使用 ftrace

ftrace 是一种调试工具，用于了解 Linux 内核中的情况。以下部分详细介绍了 ftrace 的基本功能、ftrace 与 atrace（捕获内核事件）如何配合使用，以及动态 ftrace。

如需详细了解 systrace 中没有的 ftrace 高级功能，请参阅 ftrace 文档：**<kernel tree>/Documentation/trace/ftrace.txt** (https://www.kernel.org/doc/Documentation/trace/ftrace.txt)。

通过 atrace 捕获内核事件

atrace (**frameworks/native/cmds/atrace**) 使用 ftrace 来捕获内核事件。反之，systrace.py（或更高版本的 Catapult (https://github.com/catapult-project/catapult) 中的 run_systrace.py）使用 adb 在设备上运行 atrace。atrace 会执行以下操作：

- 通过设置属性 (**debug.atrace.tags.enableflags**) 来设置用户模式跟踪。
- 通过写入相应的 ftrace sysfs 节点来启用所需的 ftrace 功能。不过，由于 ftrace 支持的功能更多，您可以自行设置一些 sysfs 节点，然后使用 atrace。

您可以使用 atrace 将该属性设置为适当的值，但启动时跟踪除外。该属性是一个位掩码，除了查看相应的标头（在不同的 Android 版本之间会有所变化），没有确定正确值更好的办法了。

启用 ftrace 事件

ftrace sysfs 节点位于 **/d/tracing** 中，trace 事件在 **/d/tracing/events** 中被划分为多个类别。

要按类别启用事件，请使用：

```
$ echo 1 > /d/tracing/events/irq/enable
```

要按事件启用事件，请使用：

```
$ echo 1 > /d/tracing/events/sched/sched_wakeup/enable
```

如果通过写入 sysfs 节点启用了额外的事件，这些事件将**不会**被 atrace 重置。Qualcomm 设备启动的常见模式是启用 **kgsl** (GPU) 和 **mdss**（显示管道）跟踪点，然后使用 atrace 或 systrace (https://source.android.google.cn/devices/tech/debug/systrace?hl=zh-cn)：

```
$ adb shell "echo 1 > /d/tracing/events/mdss/enable"
$ adb shell "echo 1 > /d/tracing/events/kgsl/enable"
$ ./systrace.py sched freq idle am wm gfx view binder_driver irq workq ss sync
```

您还可以单独使用 ftrace（不使用 atrace 或 systrace），这在您需要仅限内核的跟踪（或者您已经花时间手动写入用户模式跟踪属性）时很有帮助。如需只运行 ftrace，请执行以下操作：

1. 将缓冲区大小设置为足以用于您跟踪的值：

   ```
   $ echo 96000 > /d/tracing/buffer_size_kb
   ```

2. 启用跟踪：

   ```
   $ echo 1 > /d/tracing/tracing_on
   ```

3. 运行您的测试，然后停用跟踪：

   ```
   $ echo 0 > /d/tracing/tracing_on
   ```

4. 转储跟踪：

   ```
   $ cat /d/tracing/trace > /data/local/tmp/trace_output
   ```

trace_output 以文本形式提供跟踪记录。要使用 Catapult 将其可视化，请从 Github 获取 Catapult 代码库 (https://github.com/catapult-project/catapult/tree/master/)并运行 trace2html：

```
$ catapult/tracing/bin/trace2html ~/path/to/trace_file
```

默认情况下，此操作会将 **trace_file.html** 写入同一目录中。

相关事件

同时查看 Catapult 可视化内容和 ftrace 日志通常很有帮助，例如，某些 ftrace 事件（尤其是特定于供应商的事件）未经 Catapult 可视化。不过，Catapult 的时间戳与跟踪记录中的第一个事件或 atrace 所转储的特定时间戳相关，而原始的 ftrace 时间戳则基于 Linux 内核中的特定绝对时钟源。

要从 Catapult 事件中查找特定的 ftrace 事件，请执行以下操作：

1. 打开原始的 ftrace 日志。最新版本的 systrace 中的跟踪记录默认经过压缩：
    - 如果您使用 **`--no-compress`** 捕获 systrace，则跟踪记录位于 html 文件中以 BEGIN TRACE 开头的部分。
    - 如果没有，请从 Catapult 树 (https://github.com/catapult-project/catapult/tree/master/) (**`tracing/bin/html2trace`**) 运行 html2trace 来解压缩跟踪记录。
2. 在 Catapult 可视化中查找相对时间戳。
3. 在跟踪记录的开头找到一行包含 **`tracing_mark_sync`** 的内容。该行应该与以下内容相似：

    ```
    >-5134  (-----) [003] ...1    68.104349: tracing_mark_write: trace_event_
    ```

