start(ap, fmt);
	n = vsnprintf(buf, 256, fmt, ap);
	va_end(ap);

	write(trace_fd, buf, n);
}
```

A typical use case for trace_marker is found in ATrace, which is a part of Google's Perfetto tool. Perfetto is widely used for profiling Android UI performance and depends on the ATrace to trace application events - the trace_marker is the underlying mechanism for tracing. While this article will remain on basic tools, it will not dive into ATrace.

Debugging perf with ftrace

Now it's a good time point for us to apply ftrace for debugging.

The following experiment is to inspect how the AUX buffer is consumed in perf. The AUX buffer is a ring buffer designed to store hardware trace data from components such as Arm CoreSight, Arm SPE, Intel PT, etc.

After the trace data has been filled by a hardware IP, the kernel calls the perf_aux_output_end() function to update the head of the buffer and send a notification to user space.

Two buffer modes are supported: overwrite mode and normal mode. The overwrite mode is used for snapshots. For simplicity in explanation, we solely focus on the normal mode from line 493 to line 498 in the code piece below.

Line 496 saves the old head value into the aux_head variable; in line 497, rb->aux_head is updated as a new head by adding the size to the old head.

We add tracing code in lines 500 and 501 to print the old head, the new head and the size into ftrace.

```
481 void perf_aux_output_end(struct perf_output_handle *handle, unsigned long size)
482 {
483         bool wakeup = !!(handle->aux_flags & PERF_AUX_FLAG_TRUNCATED);
484         struct perf_buffer *rb = handle->rb;
485         unsigned long aux_head;
486
487         /* in overwrite mode, driver provides aux_head via handle */
488         if (rb->aux_overwrite) {
489                 handle->aux_flags |= PERF_AUX_FLAG_OVERWRITE;
490
491                 aux_head = handle->head;
492                 rb->aux_head = aux_head;
493         } else {
494                 handle->aux_flags &= ~PERF_AUX_FLAG_OVERWRITE;
495
496                 aux_head = rb->aux_head;
497                 rb->aux_head += size;
498         }
499
500         trace_printk("old_head=0x%lx new_head=0x%lx size=0x%lx\n",
501                      aux_head, rb->aux_head, size);
            ...
535 }
```

The perf tool in user space calls auxtrace_mmap__read_head() to retrieve the latest head of the buffer. As shown from lines 1869 to 1875 in the below code, it handles the overflow case by using a mask or dividing by the buffer length. Finally, a delta between the old head and the new head is calculated, taking into account any wrapping around, which is accomplished between line 1877 and line 1880.

It already contains debugging code at line 1866 for printing logs, but it only outputs messages to a terminal or a log file, we have no chance to print them with kernel logs together. This is why the trace_write() function at line 1882 is added to write logs into the ftrace buffer via the trace_maker interface.

```
1844 static int __auxtrace_mmap__read(struct mmap *map,
1845                                  struct auxtrace_record *itr,
1846                                  struct perf_tool *tool, process_auxtrace_t fn,
1847                                  bool snapshot, size_t snapshot_size)
1848 {
1849         struct auxtrace_mmap *mm = &map->auxtrace_mmap;
1850         u64 head, old = mm->prev, offset, ref;
1851         unsigned char *data = mm->base;
1852         size_t size, head_off, old_off, len1, len2, padding;
1853         union perf_event ev;
1854         void *data1, *data2;
1855         int kernel_is_64_bit = perf_env__kernel_is_64_bit(evsel__env(NULL));
1856
1857         head = auxtrace_mmap__read_head(mm, kernel_is_64_bit);
1858
1859         if (snapshot &&
1860             auxtrace_record__find_snapshot(itr, mm->idx, mm, data, &head, &old))
1861                 return -1;
1862
1863         if (old == head)
1864                 return 0;
1865
1866         pr_debug3("auxtrace idx %d old %#"PRIx64" head %#"PRIx64" diff %#"PRIx64"\n",
1867                   mm->idx, old, head, head - old);
1868
```

```
        if (mm->mask) {
                head_off = head & mm->mask;
                old_off = old & mm->mask;
        } else {
                head_off = head % mm->len;
                old_off = old % mm->len;
        }

        if (head_off > old_off)
                size = head_off - old_off;
        else
                size = mm->len - (old_off - head_off);

        trace_write("%s: old_offset=0x%lx head_offset=0x%lx size=0x%lx\n",
                      __func__, old_off, head_off, size);
       ...
}
```

Rebuild the Linux kernel and perf with the added printing code, reboot the system, then it will be ready for debugging.

To prepare a fresh context for ftrace before running the test, you can use several commands:

```
# Stop tracing
echo 0 > /sys/kernel/debug/tracing/tracing_on

# Cleanup ftrace data
echo > /sys/kernel/debug/tracing/trace

# Start tracing
echo 1 > /sys/kernel/debug/tracing/tracing_on
```

Then, run the test and stop tracing:

```
# Run test
perf record -e cs_etm// -- ls

# Stop tracing
echo 0 > /sys/kernel/debug/tracing/tracing_on
```

At last, you can dump tracing log:

```
# Dump tracing data
cat /sys/kernel/debug/tracing/trace

# tracer: nop
#
# entries-in-buffer/entries-written: 7/7 #P:6
#
#                             _-----=> irqs-off/BH-disabled
#                            / _----=> need-resched
#                           | / _---=> hardirq/softirq
#                           || / _--=> preempt-depth
#                           ||| / _-=> migrate-disable
#                           |||| /    delay
#          TASK-PID    CPU#  |||||  TIMESTAMP FUNCTION
#             | |        |   |||||     |         |
             ls-2041   [003] d..3.  220.497444: perf_aux_output_end: old_head=0x0
                 new_head=0xe26a0 size=0xe26a0
             ls-2041   [003] d..3.  220.498274: perf_aux_output_end: old_head=0xe26a0
                 new_head=0xf0da0 size=0xe700
             ls-2041   [003] d..3.  220.500091: perf_aux_output_end: old_head=0xf0da0
                 new_head=0xf61a0 size=0x5400
             ls-2041   [003] d..3.  220.500743: perf_aux_output_end: old_head=0xf61a0
                 new_head=0x108080 size=0x11ee0
             ls-2041   [003] d..3.  220.502813: perf_aux_output_end: old_head=0x108080
                 new_head=0x119dd0 size=0x11d50
             ls-2041   [003] d..1.  220.508788: perf_aux_output_end: old_head=0x119dd0
```

