Understanding Systrace

Caution: If you've never used systrace before, we strongly recommend reading the systrace overview (https://developer.android.com/studio/profile/systrace.html) before continuing.

systrace is the primary tool for analyzing Android device performance. However, it's really a wrapper around other tools: It is the host-side wrapper around *atrace*, the device-side executable that controls userspace tracing and sets up *ftrace*, the primary tracing mechanism in the Linux kernel. systrace uses atrace to enable tracing, then reads the ftrace buffer and wraps it all in a self-contained HTML viewer. (While newer kernels have support for Linux Enhanced Berkeley Packet Filter (eBPF), the following documentation pertains to the 3.18 kernel (no eFPF) as that's what was used on the Pixel/Pixel XL.)

systrace is owned by the Google Android and Google Chrome teams and is developed in the open as part of the Catapult project (https://github.com/catapult-project/catapult). In addition to systrace, Catapult includes other useful utilities. For example, ftrace has more features than can be directly enabled by systrace or atrace and contains some advanced functionality that is critical to debugging performance problems. (These features require root access and often a new kernel.)

Running systrace

When debugging jitter on Pixel/Pixel XL, start with the following command:

```
$ ./systrace.py sched freq idle am wm gfx view sync binder_driver irq workq input -b 96000
```

When combined with the additional tracepoints required for GPU and display pipeline activity, this gives you the ability to trace from user input to frame displayed on screen. Set the buffer size to something large to avoid losing events (which usually manifests as some CPUs containing no events after some point in the trace).

When going through systrace, keep in mind that **every event is triggered by something on the CPU**.

Note: Hardware interrupts are not controlled by the CPU and do trigger things in ftrace, but the actual commit to the trace log is done by the interrupt handler, which could have been delayed if your interrupt arrived while (for example) some other bad driver had interrupts disabled. The critical element is the CPU.

Because systrace is built on top of ftrace and ftrace runs on the CPU, something on the CPU must write the ftrace buffer that logs hardware changes. This means that if you're curious about why a display fence changed state, you can see what was running on the CPU at the exact point of its transition (something that is running on the CPU triggered that change in the log). This concept is the foundation of analyzing performance using systrace.

Example: Working frame

This example describes a systrace for a normal UI pipeline. 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), upzip the file, and open the systrace_tutorial.html file in your browser. Be warned this systrace is a large file; unless you use systrace in your day-to-day work, this is likely a much bigger trace with much more information than you've ever seen in a single trace before.

For a consistent, periodic workload such as TouchLatency, the UI pipeline contains the following:

1. EventThread in SurfaceFlinger wakes the app UI thread, signaling it's time to render a new frame.
2. App renders frame in UI thread, RenderThread, and hwuiTasks, using CPU and GPU resources. This is the bulk of the capacity spent for UI.
3. App sends rendered frame to SurfaceFlinger via binder and goes to sleep.
4. A second EventThread in SurfaceFlinger wakes SurfaceFlinger to trigger composition and display output. If SurfaceFlinger determines there is no work to be done, it goes back to sleep.
5. SurfaceFlinger handles composition via HWC/HWC2 or GL. HWC/HWC2 composition is faster and lower power but has limitations depending on the SOC. This usually takes ~4-6ms, but can overlap with step 2 because Android applications are always triple-buffered. (While applications are always triple-buffered, there may only be one pending frame waiting in SurfaceFlinger, which makes it appear identical to double-buffering.)
6. SurfaceFlinger dispatches final output to display via vendor driver and goes back to sleep, waiting for EventThread wakeup.

Let's walk through the frame beginning at 15409ms:

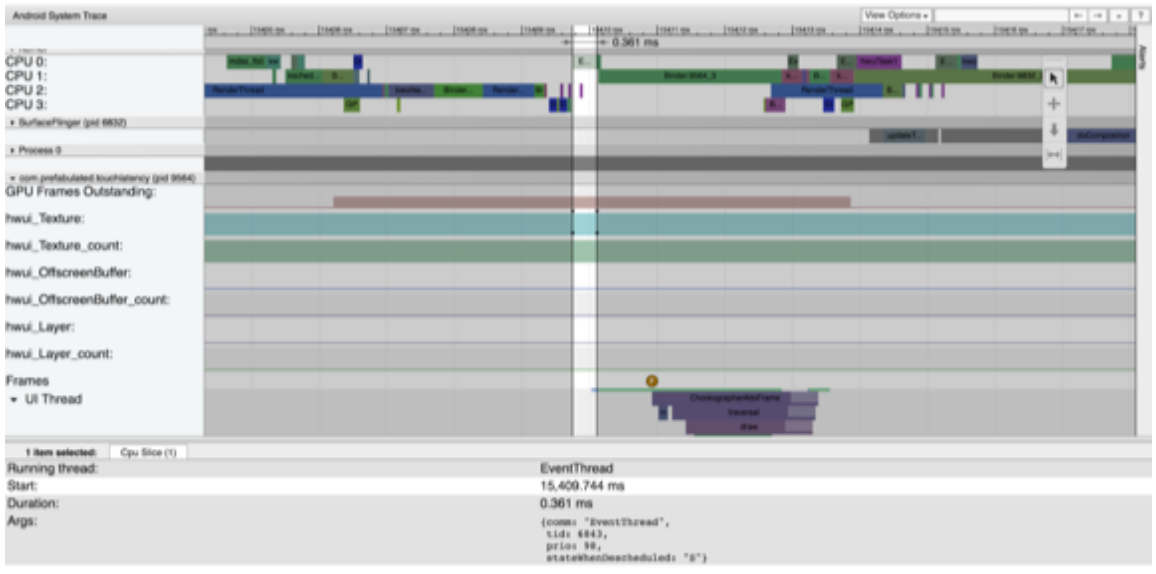

Figure 1. Normal UI pipeline, EventThread running.

