识别与抖动相关的卡顿

抖动是一种随机的系统行为，可阻止可察觉任务运行。本文将介绍如何识别和解决抖动引起的卡顿问题。

应用线程调度程序延迟

调度程序延迟是最明显的抖动征兆，具体表现为：尽管本应运行的进程处于就绪状态，但在相当长的一段时间内并未运行。根据情况的不同，延迟所造成的影响也不尽相同。例如：

- 某个应用中的某个随机帮助程序线程可能会延迟若干毫秒，而不引起任何问题。
- 某个应用的界面线程可能可以容忍 1 至 2 毫秒的抖动。
- 作为 SCHED_FIFO 运行的驱动程序 kthread，如果在进入就绪状态 500 微秒还未开始运行，则可能会引发问题。

就绪状态时间可以在 systrace 中确定，具体是通过线程运行分段前面的蓝色条进行确定。还有一种方法可以确定就绪状态时间，即通过某个线程的 **sched_wakeup** 事件与表示线程开始执行的 **sched_switch** 事件之间的时间长度来确定。

线程运行时间过长

应用界面线程如果处于就绪状态的时间过长，可能会引起问题。不同的低级线程长时间处于就绪状态的原因通常是不同的，不过，如果您想尝试将界面线程的就绪状态时间降至零，可能需要修复导致低级线程长时间处于就绪状态的一些相同问题。要降低延迟，可以尝试以下方法：

1. 使用温控调频 (https://source.android.google.cn/devices/tech/debug/jank_capacity?hl=zh-cn#thermal-throttling) 中描述的 cpuset。
2. 增加 CONFIG_HZ 的值。
    - 过去，在 arm 和 arm64 平台上，该值曾设为 100。不过，这只是一个巧合，该值其实并不适合交互式设备。如果 CONFIG_HZ = 100，则表示 jiffy 的长度为 10 毫秒，即 CPU 之间的负载平衡可能需要 20 毫秒的时间（两个 jiffy）。这极大增加了负载系统发生卡顿的几率。
    - 近期推出的设备（如 Nexus 5X、Nexus 6P、Pixel、Pixel XL 等），出厂设置为 CONFIG_HZ = 300。该设置不仅可以显著改善就绪状态时间的问题，同时造成的

功耗可以忽略不计。如果在更改 CONFIG_HZ 之后，出现功耗明显增加或性能问题，很可能是因为您的某一个驱动程序使用的是以原始 jiffy（而非毫秒）为单位的定时器，并会转换为 jiffy。通常情况下，要修复这个问题并不难（请参阅补丁程序 (https://android.googlesource.com/kernel/msm.git/+/daf0cdf29244ce4098cff186d16df23cfa782993%5E!/)，该补丁程序可修复转换至 CONFIG_HZ = 300 时，Nexus 5X 和 6P 上出现的 kgsl 定时器问题）。

- 最后，我们在 Nexus/Pixel 上尝试设置 CONFIG_HZ = 1000，发现由于 RCU 开销的降低，该设置可显著提升性能并降低功耗。

您只需完成以上两项更改，便可改善设备在负载情况下的界面线程就绪状态时间问题。

使用 sys.use_fifo_ui

您可以尝试将 `sys.use_fifo_ui` 属性设置为 1，从而将界面线程处于就绪状态的时间降至零。

警告：除非您有容量感知型 RT 调度程序，否则不能在异构 CPU 配置上使用此选项。而且，**目前市面上的 RT 调度程序均不具备容量感知功能**。我们正在努力针对 EAS 开发这种类型的调度程序，但尚未推出。默认的 RT 调度程序完全基于两个因素：RT 优先级，以及 CPU 是否已经具有相同或更高优先级的 RT 线程。

因此，如果在同一个大核心上有较高优先级的 FIFO kthread 被唤醒，则默认的 RT 调度程序会将运行时间相对较长的界面线程从高频大核心上移到在最低频率运行的小核心上。**这会使得性能明显降低**。鉴于目前市面上的 Android 设备尚未使用上述功能，如果您想要使用该功能，请与 Android 性能团队联系，让他们帮助您验证此功能。

启用 `sys.use_fifo_ui` 后，ActivityManager 会跟踪首要应用的界面线程和 RenderThread（对界面最为关键的两个线程），并将这些线程设为 SCHED_FIFO 而非 SCHED_OTHER。这种方法可以有效地消除界面线程和 RenderThread 造成的抖动；在启用该选项的情况下，我们收集的跟踪记录显示微秒级（而非毫秒级）就绪状态时间。