    如果此行不存在（或者您在使用 ftrace 时没有使用 atrace），则时间将相对于 ftrace 日志中的第一个事件。
    a. 将相对时间戳（以毫秒为单位）添加到 **`parent_ts`**（以秒为单位）中的值。
    b. 搜索新时间戳。

执行这些步骤之后，您应该会找到（或至少非常接近于）要找的事件。

使用动态 ftrace

当 systrace 和标准 ftrace 不能满足需求时，还有最后一种可用资源：动态 ftrace。动态 ftrace 涉及在启动后重写内核代码，因此出于安全考虑，它在正式版内核中不可用。不过，2015 和 2016 版中的每个严重性能错误最终都使用动态 ftrace 找出了根本原因。它在调试不

间断休眠方面的功能尤其强大，因为每次命中触发不间断休眠的函数时，您都会在内核中获得堆栈跟踪记录。您还可以调试中断和抢占被停用的部分，这对证明问题非常有帮助。

要开启动态 ftrace，请修改您内核的 defconfig：

1. 移除 CONFIG_STRICT_MEMORY_RWX（如果存在）。如果您使用的是 3.18 或更新版本以及 arm64，则不存在。
2. 添加以下内容：CONFIG_DYNAMIC_FTRACE=y、CONFIG_FUNCTION_TRACER=y、CONFIG_IRQSOFF_TRACER=y、CONFIG_FUNCTION_PROFILER=y 和 CONFIG_PREEMPT_TRACER=y
3. 重新编译和启动新内核。
4. 运行以下代码以检查可用的跟踪工具：

```
$ cat /d/tracing/available_tracers
```

5. 确认命令返回 **function**、**irqsoff**、**preemptoff** 和 **preemptirqsoff**。
6. 运行以下命令以确保动态 ftrace 正常运行：

```
$ cat /d/tracing/available_filter_functions | grep <a function you care a
```

完成这些步骤之后，动态 ftrace、函数分析器、irqsoff 分析器以及 preemptoff 分析器便处于可用状态。我们**强烈建议**您在使用之前阅读有关这些主题的 ftrace 文档，因为它们虽然功能强大，但比较复杂。irqsoff 和 preemptoff 主要用于确认驱动程序是否会使中断或抢占处于关闭状态的时间过长。

函数分析器是诊断性能问题的最佳选择，通常用于查找函数被调用的位置。

+ 显示问题：HDR 照片 + 旋转取景器

对于此问题，每次使用 Pixel XL 拍摄 HDR + 照片后立即旋转取景器时都会造成卡顿。我们使用函数分析器不到一个小时就调试了该问题。要按照示例进行操作，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)（其中包括本部分中提到的其他跟踪记录），解压缩下载的文件，然后在浏览器中打开 trace_30898724.html 文件。

跟踪记录显示 cameraserver 进程中的一些线程在 **ion_client_destroy** 上的不间断休眠模式下被屏蔽。此为高代价函数，只能极为偶尔地调用，因为 ion 客户端应该包含多项分配。

最初是将原因归咎于 Halide 中的 Hexagon 代码，它确实是其中一个原因（它为每个 ion 分配创建了新的客户端，并在分配被释放时破坏相应客户端，其代价太高了）。改为针对所有 Hexagon 分配使用单个 ion 客户端改善了这种情况，但卡顿问题并没有得到解决。

此时，我们需要知道谁在调用 **ion_client_destroy**，这就要使用函数分析器：

1. 由于编译器有时会对函数进行重命名，因此通过使用以下命令确认 **ion_client_destroy** 是否存在：

```
$ cat /d/tracing/available_filter_functions | grep ion_client_destroy
```

2. 确认存在之后，将其用作 ftrace 过滤器：

```
$ echo ion_client_destroy > /d/tracing/set_ftrace_filter
```

3. 开启函数分析器：

```
$ echo function > /d/tracing/current_tracer
```

4. 每次调用筛选器函数时均开启堆栈跟踪：

```
$ echo func_stack_trace > /d/tracing/trace_options
```

5. 增加缓冲区大小：

```
$ echo 64000 > /d/tracing/buffer_size_kb
```

6. 开启跟踪功能：

```
$ echo 1 > /d/tracing/trace_on
```

7. 运行测试并获取跟踪记录：

```
$ cat /d/tracing/trace > /data/local/tmp/trace
```

8. 查看跟踪，以查看大量堆栈跟踪记录：

```
   cameraserver-643   [003] ...1    94.192991: ion_client_destroy <-ion_
   cameraserver-643   [003] ...1    94.192997: <stack trace>
=> ftrace_ops_no_ops
=> ftrace_graph_call
=> ion_client_destroy
=> ion_release
=> __fput
=> ____fput
=> task_work_run
=> do_notify_resume
=> work_pending
```