Figure 1 is a normal frame surrounded by normal frames, so it's a good starting point for understanding how the UI pipeline works. The UI thread row for TouchLatency includes different colors at different times. Bars denote different states for the thread:

- **Gray**. Sleeping.
- **Blue**. Runnable (it could run, but the scheduler hasn't picked it to run yet).
- **Green**. Actively running (the scheduler thinks it's running).

★ **Note**: Interrupt handlers aren't shown in the normal per-CPU timeline, so it's possible that you're actually running interrupt handlers or softirqs during some portion of a thread's runtime. Check the irq section of the trace (under process 0) to confirm whether an interrupt is running instead of a standard thread.

- **Red**. Uninterruptible sleep (generally sleeping on a lock in the kernel). Can be indicative of I/O load. Extremely useful for debugging performance issues.
- **Orange**. Uninterruptible sleep due to I/O load.

To view the reason for uninterruptible sleep (available from the `sched_blocked_reason` tracepoint), select the red uninterruptible sleep slice.

While EventThread is running, the UI thread for TouchLatency becomes runnable. To see what woke it, click the blue section:

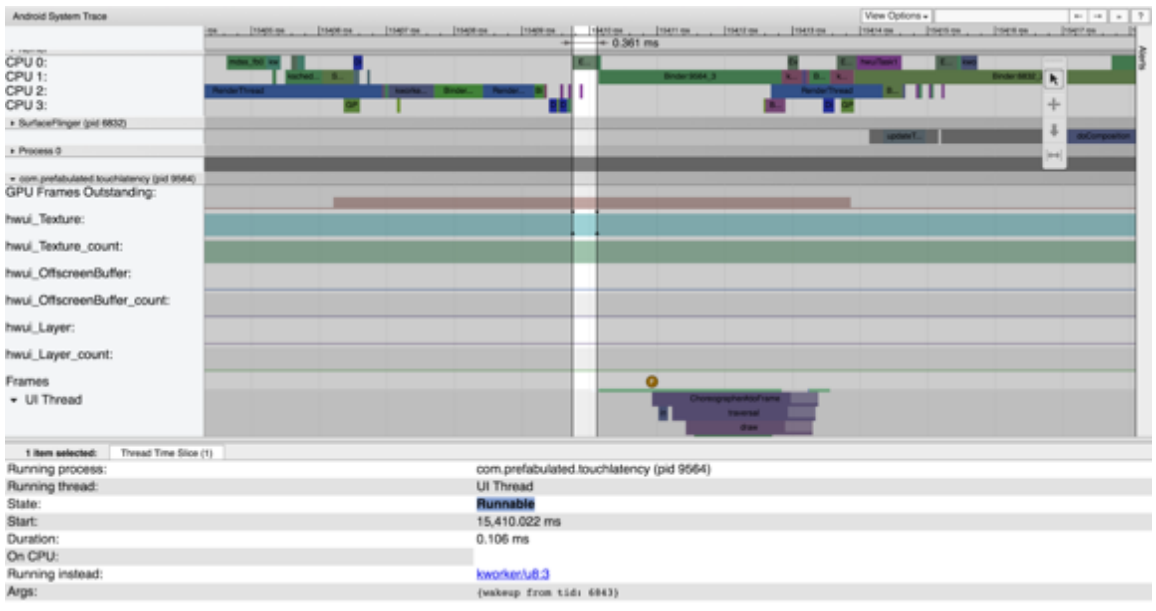

Figure 2. UI thread for TouchLatency.

Figure 2 shows the TouchLatency UI thread was woken by tid 6843, which corresponds to EventThread. The UI thread wakes:

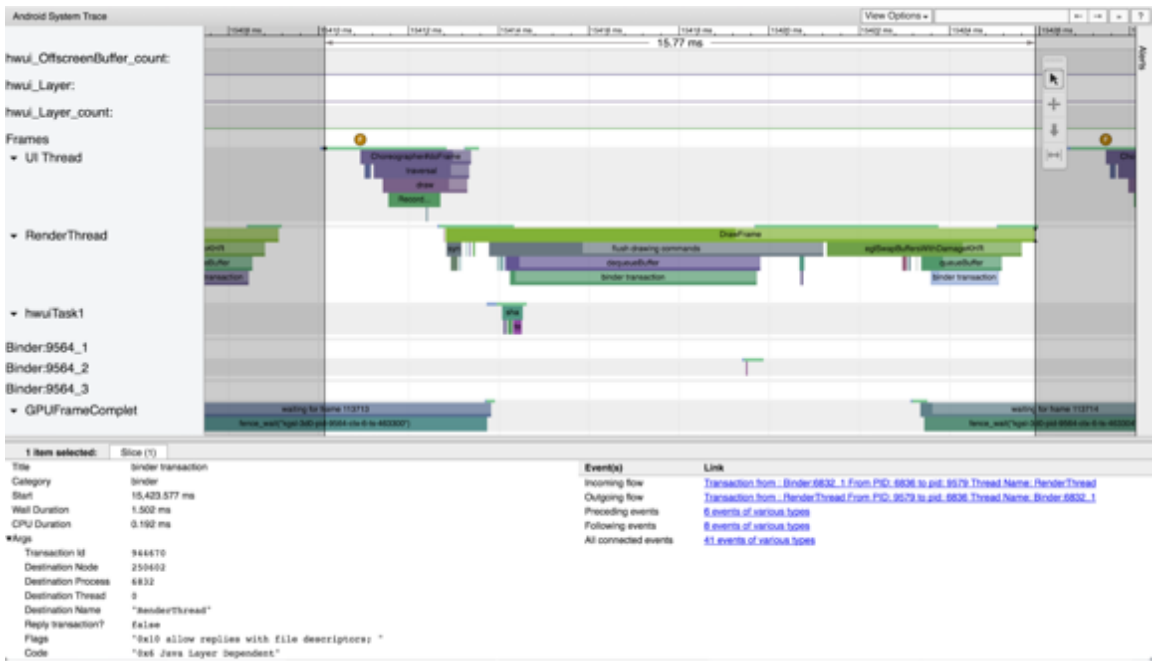

Figure 3. UI thread wakes, renders a frame, and enqueues it for SurfaceFlinger to consume.

If the `binder_driver` tag is enabled in a trace, you can select a binder transaction to view information on all of the processes involved in that transaction:

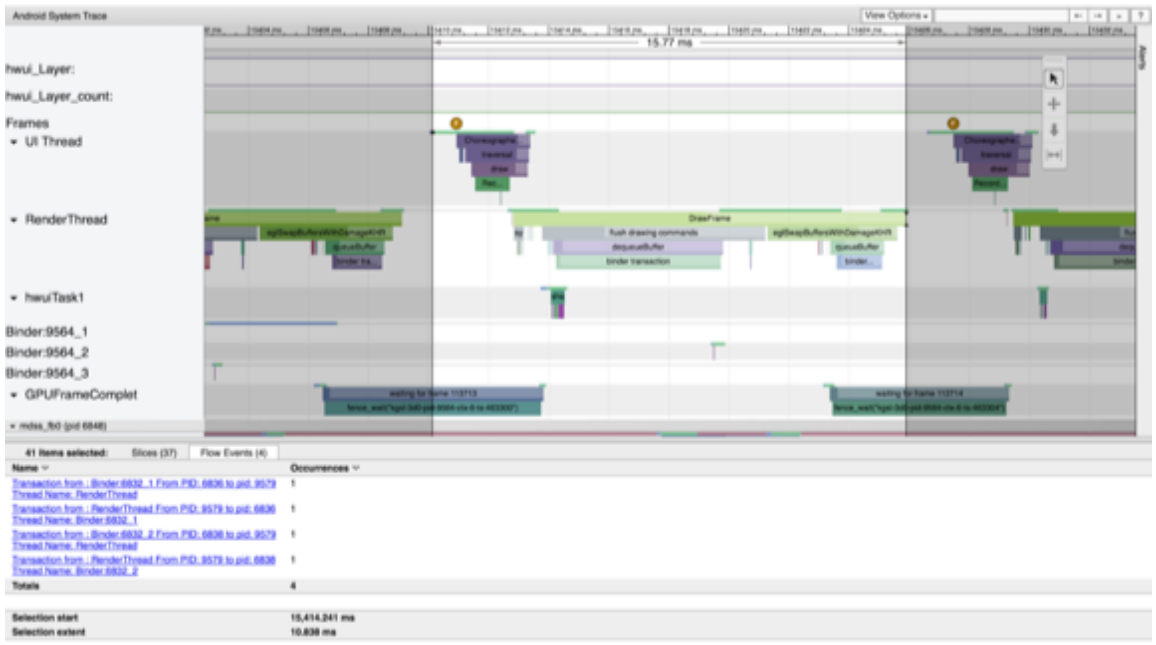

Figure 4. Binder transaction.

Figure 4 shows that, at 15,423.65ms, Binder:6832_1 in SurfaceFlinger becomes runnable because of tid 9579, which is TouchLatency's RenderThread. You can also see queueBuffer on both sides of the binder transaction.

During the queueBuffer on the SurfaceFlinger side, the number of pending frames from TouchLatency goes from 1 to 2:

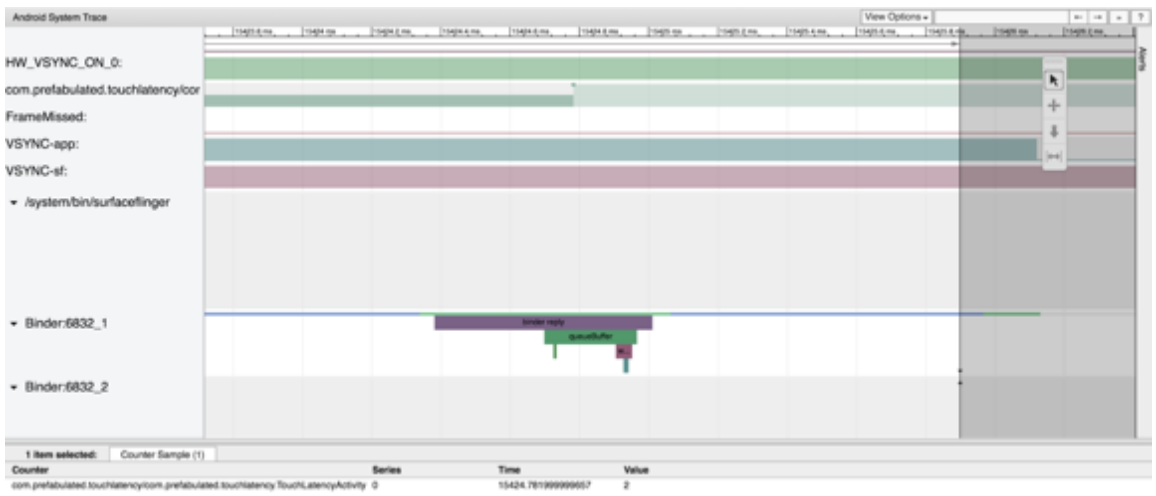

Figure 5. Pending frames goes from 1 to 2.