```
            new_head=0x1d5310 size=0xbb540
        perf-2040   [002] .....  220.508985: tracing_mark_write:
           __auxtrace_mmap__read: old_offset=0x0 head_offset=0x1d5310 size=0
           x1d5310
```

The logs show that the perf_aux_output_end() function has been invoked multiple times in the kernel. The perf tool called __auxtrace_mmap__read() once to read out all trace data from the AUX buffer. The writing to the AUX buffer in the kernel and the reading in user space are not paired. It's apparent that the perf tool is not necessarily woken up every time the kernel stores trace data. Nevertheless, we still don't know the scheduling within this flow, which will be discussed soon.

Tracepoints in ftrace

In the log above, "tracer: nop" indicates that no tracer is enabled. The naming 'ftrace' is derived from 'function trace', users can enable function tracer or function graph tracer for function-based tracing. Over time, ftrace has extended to support other tracers, for example, the latency tracer is for profiling scheduling latency.

Furthermore, ftrace provides tracepoints which are predefined and invoked in the kernel, which are commonly known as "static tracepoints". After the system boots up, you can see the available tracepoints in the subfolder events under the ftrace's debugfs folder.

More importantly, ftrace can combine printing, tracers, and tracepoints together for debugging. The previous section was absent to show how the perf tool is waken up in the test, the scheduler tracepoints can help us to easily understand scheduling behaviours. By enabling the tracepoints, we get logs:

```
# Enable scheduler tracepoints
echo 1 > /sys/kernel/debug/tracing/events/sched/enable

# Run test
perf record -e cs_etm// -- ls

# Stop tracing
echo 0 > /sys/kernel/debug/tracing/tracing_on

# Dump tracing data
cat /sys/kernel/debug/tracing/trace

#                            _-----=> irqs-off/BH-disabled
#                           / _----=> need-resched
#                          | / _---=> hardirq/softirq
#                          || / _--=> preempt-depth
#                          ||| / _-=> migrate-disable
#                          |||| /     delay
#         TASK-PID    CPU#  |||||  TIMESTAMP  FUNCTION
#            | |        |   |||||     |         |
...
           ls-2660    [003] d..1. 4755.179918: perf_aux_output_end: old_head=0x11c460
               new_head=0x21d7a0 size=0x101340
           ls-2660    [003] d.h4. 4755.179979: sched_waking: comm=perf pid=2659 prio
               =120 target_cpu=001
        <idle>-0      [001] dNh2. 4755.179998: sched_wakeup: comm=perf pid=2659 prio
            =120 target_cpu=001
        <idle>-0      [001] d..2. 4755.180002: sched_switch: prev_comm=swapper/1
            prev_pid=0 prev_prio=120 prev_state=R ==> next_comm=perf next_pid=2659
            next_prio=120
           ls-2660    [003] d.h2. 4755.180014: sched_stat_runtime: comm=ls pid=2660
               runtime=4493620 [ns]
```

```
perf-2659   [001] ..... 4755.180089: tracing_mark_write:
     __auxtrace_mmap__read: old_offset=0x0 head_offset=0x21d7a0 size=0
     x21d7a0
```

The logs show that the profiled program *ls* stored hardware tracing data into AUX buffer. Then it woke up the perf process in the `sched_waking` event. Afterwards, the $CPU1$ was pulled out from idle, and the scheduler placed the perf process to run on it. We can know that the *ls* program and the perf running on two different CPUs, so that avoid performance degradation due to parallel execution during profiling.

However, the printing and static tracepoints in ftrace are not efficient, as we must rebuild the source code to add tracing. We will explore dynamic tracing to add tracepoints on the fly.

Dynamic tracing

If you have experience with debugger, an often used feature is breakpoint. You select a code line, set a breakpoint, and then kick off the program to run. When the program reaches the breakpoint, it halts, and the debugger takes over control. At this point, since the program is stopped, you can take your time to read variables, review memory content, and dump CPU general registers.

An aspect of a breakpoint is it can be set as either a hardware breakpoint or a software breakpoint. A hardware breakpoint is to set an address in the CPU's debug register, while a software breakpoint uses the break instruction (the Arm instruction is `BRK`) to replace an original instruction at the specified address. Both methods ultimately interrupt the program execution and transfer control to the debugger for further inspection.

We can take dynamic tracing as a self-hosted debugger, often referred to as *probe* in Linux. When we add a probe, a break instruction is injected into a specified address, and an event is attached to it for accessing additional data. The original instruction is copied to somewhere for a single-step execution.

Ftrace provides *kprobe* and $uprobe$ for adding probe in the kernel and user space respectively. We will demonstrate how to use them.

Adding probe in the kernel

The kprobe provides the sysfs node `kprobe_events` under the ftrace's umbrella for adding dynamic tracepoints.

Adding a probe requires specifying two things: the probed address and the inspected data. The file `Documentation/trace/kprobetrace.rst` in the kernel tree explains the kprobe syntax.

An obstacle to using kprobe is figuring out the appropriate address and determining where the interested variables are stored. Tracing a function's entry or return is straightforward, but determining the address becomes challenging when intending to observe in the middle of the function. Arguments or local variables might be stored in general registers or in the stack, and it's possible that a variable is located in the heap.

Therefore, we need to understand how a Linux kernel image is compiled. We can disassemble a kernel ELF file with the `objdump` command. When working in a cross compilation environment for Arm64, you need to use the command `aarch64-linux-gnu-objdump` instead. The command below uses two options for the disassembly: the option `'-d'` is for displaying assembler, and the option `'-S'` is for dumping the source code so we can intermix C code with assembly instructions.

