Identifying Jitter-Related Jank

Jitter is the random system behavior that prevents perceptible work from running. This page describes how to identify and address jitter-related jank issues.

Application thread scheduler delay

Scheduler delay is the most obvious symptom of jitter: A process that should be run is made runnable but does not run for some significant amount of time. The significance of the delay varies according to the context. For example:

- A random helper thread in an app can probably be delayed for many milliseconds without an issue.
- An application's UI thread may be able to tolerate 1-2ms of jitter.
- Driver kthreads running as SCHED_FIFO may cause issues if they are runnable for 500us before running.

Runnable times can be identified in systrace by the blue bar preceding a running segment of a thread. A runnable time can also be determined by the length of time between the **`sched_wakeup`** event for a thread and the **`sched_switch`** event that signals the start of thread execution.

Threads that run too long

Application UI threads that are runnable for too long can cause problems. Lower-level threads with long runnable times generally have different causes, but attempting to push UI thread runnable time toward zero may require fixing some of the same issues that cause lower level threads to have long runnable times. To mitigate delays:

1. Use cpusets as described in Thermal throttling (https://source.android.com/devices/tech/debug/jank_capacity.html#thermal-throttling).
2. Increase the CONFIG_HZ value.
    - Historically, the value has been set to 100 on arm and arm64 platforms. However, this is an accident of history and is not a good value to use for interactive devices. CONFIG_HZ=100 means that a jiffy is 10ms long, which means that load balancing between CPUs may take 20ms (two jiffies) to happen. This can significantly contribute to jank on a loaded system.
    - Recent devices (Nexus 5X, Nexus 6P, Pixel, and Pixel XL) have shipped with CONFIG_HZ=300. This should have a negligible power cost while significantly improving runnable times. If you do see significant increases in power consumption or performance issues after changing CONFIG_HZ, it's likely one of your drivers is using a timer based on raw jiffies instead of milliseconds and converting to jiffies. This is usually an easy fix (see the patch (https://android.googlesource.com/kernel/msm.git/+/daf0cdf29244ce4098cff186d16df23cfa782993%5E!/) that fixed kgsl timer issues on Nexus 5X and 6P when converting to CONFIG_HZ=300).
    - Finally, we've experimented with CONFIG_HZ=1000 on Nexus/Pixel and found it offers a noticeable performance and power reduction due to decreased RCU overhead.

With those two changes alone, a device should look much better for UI thread runnable time under load.

Using sys.use_fifo_ui

You can try to drive UI thread runnable time to zero by setting the **`sys.use_fifo_ui`** property to 1.

Warning: Do not use this option on heterogeneous CPU configurations unless you have a capacity-aware RT scheduler. And, at this moment, **NO CURRENTLY SHIPPING RT SCHEDULER IS CAPACITY-AWARE**. We're working on one for EAS, but it's not yet available. The default RT scheduler is based purely on RT priorities and whether a CPU already has a RT thread of equal or higher priority.

As a result, the default RT scheduler will happily move your relatively-long-running UI thread from a high-frequency big core to a little core at minimum frequency if a higher priority FIFO kthread happens to wake up on the same big core. **This will introduce significant performance regressions**. As this option has not yet been used on a shipping Android device, if you want to use it get in touch with the Android performance team to help you validate it.

When **`sys.use_fifo_ui`** is enabled, ActivityManager tracks the UI thread and RenderThread (the two most UI-critical threads) of the top application and makes those threads SCHED_FIFO instead of SCHED_OTHER. This effectively eliminates jitter from UI and RenderThreads; the traces we've gathered with this option enabled show runnable times on the order of microseconds instead of milliseconds.

However, because the RT load balancer was not capacity-aware, there was a 30% reduction in application startup performance because

the UI thread responsible for starting up the app would be moved from a 2.1Ghz gold Kryo core to a 1.5GHz silver Kryo core. With a capacity-aware RT load balancer, we see equivalent performance in bulk operations and a 10-15% reduction in 95th and 99th percentile frame times in many of our UI benchmarks.

Interrupt traffic

Because ARM platforms deliver interrupts to CPU 0 only by default, we recommend the use of an IRQ balancer (irqbalance or msm_irqbalance on Qualcomm platforms).