Figure 5 shows triple-buffering, where there are two completed frames and the app will soon start rendering a third. This is because we've already dropped some frames, so the app keeps two pending frames instead of one to try to avoid further dropped frames.

Soon after, SurfaceFlinger's main thread is woken by a second EventThread so it can output the older pending frame to the display:

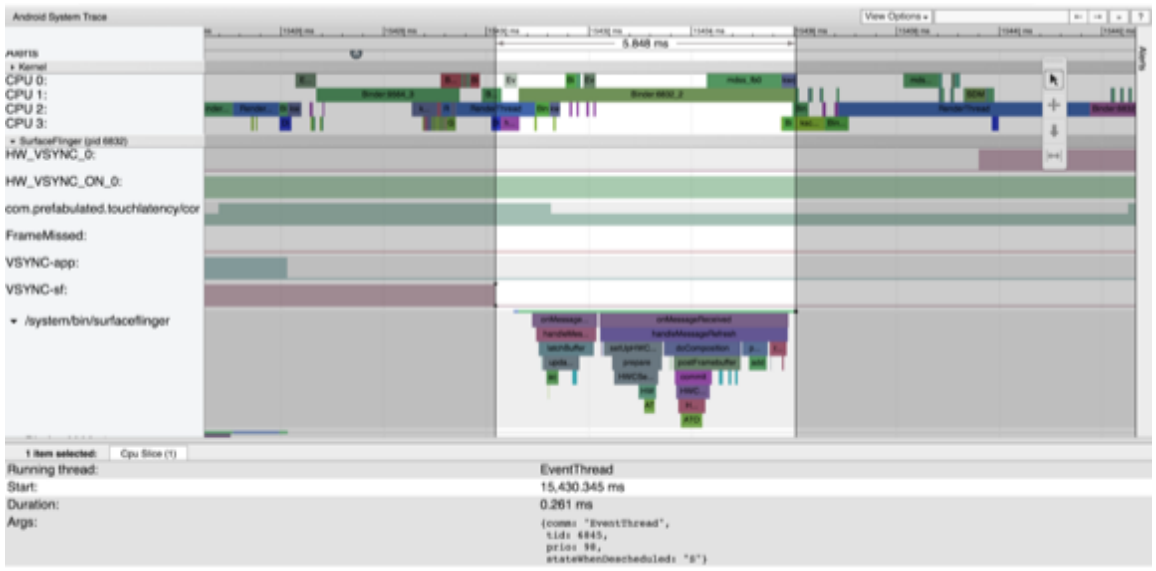

Figure 6. SurfaceFlinger's main thread is woken by a second EventThread.

SurfaceFlinger first latches the older pending buffer, which causes the pending buffer count to decrease from 2 to 1:

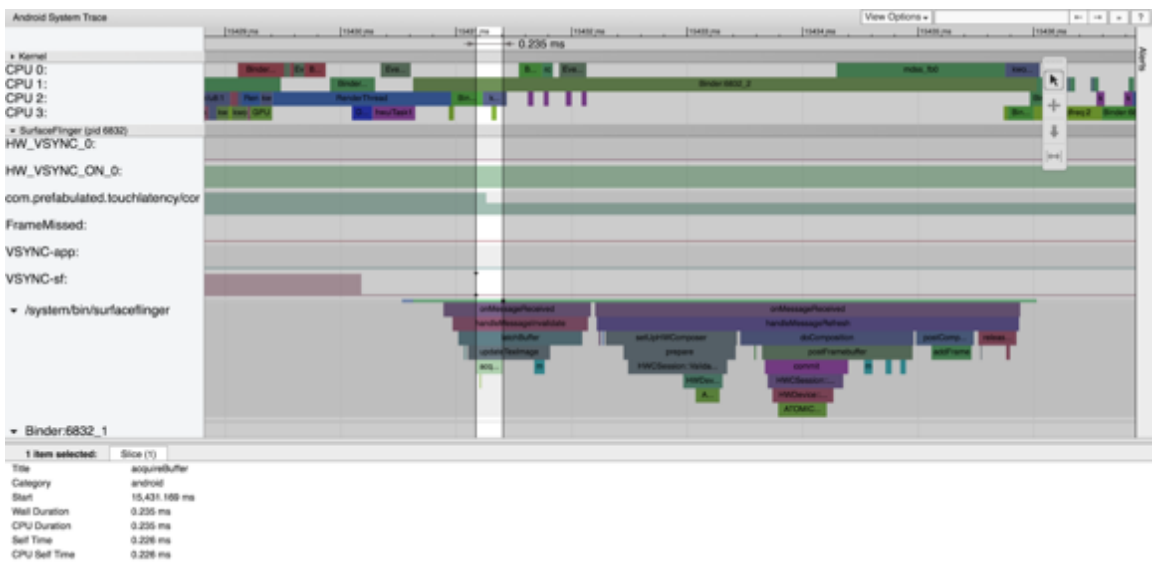

Figure 7. SurfaceFlinger first latches the older pending buffer.

After latching the buffer, SurfaceFlinger sets up composition and submits the final frame to the display. (Some of these sections are enabled as part of the `mdss` tracepoint, so they may not be there on your SOC.)

