密级状态：绝密(　　)　　秘密(　　)　　内部(　)　　公开(√)

RK3399 性能优化方法

（技术部，第二系统产品部）

文件状态： [　] 正在修改 [√] 正式发布	当前版本：	V1.5
	作　　者：	郑建、刘益星、张文平
	完成日期：	2019-07-01
	审　　核：	张文平、陈海燕
	完成日期：	20190701

福州瑞芯微电子股份有限公司

Fuzhou Rockchips Semiconductor Co., Ltd

版 本 历 史

版本号	作者	修改日期	修改说明	备注
V1.0	刘益星	2018-6-6	发布初始版本	
V1.1	郑建	2018-10-11	方法三：增加指定线程绑定大核的 demo 以及说明	
V1.2	郑建	2018-12-29	补充 target_load 的说明	
V1.3	郑建	2018-01-16	补充性能监测工具	
V1.4	张文平	2019-5-28	补充 Touch Boost 优化方法	
V1.5	郑建	2019-5-28	补充配置进程名和线程名绑定功能说明	

目 录

概述

本文主要介绍针对 android 系统的一些常用的性能观测和优化方法。

性能监测工具

mpstat

mpstat 是一个常用的多核 CPU 性能分析工具，用来实时查看每个 CPU 的性能指标，以及所有 CPU 的平均指标，一般的，建议先用 mpstat 来初步快速地统计 CPU 的负载，方便下一步的定位。

```
busybox mpstat
```

当 mpstat 不带参数时，输出为从系统启动以来的平均值，可作为平均负载参考。

```
127|rk3399_all:/system/bin # busybox mpstat
Linux 4.4.126 (localhost)       01/19/13        _aarch64_       (6 CPU)

14:51:25     CPU    %usr   %nice    %sys %iowait    %irq   %soft  %steal  %guest   %idle
14:51:25     all   13.29    0.02    2.22    0.01    0.00    0.03    0.00    0.00   84.44
```

-P ALL 表示监控所有 CPU，后面数字 2 表示间隔 2 秒后输出一组数据。

```
busybox mpstat -P ALL 2
```

```
rk3399_all:/system/bin # busybox mpstat -P ALL 2
Linux 4.4.126 (localhost)       01/19/13        _aarch64_       (6 CPU)

14:46:31     CPU    %usr   %nice    %sys %iowait    %irq   %soft  %steal  %guest   %idle
14:46:33     all   11.05    0.00    8.77    0.00    0.00    0.17    0.00    0.00   80.02
14:46:33       0   22.77    0.00   14.85    0.00    0.00    0.99    0.00    0.00   61.39
14:46:33       1   20.00    0.00   11.50    0.00    0.00    0.00    0.00    0.00   68.50
14:46:33       2   17.62    0.00   18.13    0.00    0.00    0.00    0.00    0.00   64.25
14:46:33       3    5.08    0.00    6.09    0.00    0.00    0.00    0.00    0.00   88.83
14:46:33       4    0.51    0.00    2.53    0.00    0.00    0.00    0.00    0.00   96.97
14:46:33       5    0.51    0.00    0.00    0.00    0.00    0.00    0.00    0.00   99.49

14:46:33     CPU    %usr   %nice    %sys %iowait    %irq   %soft  %steal  %guest   %idle
14:46:35     all   10.07    0.09    9.81    0.00    0.00    0.09    0.00    0.00   79.95
14:46:35       0   16.95    0.56   18.64    0.00    0.00    0.56    0.00    0.00   63.28
14:46:35       1   13.24    0.00   15.20    0.00    0.00    0.00    0.00    0.00   71.57
14:46:35       2   15.43    0.00   12.77    0.00    0.00    0.00    0.00    0.00   71.81
14:46:35       3   12.76    0.00   11.22    0.00    0.00    0.00    0.00    0.00   76.02
14:46:35       4    1.03    0.00    1.03    0.00    0.00    0.00    0.00    0.00   97.95
14:46:35       5    1.50    0.00    0.50    0.00    0.00    0.00    0.00    0.00   98.00

14:46:35     CPU    %usr   %nice    %sys %iowait    %irq   %soft  %steal  %guest   %idle
14:46:37     all    8.55    0.00    8.11    0.00    0.00    0.09    0.00    0.00   83.25
14:46:37       0   12.92    0.00   14.04    0.00    0.00    0.56    0.00    0.00   72.47
14:46:37       1   18.58    0.00   19.13    0.00    0.00    0.00    0.00    0.00   62.30
14:46:37       2    7.03    0.00    7.03    0.00    0.00    0.00    0.00    0.00   85.95
14:46:37       3   12.82    0.00    8.21    0.00    0.00    0.00    0.00    0.00   78.97
14:46:37       4    1.02    0.00    1.52    0.00    0.00    0.00    0.00    0.00   97.46
14:46:37       5    0.50    0.00    0.50    0.00    0.00    0.00    0.00    0.00   98.99
```

pidstat

pidstat 是一个常用的进程性能分析工具，用来实时查看进程的 CPU、内存、I/O 以及上下文切换等性能指标，用于初步定位系统负载瓶颈在 CPU、内存或者磁盘 IO，缩小排查范围，建议在使用 mpstat 初步定位之后，使用 pidstat 进一步定位负载瓶颈。

pidstat 主要用于监控全部或指定进程占用系统资源的情况，如 CPU、内存、设备 IO、任务切换、线程等。pidstat 首次运行时显示自系统启动开始的各项统计信息，之后运行 pidstat 将显示自上次运行该命令以后的统计信息。用户可以通过指定统计的次数和时间来获得所需的统计信息。