不过，由于 RT 负载平衡程序不具备容量感知能力，应用的启动性能降低了 30%。这是因为负责启动应用的界面线程会从 2.1 Ghz 金级 Kryo 核心移至 1.5 GHz 银级 Kryo 核心。我们发现，在许多界面基准测试中，通过采用容量感知型 RT 负载平衡程序，在批量操作中取得了相同的性能，并且第 95 和第 99 个百分位的帧时间缩短了 10-15%。

中断流量

由于 ARM 平台仅在默认情况下才会对 CPU 0 造成中断，因此，我们建议使用 IRQ 平衡程序（在高通平台上，使用 irqbalance 或 msm_irqbalance）。

在 Pixel 开发期间，我们发现，导致卡顿的直接原因是 CPU 0 负载过重而出现中断。例如，如果在 CPU 0 上调度了 `mdss_fb0` 线程，那么在扫描输出之前，显示屏几乎会立即触发中断，从而大大增加了发生卡顿的概率。`mdss_fb0` 将处于工作进程中，且截止时间非常紧迫，而它又会因为 MDSS 中断处理程序损失部分时间。为了解决这个问题，我们最初试图将 mdss_fb0 线程的 CPU 亲和性设置为 CPU 1-3，以此来避免因中断而出现争用，但后来我们意识到，我们还未启用 msm_irqbalance。启用 msm_irqbalance 后，由于其他中断引起的争用减少，因此，即使 mdss_fb0 和 MDSS 中断出现在同一个 CPU 上，卡顿问题依然得到了明显的改善。

这可以通过查看 systrace 中的 sched 和 irq 区段来确定。sched 区段显示的是已调度的内容，但 irq 区段中的重叠区域表示在该时间段内正在运行中断，而不是正常调度的进程。如果您发现某个中断占用了大量时间，您有以下选择：

- 提高中断处理程序的速度。
- 在第一时间阻止中断发生。
- 如果是常规中断，则将中断频率更改为与可能受中断干扰的其他常规工作不同步。
- 直接设置中断的 CPU 亲和性，并防止进行负载平衡。
- 设置受中断干扰的线程的 CPU 亲和性，以避开中断。
- 借助中断平衡程序，将中断移至负载较少的 CPU。

通常我们不建议设置 CPU 亲和性，但在特定情况下，该操作可能会有所帮助。一般来说，我们很难预测系统在大多数常见中断下的状态，但是，对于某些会使系统受到超过正常水平限制（例如 VR）的中断，如果您掌握一套非常具体的触发条件，那么指定 CPU 亲和性可能会是一个很好的解决方案。

长 softirq

当 softirq 运行时，它会停用抢占。softirq 也可以从内核中的多个位置触发，而且可以在用户进程内运行。如果 softirq 活动充分，则用户进程会停止运行 softirq，同时 ksoftirqd 会被唤醒，以运行 softirq 并进行负载平衡。通常情况下，该过程不会出现问题。但是，单个的超长 softirq 可能会对系统造成严重破坏。

+ 显示问题：通过 WLAN 进行数据流式传输时，头部追踪会发生卡顿

我们发现，对于这个问题，在特定网络条件 (WLAN) 下，VR 性能不一致。通过跟踪，我们发现单个 NET_RX softirq 可以运行超过 30 毫秒。我们最后发现，该问题是由接收数据包转向（一种高通 WLAN 功能）造成的，该功能将多个 NET_RX softirq 合并成一个 softirq。所生成的 softirq 在合适的条件下，可能会拥有超长（有可能为无限长）的运行时间。

尽管此功能可能会减少花费在联网上的总 CPU 周期，但是它会使系统无法在正确的时间运行正确的内容。停用该功能不仅不会对网络吞吐量或电池续航造成影响，而且可以使 ksoftirqd 对 softirq 进行负载平衡（而非绕过 softirq），从而可以解决 VR 的头部追踪卡顿问题。

softirq 会在跟踪记录的 irq 区段内显示，因此，如果能够在跟踪过程中重现该问题，就可以轻松找到 softirq。由于 softirq 可以在用户进程内运行，因此一个不良 softirq 也可能在没有明显原因的情况下，在用户进程内表现为额外的运行时。如果发生这种情况，请查看 irq 区段，确认该问题是否是由 softirq 引起的。

驱动程序停用抢占或 IRQ 的时间过长

如果停用抢占或中断的时间过长（达到数十毫秒），就会导致卡顿。通常情况下，卡顿表现为某个进入就绪状态却不在特定 CPU 上运行的线程，即使处于就绪状态的线程的优先级（即 SCHED_FIFO）远高于另一线程。

