Identifying Capacity-Related Jank

Capacity is the total amount of some resource (CPU, GPU, etc.) a device possesses over some amount of time. This page describes how to identify and address capacity-related jank issues.

Governor slow to react

To avoid jank, the CPU frequency governor needs to be able to respond quickly to bursty workloads. Most UI applications follow the same basic pattern:

1. User is reading the screen.
2. User touches the screen: taps a button, scrolls, etc.
3. Screen scrolls, changes activity, or animates in some way in response to input.
4. System quiesces as new content is displayed.
5. User goes back to reading the screen.

Pixel and Nexus devices implement touch boost to modify CPU frequency governor (and scheduler) behavior on touch. To avoid a slow ramp to a high clock frequency (which could cause the device to drop frames on touch), touch boost usually sets a frequency floor on the CPU to ensure plenty of CPU capacity is available on touch. A floor lasts for some amount of time after touch (usually around two seconds).

Pixel also uses the schedtune cgroup provided by Energy Aware Scheduling (EAS) as an additional touch boost signal: Top applications get additional weight via schedtune to ensure they get enough CPU capacity to run quickly. The Nexus 5X and 6P have a much bigger performance gap between the little and big CPU clusters (A53 and A57, respectively) than the Pixel with the Kryo CPU. We found that the little CPU cluster was not always adequate for smooth UI rendering, especially given other sources of jitter on the device.

Accordingly, on the Nexus 5X and 6P, touch boost modifies the scheduler behavior to make it more likely for foreground applications to move to the big cores (this is conceptually similar to the floor on CPU frequency). Without the scheduler change to make foreground applications more likely to move to the big CPU cluster, foreground applications may have insufficient CPU capacity to render until the scheduler decided to load balance the thread to a big CPU core. By changing the scheduler behavior during touch boost, it is more likely for a UI thread to immediately run on a big core and avoid jank while not forcing it to always run on a big core, which could have severe impacts on power consumption.

Thermal throttling

Thermal throttling occurs when the device must reduce its overall thermal output, usually performed by reducing CPU, GPU, and DRAM clocks. Unsurprisingly, this often results in jank as the system may no longer be able to provide enough capacity to render within a given timeslice. The only way to avoid thermal throttling is to use less power. There are not a lot of ways to do this, but based on our experiences with past SOCs, we have a few recommendations for system vendors.

First, when building a new SOC with heterogeneous CPU architectures, ensure the performance/W curves of the CPU clusters overlap. The overall performance/W curve for the entire processor should be a continuous line. Discontinuities in the perf/W curve force the scheduler and frequency governor to guess what a workload needs; to prevent jank, the scheduler and frequency governor err on the side of giving the workload more capacity than it requires. This results in spending too much power, which contributes to thermal throttling.

Imagine a hypothetical SOC with two CPU clusters:

- Cluster 1, the little cluster, can spend between 100-300mW and scores 100-300 in a throughput benchmark depending on clocks.
- Cluster 2, the big cluster, can spend between 1000 and 1600mW and scores between 800 and 1200 in the same throughput benchmark depending on clocks.

In this benchmark, a higher score is faster. While not more desirable than slower, faster == greater power consumption.

If the scheduler believes a UI workload would require the equivalent of a score of 310 on that throughput benchmark, its best option to avoid jank is to run the big cluster at the lowest frequency, wasting significant power. (This depends on cpuidle behavior and race to idle; SOCs with continuous perf/W curves are easier to optimize for.)

Second, use cpusets. Ensure you have enabled cpusets in your kernel and in your **`BoardConfig.mk`**. You must also set up the actual cpuset assignments in your device-specific **`init.rc`**. Some vendors leave this disabled in their BSPs in the hopes they can use other hints to

influence scheduler behavior; we feel this doesn't make sense. cpusets are useful for ensuring load balancing between CPUs is done in a way that reflects what the user is actually doing on the device.

ActivityManager assigns apps to different cpusets based on the relative importance of those apps (top, foreground, background), with more important applications getting more access to CPU cores. This helps ensure quality of service for foreground and top apps.

cpusets are useful on homogeneous CPU configurations, but you should not ship a device with a heterogeneous CPU configuration without cpusets enabled. Nexus 6P is a good model for how to use cpusets on heterogeneous CPU configurations; use that as a basis for your own device's configuration.

cpusets also offer power advantages by ensuring background threads that are not performance-critical never get load balanced to big CPU cores, where they could spend significantly more power without any user-perceived benefit. This can help to avoid thermal throttling as well. While thermal throttling is a capacity issue, jitter improvements have an outsize impact on UI performance when thermal throttling. Because the system will be running closer to its ability to render 60FPS, it takes less jitter to cause a dropped frame.

Last updated April 26, 2017.