默认参数

执行 pidstat，将输出系统启动后所有活动进程的 CPU 统计信息：

```
pidstat 1
```

命令输出进程的 CPU 占用率，该命令会持续输出，并且不会覆盖之前的数据，可以方便观察系统动态。

```
rk3399_all:/system/bin # pidstat 1
Linux 4.4.126 (localhost)       01/19/13        _armv81_        (6 CPU)

15:01:40      UID       PID    %usr %system  %guest   %wait    %CPU   CPU  Command
15:01:41        0         7    0.00    0.95    0.00    1.90    0.95     2  rcu_preempt
15:01:41        0       165    0.00    0.95    0.00    0.00    0.95     0  irq/122-inv_irq
15:01:41     1000       233   17.14   15.24    0.00    0.00   32.38     3  surfaceflinger
15:01:41     1041      1535    4.76    1.90    0.00    0.00    6.67     0  audioserver
15:01:41     1000      1675    2.86    1.90    0.00    0.00    4.76     3  system_server
15:01:41    10057      3979   36.19   33.33    0.00    0.00   69.52     0  com.tencent.wok
15:01:41        0      5025    0.00    0.95    0.00    0.00    0.95     1  kworker/u13:1
15:01:41        0      5133    0.00    1.90    0.00    0.95    1.90     2  kworker/u13:2
15:01:41        0      5135    3.81    2.86    0.00    0.00    6.67     5  pidstat

15:01:41      UID       PID    %usr %system  %guest   %wait    %CPU   CPU  Command
15:01:42        0         7    0.00    0.99    0.00    0.99    0.99     1  rcu_preempt
15:01:42        0       165    0.00    0.99    0.00    0.00    0.99     0  irq/122-inv_irq
15:01:42     1000       233   18.81   13.86    0.00    0.00   32.67     3  surfaceflinger
15:01:42     1041      1535    2.97    2.97    0.00    0.00    5.94     0  audioserver
15:01:42     1000      1675    1.98    1.98    0.00    0.00    3.96     3  system_server
15:01:42    10057      3979   36.63   31.68    0.00    0.00   68.32     0  com.tencent.wok
15:01:42        0      5025    0.00    1.98    0.00    0.00    1.98     2  kworker/u13:1
15:01:42        0      5105    0.00    0.99    0.00    0.99    0.99     2  kworker/u12:0
15:01:42        0      5133    0.00    2.97    0.00    0.00    2.97     2  kworker/u13:2
15:01:42        0      5135    1.98    3.96    0.00    0.00    5.94     5  pidstat
```

指定采样周期和采样次数

pidstat 命令指定采样周期和采样次数，命令形式为”pidstat [option] interval [count]”，以下 pidstat 输出以 2 秒为采样周期，输出 10 次 CPU 使用统计信息：

```
pidstat 2 10
```

CPU 使用情况统计(-u)

使用-u 选项，pidstat 将显示各活动进程的 CPU 使用统计，执行”pidstat -u”与单独执行”pidstat”的效果一样。

内存使用情况统计(-r)

使用-r 选项，pidstat 将显示各活动进程的内存使用统计，针对特定进程统计(-p)：

```
pidstat -r -p 233 1
```

```
rk3399_all:/system/bin # pidstat -r -p 233 1
Linux 4.4.126 (localhost)       01/20/13        _armv81_        (6 CPU)

06:45:11      UID       PID  minflt/s  majflt/s      VSZ      RSS   %MEM  Command
06:45:12     1000       233      0.00      0.00   713324   131452   3.34  surfaceflinger
06:45:13     1000       233    109.00      0.00   685252   140876   3.58  surfaceflinger
```

以上各列输出的含义如下：

minflt/s：每秒次缺页错误次数(minor page faults)，次缺页错误次数意即虚拟内存地址映射成物理内存地址产生的 page fault 次数

majflt/s：每秒主缺页错误次数(major page faults)，当虚拟内存地址映射成物理内存地址时，相应的 page 在 swap 中，这样的 page fault 为 major page fault，一般在内存使用紧张时产生

VSZ：　　该进程使用的虚拟内存(以 kB 为单位)

RSS：　　该进程使用的物理内存(以 kB 为单位)

%MEM：　　该进程使用内存的百分比；

Command：　拉起进程对应的命令

IO 情况统计(-d)

使用-d 选项，我们可以查看进程 IO 的统计信息：

```
pidstat –d 1
```

```
rk3399_all:/system/bin # pidstat -d 1
Linux 4.4.126 (localhost)       01/20/13        _armv81_        (6 CPU)

06:49:57      UID       PID   kB_rd/s   kB_wr/s kB_ccwr/s iodelay  Command
06:49:58    10057      8911      0.00     62.96      0.00       0  com.tencent.wok

06:49:58      UID       PID   kB_rd/s   kB_wr/s kB_ccwr/s iodelay  Command
06:49:59        0       320      0.00      4.00      0.00       0  logcatext

06:49:59      UID       PID   kB_rd/s   kB_wr/s kB_ccwr/s iodelay  Command

06:50:00      UID       PID   kB_rd/s   kB_wr/s kB_ccwr/s iodelay  Command
06:50:01        0      8791      0.00      8.00      0.00       0  kworker/u12:0
^C

Average:      UID       PID   kB_rd/s   kB_wr/s kB_ccwr/s iodelay  Command
Average:        0       320      0.00      0.98      0.00       0  logcatext
Average:        0      8791      0.00      1.96      0.00       0  kworker/u12:0
Average:    10057      8911      0.00     16.67      0.00       0  com.tencent.wok
```

输出信息含义：

kB_rd/s：每秒进程从磁盘读取的数据量(以 kB 为单位)

kB_wr/s：每秒进程向磁盘写的数据量(以 kB 为单位)

Command：拉起进程对应的命令

pidstat 常用命令

使用 pidstat 进行问题定位时，以下命令常被用到：

```
pidstat -u 1

pidstat -r 1

pidstat -d 1
```

以上命令以 1 秒为信息采集周期，分别获取 CPU、内存和磁盘 IO 的统计信息。

Android Simpleperf 查找热点函数

什么是 simpleperf

Simpleperf 是 Android 平台的一个强大的本地层性能分析工具，它包含在 NDK 中，可以帮助我们分析应用的 CPU 性能。**Simpleperf** 可以帮助我们找到应用的热点，而热点往往与性能问题相关，这样我们就可以分析修复热点源。它的命令行界面支持与 linux-tools perf 大致相同的选项，但是它还支持许多 Android 特有的改进。