During Pixel development, we saw jank that could be directly attributed to overwhelming CPU 0 with interrupts. For example, if the `mdss_fb0` thread was scheduled on CPU 0, there was a much greater likelihood to jank because of an interrupt that is triggered by the display almost immediately before scanout. `mdss_fb0` would be in the middle of its own work with a very tight deadline, and then it would lose some time to the MDSS interrupt handler. Initially, we attempted to fix this by setting the CPU affinity of the mdss_fb0 thread to CPUs 1-3 to avoid contention with the interrupt, but then we realized that we had not yet enabled msm_irqbalance. With msm_irqbalance enabled, jank was noticeably improved even when both mdss_fb0 and the MDSS interrupt were on the same CPU due to reduced contention from other interrupts.

This can be identified in systrace by looking at the sched section as well as the irq section. The sched section shows what has been scheduled, but an overlapping region in the irq section means an interrupt is running during that time instead of the normally scheduled process. If you see significant chunks of time taken during an interrupt, your options include:

- Make the interrupt handler faster.
- Prevent the interrupt from happening in the first place.
- Change the frequency of the interrupt to be out of phase with other regular work that it may be interfering with (if it is a regular interrupt).
- Set CPU affinity of the interrupt directly and prevent it from being balanced.
- Set CPU affinity of the thread the interrupt is interfering with to avoid the interrupt.
- Rely on the interrupt balancer to move the interrupt to a less loaded CPU.

Setting CPU affinity is generally not recommended but can be useful for specific cases. In general, it's too hard to predict the state of the system for most common interrupts, but if you have a very specific set of conditions that triggers certain interrupts where the system is more constrained than normal (such as VR), explicit CPU affinity may be a good solution.

Long softirqs

While a softirq is running, it disables preemption. softirqs can also be triggered at many places within the kernel and can run inside of a user process. If there's enough softirq activity, user processes will stop running softirqs, and ksoftirqd wakes to run softirqs and be load balanced. Usually, this is fine. However, a single very long softirq can wreak havoc on the system.

Show Issue: Janky headtracking while streaming data over Wi-Fi

In this issue, we saw that VR performance was inconsistent under specific network conditions (Wi-Fi). What we found in the trace was that a single NET_RX softirq could run for 30+ milliseconds. This was eventually tracked down to Receive Packet Steering, a Qualcomm Wi-Fi feature that coalesces many NET_RX softirqs into a single softirq. The resulting softirq could, under the right conditions, have a very large (potentially unbounded) runtime.

While this feature may have reduced total CPU cycles spent on networking, it prevented the system from running the right things at the right time. Disabling the feature didn't impact network throughput or battery life, but it did fix the VR headtracking jank by allowing ksoftirqd to load balance the softirqs instead of circumventing it.

softirqs are visible within the irq section of a trace, so they are easy to spot if the issue can be reproduced while tracing. Because a softirq can run within a user process, a bad softirq can also manifest as extra runtime inside of a user process for no obvious reason. If you see that, check the irq section to see if softirqs are to blame.

Drivers leaving preemption or IRQs disabled too long

Disabling preemption or interrupts for too long (tens of milliseconds) results in jank. Typically, the jank manifests as a thread becoming runnable but not running on a particular CPU, even if the runnable thread is significantly higher priority (or SCHED_FIFO) than the other thread.

Some guidelines:

- If runnable thread is SCHED_FIFO and running thread is SCHED_OTHER, the running thread has preemption or interrupts disabled.
- If runnable thread is significantly higher priority (100) than the running thread (120), the running thread likely has preemption or interrupts disabled if the runnable thread doesn't run within two jiffies.
- If the runnable thread and the running thread have the same priority, the running thread likely has preemption or interrupts disabled if the runnable thread doesn't run within 20ms.

Keep in mind that running an interrupt handler prevents you from servicing other interrupts, which also disables preemption.

Show Issue: CONFIG_BUS_AUTO_SUSPEND causes significant jank

In this issue, we identified a major source of jank in Pixel during bringup. 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 trace_30293222.html file in your browser.

In the trace, locate the SurfaceFlinger EventThread beginning at 2235.195 ms. When performing bouncing ball tuning, we often saw one dropped frame when either SurfaceFlinger or a UI-critical thread ran after being runnable for 6.6ms (two jiffies at CONFIG_HZ=300). Critical SurfaceFlinger threads and the app's UI thread were SCHED_FIFO at the time.

According to the trace, the thread would wake on a particular CPU, remain runnable for two jiffies, and get load balanced to a different CPU, where it would then run. The thread that was running while the UI thread and SurfaceFlinger were runnable was always a priority 120 kworker in pm_runtime_work.

