Using ftrace

ftrace is a debugging tool for understanding what is going on inside the Linux kernel. The following sections detail basic ftrace functionality, ftrace usage with atrace (which captures kernel events), and dynamic ftrace.

For details on advanced ftrace functionality that is not available from systrace, refer to the ftrace documentation at **<kernel tree>/Documentation/trace/ftrace.txt** (https://www.kernel.org/doc/Documentation/trace/ftrace.txt).

Capturing kernel events with atrace

atrace (**frameworks/native/cmds/atrace**) uses ftrace to capture kernel events. In turn, systrace.py (or run_systrace.py in later versions of Catapult (https://github.com/catapult-project/catapult)) uses adb to run atrace on the device. atrace does the following:

- Sets up user-mode tracing by setting a property (**debug.atrace.tags.enableflags**).
- Enables the desired ftrace functionality by writing to the appropriate ftrace sysfs nodes. However, as ftrace supports more features, you might set some sysfs nodes yourself then use atrace.

With the exception of boot-time tracing, rely on using atrace to set the property to the appropriate value. The property is a bitmask and there's no good way to determine the correct values other than looking at the appropriate header (which could change between Android releases).

Enabling ftrace events

The ftrace sysfs nodes are in **/d/tracing** and trace events are divided into categories in **/d/tracing/events**.

To enable events on a per-category basis, use:

```
$ echo 1 > /d/tracing/events/irq/enable
```

To enable events on per-event basis, use:

```
$ echo 1 > /d/tracing/events/sched/sched_wakeup/enable
```

If extra events have been enabled by writing to sysfs nodes, they will **not** be reset by atrace. A common pattern for Qualcomm device bringup is to enable **kgsl** (GPU) and **mdss** (display pipeline) tracepoints and then use atrace or systrace (https://source.android.com/devices/tech/debug/systrace.html):

```
$ adb shell "echo 1 > /d/tracing/events/mdss/enable"
$ adb shell "echo 1 > /d/tracing/events/kgsl/enable"
$ ./systrace.py sched freq idle am wm gfx view binder_driver irq workq ss sync -t 10 -b 96000 -o full_trace.html
```

You can also use ftrace without atrace or systrace, which is useful when you want kernel-only traces (or if you've taken the time to write the user-mode tracing property by hand). To run just ftrace:

1. Set the buffer size to a value large enough for your trace:

   ```
   $ echo 96000 > /d/tracing/buffer_size_kb
   ```

2. Enable tracing:

   ```
   $ echo 1 > /d/tracing/tracing_on
   ```

3. Run your test, then disable tracing:

   ```
   $ echo 0 > /d/tracing/tracing_on
   ```

4. Dump the trace:

   ```
   $ cat /d/tracing/trace > /data/local/tmp/trace_output
   ```

The trace_output gives the trace in text form. To visualize it using Catapult, get the Catapult repository (https://github.com/catapult-project/catapult/tree/master/) from Github and run trace2html:

```
$ catapult/tracing/bin/trace2html ~/path/to/trace_file
```

By default, this writes **`trace_file.html`** in the same directory.

Correlating events

It is often useful to look at the Catapult visualization and the ftrace log simultaneously; for example, some ftrace events (especially vendor-specific ones) are not visualized by Catapult. However, Catapult's timestamps are relative either to the first event in the trace or to a specific timestamp dumped by atrace, while the raw ftrace timestamps are based on a particular absolute clock source in the Linux kernel.

To find a given ftrace event from a Catapult event:

1. Open the raw ftrace log. Traces in recent versions of systrace are compressed by default:
    - If you captured your systrace with **`--no-compress`**, this is in the html file in the section beginning with BEGIN TRACE.
    - If not, run html2trace from the Catapult tree (https://github.com/catapult-project/catapult/tree/master/) (**`tracing/bin/html2trace`**) to uncompress the trace.
2. Find the relative timestamp in the Catapult visualization.
3. Find a line at the beginning of the trace containing **`tracing_mark_sync`**. It should look something like this:

    ```
    <5134>-5134  (-----) [003] ...1    68.104349: tracing_mark_write: trace_event_clock_sync: parent_ts=68.104286
    ```

    If this line does not exist (or if you used ftrace without atrace), then timings will be relative from the first event in the ftrace log.
    a. Add the relative timestamp (in milliseconds) to the value in **`parent_ts`** (in seconds).
    b. Search for the new timestamp.

