Evaluating Performance

There are two user-visible indicators of performance:

- **Predictable, perceptible performance**. Does the user interface (UI) drop frames or consistently render at 60FPS? Does audio play without artifacts or popping? How long is the delay between the user touching the screen and the effect showing on the display?
- **Length of time required for longer operations** (such as opening applications).

The first is more noticeable than the second. Users typically notice jank but they won't be able to tell 500ms vs 600ms application startup time unless they are looking at two devices side-by-side. Touch latency is immediately noticeable and significantly contributes to the perception of a device.

As a result, in a fast device, the UI pipeline is the most important thing in the system other than what is necessary to keep the UI pipeline functional. This means that the UI pipeline should preempt any other work that is not necessary for fluid UI. To maintain a fluid UI, background syncing, notification delivery, and similar work must all be delayed if UI work can be run. It is acceptable to trade the performance of longer operations (HDR+ runtime, application startup, etc.) to maintain a fluid UI.

Capacity vs jitter

When considering device performance, *capacity* and *jitter* are two meaningful metrics.

Capacity

Capacity is the total amount of some resource that the device possesses over some amount of time. This can be CPU resources, GPU resources, I/O resources, network resources, memory bandwidth, or any similar metric. When examining whole-system performance, it can be useful to abstract the individual components and assume a single metric that determines performance (especially when tuning a new device because the workloads run on that device are likely fixed).

The capacity of a system varies based on the computing resources online. Changing CPU/GPU frequency is the primary means of changing capacity, but there are others such as changing the number of CPU cores online. Accordingly, the capacity of a system corresponds with power consumption; **changing capacity always results in a similar change in power consumption.**

The capacity required at a given time is overwhelmingly determined by the running application. As a result, the platform can do little to adjust the capacity required for a given workload, and the means to do so are limited to runtime improvements (Android framework, ART, Bionic, GPU compiler/drivers, kernel).

Jitter

While the required capacity for a workload is easy to see, jitter is a more nebulous concept. For a good introduction to jitter as an impediment to fast systems, refer to *THE CASE OF THE MISSING SUPERCOMPUTER PERFORMANCE: ACHIEVING OPTIMAL PERFORMANCE ON THE 8,192 PROCESSORS OF ASCI Q* (http://permalink.lanl.gov/object/tr?what=info:lanl-repo/lareport/LA-UR-03-3116). (It's an investigation of why the ASCI Q supercomputer did not achieve its expected performance and is a great introduction to optimizing large systems.)

This page uses the term jitter to describe what the ASCI Q paper calls *noise*. Jitter is the random system behavior that prevents perceptible work from running. It is often work that must be run, but it may not have strict timing requirements that cause it to run at any particular time. Because it is random, it is extremely difficult to disprove the existence of jitter for a given workload. It is also extremely difficult to prove that a known source of jitter was the cause of a particular performance issue. The tools most commonly used for diagnosing causes of jitter (such as tracing or logging) can introduce their own jitter.

Sources of jitter experienced in real-world implementations of Android include:

- Scheduler delay
- Interrupt handlers
- Driver code running for too long with preemption or interrupts disabled
- Long-running softirqs
- Lock contention (application, framework, kernel driver, binder lock, mmap lock)
- File descriptor contention where a low-priority thread holds the lock on a file, preventing a high-priority thread from running
- Running UI-critical code in workqueues where it could be delayed

- CPU idle transitions
- Logging
- I/O delays
- Unnecessary process creation (e.g., CONNECTIVITY_CHANGE broadcasts)
- Page cache thrashing caused by insufficient free memory

The required amount of time for a given period of jitter may or may not decrease as capacity increases. For example, if a driver leaves interrupts disabled while waiting for a read from across an i2c bus, it will take a fixed amount of time regardless of whether the CPU is at 384MHz or 2GHz. Increasing capacity is not a feasible solution to improve performance when jitter is involved. As a result, **faster processors will not usually improve performance in jitter-constrained situations.**

Finally, unlike capacity, jitter is almost entirely within the domain of the system vendor.

Memory consumption

Memory consumption is traditionally blamed for poor performance. While consumption itself is not a performance issue, it can cause jitter via lowmemorykiller overhead, service restarts, and page cache thrashing. Reducing memory consumption can avoid the direct causes of poor performance, but there may be other targeted improvements that avoid those causes as well (for example, pinning the framework to prevent it from being paged out when it will be paged in soon after).

Analyzing initial device performance

Starting from a functional but poorly-performing system and attempting to fix the system's behavior by looking at individual cases of user-visible poor performance is **not** a sound strategy. Because poor performance is usually not easily reproducible (i.e., jitter) or an application issue, too many variables in the full system prevent this strategy from being effective. As a result, it's very easy to misidentify causes and make minor improvements while missing systemic opportunities for fixing performance across the system.