```
aarch64-linux-gnu-objdump -S -d vmlinux > kernel.objdump
```

In addition to the knowledge of assembly language, understanding the general-purpose register usage in the procedure call is crucial for reading disassembly. The documentation AAPCS64 defines the Procedure Call Standard for AArch64. Applying the prerequisite knowledge, let's take a closer look at the disassembly of the `perf_aux_output_end()` function.

```
ffff8000802a0ee0 <perf_aux_output_end>:
ffff8000802a0ee0: d503201f  nop
ffff8000802a0ee4: d503201f  nop
{
ffff8000802a0ee8: d503233f  paciasp
ffff8000802a0eec: a9bd7bfd  stp  x29, x30, [sp, #-48]!
ffff8000802a0ef0: aa0103e2  mov  x2, x1
ffff8000802a0ef4: 910003fd  mov  x29, sp
ffff8000802a0ef8: a90153f3  stp  x19, x20, [sp, #16]
ffff8000802a0efc: aa0003f3  mov  x19, x0
ffff8000802a0f00: f90013f5  str  x21, [sp, #32]
   struct perf_buffer *rb = handle->rb;
ffff8000802a0f04: f9400414  ldr  x20, [x0, #8]
   bool wakeup = !!(handle->aux_flags & PERF_AUX_FLAG_TRUNCATED);
ffff8000802a0f08: f9401000  ldr  x0, [x0, #32]
   if (rb->aux_overwrite) {
ffff8000802a0f0c: b940b681  ldr  w1, [x20, #180]
   bool wakeup = !!(handle->aux_flags & PERF_AUX_FLAG_TRUNCATED);
ffff8000802a0f10: 12000015  and  w21, w0, #0x1
   if (rb->aux_overwrite) {
ffff8000802a0f14: 34000741  cbz  w1, ffff8000802a0ffc <perf_aux_output_end+0x11c>
      aux_head = handle->head;
ffff8000802a0f18: f9401661  ldr  x1, [x19, #40]
      handle->aux_flags |= PERF_AUX_FLAG_OVERWRITE;
ffff8000802a0f1c: b27f0000  orr  x0, x0, #0x2
ffff8000802a0f20: f9001260  str  x0, [x19, #32]
      rb->aux_head = aux_head;
ffff8000802a0f24: aa0103e0  mov  x0, x1
ffff8000802a0f28: f9004a80  str  x0, [x20, #144]

...

      handle->aux_flags &= ~PERF_AUX_FLAG_OVERWRITE;
ffff8000802a0ffc: 927ef800  and  x0, x0, #0xfffffffffffffffd
ffff8000802a1000: f9001260  str  x0, [x19, #32]
      aux_head = rb->aux_head;
ffff8000802a1004: f9404a81  ldr  x1, [x20, #144]
      rb->aux_head += size;
ffff8000802a1008: 8b020020  add  x0, x1, x2
ffff8000802a100c: 17ffffc7  b ffff8000802a0f28 <perf_aux_output_end+0x48>
ffff8000802a1010: d503201f  nop
ffff8000802a1014: d503201f  nop

ffff8000802a1018 <rb_alloc>:
ffff8000802a1018: d503201f  nop
ffff8000802a101c: d503201f  nop
   page->mapping = NULL;
   __free_page(page);
}
```

AAPCS64 defines that registers x0 - x7 are used to pass function arguments. `perf_aux_output_end()` has two arguments: the first one is an output handler, and the second one is the filled buffer size. When it is called, the registers x0 and x1 hold values for these two arguments, respectively. At the address 0xffff8000802a0ef0, the instruction is "move x2, x1", which moves the size value in the register x1 into x2. Later in the function, the register x2 holds this value while x1 is assigned to intermediate values.

The register x20 is assigned at 0xffff8000802a0f04 for a pointer value pointing to structure perf_buffer. Then, at the address 0xffff8000802a0f0c, the rb->aux_overwrite is loaded. Since this field has an offset of 180 in the structure perf_buffer, the instruction "ldr w1, [x20, #180]" loads it into the register w1 (using w1 as target register means loading value into the x1 with upper 4 bytes zeroed).

At 0xffff8000802a0f14, the instruction "cbz w1, ffff8000802a0ffc" compares the buffer mode. Now we are only interesed in the normal mode and w1 is zero, as a result, the instruction will jump to 0xffff8000802a0ffc.

From there, it sets the handle->aux_flags and retrieves the old buffer head into the register x1 at 0xffff8000802a1004 - the instruction is "ldr x1, [x20, #144]", 144 is the offset of the buffer's head in the structure.

As we know, x2 contains the written buffer size. It is added to the old buffer head in x1 for a new buffer head and is stored back into the register x0. This operation is accomplished at 0xffff8000802a1008.

The consecutive address 0xffff8000802a100c would be a good trace point, with registers x0, x1 and x2 containing the values we want to dump. To achieve this, we can add a probe with the following command:

```
cd /sys/kernel/debug/tracing/
echo 'p:myprobe 0xffff8000802a100c new_head=%x0:x64 old_head=%x1:x64 size=%x2:x64' \
      > kprobe_events
```

In the command, 'p' means to add a probe, and 'myprobe' is the event name for the probe. The address 0xffff8000802a100c is where the probe is inserted. The subsequent arguments are for dumping values. The option 'new_head=%x0:x64' means an argument named as 'new_head', reading value from the register x0 and printing as the 64-bit hexadecimal type 'x64'. The later options follow the same format.

Improving readability for the kprobe command is plausible. We can use the 'function name + offset' format to replace an arbitrary address for specifying a probe address. This is more readable and will not be bothered by address alterations caused by a rebuild:

```
echo 'p:myprobe perf_aux_output_end+0x12c new_head=%x0:x64 \
      old_head=%x1:x64 size=%x2:x64' > kprobe_events
```

Moreover, the probe supports fetching memory with a fetch register and an offset (the syntax is '+/-offset(REG)'). In this case, the old head is kept in two places: it is loaded into x1, and it's stored in the address pointed to by x20 plus 144. Instead of accessing register x1 to retrieve the old head, we can use '+144(%x20)' for the same purpose. Thus, the command can be updated as:

```
echo 'p:myprobe perf_aux_output_end+0x12c new_head=%x0:x64 \
      old_head=+144(%x20):x64 size=%x2:x64' > kprobe_events
```

Adding probe in user space

Similarly to kprobe, ftrace provides a sysfs node `uprobe_events` for inserting a probe into a user space program. We can apply the methodology discussed in the previous section to analyze the disassembly of a program running in user space.

Firstly, generate the disassembly for the perf with the command:

```
aarch64-linux-gnu-objdump -S -d perf > perf.objdump
```

In the dump file, we can get the disasembly for `__auxtrace_mmap__read()`.