These steps should put you at (or at least very close to) the event.

Using dynamic ftrace

When systrace and standard ftrace are insufficient, there is one last recourse available: *dynamic ftrace*. Dynamic ftrace involves rewriting of kernel code after boot, and as a result it is not available in production kernels for security reasons. However, every single difficult performance bug in 2015 and 2016 was ultimately root-caused using dynamic ftrace. It is especially powerful for debugging uninterruptible sleeps because you can get a stack trace in the kernel every time you hit the function triggering uninterruptible sleep. You can also debug sections with interrupts and preemptions disabled, which can be very useful for proving issues.

To turn on dynamic ftrace, edit your kernel's defconfig:

1. Remove CONFIG_STRICT_MEMORY_RWX (if it's present). If you're on 3.18 or newer and arm64, it's not there.
2. Add the following: CONFIG_DYNAMIC_FTRACE=y, CONFIG_FUNCTION_TRACER=y, CONFIG_IRQSOFF_TRACER=y, CONFIG_FUNCTION_PROFILER=y, and CONFIG_PREEMPT_TRACER=y
3. Rebuild and boot the new kernel.
4. Run the following to check for available tracers:

    ```
    $ cat /d/tracing/available_tracers
    ```

5. Confirm the command returns **`function`**, **`irqsoff`**, **`preemptoff`**, and **`preemptirqsoff`**.
6. Run the following to ensure dynamic ftrace is working:

    ```
    $ cat /d/tracing/available_filter_functions | grep <a function you care about>
    ```

After completing these steps, you have dynamic ftrace, the function profiler, the irqsoff profiler, and the preemptoff profiler available. We **strongly recommend** reading ftrace documentation on these topics before using them as they are powerful but complex. irqsoff and preemptoff are primarily useful for confirming that drivers may be leaving interrupts or preemption turned off for too long.

The function profiler is the best option for performance issues and is often used to find out where a function is being called.

Show Issue: HDR photo + rotating viewfinder

In this issue, using a Pixel XL to take an HDR+ photo then immediately rotating the viewfinder caused jank every time. We used the function profiler to debug the issue in less than one hour. To follow along with the example, download the zip file (https://source.android.com/devices/tech/debug/perf_traces.zip) of traces (which also includes other traces referred to in this section), unzip the file, and open the trace_30898724.html file in your browser.

The trace shows several threads in the cameraserver process blocked in uninterruptible sleep on **`ion_client_destroy`**. That's an expensive function, but it should be called very infrequently because ion clients should encompass many allocations. Initially, the blame fell on the Hexagon code in Halide, which was indeed one of the culprits (it created a new client for every ion allocation and destroyed that client when the allocation was freed, which was way too expensive). Moving to a single ion client for all Hexagon allocations improved the situation, but the jank wasn't fixed.

At this point we need to know who is calling **`ion_client_destroy`**, so it's time to use the function profiler:

1. As functions are sometimes renamed by the compiler, confirm **`ion_client_destroy`** is there by using:

   ```
   $ cat /d/tracing/available_filter_functions | grep ion_client_destroy
   ```