Simpleperf 是 Android 开源项目（AOSP）的一部分。其源代码 位于。其最新的文档 位于。Bugs 和 功能需求可以提交到 githb 上。

Simpleperf 分为 simpleperf bin 文件和 Simpleperf 的 sdk 工具包两部分:

(1)simpleperf bin 文件的 sdk 源码路径：system/extras/simpleperf（如需进一步研究，请参考 google 官方文档 1）。

github:https://android.googlesource.com/platform/system/extras/+/master/simpleperf/

(2)Simpleperf 的 sdk 工具包（为方便简单使用 simpleperf，这里不对工具包进行介绍和使用，simpleperf 提供十分强大的分析能力，如需进一步使用，请参考 google 官方文档 2）

github:https://android.googlesource.com/platform/prebuilts/simpleperf

环境要求

为了使用 Simpleperf，需要以下环境：

- 待分析的 App 应运行在 Android 5.0 或者更高版本的设备上
- 使能手机的 **USB debugging** 连接到宿主机器
- 为了能够运行 **Python scripts**，宿主机器应安装：
  - **Python** 2.7 或者更高版本
  - 最新版本的 Android SDK 以及 NDK

如果你刚开始使用 Simpleperf，下面列出了一些对查找特定内容比较有用的命令，更多命令请参考 Simpleperf Command Reference。

simpleperf 的特性

simpleperf 的工作方式与 linux-tools-perf 类似，但它有如下的提升：

1. Aware of Android environment。Simpleperf 在剖析时处理一些 Android 特有的情形。例如，它可以剖析 apk 中嵌入的共享库，从.gnu_debugdata 段读取符号表和调试信息。如果可能，当出现错误时它会给出一些提示，比如，如何禁用 perf_harden 启用分析。

2. 记录时支持展开。如果我们想使用-g 选项来记录并报告一个程序的调用图，我们需要转储每个记录中的用户栈和寄存器集合，然后展开栈来查找调用链。Simpleperf 支持在记录时展开，因此它无需在 perf.data 中存储用户栈。因而我们可以在设备上有限的空间内剖析更长的时间。
3. 支持脚本使 Android 上更方便的剖析。
4. 编译进静态的二进制文件。Simpleperf 是一个静态库，因此它无需支持运行共享库。这意味着对于 simpleperf 可以运行的 Android 版本没有限制，尽管有些设备不支持剖析。

Simpleperf 小技巧

1. 编译 simpleperf，并运行 simpleperf record 收集 perf.data。

```
mmma system/extras/simpleperf –j20
```

将生成的 out/target/product/rk3399_all/system/xbin/ simpleperf 文件 push 到在 Android 设备上的/ system/xbin/目录。

```
cd data
```

在 Android 设备上运行 simpleperf record -a -g 命令，CTRL+C 结束收集，如果执行成功将生成/data/perf.data 文件（存放收集的信息）。

```
simpleperf record -a –g
```

```
rk3399_all:/data # simpleperf record -a -g
```

2. 查看线程耗时百分比

```
simpleperf report --sort tid,comm
```

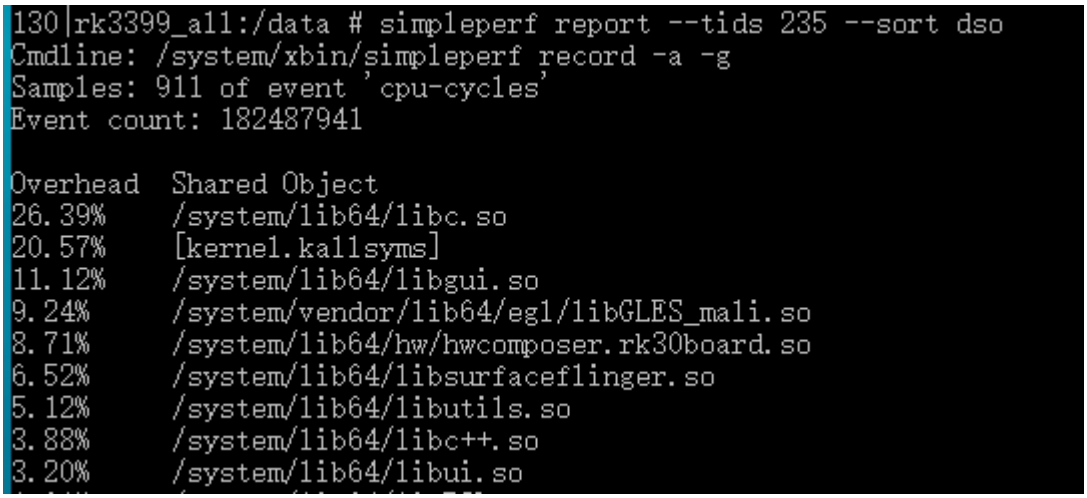
```
^C^Crk3399_all:/data # simpleperf report --sort tid,comm
Cmdline: /system/xbin/simpleperf record -a -g
Samples: 20334 of event 'cpu-cycles'
Event count: 5995117305

Overhead  Tid   Command
61.94%    1500  simpleperf
8.76%     0     swapper
4.38%     1257  RenderThread
4.17%     1225  droid.launcher3
3.04%     235   surfaceflinger
2.14%     260   surfaceflinger
1.88%     1268  mali-cmar-backe
1.36%     628   SensorService
1.25%     244   Binder:235_2
```

3. 查找线程执行最耗时的目标模块（共享库）

在找到了最耗时的线程之后，使用该命令可以查找线程中最耗时的共享库，此处以 surfaceflinger 为例：

```
simpleperf report --tids 235 --sort dso
```



4. 查找最耗时的函数

这里以/system/lib64/hw/hwcomposer.rk30board.so 为例，由于设备上使用的 hwcomposer.rk30board.so 已经抛离了.symbol 段。我们可以将带有调试信息的 hwcomposer.rk30board.so 下载到设备上：