Instead, use the following general approach when bringing up a new device:

1. Get the system booting to UI with all drivers running and some basic frequency governor settings (if you change the frequency governor settings, repeat all steps below).
2. Ensure the kernel supports the `sched_blocked_reason` tracepoint as well as other tracepoints in the display pipeline that denote when the frame is delivered to the display.
3. Take long traces of the entire UI pipeline (from receiving input via an IRQ to final scanout) while running a lightweight and consistent workload (e.g., UiBench (https://android.googlesource.com/platform/frameworks/base.git/+/master/tests/UiBench/) or the ball test in TouchLatency) (#touchlatency).
4. Fix the frame drops detected in the lightweight and consistent workload.
5. Repeat steps 3-4 until you can run with zero dropped frames for 20+ seconds at a time.
6. Move on to other user-visible sources of jank.

Other simple things you can do early on in device bringup include:

- Ensure your kernel has the sched_blocked_reason tracepoint patch (https://android.googlesource.com/kernel/msm/+/c9f00aa0e25e397533c198a0fcf6246715f99a7b%5E!/). This tracepoint is enabled with the sched trace category in systrace and provides the function responsible for sleeping when that thread enters uninterruptible sleep. It is critical for performance analysis because uninterruptible sleep is a very common indicator of jitter.
- Ensure you have sufficient tracing for the GPU and display pipelines. On recent Qualcomm SOCs, tracepoints are enabled using:

  ```
  $ adb shell "echo 1 > /d/tracing/events/kgsl/enable"
  $ adb shell "echo 1 > /d/tracing/events/mdss/enable"
  ```

  These events remain enabled when you run systrace so you can see additional information in the trace about the display pipeline (MDSS) in the `mdss_fb0` section. On Qualcomm SOCs, you won't see any additional information about the GPU in the standard systrace view, but the results are present in the trace itself (for details, see Understanding systrace (https://source.android.com/devices/tech/debug/systrace.html)).

  What you want from this kind of display tracing is a single event that directly indicates a frame has been delivered to the display. From there, you can determine if you've hit your frame time successfully; if event X*n* occurs less than 16.7ms after event X*n-1*

(assuming a 60Hz display), then you know you did not jank. If your SOC does not provide such signals, work with your vendor to get them. Debugging jitter is extremely difficult without a definitive signal of frame completion.

Using synthetic benchmarks

Synthetic benchmarks are useful for ensuring a device's basic functionality is present. However, treating benchmarks as a proxy for perceived device performance is not useful.

Based on experiences with SOCs, differences in synthetic benchmark performance between SOCs is not correlated with a similar difference in perceptible UI performance (number of dropped frames, 99th percentile frame time, etc.). Synthetic benchmarks are capacity-only benchmarks; jitter impacts the measured performance of these benchmarks only by stealing time from the bulk operation of the benchmark. As a result, synthetic benchmark scores are mostly irrelevant as a metric of user-perceived performance.

Consider two SOCs running Benchmark X that renders 1000 frames of UI and reports the total rendering time (lower score is better).

- SOC 1 renders each frame of Benchmark X in 10ms and scores 10,000.
- SOC 2 renders 99% of frames in 1ms but 1% of frames in 100ms and scores 19,900, a dramatically better score.

If the benchmark is indicative of actual UI performance, SOC 2 would be unusable. Assuming a 60Hz refresh rate, SOC 2 would have a janky frame every 1.5s of operation. Meanwhile, SOC 1 (the slower SOC according to Benchmark X) would be perfectly fluid.

Using bug reports

Bug reports are sometimes useful for performance analysis, but because they are so heavyweight, they are rarely useful for debugging sporadic jank issues. They may provide some hints on what the system was doing at a given time, especially if the jank was around an application transition (which is logged in a bug report). Bug reports can also indicate when something is more broadly wrong with the system that could reduce its effective capacity (such as thermal throttling or memory fragmentation).

Using TouchLatency

Several examples of bad behavior come from TouchLatency, which is the preferred periodic workload used for the Pixel and Pixel XL. It's available at `frameworks/base/tests/TouchLatency` and has two modes: touch latency and bouncing ball (to switch modes, click the button in the upper-right corner).

The bouncing ball test is exactly as simple as it appears: A ball bounces around the screen forever, regardless of user input. It is usually also **by far** the hardest test to run perfectly, but the closer it comes to running without any dropped frames, the better your device will be. The bouncing ball test is difficult because it is a trivial but perfectly consistent workload that runs at a very low clock (this assumes device has a frequency governor; if the device is instead running with fixed clocks, downclock the CPU/GPU to near-minimum when running the bouncing ball test for the first time). As the system quiesces and the clocks drop closer to idle, the required CPU/GPU time per frame increases. You can watch the ball and see things jank, and you'll be able to see missed frames in systrace as well.

Because the workload is so consistent, you can identify most sources of jitter much more easily than in most user-visible workloads by tracking what exactly is running on the system during each missed frame instead of the UI pipeline. **The lower clocks amplify the effects of jitter by making it more likely that any jitter causes a dropped frame.** As a result, the closer TouchLatency is to 60FPS, the less likely you are to have bad system behaviors that cause sporadic, hard-to-reproduce jank in larger applications.

As jitter is often (but not always) clockspeed-invariant, use a test that runs at very low clocks to diagnose jitter for the following reasons:

- Not all jitter is clockspeed-invariant; many sources just consume CPU time.
- The governor should get the average frame time close to the deadline by clocking down, so time spent running non-UI work can push it over the edge to dropping a frame.