一些准则如下：

- 如果处于就绪状态的线程是 SCHED_FIFO，而正在运行的线程是 SCHED_OTHER，则正在运行的线程已停用抢占或中断。
- 如果处于就绪状态的线程的优先级 (100) 远高于正在运行的线程 (120)，且处于就绪状态的线程没有在两个 jiffy 的时间内运行，则正在运行的线程很可能已停用抢占或中断。
- 如果处于就绪状态的线程与正在运行的线程具有相同的优先级，且处于就绪状态的线程没有在 20 毫秒内运行，则正在运行的线程可能已停用抢占或中断。

请注意，运行中断处理程序时，您无法为其他中断提供服务，因为中断处理程序也会停用抢占。

+ 显示问题：CONFIG_BUS_AUTO_SUSPEND 造成严重卡顿

在这个问题中，我们发现了在 Pixel 启动时造成卡顿的一个主要原因。要按照示例操作，请下载跟踪记录的 ZIP 文件 (https://source.android.google.cn/devices/tech/debug/perf_traces.zip?hl=zh-cn)（包含本节中提及的其他跟踪记录），将文件解压缩，然后在浏览器中打开 trace_30293222.html 文件。

在跟踪记录中，找到从 2235.195 毫秒开始的 SurfaceFlinger EventThread。在进行弹力球调整时，当 SurfaceFlinger 或界面关键线程在进入就绪状态 6.6 毫秒（两个 jiffy，CONFIG_HZ = 300）后运行时，我们经常会看到丢失一帧的现象。此时，关键 SurfaceFlinger 线程和应用的界面线程为 SCHED_FIFO。

根据跟踪记录，该线程会在特定 CPU 上被唤醒并进入就绪状态，然后在等待两个 jiffy 后被负载平衡到其他 CPU 上开始运行。在界面线程和 SurfaceFlinger 处于就绪状态时正在运行的线程在 pm_runtime_work 中的优先级始终保持为 120 kworker。

通过查看内核了解 pm_runtime_work 的实际运作，我们发现，WLAN 驱动程序中的电源管理是通过 pm_runtime_work 进行的。于是我们停用了 WLAN，然后再获取其他跟踪记录，这时我们发现卡顿消失了。为进一步确认，我们停用了内核中的 WLAN 驱动程序的电源管理功能，并在连接 WLAN 的情况下获取更多跟踪记录，结果发现卡顿同样不见了。高通因此能找到停用抢占的问题区域，并对此进行修复，我们也能在发布时重新启用该选项。

识别问题区域的另一个方式是使用 preemptirqsoff 跟踪程序（请参阅使用动态 ftrace (https://source.android.google.cn/devices/tech/debug/ftrace?hl=zh-cn#dftrace)）。通过此跟踪程序可以更深入地了解无法中断的区域的根本原因（例如函数名称），但要实现这一点需要进行更多激进的工作。虽然这可能会对性能造成更大影响，但绝对值得一试。

错误使用工作队列

中断处理程序通常需要执行可在中断上下文环境之外进行的工作，从而将工作分包给内核中的不同线程。驱动程序开发人员可能会注意到，内核有一个非常方便的全系统异步任务功能（称为工作队列），可用于执行与中断相关的任务。

然而，工作队列几乎从来不是这类问题的解决方法，因为工作队列都是 SCHED_OTHER。而许多硬件中断都是出现在影响性能的关键路径上，因此必须能够立即运行。但工作队列无法保证其运行时间。无论对于何种设备，只要在影响性能的关键路径上发现有工作队列存在，便可断定它就是造成不时出现卡顿的原因。我们发现，在采用旗舰处理器的 Pixel 上，如果设备处于负载状态（负载由调度程序行为和系统上运行的其他进程决定），则单个工作队列的延迟可达 7 毫秒。

如果不是工作队列，而是驱动程序需要在单独的线程中处理类似中断的工作，则驱动程序应该创建自己的 SCHED_FIFO kthread。如需获得关于如何通过 kthread_work 函数实现此操作的帮助信息，请参阅此补丁程序 (https://android.googlesource.com/kernel/msm/+/1a7a93bd33f48a369de29f6f2b56251127bf6ab4%5E!/)