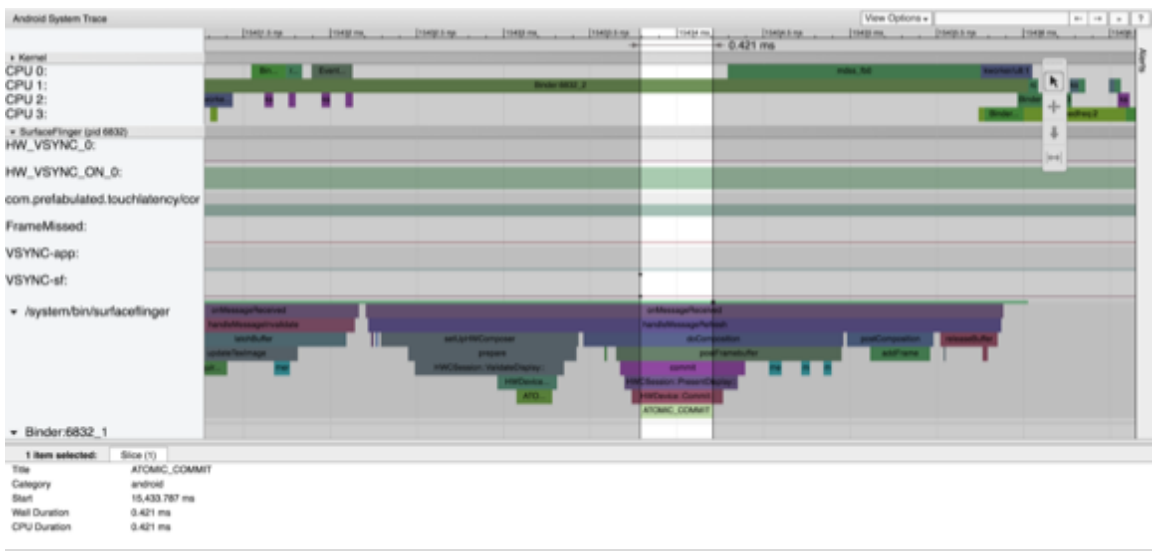

Figure 8. SurfaceFlinger sets up composition and submits the final frame.

Next, `mdss_fb0` wakes on CPU 0. `mdss_fb0` is the display pipeline's kernel thread for outputting a rendered frame to the display. We can see `mdss_fb0` as its own row in the trace (scroll down to view).

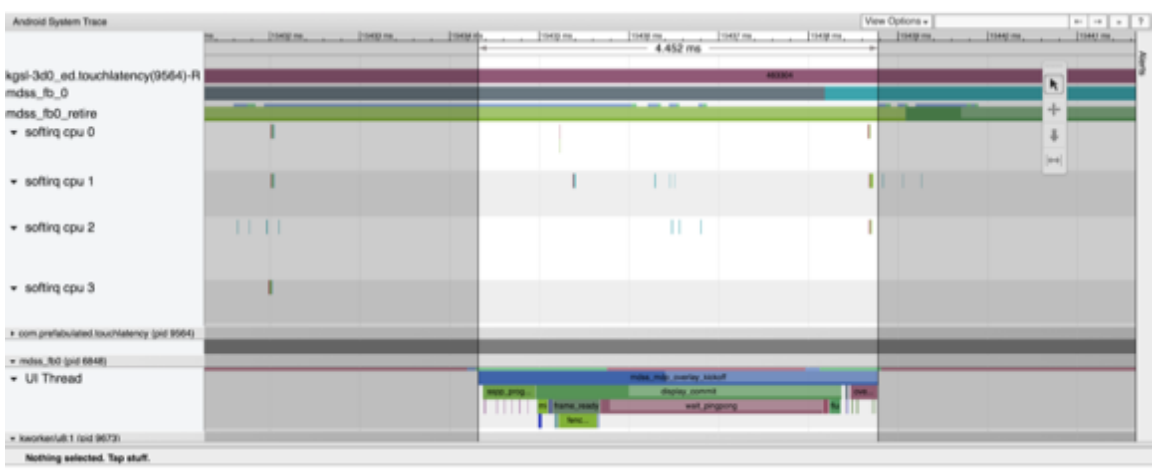

Figure 9. `mdss_fb0` wakes on CPU 0.

`mdss_fb0` wakes up, runs for a bit, enters uninterruptible sleep, then wakes again.

Note: From this point forward, the trace is more complicated as the final work is split between `mdss_fb0`, interrupts, and workqueue functions. If you need that level of detail, refer to the exact characteristics of the driver stack for your SOC (as what happens on the Pixel XL might not be useful to you).

Example: Non-working frame

This example describes a systrace used to debug Pixel/Pixel XL jitter. 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), upzip the file, and open the systrace_tutorial.html file in your browser.

When you first open the systrace, you'll see something like this:

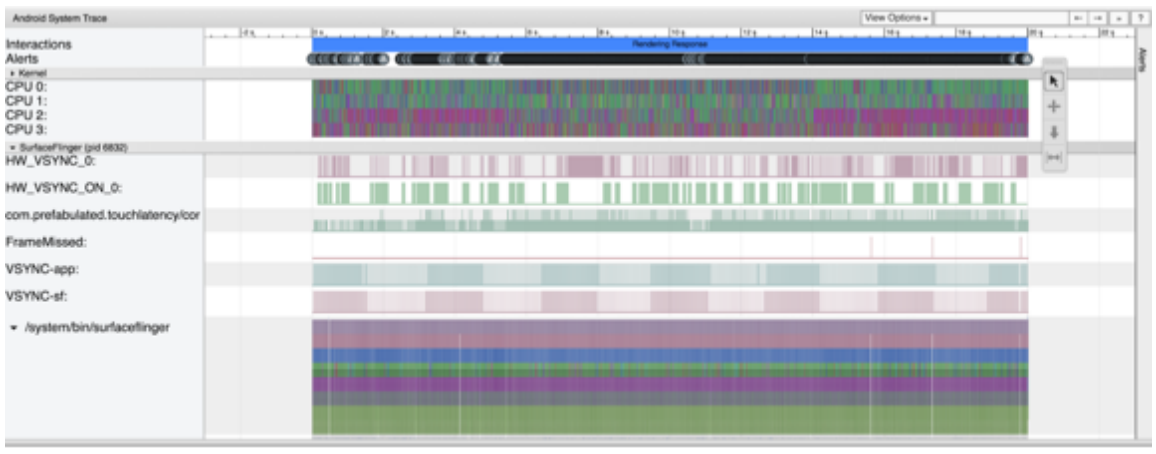

Figure 10. TouchLatency running on Pixel XL (most options enabled, including mdss and kgsl tracepoints).

When looking for jank, check the FrameMissed row under SurfaceFlinger. FrameMissed is a quality-of-life improvement provided by Hardware Composer 2 (HWC2). As of December 2016, HWC2 is used only on Pixel/Pixel XL; when viewing systrace for other devices, the FrameMissed row may not be present. In either case, FrameMissed is correlated with SurfaceFlinger missing one of its extremely-regular runtimes and an unchanged pending-buffer count for the app (`com.prefabulated.touchlatency`) at a vsync:

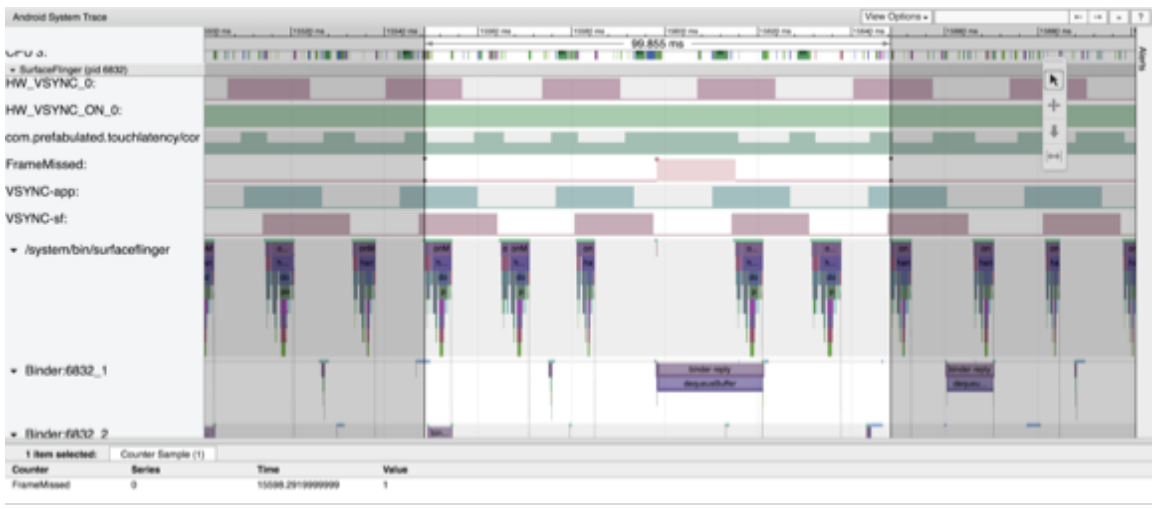

Figure 11. FrameMissed correlation with SurfaceFlinger.

Figure 11 shows a missed frame at 15598.29ms. SurfaceFlinger woke briefly at the vsync interval and went back to sleep without doing any work, which means SurfaceFlinger determined it was not worth trying to send a frame to the display again. Why?

To understand how the pipeline broke down for this frame, first review the working frame example above to see how a normal UI pipeline appears in systrace. When ready, return to the missed frame and work backwards. Notice that SurfaceFlinger wakes and immediately goes to sleep. When viewing the number of pending frames from TouchLatency, there are two frames (a good clue to help figure out what's going on).

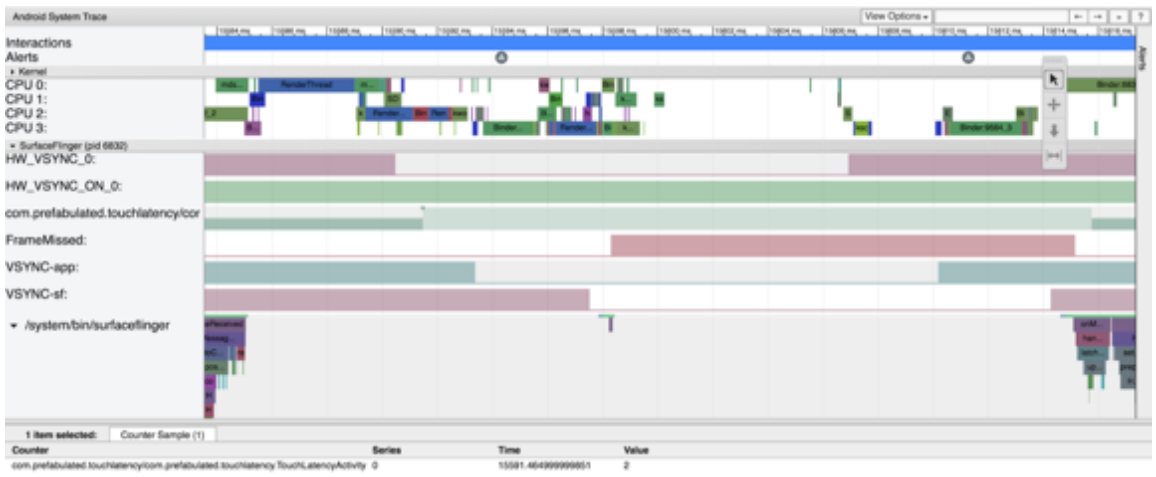

Figure 12. SurfaceFlinger wakes and immediately goes to sleep.