While digging through the kernel to see what pm_runtime_work was actually doing, we discovered the Wi-Fi driver's power management was handled through pm_runtime_work. We took additional traces with Wi-Fi disabled and the jank disappeared. To double-check, we also disabled the Wi-Fi driver's power management in the kernel and took more traces with Wi-Fi connected, and the jank was still gone. Qualcomm was then able to find the offending region with preemption disabled and fix it, and we were able to reenable that option for launch.

Another option for identifying offending regions is with the preemptirqsoff tracer (see Using dynamic ftrace (https://source.android.com/devices/tech/debug/ftrace.html#dftrace)). This tracer can give a much greater insight into the root cause of an uninterruptible region (such as function names), but requires more invasive work to enable. While it may have more of a performance impact, it is definitely worth trying.

Incorrect use of workqueues

Interrupt handlers often need to do work that can run outside of an interrupt context, enabling work to be farmed out to different threads in the kernel. A driver developer may notice the kernel has a very convenient system-wide asynchronous task functionality called *workqueues* and might use that for interrupt-related work.

However, workqueues are almost always the wrong answer for this problem because they are always SCHED_OTHER. Many hardware interrupts are in the critical path of performance and must be run immediately. Workqueues have no guarantees about when they will be run. Every time we've seen a workqueue in the critical path of performance, it's been a source of sporadic jank, regardless of the device. On Pixel, with a flagship processor, we saw that a single workqueue could be delayed up to 7ms if the device was under load depending on scheduler behavior and other things running on the system.

Instead of a workqueue, drivers that need to handle interrupt-like work inside a separate thread should create their own SCHED_FIFO kthread. For help doing this with kthread_work functions, refer to this patch (https://android.googlesource.com/kernel/msm/+/1a7a93bd33f48a369de29f6f2b56251127bf6ab4%5E!/).

Framework lock contention

Framework lock contention can be a source of jank or other performance issues. It's usually caused by the ActivityManagerService lock but can be seen in other locks as well. For example, the PowerManagerService lock can impact screen on performance. If you're seeing

this on your device, there's no good fix because it can only be improved via architectural improvements to the framework. However, if you are modifying code that runs inside of system_server, it's critical to avoid holding locks for a long time, especially the ActivityManagerService lock.

Binder lock contention

Historically, binder has had a single global lock. If the thread running a binder transaction was preempted while holding the lock, no other thread can perform a binder transaction until the original thread has released the lock. This is bad; binder contention can block everything in the system, including sending UI updates to the display (UI threads communicate with SurfaceFlinger via binder).

Android 6.0 included several patches to improve this behavior by disabling preemption while holding the binder lock. This was safe only because the binder lock should be held for a few microseconds of actual runtime. This dramatically improved performance in uncontended situations and prevented contention by preventing most scheduler switches while the binder lock was held. However, preemption could not be disabled for the entire runtime of holding the binder lock, meaning that preemption was enabled for functions that could sleep (such as copy_from_user), which could cause the same preemption as the original case. When we sent the patches upstream, they promptly told us that this was the worst idea in history. (We agreed with them, but we also couldn't argue with the efficacy of the patches toward preventing jank.)

fd contention within a process

This is rare. Your jank is probably not caused by this.

That said, if you have multiple threads within a process writing the same fd, it is possible to see contention on this fd, however the only time we saw this during Pixel bringup is during a test where low-priority threads attempted to occupy all CPU time while a single high-priority thread was running within the same process. All threads were writing to the trace marker fd and the high-priority thread could get blocked on the trace marker fd if a low-priority thread was holding the fd lock and was then preempted. When tracing was disabled from the low-priority threads, there was no performance issue.

We weren't able to reproduce this in any other situation, but it's worth pointing out as a potential cause of performance issues while tracing.

Unnecessary CPU idle transitions

When dealing with IPC, especially multi-process pipelines, it's common to see variations on the following runtime behavior:

1. Thread A runs on CPU 1.
2. Thread A wakes up thread B.
3. Thread B starts running on CPU 2.
4. Thread A immediately goes to sleep, to be awakened by thread B when thread B has finished its current work.

A common source of overhead is between steps 2 and 3. If CPU 2 is idle, it must be brought back to an active state before thread B can run. Depending on the SOC and how deep the idle is, this could be tens of microseconds before thread B begins running. If the actual runtime of each side of the IPC is close enough to the overhead, the overall performance of that pipeline can be significantly reduced by CPU idle transitions. The most common place for Android to hit this is around binder transactions, and many services that use binder end up looking like the situation described above.

First, use the `wake_up_interruptible_sync()` function in your kernel drivers and support this from any custom scheduler. Treat this as a requirement, not a hint. Binder uses this today, and it helps a lot with synchronous binder transactions avoiding unnecessary CPU idle transitions.