对 ion 驱动程序进行检查后，我们可以发现，**ion_client_destroy** 被将 fd 关闭到 **/dev/ion**（而非随机内核驱动程序）的用户空间函数所破坏。通过在 Android 代码库中搜索 **\"/dev/ion\"**，我们发现一些供应商驱动程序与 Hexagon 驱动程序的功能相同，每次在需要新的 ion 分配时打开/关闭 **/dev/ion**（创建和破坏新的 ion 客户端）。将其更改为在进程的生命周期中使用单个 ion 客户端 (https://android.googlesource.com/platform/hardware/qcom/camera/+/8f7984018b6643f430c229725a58d3c6bb04acab)
即修复了该错误。

如果函数分析器的数据不够具体，您可以将 ftrace 跟踪点与函数分析器结合使用。此外，还可以像平常一样启用 ftrace 事件，这些事件会与您的跟踪记录交错。如果您要调试的特定函数中偶尔存在不间断的长时间休眠，这会非常有用：将 ftrace 过滤器设置为所需函数，启用跟踪点，然后进行跟踪。您可以使用 **trace2html** 解析得出的跟踪记录，查找所需的事件，然后获取原始跟踪记录中相邻的堆栈跟踪记录。

使用 lockstat

有时，只有 ftrace 是不够的，您必须调试内核锁争用。还有一种内核选项值得尝试：**CONFIG_LOCK_STAT**。这是最后一种方法，因为要在 Android 设备上应用这一方法非常困难，原因是它会使内核的大小超出大多数设备可以处理的范围。

不过，lockstat 使用调试锁基础设施，这对很多其他应用都有帮助。负责设备启动的每个人都应该设法让该选项适用于每台设备，因为，如果**以后**碰到这方面的事情，您可能会想“要是能开启 `LOCK_STAT`，我就可以在 5 分钟（而不是 5 天）内判断是不是这方面的问题”。

显示问题：当内核以最大负载运行 non-SCHED_FIFO 时，SCHED_FIFO 终止

在这个问题中，当所有内核以最大负载运行 non-SCHED_FIFO 线程时，SCHED_FIFO 线程便会终止。我们的跟踪记录显示 VR 应用中的 fd 存在明显的锁争用，但是我们无法轻松确定相关 fd。要按照示例进行操作，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)（其中包括本部分中提到的其他跟踪记录），解压下载文件，然后在浏览器中打开 trace_30905547.html 文件。

我们假设 ftrace 本身是锁争用的来源，当优先级较低的线程开始写入 ftrace 管道时，却在能够释放锁之前被抢占。这是最糟糕的一种情况，一些优先级极低的线程写入 ftrace 标记，同时另一些优先级较高的线程在 CPU 上旋转以模拟完全负载的设备，从而进一步加重情况。

由于我们无法使用 ftrace 进行调试，因此运行 `LOCK_STAT`，然后关闭应用的所有其他跟踪功能。结果表明，锁争用实际上来自 ftrace，因为当 ftrace 不运行时，锁跟踪记录中没有出现任何争用情况。

如果您可以使用配置选项启动内核，则锁跟踪与 ftrace 类似：

1. 启用跟踪：

```
$ echo 1 > /proc/sys/kernel/lock_stat
```

2. 运行您的测试。
3. 停用跟踪：

```
$ echo 0 > /proc/sys/kernel/lock_stat
```

4. 转储您的跟踪记录：

```
$ cat /proc/lock_stat > /data/local/tmp/lock_stat
```

如需获取关于解读所生成的输出结果的帮助信息，请参阅 lockstat 文档：
<kernel>/Documentation/locking/lockstat.txt
(https://www.kernel.org/doc/Documentation/locking/lockstat.txt)。

使用供应商跟踪点

首先使用上游跟踪点，但有时需要使用供应商跟踪点：

```
{ "gfx",        "Graphics",         ATRACE_TAG_GRAPHICS, {
      { OPT,      "events/mdss/enable" },
      { OPT,      "events/sde/enable" },
      { OPT,      "events/mali_systrace/enable" },
  } },
```

跟踪点可以通过 HAL 服务扩展，这样您就可以添加设备专用的跟踪点/类别。跟踪点与 perfetto、atrace/systrace 以及设备上的系统跟踪应用集成。

用于实现跟踪点/类别的 API 有：

- listCategories()generates (vec<TracingCategory> categories);
- enableCategories(vec<string> categories) generates (Status status);
- disableAllCategories() generates (Status status);

如需了解详情，请参阅 AOSP
(https://android-review.googlesource.com/q/topic:%22atracehal_aosp%22+(status:open%20OR%20status:merged))
中的 HAL 定义和默认实现。