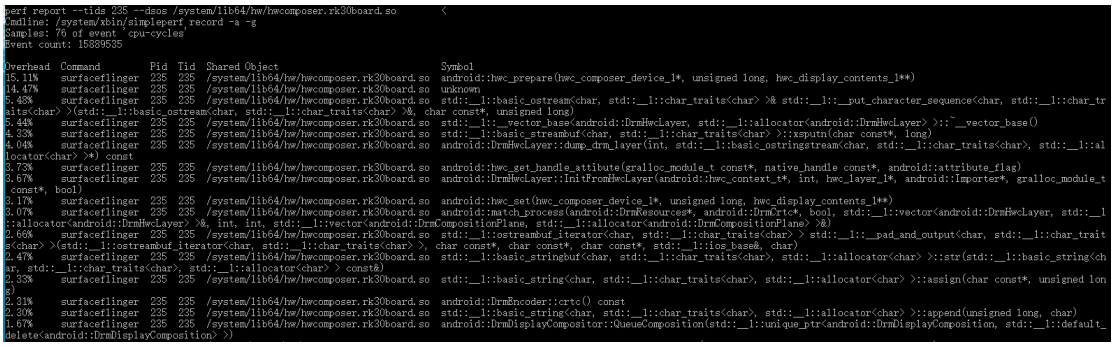
```
adb push
out\target\product\rk3399_all\symbols\system\lib64\hw\hwcomposer.rk30board.so /system/lib64/hw/hwcomposer.rk30board.so
```

然后运行：

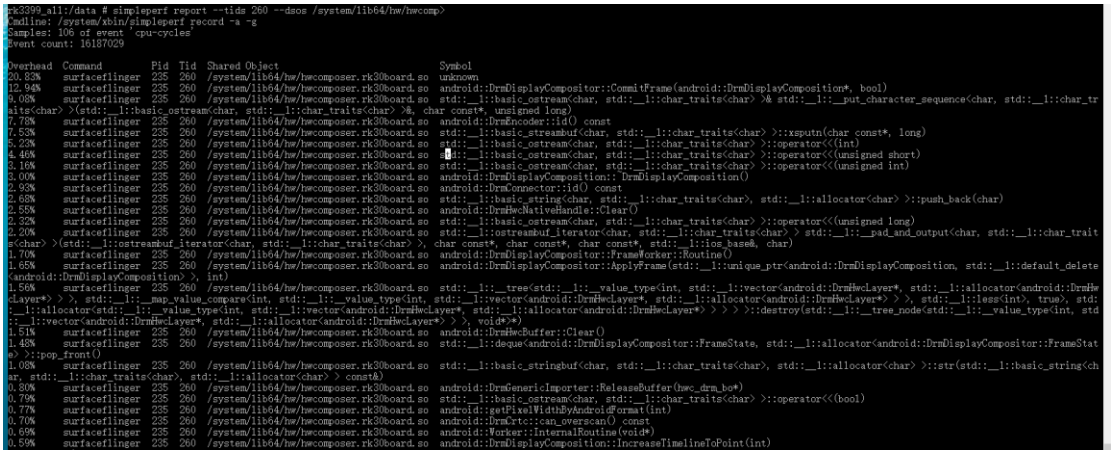
```
simpleperf report --tids 235 --dsos
/system/lib64/hw/hwcomposer.rk30board.so
```

可能会报 Symbol addresses in /proc/kallsyms are all zero. Check /proc/sys/kernel/kptr_restrict if possible.则先输入下面命令，再用 simpleperf report 指令解析 perf.data。

```
echo 1 > /proc/sys/kernel/kptr_restrict
```



simpleperf report --tids 260 --dsos /system/lib64/hw/hwcomposer.rk30board.so



5. 查看函数调用关系

simpleperf report –g>report.txt

性能优化方法

方法一：修改 **interactive** 调频策略的 **target_loads**

RK3399 默认的调频策略是 interactive，此策略同时提供了一些参数供修改，其中最容易理解和修改的参数就是 target_loads，介绍如下：

Kernel/Documentation/cpu-freq/governors.txt：

2.6 Interactive

Documentation/cpu-freq/governors.txt

target_loads: CPU load values used to adjust speed to influence the current CPU load toward that value. In general, the lower the target load, the more often the governor will raise CPU speeds to bring load below the target. The format is a single target load, optionally followed by pairs of CPU speeds and CPU loads to target at or above those speeds. Colons can be used between the speeds and associated target loads for readability. For example:

85 1000000:90 1700000:99

targets CPU load 85% below speed 1GHz, 90% at or above 1GHz, until 1.7GHz and above, at which load 99% is targeted. If speeds are specified these must appear in ascending order. Higher target load values are typically specified for higher speeds, that is, target load values also usually appear in an ascending order. The default is target load 90% for all speeds.

一般情况下，调速器根据 target_loads 参数调整频率，负载超过设定值时提高频率，反之则下降频率；该值设置的越低，CPU 越容易提升频率；单位:%，频率单位:KHz。

格式是单个目标负载，可选地，后面是 CPU 速度对和以这些速度或以上为目标的 CPU 负载。冒号可以在速度和相关目标负载之间使用，以提高可读性。例如:85 1000000:90 1700000:99 目标 CPU 负载 85%低于 1GHz 的速度，90%在 1GHz 或以上，直到 1.7GHz 及以上，目标负载 99%。如果指定了速度，则必须按升序显示。更高的目标负载值通常用于更高的速度，也就是说，目标负载值通常也以升序出现。默认情况下，所有速度的目标负载为 90%。

修改 target_loads 方法，如下红色字体：

在 device/rockchip/rk3399/init.tablet.rc 中：

```
on boot
# update cpusets feature nodes for rk3399 tablet
write /dev/cpuset/foreground/cpus 0-5
write /dev/cpuset/foreground/boost/cpus 4-5
```

```
    write /dev/cpuset/background/cpus 0
    write /dev/cpuset/system-background/cpus 0-3
    write /dev/cpuset/top-app/cpus 4-5
    write /sys/devices/system/cpu/cpufreq/policy4/interactive/target_loads
"65 1008000:70 1200000:75 1416000:80 1608000:90"
```