Because we have frames in SurfaceFlinger, it's not an app issue. In addition, SurfaceFlinger is waking at the correct time, so it's not a SurfaceFlinger issue. If SurfaceFlinger and the app are both looking normal, it's probably a driver issue.

Because the `mdss` and `sync` tracepoints are enabled, we can get information about the fences (shared between the display driver and SurfaceFlinger) that control when frames are actually submitted to the display. The fences we care about are listed under `mdss_fb0_retire`, which denotes when a frame is actually on the display. These fences are provided as part of the `sync` trace category. Which fences correspond to particular events in SurfaceFlinger depends on your SOC and driver stack, so work with your SOC vendor to understand the meaning of fence categories in your traces.

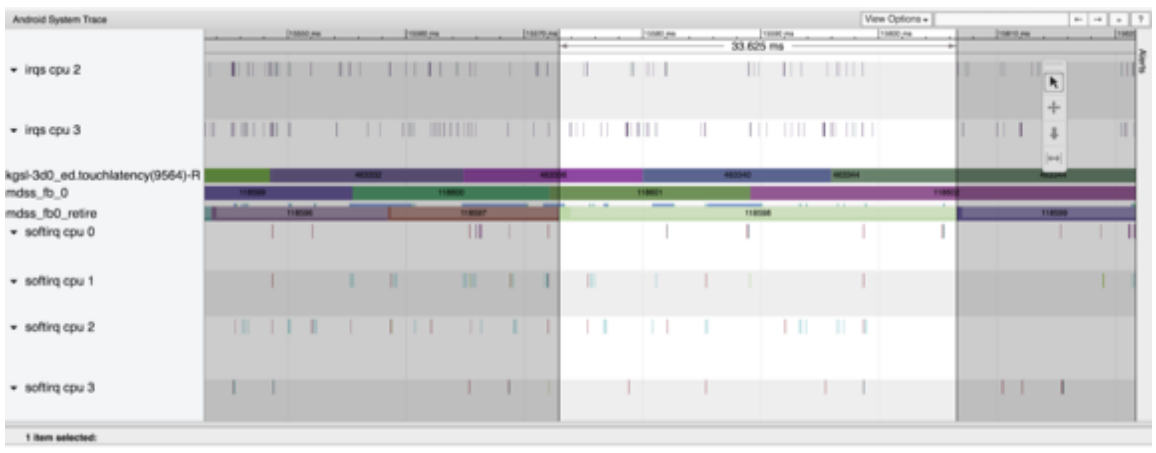

Figure 13. `mdss_fb0_retire` fences.

Figure 13 shows a frame that was displayed for 33ms, not 16.7ms as expected. Halfway through that slice, that frame should have been replaced by a new one but wasn't. View the previous frame and look for anything interesting:

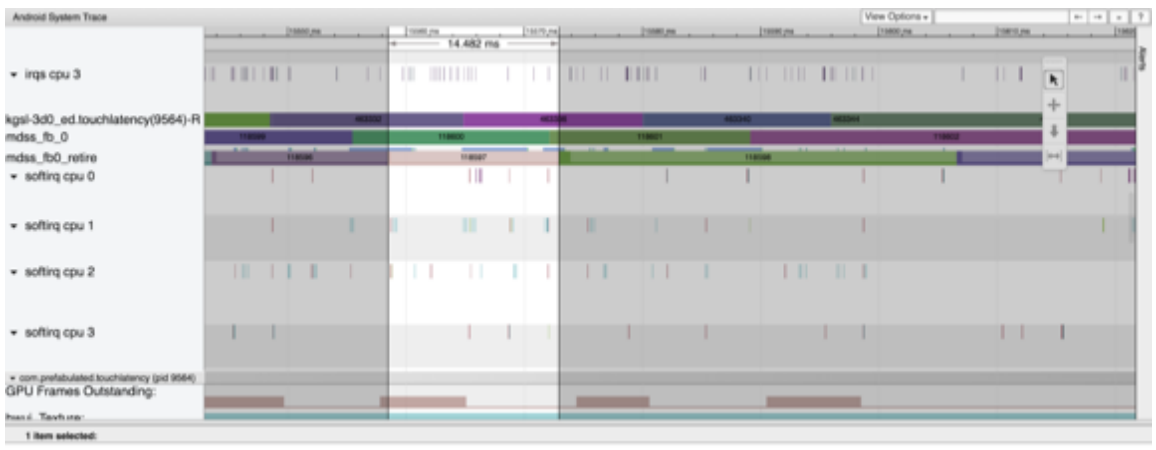

Figure 14. Frame previous to busted frame.

Figure 14 shows 14.482ms a frame. The busted two-frame segment was 33.6ms, which is roughly what we would expect for two frames (we render at 60Hz, 16.7ms a frame, which is close). But 14.482ms is not anywhere close to 16.7ms, which suggests that something is very wrong with the display pipe.