。

框架锁争用

框架锁争用可能造成卡顿或其他性能问题。框架锁争用通常是由 ActivityManagerService 锁引起的，但也可能出现在其他锁中。例如，PowerManagerService 锁可能会影响屏幕开启性能。如果您在设备上遇到这个问题，目前还没有什么有效的解决办法，因为该问题只能通过改进框架的架构来加以改善。但是，如果您正在修改在 system_server 内部运行的代码，切记要避免长时间持有锁（尤其是 ActivityManagerService 锁），这非常重要。

Binder 锁争用

过去，binder 有一个单独的全局锁。如果运行 binder 事务的线程在持有锁时被抢占，则在该原始线程释放该锁之前，其他线程都无法执行 binder 事务。这是非常糟糕的情况，因为 binder 争用会阻止系统中的所有活动，包括向显示屏发送界面更新（界面线程通过 binder 与 SurfaceFlinger 通信）。

Android 6.0 包含了几个补丁程序，可以在持有 binder 锁时停用抢占，从而改善此问题。这样做之所以是安全的，只是因为应该在实际运行时的几微秒内持有 binder 锁。这可以极大地提高在无争用情况下的性能，而且还可以在 binder 锁被持有期间阻止大多数调度程序切换，从而防止发生争用。但是，由于无法在持有 binder 锁的整个运行时内停用抢占，因此对于那些可以进入睡眠状态的函数（例如 copy_from_user）而言，抢占仍为启用状态，而这可能会导致和初始情况一样的抢占。当我们向上游部门提交补丁程序时，他们很快就告诉我们这是最不明智的做法。（我们同意他们的观点，但我们也不能否认这些补丁程序可以有效地预防卡顿。）

进程内的 fd 争用

这种情况很罕见。通常情况下不会因此造成卡顿。

也就是说，如果您在一个进程中有多个线程在写入相同的 fd，就有可能在此 fd 上出现争用。但是，在 Pixel 启动期间，我们只在一个测试中看到过这个问题，在该测试中，低优先级线程尝试占用所有 CPU 时间，而此时单个高优先级线程正在同一进程中运行。所有线程都在写入跟踪标记 fd，如果一个低优先级线程正持有 fd 锁，然后被抢占，则高优先级线程就可能在跟踪标记 fd 上受到阻止。如果低优先级线程被停用跟踪，则不会出现性能问题。

我们在其他任何情况下都没能重现此问题，但该问题依然值得一提，因为这也是跟踪期间可能造成性能问题的一个潜在原因。

不必要的 CPU 空闲转换

在处理 IPC 时，尤其是处理多进程通路时，经常可以看到以下运行时行为的变化：

1. 线程 A 在 CPU 1 上运行。
2. 线程 A 唤醒线程 B。
3. 线程 B 开始在 CPU 2 上运行。
4. 线程 A 立即进入睡眠状态，待线程 B 完成其当前工作后，由线程 B 将线程 A 唤醒。

开销通常在步骤 2 和步骤 3 之间产生。如果 CPU 2 空闲，它必须回到活动状态，然后线程 B 才可以运行。根据 SOC 和空闲的深度，线程 B 可能需要等待几十微秒才能开始运行。如果 IPC 每一侧的实际运行时与开销足够接近，CPU 空闲转换将会大幅度降低该通路的整体性能。Android 最常发生上述情况的位置是 binder 事务周围，而许多使用 binder 的服务最后都与上述情况相似。

首先，请在内核驱动程序中使用 `wake_up_interruptible_sync()` 函数，并在所有自定义调度程序中支持此函数。请将此作为一项要求，而不是一个提示。目前 binder 使用该函数，这可以避免不必要的 CPU 空闲转换，对同步 binder 事务很有帮助。

第二，请确保您的 CPU 空闲转换时间是可行的，并且 CPU 空闲调节器能正确地将这些因素纳入考虑范畴。如果您的 SOC 出现最深空闲状态的颠簸状态，则无法通过进入最深空闲状态来实现节能。

日志记录

日志记录并非不会占用 CPU 周期或内存，因此，请勿滥用日志缓冲区。日志记录会消耗应用（直接）和日志守护进程的周期。在设备出库之前，请删除全部调试日志。

I/O 问题

I/O 操作是造成抖动的常见原因。如果某个线程去访问内存映射文件，而该页面不在页面缓存中，则会发生故障，并且该线程会从磁盘读取该页面。这会造成该线程被阻塞（通常会阻塞 10 毫秒以上），并且如果问题发生在界面渲染的关键路径中，则可能会导致卡顿。造成 I/O 操作的原因多种多样，在这里我们就不一一讨论了，但在尝试改进 I/O 行为时，请注意检查以下位置：