```
00000000002077f8 <__auxtrace_mmap__read>:

static int __auxtrace_mmap__read(struct mmap *map,
            struct auxtrace_record *itr,
            struct perf_tool *tool, process_auxtrace_t fn,
            bool snapshot, size_t snapshot_size)
{
...
   if (head_off > old_off)
  2079a0:  f9403fe1   ldr  x1, [sp, #120]
  2079a4:  f94043e0   ldr  x0, [sp, #128]
  2079a8:  eb00003f   cmp  x1, x0
  2079ac:  540000c9   b.ls 2079c4 <__auxtrace_mmap__read+0x1cc> // b.plast
     size = head_off - old_off;
  2079b0:  f9403fe1   ldr  x1, [sp, #120]
  2079b4:  f94043e0   ldr  x0, [sp, #128]
  2079b8:  cb000020   sub  x0, x1, x0
  2079bc:  f9003be0   str  x0, [sp, #112]
  2079c0:  14000008   b  2079e0 <__auxtrace_mmap__read+0x1e8>
   else
     size = mm->len - (old_off - head_off);
  2079c4:  f9405be0   ldr  x0, [sp, #176]
  2079c8:  f9400c01   ldr  x1, [x0, #24]
  2079cc:  f9403fe2   ldr  x2, [sp, #120]
  2079d0:  f94043e0   ldr  x0, [sp, #128]
  2079d4:  cb000040   sub  x0, x2, x0
  2079d8:  8b000020   add  x0, x1, x0
  2079dc:  f9003be0   str  x0, [sp, #112]

   if (snapshot && size > snapshot_size)
  2079e0:  39407fe0   ldrb w0, [sp, #31]
...
}
```

The truncated disassembly piece is for reading the old head and the latest head, and calculating the the filled buffer size. The `'if'` branch handles the normal case and the `'else'` branch handles the wrap-around case. In the end, both branches run to the address `2079e0` (see the branch instruction `"b 2079e0"` at `2079c0`), we can select it as the probed address.

From the load and store instructions, we can know variables are stored in the stack. If it falls into the `"if"` branch, the instruction `"ldr x1, [sp, #120]"` loads the head from the stack with an offset `120`, and `"ldr x0, [sp, #128]"` fetches the old head from the stack with an offset `128`. The filled buffer size is calculated as the delta between the head and the old head, and it is stored back into the stack with the instcution `"str x0, [sp, #112]"`, the offset is `112`. The `"else"` branch does the same for storing the variables into the stack.

Consequently, we can use the command to inject a user probe:

```
echo 'p /mnt/linux-kernel/tools/perf/perf:0x2079e0 \
      old_off=+128(%sp):x64 head_off=+120(%sp):x64 size=+112(%sp):x64' > uprobe_events
```

Now we have known how to add probes for kernel and user space, we just need to re-run the test and capture the trace log.

```
# Change to tracing folder
cd /sys/kernel/debug/tracing/

# Add kernel probe
echo 'p:myprobe perf_aux_output_end+0x12c new_head=%x0:x64 \
      old_head=+144(%x20):x64 size=%x2:x64' > kprobe_events

# Enable the kprobe tracepoint
echo 1 > events/kprobes/myprobe1/enable

# Add user space probe
echo 'p /mnt/linux-kernel/tools/perf/perf:0x2079e0 \
      old_off=+128(%sp):x64 head_off=+120(%sp):x64 size=+112(%sp):x64' > uprobe_events

# Enable the user space tracepoint
echo 1 > events/uprobes/p_perf_0x2079e0/enable

# Start tracing
echo 1 > tracing_on

# Run test
perf record -e cs_etm// -- ls

# Stop tracing
echo 0 > tracing_on

# Dump tracing data
cat trace

# tracer: nop
#
# entries-in-buffer/entries-written: 7/7 #P:6
#
#                             _-----=> irqs-off/BH-disabled
#                            / _----=> need-resched
#                           | / _---=> hardirq/softirq
#                           || / _--=> preempt-depth
#                           ||| / _-=> migrate-disable
#                           |||| /    delay
#          TASK-PID    CPU# ||||| TIMESTAMP FUNCTION
#            | |        |   |||||    |        |
            ls-3480   [005] d..3. 13312.444215: myprobe1: (perf_aux_output_end+0x12c/0
                 x138) new_head=0xd71f0 old_head=0x0 size=0xd71f0
            ls-3480   [005] d..3. 13312.444779: myprobe1: (perf_aux_output_end+0x12c/0
                 x138) new_head=0xe22a0 old_head=0xd71f0 size=0xb0b0
            ls-3480   [005] d..3. 13312.446907: myprobe1: (perf_aux_output_end+0x12c/0
                 x138) new_head=0xf2950 old_head=0xe22a0 size=0x106b0
            ls-3480   [005] d..3. 13312.449854: myprobe1: (perf_aux_output_end+0x12c/0
                 x138) new_head=0x10e470 old_head=0xf2950 size=0x1bb20
            ls-3480   [005] d..3. 13312.456476: myprobe1: (perf_aux_output_end+0x12c/0
                 x138) new_head=0x1ac6e0 old_head=0x10e470 size=0x9e270
            ls-3480   [005] d..1. 13312.460459: myprobe1: (perf_aux_output_end+0x12c/0
                 x138) new_head=0x243560 old_head=0x1ac6e0 size=0x96e80
          perf-3479   [002] DNZff 13312.460598: p_perf_0x2079e0: (0xaaaad0f979e0)
               old_off=0x0 head_off=0x243560 size=0x243560
```

Using perf to debug perf

Kprobe and uprobe inherently require developers to acquire intense knowledge before using them. Things are not perfect. This is why we move eyes to the perf tool. The perf tool is capable of reading ELF files and annotating with source code. It provides a much more user-friendly way to set up a probe.

The command ’`perf probe`’ is for both kprobe and uprobe. The two options ’`--line`’ and ’`--vars`’ in the command are quite handy. The former is for locating code lines, and the latter one gives out variables avaliable for tracing. After finalizing the traced address and variables, the option ’`--add`’ is used to add a probe. If the tracepoint is no longer needed, we can use the option ’`--del`’ to remove it.

Let’s see how to use perf to debug perf.

Building binaries with debugging info

As said, the perf tool can understand the debugging info in an ELF file. Building is required to enable debugging options so that the perf tool has sufficient information to connect the binary with source code.

The example below sets up the Linux kernel debug configurations. When `CONFIG_DEBUG_INFO` is enabled, the compiler option ’`-g`’ is turned on. The configurations `CONFIG_DEBUG_INFO_DWARF5` and `CONFIG_DEBUG_INFO_BTF` are selected to enable the DWARF5 debug data format and BPF type format. At last, `CONFIG_DEBUG_INFO_REDUCED` is disabled to avoid stripping debugging information from the `vmlinux`.