Investigate exactly where that fence ends to determine what controls it:

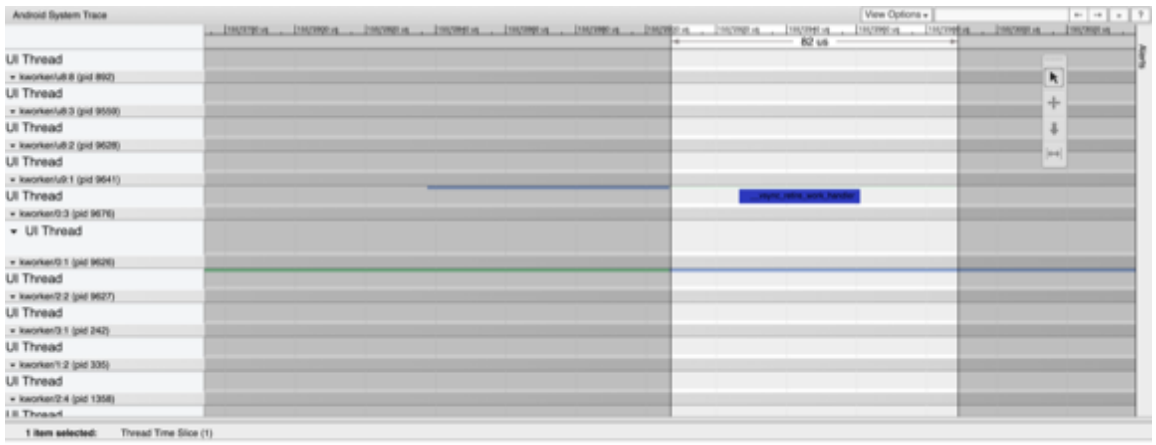

Figure 15. Investigate fence end.

A workqueue contains a `__vsync_retire_work_handler` that is running when the fence changes. Looking through the kernel source, you can see it's part of the display driver. It definitely appears to be on the critical path for the display pipeline, so it must run as quickly as possible. It's runnable for 70us or so (not a long scheduling delay), but it's a workqueue and might not get scheduled accurately.

Check the previous frame to determine if that contributed; sometimes jitter can add up over time and eventually cause us to miss a deadline.

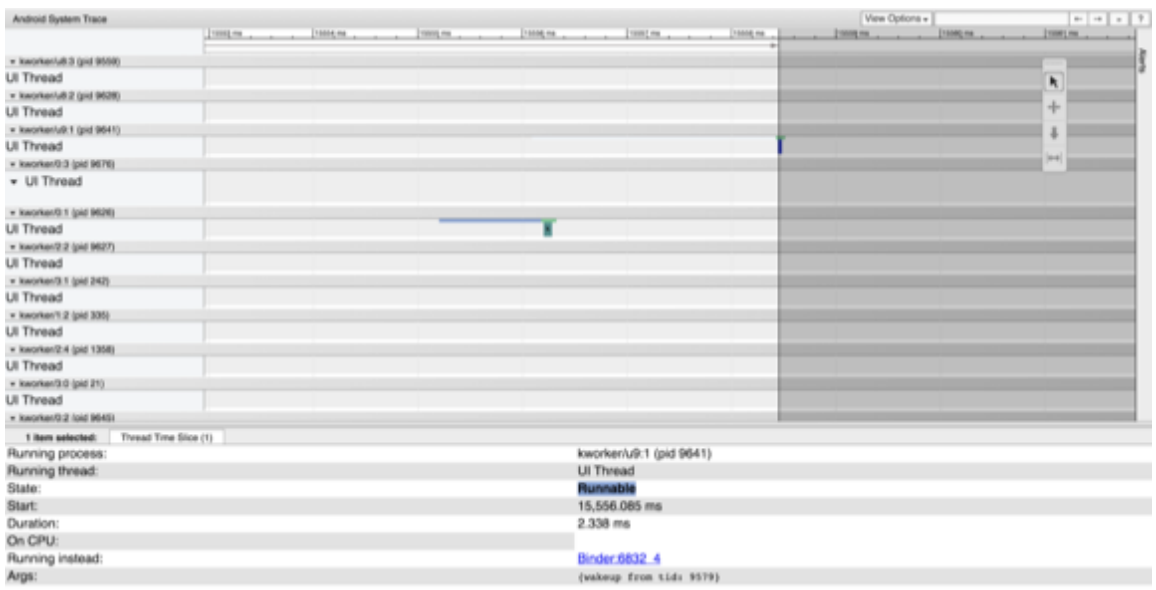

Figure 16. Previous frame.

The runnable line on the kworker isn't visible because the viewer turns it white when it's selected, but the statistics tell the story: 2.3ms of scheduler delay for part of the display pipeline critical path is **bad**. Before we do anything else, we should fix that by moving this part of the display pipeline critical path from a workqueue (which runs as a `SCHED_OTHER` CFS thread) to a dedicated `SCHED_FIFO` kthread. This function needs timing guarantees that workqueues can't (and aren't intended to) provide.

Is this the reason for the jank? It's hard to say conclusively. Outside of easy-to-diagnose cases such as kernel lock contention causing display-critical threads to sleep, traces usually won't tell you directly what the problem is. Could this jitter have been the cause of the dropped frame? Absolutely. The fence times should be 16.7ms, but they aren't close to that at all in the frames leading up to the dropped frame. (There's a 19ms fence followed by a 14ms fence.) Given how tightly coupled the display pipeline is, it's entirely possible the jitter around fence timings resulted in an eventual dropped frame.

In this example, the solution involved converting `__vsync_retire_work_handler` from a workqueue to a dedicated kthread. This resulted in noticeable jitter improvements and reduced jank in the bouncing ball test. Subsequent traces show fence timings that hover very close to 16.7ms.

Last updated April 26, 2017.