临时验证的话，手动修改 target_loads 的方法如下：

```
1）su
2）echo "65 1008000:70 1200000:75 1416000:80 1608000:90" >
   /sys/devices/system/cpu/cpufreq/policy4/interactive/target_loads
```

方法二：为不同类别的任务分配 CPU 核资源

如上 init.tablet.rc 看到的，可以通过 linux 系统的 cpuset 子系统为不同任务分配 RK3399 的大小核资源（0 到 3 为小核，4 到 5 为大核）：

以 rk3399 行业 sdk 为例：

```
在 device/rockchip/rk3399/init.tablet.rc 中：
on boot
    # update cpusets feature nodes for rk3399 tablet
    write /dev/cpuset/foreground/cpus 0-5
    write /dev/cpuset/foreground/boost/cpus 0-5
    write /dev/cpuset/background/cpus 0
    write /dev/cpuset/system-background/cpus 0-3
    write /dev/cpuset/top-app/cpus 0-5
    write /sys/devices/system/cpu/cpufreq/policy4/interactive/target_loads 65
```

方法三：指定线程绑定大核

方法二中我们看到，android 上面可以把一类的进程指定到特定的 CPU 上运行以达到更优的运行效果，实现该功能的底层 API 函数其实是 sched_setaffinity(pid_t pid, unsigned int cpusetsize, cpu_set_t *mask)。

该函数设置进程为 pid 的这个进程，让它运行在 mask 所设定的 CPU 上；如果 pid 的

值为 0，则表示指定的是当前进程，使当前进程运行在 mask 所设定的那些 CPU 上；第二个参数 cpusetsize 是 mask 所指定的数的长度，通常设定为 sizeof(cpu_set_t)；如果当前 pid 所指定的进程此时没有运行在 mask 所指定的任意一个 CPU 上，则该指定的进程会从其它 CPU 上迁移到 mask 的指定的一个 CPU 上运行。

这里提供一段代码示例：

```
cpu_set_t mask;
CPU_ZERO(&mask);
CPU_SET(4, &mask);  //0 到 3 为小核，4 到 5 为大核
CPU_SET(5, &mask);
sched_setaffinity(0, sizeof(mask), &mask);  //第一个参数是 pid，如果为 0 表示为当前进程
```

网上实例：https://blog.csdn.net/stn_lcd/article/details/78134574

命令行方式：

```
rk3399_mid:/ # taskset -p 674     //查看进程 674 分配的 CPU 核资源情况
pid 674's current affinity mask: 3f    //3f 表示该进程可以跑在 cpu 0 到 5
rk3399_mid:/ # taskset -p 30 674    //绑定线程 674 到大核上（cpu4,5）
pid 674's current affinity mask: 3f
pid 674's new affinity mask: 30
```

0 到 3 为小核，4 到 5 为大核，大核处理的性能效率更高，建议对 cpu 要求高的线程绑定到 4/5 核上，只绑定进程中 cpu 占用率高的线程。但是不建议绑定整个进程到 cpu4/5 上，因为如果进程创建了多于 2 个以上的线程，并且线程间存在互斥等待的情况，反而会导致该进程效率更低，所以建议只绑定进程中 cpu 占用率高的线程。

这里提供实际应用的代码示例：

1、将某个线程绑定到指定 CPU 的 4、5 大核，详情见下面代码：

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sched.h>
#include <ctype.h>
#include <pthread.h>
#include <unistd.h>
#include <errno.h>
```

```
#include <sys/types.h>
#include <dirent.h>
#include <grp.h>
#include <inttypes.h>
#include <pwd.h>
#include <string.h>
#include <jni.h>
#include <time.h>
#include <poll.h>
#include <fcntl.h>
#include <android/log.h>
#include <sys/stat.h>
#include <sys/syscall.h>
#include <assert.h>
#define TAG "native-lib"

#define DEBUG 1
#ifdef DEBUG
#define LOGD(...) __android_log_print(ANDROID_LOG_DEBUG,TAG,__VA_ARGS__)
#define LOGE(...) __android_log_print(ANDROID_LOG_ERROR,TAG,__VA_ARGS__)
#else
#define LOGD(...) ((void)0)
#define LOGE(...) ((void)0)
#endif

void set_cur_thread_affinity(cpu_set_t *mask)
{
    int err, syscallres;
    pid_t tid = gettid();
    print_cpu_mask(*mask);
    syscallres = syscall(__NR_sched_setaffinity, tid,
sizeof(cpu_set_t),mask);
    if (syscallres) {
        err = errno;
        LOGE("Error in the syscall setaffinity: mask = %d,
err=%d",mask,errno);
    }
    LOGD("tid = %d has setted affinity success",tid);
}

JNIEXPORT void JNICALL Java_nativelibs_Affinity_bindThreadToCpu45(JNIEnv
*env, jclass type)
{
    cpu_set_t mask;
```

```
    CPU_ZERO(&mask);
    CPU_SET(4,&mask);
    CPU_SET(5,&mask);
    set_cur_thread_affinity(&mask);
    LOGD("set affinity to %d success");
}
```

linux 下可以直接调用 pthread_setaffinity_np，将当前线程绑定在具体的 cpu 上，而 android 该 API 被屏蔽了，需要调用 sched 这个系统 API。

2、将整个进程绑定到指定 CPU 的 4、5 大核

```
JNIEXPORT int JNICALL Java_nativelibs_Affinity_bindProcessToCpu45(JNIEnv*
env, jobject obj){
    DIR *task_dir;
    struct dirent *tid_dir;
    int pid,tid;
    int ret;
    char filename[64];
    LOGD("BindThreadsToCpu45 : Current platform = rk3399");
    cpu_set_t mask;
    struct sched_param param;
    CPU_ZERO(&mask);
    CPU_SET(4, &mask);
    CPU_SET(5, &mask);
    extern int errno;
// bind whole processId to cpu 4 & 5
    pid = getpid();
    LOGD("pid = %d",pid);
    ret = sched_setaffinity(pid, sizeof(mask), &mask);
    if(ret < 0){
        LOGE("BindThreadsToCpu45 -> bind process sched_setaffinity
return %d, errno = %d\n",ret,errno);
        return -1;
    }else{
        LOGD("BindThreadsToCpu45 -> bind process sched_setaffinity
success!");
    }
    return 0;
}
```

方法四：配置进程名、线程名获得系统自动将线程绑定大核

Affinity 服务

sdk 中添加了 Affinity 框架服务，用来解析并自动绑定客户指定的线程名，通过配置 device/rockchip/rk3399/affinity_conf.xml 文件指定需要绑定哪个进程的哪个线程到大核，系统可以在指定线程启动后，自动将线程绑定到大核，编译后生成 out/target/product/rk3399_all/system/etc/affinity_conf.xml ：