2. After confirming it is there, use it as the ftrace filter:

   ```
   $ echo ion_client_destroy > /d/tracing/set_ftrace_filter
   ```

3. Turn on the function profiler:

   ```
   $ echo function > /d/tracing/current_tracer
   ```

4. Turn on stack traces whenever a filter function is called:

   ```
   $ echo func_stack_trace > /d/tracing/trace_options
   ```

5. Increase the buffer size:

   ```
   $ echo 64000 > /d/tracing/buffer_size_kb
   ```

6. Turn on tracing:

   ```
   $ echo 1 > /d/tracing/trace_on
   ```

7. Run the test and get the trace:

   ```
   $ cat /d/tracing/trace > /data/local/tmp/trace
   ```

8. View the trace to see lots and lots of stack traces:

   ```
      cameraserver-643   [003] ...1    94.192991: ion_client_destroy <-ion_release
      cameraserver-643   [003] ...1    94.192997: <stack trace>
    => ftrace_ops_no_ops
    => ftrace_graph_call
    => ion_client_destroy
    => ion_release
    => __fput
    => ____fput
    => task_work_run
    => do_notify_resume
    => work_pending
   ```

Based on inspection of the ion driver, we can see that **`ion_client_destroy`** is being spammed by a userspace function closing an fd to **`/dev/ion`**, not a random kernel driver. By searching the Android codebase for **`\"/dev/ion\"`**, we find several vendor drivers doing the same thing as the Hexagon driver and opening/closing **`/dev/ion`** (creating and destroying a new ion client) every time they need a new ion allocation. Changing those to use a single ion client (https://android.googlesource.com/platform/hardware/qcom/camera/+/8f7984018b6643f430c229725a58d3c6bb04acab) for the lifetime of the process

fixed the bug.

If the data from function profiler isn't specific enough, you can combine ftrace tracepoints with the function profiler. ftrace events can be enabled in exactly the same way as usual, and they will be interleaved with your trace. This is great if there's an occasional long uninterruptible sleep in a specific function you want to debug: set the ftrace filter to the function you want, enable tracepoints, take a trace. You can parse the resulting trace with `trace2html`, find the event you want, then get nearby stack traces in the raw trace.

Using lockstat

Sometimes, ftrace isn't enough and you really need to debug what appears to be kernel lock contention. There is one more kernel option worth trying: `CONFIG_LOCK_STAT`. This is a last resort as it is extremely difficult to get working on Android devices because it inflates the size of the kernel beyond what most devices can handle.

However, lockstat uses the debug locking infrastructure, which is useful for many other applications. Everyone working on device bringup should figure out some way to get that option working on every device because there **will** be a time when you think "If only I could turn on `LOCK_STAT`, I could confirm or refute this as the problem in five minutes instead of five days."

Show Issue: Stall in SCHED_FIFO when cores at max load with non-SCHED_FIFO

In this issue, the SCHED_FIFO thread stalled when all cores were at maximum load with non-SCHED_FIFO threads. We had traces showing significant lock contention on an fd in VR apps, but we couldn't easily identify the fd in use. To follow along with the example, download the zip file (https://source.android.com/devices/tech/debug/perf_traces.zip) of traces (which also includes other traces referred to in this section), unzip the file, and open the trace_30905547.html file in your browser.

We hypothesized that ftrace itself was the source of lock contention, when a low priority thread would start writing to the ftrace pipe and then get preempted before it could release the lock. This is a worst-case scenario that was exacerbated by a mixture of extremely low-priority threads writing to the ftrace marker along with some higher priority threads spinning on CPUs to simulate a completely loaded device.

As we couldn't use ftrace to debug, we got `LOCK_STAT` working then turned off all other tracing from the app. The results showed the lock contention was actually from ftrace because none of the contention showed up in the lock trace when ftrace was not running.

If you can boot a kernel with the config option, lock tracing is similar to ftrace:

1. Enable tracing:

   ```
   $ echo 1 > /proc/sys/kernel/lock_stat
   ```

2. Run your test.
3. Disable tracing:

   ```
   $ echo 0 > /proc/sys/kernel/lock_stat
   ```

4. Dump your trace:

   ```
   $ cat /proc/lock_stat > /data/local/tmp/lock_stat
   ```

For help interpreting the resulting output, refer to lockstat documentation at **`<kernel>/Documentation/locking/lockstat.txt`** (https://www.kernel.org/doc/Documentation/locking/lockstat.txt).