```
cd /path/to/kernel/
./scripts/config -e CONFIG_DEBUG_INFO
./scripts/config -e CONFIG_DEBUG_INFO_DWARF5
./scripts/config -e CONFIG_DEBUG_INFO_BTF
./scripts/config -d CONFIG_DEBUG_INFO_REDUCED
```

It’s also necessary to enable compiler’s debug option and mute the optimization option when building the perf. This can be achieved by simply appending the option ’`DEBUG=1`’ in the make command. The option ’`CORESIGHT=1`’ is for support Arm CoreSight decoder in the tool.

```
cd /path/to/kernel/tools/perf
make DEBUG=1 CORESIGHT=1
```

Adding a probe in the kernel

To learn which source code lines in `perf_aux_output_end()` are suitable for injecting a tracepoint, we can use option ’`--line`’ in the ’`perf probe`’ command.

Perf needs the vmlinux file. The vmlinux path can be offered by the option ’`-k`’ or ’`--vmlinux`’, if not, it is assumed to be placed in the same folder as the current working directory or in any of the predefined paths in the structure `vmlinux_paths_upd`. The option ’`-s`’ followed by a path is used to specify the Linux kernel source, which should be consistent with the built kernel image, in below case, the kernel source folder is ’`/mnt/linux-kernel/`’.

```
perf probe --line "perf_aux_output_end" -k ./vmlinux -s /mnt/linux-kernel/
```

As result, we can get output:

```
<perf_aux_output_end@.///kernel/events/ring_buffer.c:0>
```

```
      0  void perf_aux_output_end(struct perf_output_handle *handle, unsigned long size)

         {
                  bool wakeup = !!(handle->aux_flags & PERF_AUX_FLAG_TRUNCATED);
      3           struct perf_buffer *rb = handle->rb;
                  unsigned long aux_head;

                  /* in overwrite mode, driver provides aux_head via handle */
                  if (rb->aux_overwrite) {
      8                   handle->aux_flags |= PERF_AUX_FLAG_OVERWRITE;

     10                   aux_head = handle->head;
                          rb->aux_head = aux_head;
                  } else {
     13                   handle->aux_flags &= ~PERF_AUX_FLAG_OVERWRITE;

     15                   aux_head = rb->aux_head;
     16                   rb->aux_head += size;
                  }

                  /*
                   * Only send RECORD_AUX if we have something useful to communicate
                   *
                   * Note: the OVERWRITE records by themselves are not considered
                   * useful, as they don’t communicate any *new* information,
                   * aside from the short-lived offset, that becomes history at
                   * the next event sched-in and therefore isn’t useful.
                   * The userspace that needs to copy out AUX data in overwrite
                   * mode should know to use user_page::aux_head for the actual
                   * offset. So, from now on we don’t output AUX records that
                   * have *only* OVERWRITE flag set.
                   */
     31           if (size || (handle->aux_flags & ~(u64)PERF_AUX_FLAG_OVERWRITE))
     32                   perf_event_aux_event(handle->event, aux_head, size,
                                               handle->aux_flags);

     35           WRITE_ONCE(rb->user_page->aux_head, rb->aux_head);
                  if (rb_need_aux_wakeup(rb))
                          wakeup = true;

     39           if (wakeup) {
     40                   if (handle->aux_flags & PERF_AUX_FLAG_TRUNCATED)
     41                           handle->event->pending_disable = smp_processor_id();
     42                   perf_output_wakeup(handle);
                  }

     45           handle->event = NULL;

     47           WRITE_ONCE(rb->aux_nest, 0);
                  /* can’t be last */
                  rb_free_aux(rb);
     50           ring_buffer_put(rb);
         }
```

The tool annotates lines with prefixed numbers that are candidates for probes. The buffer head is calculated from line 8 to line 16. After that, line 31 is a good place to observe the calculation result.

Next, we intend to check which variables are available at the line 31 of the function. So the option ’--vars’ can be used with specifying the source code line with format ’function_name:line_number’:

```
perf probe --vars "perf_aux_output_end:31" -s /mnt/linux-kernel/
```

The output ’@<perf_aux_output_end+68>’ indicates that source code line mapped to the offset of 68 from the start of the function. Then, there are five local variables availiable for tracing. If we recall the previous analysis, we know that aux_head holds the old head, rb->aux_head is for the new head, and the variable size presents the filled size.

```
Available variables at perf_aux_output_end:31
        @<perf_aux_output_end+68>
                bool    wakeup
                long unsigned int       aux_head
                long unsigned int       size
                struct perf_buffer*     rb
                struct perf_output_handle*      handle
```

Now, use the option '--add' to add a probe in the kernel.