```
<?xml version="1.0" encoding="utf-8"?>
<xml>
    <Affinity usercode="4cb3">
        <key name="com.android.launcher3" value="RenderThread"/>
        <key name="com.android.launcher3" value="hwuiTask1"/>
        <key name="com.android.launcher3" value="hwuiTask2"/>
        <key name="com.android.launcher3" value="mali-utility-wo"/>
    </Affinity>
</xml>
```

（进程名 / 线程名）

如需做快速测试验证，可直接将修改后的 push affinity_conf.xml 到调试机的/system/etc/affinity_conf.xml，运行程序，使用 ps 和 taskset 指令查看线程是否绑定大核：

```
130|rk3399_all:/ # ps |grep launcher3
u0_a38    1275  337   2559332 197868 SyS_epoll_ 7ce5231b80 S com.android.launcher3
rk3399_all:/ # ps -t|grep 1275
u0_a38    1275  337   2559332 197868 SyS_epoll_ 7ce5231b80 S com.android.launcher3
u0_a38    1280  1275  2559332 197868 futex_wait 7ce51e2cf0 S Jit thread pool
u0_a38    1281  1275  2559332 197868 do_sigtime 7ce5231d60 S Signal Catcher
u0_a38    1282  1275  2559332 197868 unix_strea 7ce5232708 S JDWP
u0_a38    1283  1275  2559332 197868 futex_wait 7ce51e2cf0 S ReferenceQueueD
u0_a38    1284  1275  2559332 197868 futex_wait 7ce51e2cf0 S FinalizerDaemon
u0_a38    1285  1275  2559332 197868 futex_wait 7ce51e2cf0 S FinalizerWatchd
u0_a38    1286  1275  2559332 197868 futex_wait 7ce51e2cf0 S HeapTaskDaemon
u0_a38    1287  1275  2559332 197868 binder_thr 7ce5231c70 S Binder:1275_1
u0_a38    1288  1275  2559332 197868 binder_thr 7ce5231c70 S Binder:1275_2
u0_a38    1296  1275  2559332 197868 SyS_epoll_ 7ce5231b80 S launcher-loader
u0_a38    1298  1275  2559332 197868 futex_wait 7ce51e2cf0 S pool-1-thread-1
u0_a38    1300  1275  2559332 197868 futex_wait 7ce51e2cf0 S pool-1-thread-2
u0_a38    1304  1275  2559332 197868 SyS_epoll_ 7ce5231b80 S RenderThread
u0_a38    1305  1275  2559332 197868 futex_wait 7ce51e2cf0 S pool-2-thread-1
u0_a38    1307  1275  2559332 197868 futex_wait 7ce51e2cf0 S mali-mem-purge
u0_a38    1308  1275  2559332 197868 futex_wait 7ce51e2cf0 S mali-utility-wo
u0_a38    1309  1275  2559332 197868 futex_wait 7ce51e2cf0 S mali-utility-wo
u0_a38    1310  1275  2559332 197868 futex_wait 7ce51e2cf0 S mali-utility-wo
u0_a38    1311  1275  2559332 197868 futex_wait 7ce51e2cf0 S mali-utility-wo
u0_a38    1312  1275  2559332 197868 futex_wait 7ce51e2cf0 S mali-utility-wo
u0_a38    1313  1275  2559332 197868 futex_wait 7ce51e2cf0 S mali-utility-wo
u0_a38    1314  1275  2559332 197868 poll_sched 7ce5231ca0 S mali-cmar-backe
u0_a38    1315  1275  2559332 197868 futex_wait 7ce51e2cf0 S mali-hist-dump
u0_a38    1558  1275  2559332 197868 futex_wait 7ce51e2cf0 S hwuiTask1
u0_a38    1559  1275  2559332 197868 futex_wait 7ce51e2cf0 S hwuiTask2
u0_a38    1561  1275  2559332 197868 futex_wait 7ce51e2cf0 S RenderThread
u0_a38    1562  1275  2559332 197868 futex_wait 7ce51e2cf0 S RenderThread
u0_a38    1563  1275  2559332 197868 futex_wait 7ce51e2cf0 S RenderThread
u0_a38    1564  1275  2559332 197868 futex_wait 7ce51e2cf0 S RenderThread
u0_a38    1565  1275  2559332 197868 futex_wait 7ce51e2cf0 S RenderThread
u0_a38    1566  1275  2559332 197868 unix_strea 7ce52326d8 S RenderThread
rk3399_all:/ # task
taskset    taskstats
rk3399_all:/ # taskset -p 1565
pid 1565's current affinity mask: 30
rk3399_all:/ # taskset -p 1308
pid 1308's current affinity mask: 30
rk3399_all:/ # taskset -p 1558
pid 1558's current affinity mask: 30
rk3399_all:/ # taskset -p 1559
pid 1559's current affinity mask: 30
```

（mask为30（即110000）表示线程绑定到大核）

```
ps | grep Launcher3
ps –t | grep 1275
```

如何获取应用包名

输入如下指令，然后运行需要绑定的应用

```
logcat -v time |grep START
```

这里得到 Launcher3 的包名为 com.android.launcher3

如何得到线程名、如何选取需要绑定的线程

一般的，建议绑定占用 cpu 资源较多的线程，运行 app 到特定场景后，参考本文档上述的 simpleperf 查看线程占用 cpu 情况，举例：