Second, ensure your cpuidle transition times are realistic and the cpuidle governor is taking these into account correctly. If your SOC is thrashing in and out of your deepest idle state, you won't save power by going to deepest idle.

Logging

Logging is not free for CPU cycles or memory, so don't spam the log buffer. Logging costs cycles in your application (directly) and in the log daemon. Remove any debugging logs before shipping your device.

I/O issues

I/O operations are common sources of jitter. If a thread accesses a memory-mapped file and the page is not in the page cache, it faults and reads the page from disk. This blocks the thread (usually for 10+ ms) and if it happens in the critical path of UI rendering, can result in jank. There are too many causes of I/O operations to discuss here, but check the following locations when trying to improve I/O behavior:

- **PinnerService**. Added in Android 7.0, PinnerService enables the framework to lock some files in the page cache. This removes the memory for use by any other process, but if there are some files that are known a priori to be used regularly, it can be effective to mlock those files.

  On Pixel and Nexus 6P devices running Android 7.0, we mlocked four files:

  - /system/framework/arm64/boot-framework.oat
  - /system/framework/oat/arm64/services.odex
  - /system/framework/arm64/boot.oat
  - /system/framework/arm64/boot-core-libart.oat

  These files are constantly in use by most applications and system_server, so they should not be paged out. In particular, we've found that if any of those are paged out, they will be paged back in and cause jank when switching from a heavyweight application.
- **Encryption**. Another possible cause of I/O problems. We find inline encryption offers the best performance when compared to CPU-based encryption or using a hardware block accessible via DMA. Most importantly, inline encryption reduces the jitter associated with I/O, especially when compared to CPU-based encryption. Because fetches to the page cache are often in the critical path of UI rendering, CPU-based encryption introduces additional CPU load in the critical path, which adds more jitter than just the I/O fetch.

  DMA-based hardware encryption engines have a similar problem, since the kernel has to spend cycles managing that work even if other critical work is available to run. We strongly recommend any SOC vendor building new hardware to include support for inline encryption.

Aggressive small task packing

Some schedulers offer support for packing small tasks onto single CPU cores to try to reduce power consumption by keeping more CPUs idle for longer. While this works well for throughput and power consumption, it can be *catastrophic* to latency. There are several short-running threads in the critical path of UI rendering that can be considered small; if these threads are delayed as they are slowly migrated to other CPUs, it **will** cause jank. We recommend using small task packing very conservatively.

Page cache thrashing

A device without enough free memory may suddenly become extremely sluggish while performing a long-running operation, such as opening a new application. A trace of the application may reveal it is consistently blocked in I/O during a particular run even when it often is not blocked in I/O. This is usually a sign of page cache thrashing, especially on devices with less memory.

One way to identify this is to take a systrace using the pagecache tag and feed that trace to the script at `system/extras/pagecache/pagecache.py`. pagecache.py translates individual requests to map files into the page cache into aggregate per-file statistics. If you find that more bytes of a file have been read than the total size of that file on disk, you are definitely hitting page cache thrashing.

What this means is that the working set required by your workload (typically a single application plus system_server) is greater than the amount of memory available to the page cache on your device. As a result, as one part of the workload gets the data it needs in the page cache, another part that will be used in the near future will be evicted and will have to be fetched again, causing the problem to occur again until the load has completed. This is the fundamental cause of performance problems when not enough memory is available on a device.

There's no foolproof way to fix page cache thrashing, but there are a few ways to try to improve this on a given device.

- Use less memory in persistent processes. The less memory used by persistent processes, the more memory available to applications and the page cache.
- Audit the carveouts you have for your device to ensure you are not unnecessarily removing memory from the OS. We've seen situations where carveouts used for debugging were accidentally left in shipping kernel configurations, consuming tens of megabytes of memory. This can make the difference between hitting page cache thrashing and not, especially on devices with less memory.

- If you're seeing page cache thrashing in system_server on critical files, consider pinning those files. This will increase memory pressure elsewhere, but it may modify behavior enough to avoid thrashing.
- Retune lowmemorykiller to try to keep more memory free. lowmemorykiller's thresholds are based on both absolute free memory and the page cache, so increasing the threshold at which processes at a given oom_adj level are killed may result in better behavior at the expense of increased background app death.
- Try using ZRAM. We use ZRAM on Pixel, even though Pixel has 4GB, because it could help with rarely used dirty pages.

Last updated April 26, 2017.