```
perf probe --add "perf_aux_output_end:31 old_head=aux_head \
                  new_head=rb->aux_head:x64 size=size:x64" -s /mnt/linux-kernel/
Added new event:
  probe:perf_aux_output_end_L31 (on perf_aux_output_end:31 with old_head=aux_head
       new_head=rb->aux_head:x64 size=size:x64)

You can now use it in all perf tools, such as:

   perf record -e probe:perf_aux_output_end_L31 -aR sleep 1
```

The advantage of perf is that it allows users to write readable expression and converts to the low-level probe syntax with raw addresses and registers. This can significantly reduce the difficulty of setting up probes.

Adding a probe in the perf

The rest is to apply the same steps for adding a probe in the perf tool.

Firstly, we display the source code lines for the __auxtrace_mmap__read() function. To do this, We need to specify the traced executable with the option '-x' or '--exec'.

```
perf probe -x /mnt/linux-kernel/tools/perf/perf --line "__auxtrace_mmap__read"
```

The dumping below shows that the size calculation extends until line 36, and line 38 would be a suitable location for injecting a probe.

```
<__auxtrace_mmap__read@/mnt/linux-kernel/tools/perf/util/auxtrace.c:0>
      0  static int __auxtrace_mmap__read(struct mmap *map,
<__auxtrace_mmap__read@/mnt/linux-kernel/tools/perf/util/auxtrace.c:0>
      0  static int __auxtrace_mmap__read(struct mmap *map,
                                          struct auxtrace_record *itr,
                                          struct perf_tool *tool, process_auxtrace_t fn,
                                          bool snapshot, size_t snapshot_size)
      4  {
      5          struct auxtrace_mmap *mm = &map->auxtrace_mmap;
      6          u64 head, old = mm->prev, offset, ref;
      7          unsigned char *data = mm->base;
                 size_t size, head_off, old_off, len1, len2, padding;
                 union perf_event ev;
                 void *data1, *data2;
     11          int kernel_is_64_bit = perf_env__kernel_is_64_bit(evsel__env(NULL));

     13          head = auxtrace_mmap__read_head(mm, kernel_is_64_bit);

     15          if (snapshot &&
     16              auxtrace_record__find_snapshot(itr, mm->idx, mm, data, &head, &old))
     17                  return -1;

     19          if (old == head)
     20                  return 0;

     22          pr_debug3("auxtrace idx %d old %#"PRIx64" head %#"PRIx64" diff %#"PRIx64
             "\n",
                         mm->idx, old, head, head - old);

     25        if (mm->mask) {
     26                head_off = head & mm->mask;
     27                old_off = old & mm->mask;
                 } else {
     29                head_off = head % mm->len;
     30                old_off = old % mm->len;
                 }

     33        if (head_off > old_off)
     34                size = head_off - old_off;
                 else
     36                size = mm->len - (old_off - head_off);

     38        if (snapshot && size > snapshot_size)
     39                size = snapshot_size;
      ...
         }
```

We confirm which variables are accessible at the line 38 of the function with option '--vars'.

```
perf probe -x /mnt/linux-kernel/tools/perf/perf --vars "__auxtrace_mmap__read:38"

Available variables at __auxtrace_mmap__read:38
        @<__auxtrace_mmap__read+488>
                (unknown_type) data1
                (unknown_type) data2
                _Bool   snapshot
                int     kernel_is_64_bit
                process_auxtrace_t    fn
                size_t head_off
                size_t len1
                size_t len2
                size_t old_off
                size_t padding
                size_t size
                size_t snapshot_size
                struct auxtrace_mmap* mm
                struct auxtrace_record* itr
                struct mmap*   map
                struct perf_tool*     tool
                u64     head
                u64     offset
                u64     old
                u64     ref
                union perf_event      ev
                unsigned char* data
        @<__auxtrace_mmap__read+500>
                (unknown_type) data1
                (unknown_type) data2
                _Bool   snapshot
                int     kernel_is_64_bit
                process_auxtrace_t    fn
                size_t head_off
                size_t len1
                size_t len2
                size_t old_off
                size_t padding
                size_t size
                size_t snapshot_size
                struct auxtrace_mmap* mm
                struct auxtrace_record* itr
                struct mmap*   map
                struct perf_tool*     tool
                u64     head
                u64     offset
                u64     old
                u64     ref
                union perf_event      ev
```

```
              unsigned char* data
```

It's a bit suprising that the tool dumps out two different offsets (one is +488 and another is +500) for line 38, with the associated variables for each offset. Due to line 38 having two conditions checking compiled with multiple branch instructions, adding a probe for this line will automatically inject tracepoints for all relevant offsets.

As a side topic, when you attempt to show variables for a probe point in an inline function, it's also possible to output multiple probe points. This is because an inline function can be compiled into different functions, the functions plus offset will be displayed in this case.

Instead of using 'function_name:line_number', we can use the format 'function_name+offset' to specify an accurate probe point. Here, we select the offset 488 for tracing. Therefore, the adding probe command is:

```
perf probe -x /mnt/linux-kernel/tools/perf/perf --add "__auxtrace_mmap__read+488
     head_off=head_off old_off=old_off size=size"
Added new event:
  probe_perf:__auxtrace_mmap__read (on __auxtrace_mmap__read+488 in /mnt/linux-kernel/
       tools/perf/perf with head_off=head_off old_off=old_off size=size)

You can now use it in all perf tools, such as:

   perf record -e probe_perf:__auxtrace_mmap__read -aR sleep 1
```

Tracing with the perf

After setting up probes, now we need to consume these tracepoints. Let's begin by reproducing the steps via the tracing virtual file system.