```
simpleperf report --sort tid,comm
```

```
^C^Crk3399_all:/data # simpleperf report --sort tid,comm
Cmdline: /system/xbin/simpleperf record -a -g
Samples: 20334 of event 'cpu-cycles'
Event count: 5995117305

Overhead  Tid   Command
61.94%    1500  simpleperf
8.76%     0     swapper
4.38%     1257  RenderThread
4.17%     1225  droid.launcher3
3.04%     235   surfaceflinger
2.14%     260   surfaceflinger
1.88%     1268  mali-cmar-backe
1.36%     628   SensorService
1.25%     244   Binder:235_2
```

可以看到 Launcher3 主线程、RenderThread、surfaceflinger 等线程占用 cpu

Tips：

- RenderThread、wuiTask、mali-utility-wo 是与绘图相关的线程，可以先绑定上之后，根据实际测试表现来决定是否绑定大核。
- 若需绑定多个同名的线程，可以只加线程前缀，本例中有 hwuiTask1、hwuiTask2，可在配置文件中只写 hwuiTask 即可。

方法五：设置线程优先级

POSIX 标准指定了三种调度策略：先入先出策略 (SCHED_FIFO)、循环策略 (SCHED_RR) 和自定义策略 (SCHED_OTHER) 。SCHED_FIFO 和 SCHED_RR 属于

RT 线程（实时线程），Android 有很严格的系统权限限制，上层的线程是无法设置为 RT 线程，可设置为普通线程中的高优先级线程，普通线程的最高优先级线程接近于 RT 线程。Android 线程优先级设置方法下面也会介绍。

API 函数：int sched_setscheduler(pid_t pid, int policy,

const struct sched_param *param);

函数将 pid 所指定进程的调度策略和调度参数分别设置为 policy 和 param 指向的 sched_param 结构中指定的参数。sched_param 结构中的 sched_priority 成员的值可以为任何整数，该整数位于 policy 所指定调度策略的优先级范围内(含边界值)。policy 参数的可能值在头文件<sched.h>中定义。

简单例子：

```
struct sched_param param = {
        .sched_priority = 90,
  };
  sched_setscheduler(0, SCHED_FIFO, &param);  //第一个参数为 pid，0 表示为当前进程
```

网上实例：https://blog.csdn.net/allwtg/article/details/5254306

命令行方式：

```
rk3399_mid:/ # busybox chrt -p 674
pid 674's current scheduling policy: SCHED_OTHER
pid 674's current scheduling priority: 0
1|rk3399_mid:/ # busybox chrt -f -p 90 674
pid 674's current scheduling policy: SCHED_OTHER
pid 674's current scheduling priority: 0
pid 674's new scheduling policy: SCHED_FIFO
pid 674's new scheduling priority: 90
```

android 系统在进程创建的时候实际上也有不同的优先级可以指定，frameworks/base/core/java/android/os/Process.java 中定义了下面这些优先级：

```
public static final int THREAD_PRIORITY_DEFAULT = 0;  应用的默认优先级
public static final int THREAD_PRIORITY_LOWEST = 19;  线程的最低优先级
```

public static final int THREAD_PRIORITY_BACKGROUND = 10; 后台线程的默认优先级

public static final int THREAD_PRIORITY_FOREGROUND = -2; 前台进程的标准优先级

public static final int THREAD_PRIORITY_DISPLAY = -4; 系统用于显示功能的优先级

public static final int THREAD_PRIORITY_URGENT_DISPLAY = -8; 系统用于重要显示功能的优先级

public static final int THREAD_PRIORITY_AUDIO = -16; 音频线程默认优先级

public static final int THREAD_PRIORITY_URGENT_AUDIO = -19; 重要音频线程默认优先级

这些优先级的设定，通过层层调用，实际上最终还是由 sched_setscheduler 执行。

简单的例子（如下红色字体）：

```
@@ -3,6 +3,7 @@ package com.google.vrtoolkit.cardboard.sensors;
 import android.hardware.*;
 import java.util.*;
 import android.os.*;
+import android.os.Process;
 import android.util.*;

 public class DeviceSensorLooper implements SensorEventProvider
@@ -49,7 +50,7 @@ public class DeviceSensorLooper implements
SensorEventProvider
                     }
                 }
             };
-        final HandlerThread sensorThread = new HandlerThread("sensor")
{
+       final HandlerThread sensorThread = new HandlerThread("sensor",
Process.THREAD_PRIORITY_URGENT_AUDIO) {
```