- **PinnerService**：PinnerService 是 Android 7.0 中新增的一项功能，可以让框架锁定页面缓存中的某些文件。这会清除这部分内存，以供任何其他进程使用。但是，如果事先已知会定期使用某些文件，则可以有效地 mlock（内存锁定）这些文件。

在运行 Android 7.0 的 Pixel 和 Nexus 6P 设备上，我们 mlock 了以下四个文件：

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

这些文件会被大多数应用和 system_server 持续使用，因此不应该被移出页面。并且我们发现，如果上述任何一个文件被移出页面，当从重型应用切换时，它们会被重新移回页面，从而引起卡顿。

- **加密**：这是造成 I/O 问题的另一个可能原因。我们发现，与基于 CPU 的加密或使用可通过 DMA 访问的硬件块相比，内嵌加密可提供最佳的性能。最重要的是，内嵌加密可以减少与 I/O 相关的抖动，尤其是跟基于 CPU 的加密相比。由于提取页面缓存通常发生在界面渲染的关键路径中，而基于 CPU 的加密会在关键路径中引入额外的 CPU 负载，因此会比 I/O 提取造成更多的抖动。

  基于 DMA 的硬件加密引擎也有类似问题，因为即使其他关键工作可以运行，内核仍然需要花费时间来管理此类工作。我们强烈建议所有 SOC 供应商在构建新硬件时，都应支持内嵌加密功能。

激进式小任务打包

一些调度程序支持将小任务打包到单个 CPU 内核上，从而让更多 CPU 在更长时间内保持空闲，以此降低功耗。虽然这样可以有效改善吞吐量和功耗，但却会造成严重的延迟。如果界面渲染的关键路径中存在多个短时间运行的线程，可以将其视为小任务；但如果这些线程在缓慢迁移至其他 CPU 的过程中出现延迟，则**会**导致卡顿。我们建议谨慎使用小任务打包。

页面缓存颠簸

如果一个设备没有足够的可用内存，当它执行长时间运行的操作（例如，打开一个新应用）时，就可能会突然变得非常缓慢。应用的跟踪记录可能显示，在执行特定运行操作期间，它在 I/O 中始终处于被阻止状态，尽管一般情况下它不会在 I/O 中被阻止。这通常可视为页面缓存颠簸的标志，尤其是在内存较少的设备上。

识别此问题的方法之一，就是使用 pagecache 标记来执行 systrace，并将跟踪记录提供给位于 `system/extras/pagecache/pagecache.py` 的脚本。pagecache.py 会使将文件映射到页

面缓存的单个请求转换成汇总的每个文件的统计信息。如果您发现某个文件被读取的字节数超过了磁盘上该文件的总大小，毫无疑问，您遇到了页面缓存颠簸问题。

这意味着：您的工作负载（通常是单个应用与 system_server 之和）所要求的工作集超出了您的设备上可供页面缓存使用的内存容量。因此，当工作负载的一部分在页面缓存中取得其需要的数据时，将在不久之后使用的另一部分工作负载则会被逐出，并不得不再次被提取，这将导致问题再次发生，直到工作负载完成。当设备上没有足够的可用内存时，这将是导致性能问题的根本原因。

对于解决页面缓存颠簸问题，目前尚没有万全之策，但可以尝试通过以下方法在给定的设备上改善此问题。

- 在持续性进程中使用较少内存。持续性进程所占用的内存越少，应用和页面缓存可使用的内存就越多。
- 审核您设备的 carveout，确保您没有从操作系统中删除不必要的内存。我们看到过这种情况，用于调试的 carveout 被无意中遗留在出库设备的内核配置中，结果占用了上百兆字节的内存。这决定了是否会出现页面缓存颠簸问题，尤其是在内存较少的设备上。
- 如果您发现 system_server 的关键文件上存在页面缓存颠簸问题，请考虑固定此类文件。尽管这样做会增加其他地方的内存压力，但是可以修改行为以达到避免颠簸的目的。
- 重新调试 lowmemorykiller，让更多内存保持可用。lowmemorykiller 的阈值由绝对可用内存和页面缓存决定，因此，提高给定 oom_adj 水平下的进程受到终止的阈值，可以改善行为，但代价是会提高后台应用被终止的数量。
- 尝试使用 ZRAM。尽管 Pixel 有 4 GB 内存，但我们仍在 Pixel 上使用 ZRAM，因为这有助于处理极少使用的脏页。

Last updated 2019-10-02.