```
# Change to tracing folder
cd /sys/kernel/debug/tracing/

# Enable the kprobe tracepoint
echo 1 > events/probe/perf_aux_output_end_L31/enable

# Enable the user space tracepoint
echo 1 > events/probe_perf/__auxtrace_mmap__read/enable

# Start tracing
echo 1 > tracing_on

# Run test
perf record -e cs_etm// -- ls

# Stop tracing
echo 0 > tracing_on

# Dump tracing data
cat trace
# tracer: nop
#
# entries-in-buffer/entries-written: 6/6 #P:6
#
#                             _-----=> irqs-off/BH-disabled
#                            / _----=> need-resched
#                           | / _---=> hardirq/softirq
#                           || / _--=> preempt-depth
#                           ||| / _-=> migrate-disable
#                           |||| /    delay
#          TASK-PID    CPU# ||||| TIMESTAMP FUNCTION
#             | |        |  |||||    |         |
            ls-4889   [005] d..3. 50585.840092: perf_aux_output_end_L31: (
                 perf_aux_output_end+0x4c/0x138) old_head=0x0 new_head=0xe4cd0 size=0
                 xe4cd0
            ls-4889   [005] d..3. 50585.841170: perf_aux_output_end_L31: (
                 perf_aux_output_end+0x4c/0x138) old_head=0xe4cd0 new_head=0xf0750
                 size=0xba80
            ls-4889   [005] d..3. 50585.843651: perf_aux_output_end_L31: (
                 perf_aux_output_end+0x4c/0x138) old_head=0xf0750 new_head=0x100eb0
                 size=0x10760
            ls-4889   [005] d..3. 50585.847108: perf_aux_output_end_L31: (
                 perf_aux_output_end+0x4c/0x138) old_head=0x100eb0 new_head=0x11c870
                 size=0x1b9c0
            ls-4889   [005] d..1. 50585.857568: perf_aux_output_end_L31: (
                 perf_aux_output_end+0x4c/0x138) old_head=0x11c870 new_head=0x219810
                 size=0xfcfa0
          perf-4888   [002] DNZff 50585.857721: __auxtrace_mmap__read: (0xaaaab03279e0
               ) head_off=0x219810 old_off=0x0 size=0x219810
```

In fact, with perf, it's no need to directly use ftrace knobs anymore. The perf tool can handle everything, including recording and reporting the trace data. We can use perf to debug perf!

Using perf to debug perf involves two perf programs running. In the current case, the first perf program opens `cs_etm` PMU event and records Arm CoreSight hardware trace data for program `ls`. The command is `'perf record -o perf.data.etm -e cs_etm// -- ls'`. Note that `'--'` is a separator followed by a traced program and its arguments (if there are any).

The second perf program is for debugging purpose. It opens the pre-configured probes and takes the first perf program as its debugging target. The two perf programs are separated by another separator `'--'`.

By default, perf saves recording into the file `'perf.data'` if no file name is given. But, if two perf programs use the same file for saving trace data, it can cause mess. Therefore, we specify distinct output file names using the '-o' option for each perf program. One is `'perf.data.etm'` for the session related to the `cs_etm` PMU event, and the another is `'perf.data.dbg'` for the debugging program.

With the above explanation, the following command kicks off the both perf programs in one go:

```
perf record -o perf.data.dbg -e probe:perf_aux_output_end_L31 \
           -e probe_perf:__auxtrace_mmap__read -- \
           perf record -o perf.data.etm -e cs_etm// -- ls
```

After recording, the `'perf script'` command can help us print out the trace data. It outputs the details for the events, as shown below, in the same format as ftrace dumping.

```
perf script -i perf.data.dbg
            ls   10102 [000] 54342.677011: probe:perf_aux_output_end_L31: (
                 ffff8000802a0f2c) old_head=0x0 new_head=0xead30 size=0xead30
            ls   10102 [001] 54342.677796: probe:perf_aux_output_end_L31: (
                 ffff8000802a0f2c) old_head=0x0 new_head=0xbb90 size=0xbb90
            ls   10102 [001] 54342.680020: probe:perf_aux_output_end_L31: (
                 ffff8000802a0f2c) old_head=0xbb90 new_head=0x1bda0 size=0x10210
            ls   10102 [003] 54342.686812: probe:perf_aux_output_end_L31: (
                 ffff8000802a0f2c) old_head=0x0 new_head=0x1c990 size=0x1c990
            ls   10102 [003] 54342.695337: probe:perf_aux_output_end_L31: (
                 ffff8000802a0f2c) old_head=0x1c990 new_head=0x9cb40 size=0x801b0
            ls   10102 [003] 54342.698274: probe:perf_aux_output_end_L31: (
                 ffff8000802a0f2c) old_head=0x9cb40 new_head=0x1095f0 size=0x6cab0
          perf   10100 [001] 54342.698429: probe_perf:__auxtrace_mmap__read: (
              aaaad68236fc) head_off=0xead30 old_off=0x0 size=0xead30
          perf  10100 [001] 54342.701141: probe_perf:__auxtrace_mmap__read: (
              aaaad68236fc) head_off=0x1bda0 old_off=0x0 size=0x1bda0
          perf  10100 [001] 54342.701500: probe_perf:__auxtrace_mmap__read: (
              aaaad68236fc) head_off=0x1095f0 old_off=0x0 size=0x1095f0
```

Sometimes, we may be curious about how a function is invoked in a flow. We can append the configuration 'call-graph=fp' for a tracepoint in the recording. When the tracepoint is triggered, its call graph will be captured as well. The `perf record` command is updated for call stack tracing:

```
perf record -o perf.data.dbg -e probe:perf_aux_output_end_L31/call-graph=fp/ \
          -e probe_perf:__auxtrace_mmap__read/call-graph=fp/ -- \
          perf record -o perf.data.etm -e cs_etm// -- ls
```

We can then use the `'perf script'` command to review the trace data again. This time, call chains of the traced functions are included. Since perf collects all objects, including the executables and libraries in the user space and the Linux image, it has the capability to parse symbols for the entire system. That's why, when reviewing the call chain of `perf_aux_output_end()` in the log below, you can see that the call originates from the user space. The process `ls` then attempts to perform scheduling, and during the context switching out, the Arm CoreSight trace data is recorded. This provides insights into the flow in the first perf program.