```
                protected void onLooperPrepared() {
                    final Handler handler = new Handler(Looper.myLooper());
                    final Sensor accelerometer =
DeviceSensorLooper.this.sensorManager.getDefaultSensor(1);
```

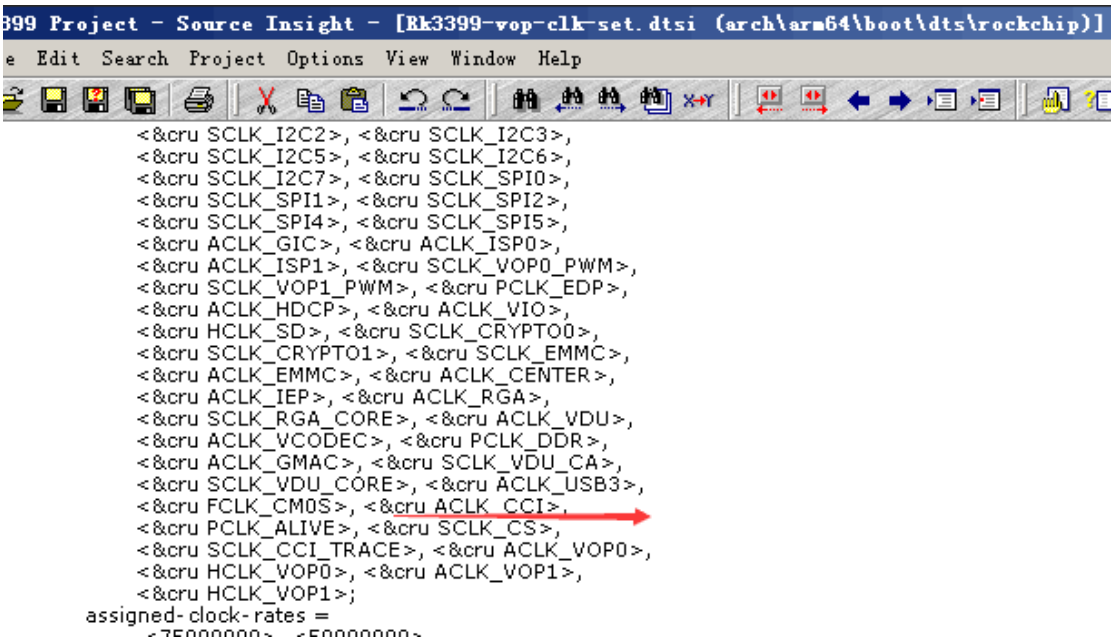
方法六：提高 CCI 频率

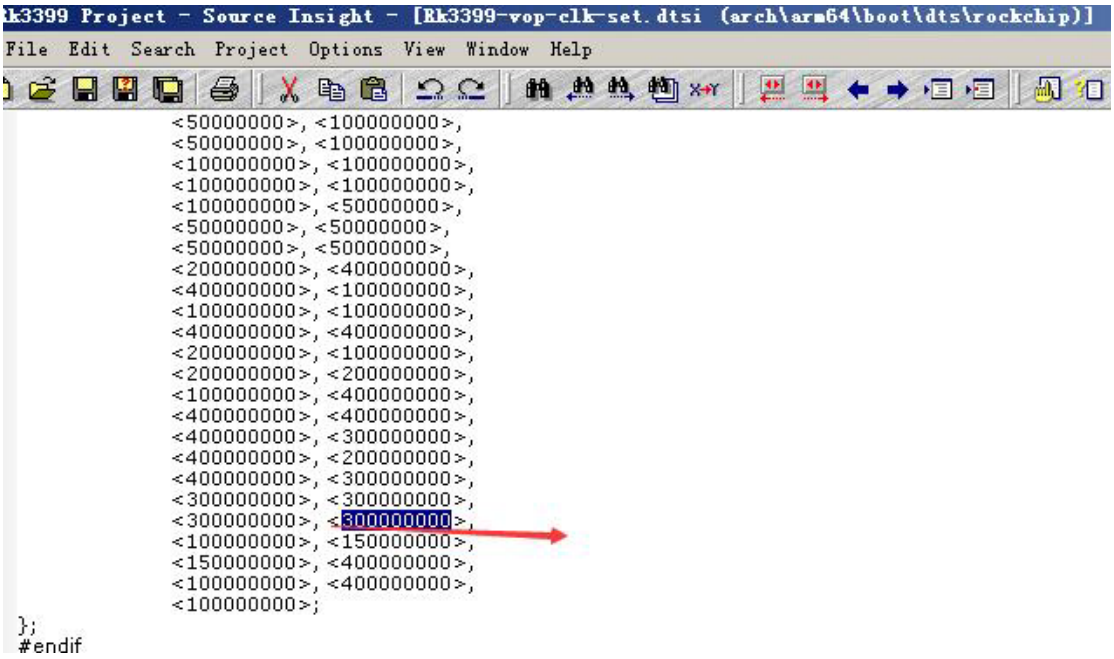
如果关注 ARM 的 DDR 吞吐率，可以提高 CCI 频率来提升，也就是说对于那些瓶颈在 memory bandwidth 的应用，需要把 cci 的频率提高到和 ddr 一样，例如，目前默认配置下 ddr 频率跑 800M，cci 频率是 300M，把 cci 改成 800M 则可以显著提高 ddr 的吞吐率。

CCI 频率修改方法：

```
--- a/arch/arm64/boot/dts/rockchip/rk3399-vop-clk-set.dtsi
+++ b/arch/arm64/boot/dts/rockchip/rk3399-vop-clk-set.dtsi
@@ -171,7 +171,7 @@
                        <400000000>, <200000000>,
                        <400000000>, <300000000>,
                        <300000000>, <300000000>,
-                       <300000000>, <300000000>,
+                       <300000000>, <800000000>,
                        <100000000>, <150000000>,
                        <150000000>, <400000000>,
                        <100000000>, <400000000>,
```

对应关系：





方法七：Touch Boost 功能

如果客户比较关注的是点击或者滑动时的瞬时响应，可以考虑加入 Touch Boost 功能，该功能会在有用户进行触摸操作（或者鼠标操作） 动作时，触发 cpu 的抬频动作，并且持续一段时间。SDK 默认带了该功能，只是在 Boost 时，只对 cpuL（也就是 cpu 小核）进行抬频，并且抬升频率只到 1.2G，下面的内核补丁可以让 cpu 大核以及小核的频率抬升到最高频，并且每次抬频动作持续 500ms，500ms 之后再根据负载进行变频：

```
diff --git a/drivers/cpufreq/cpufreq_interactive.c b/drivers/cpufreq/cpufreq_interactive.c
    index 182b31c..873cccb 100644
    --- a/drivers/cpufreq/cpufreq_interactive.c
    +++ b/drivers/cpufreq/cpufreq_interactive.c
    @@ -1304,9 +1304,11 @@ static void rockchip_cpufreq_policy_init(struct
cpufreq_policy *policy)
            if (policy->cpu == 0) {
                    tunables->hispeed_freq = 1008000;
                    tunables->touchboostpulse_duration_val = 500 * USEC_PER_MSEC;
    -               tunables->touchboost_freq = 1200000;
    +               tunables->touchboost_freq = 1416000;
            } else {
                    tunables->hispeed_freq = 816000;
    +                 tunables->touchboostpulse_duration_val = 500 * USEC_PER_MSEC;
    +                 tunables->touchboost_freq = 1800000;
            }

            index = (policy->cpu == 0) ? 0 : 1;
```

注意，上面这个策略会引起功耗升高，但是对于需要瞬时响应的场景，会有很明显的优化。