```
perf script -i perf.data.dbg

ls  10028 [003] 53664.368328: probe:perf_aux_output_end_L31: (ffff8000802a0f2c)
    old_head=0x0 new_head=0xed100 size=0xed100
      ffff8000802a0f2c perf_aux_output_end+0x4c (vmlinux)
      ffff800081052240 etm_event_stop+0x140 (vmlinux)
      ffff8000810522d4 etm_event_del+0x1c (vmlinux)
      ffff800080294384 event_sched_out+0x9c (vmlinux)
      ffff800080294574 group_sched_out+0x5c (vmlinux)
      ffff800080294930 __pmu_ctx_sched_out+0xe8 (vmlinux)
      ffff800080294a48 ctx_sched_out+0xc8 (vmlinux)
      ffff800080294b50 task_ctx_sched_out+0x38 (vmlinux)
      ffff800080296a98 __perf_event_task_sched_out+0x1a0 (vmlinux)
      ffff8000812f40a0 __schedule+0x400 (vmlinux)
      ffff8000812f4814 schedule+0x3c (vmlinux)
      ffff80008126f330 rpc_wait_bit_killable+0x20 (vmlinux)
      ffff8000812f5038 __wait_on_bit+0x58 (vmlinux)
      ffff8000812f51ec out_of_line_wait_on_bit+0x8c (vmlinux)
      ffff800081279020 __rpc_execute+0x120 (vmlinux)
      ffff8000812797d4 rpc_execute+0x164 (vmlinux)
      ffff80008125971c rpc_run_task+0x12c (vmlinux)
      ffff800080537fb4 nfs4_do_call_sync+0x7c (vmlinux)
      ffff8000805380f8 _nfs4_proc_getattr+0xf0 (vmlinux)
      ffff80008054116c nfs4_proc_getattr+0x7c (vmlinux)
      ffff80008050e944 __nfs_revalidate_inode+0xe4 (vmlinux)
      ffff800080502cdc nfs_lookup_verify_inode+0x94 (vmlinux)
      ffff800080502d64 nfs_weak_revalidate+0x5c (vmlinux)
      ffff8000803a3b94 complete_walk+0x9c (vmlinux)
      ffff8000803a8794 path_openat+0x82c (vmlinux)
      ffff8000803a9834 do_filp_open+0xa4 (vmlinux)
      ffff80008038f948 do_sys_openat2+0xc8 (vmlinux)
      ffff80008038fccc __arm64_sys_openat+0x6c (vmlinux)
      ffff800080029a30 invoke_syscall+0x50 (vmlinux)
      ffff800080029bd0 el0_svc_common.constprop.0+0xc8 (vmlinux)
      ffff800080029c1c do_el0_svc+0x24 (vmlinux)
      ffff8000812ec134 el0_svc+0x34 (vmlinux)
      ffff8000812ec568 el0t_64_sync_handler+0x100 (vmlinux)
      ffff800080011d50 el0t_64_sync+0x190 (vmlinux)
          ffff880b5c70 __open64_nocancel+0x48 (/usr/lib/aarch64-linux-gnu/libc-2.28.so
              )
          ffff8808c824 __opendir+0x1c (/usr/lib/aarch64-linux-gnu/libc-2.28.so)
          aaaacaa8a4d8 [unknown] (/usr/bin/ls)
          aaaacaa843f0 [unknown] (/usr/bin/ls)
          ffff8800fd24 __libc_start_main+0xe4 (/usr/lib/aarch64-linux-gnu/libc-2.28.so
              )
          aaaacaa8584c [unknown] (/usr/bin/ls)

...

perf  10026 [001] 53664.385318: probe_perf:__auxtrace_mmap__read: (aaaae2d036fc)
    head_off=0x21fef0 old_off=0x0 size=0x21fef0
          aaaae2d036fc __auxtrace_mmap__read+0x1e8 (/mnt/linux-kernel/tools/perf/perf)
          aaaae2d03a44 auxtrace_mmap__read+0x48 (/mnt/linux-kernel/tools/perf/perf)
          aaaae2b141bc record__auxtrace_mmap_read+0x40 (/mnt/linux-kernel/tools/perf/
              perf)
          aaaae2b16e98 record__mmap_read_evlist+0x274 (/mnt/linux-kernel/tools/perf/
              perf)
          aaaae2b16fb4 record__mmap_read_all+0x40 (/mnt/linux-kernel/tools/perf/perf)
          aaaae2b19ff4 __cmd_record+0x9ac (/mnt/linux-kernel/tools/perf/perf)
          aaaae2b1dc0c cmd_record+0xb78 (/mnt/linux-kernel/tools/perf/perf)
          aaaae2c04f90 run_builtin+0x110 (/mnt/linux-kernel/tools/perf/perf)
          aaaae2c0523c handle_internal_command+0xf4 (/mnt/linux-kernel/tools/perf/perf
              )
          aaaae2c053f4 run_argv+0x40 (/mnt/linux-kernel/tools/perf/perf)
          aaaae2c0570c main+0x248 (/mnt/linux-kernel/tools/perf/perf)
          ffff9d4aed24 __libc_start_main+0xe4 (/usr/lib/aarch64-linux-gnu/libc-2.28.so
              )
          aaaae2afd7c4 _start+0x34 (/mnt/linux-kernel/tools/perf/perf)
```

When you've read to this point, I hope you now have a brief understanding of how to use perf for tracing and debugging, and you might be interested in exploring more with perf, but I'll stop here ;)

Acknowledgement

I am grateful for the chance to write this article, using a small case to chain up several important Linux debugging tools.

I believe many words in this article are borrowed from Daniel Thompson during the time when I worked on the Linaro training project. Without Daniel's guidance, I might never have had the chance to establish a systemic view for Linux kernel debugging. I hope to record some of the ideas and insights I learned in my conversations with Daniel.

I also appreciate Mathieu Poirier for providing help when I submitted my first patch to Arm CoreSight. Mathieu encouraged me to keep going and explore more development in integration Arm CoreSight with perf. This led me to have much fun in perf!