北京大学
PEKING UNIVERSITY

Android内核打补丁示例+期末报告模板

荆琦

期末报告文档

一、基本原理分析
二、主要数据结构概述
三、代码分析
3.1 代码概述
3.1.1 函数调用分析
3.1.2 关键函数分析
3.1.3 相关影响分析
3.2 代码行注释
四、分析结果验证
五、问题及解决

期末报告文档—基本原理分析

特性原理分析

- 1、若是特性提供了参考论文，首先阅读该论文，然后阅读相关参考论文。力求通过这些论文的阅读，达到对特性功能和架构的理解。这些论文是最权威也是最贴近特性本质的。

- 2、利用网络的便利，使用谷歌学术、百度学术、北大图书馆论文搜索等工具搜索特性相关论文和书籍，并通过概述段落进行快速筛选，留下相关度高的论文。再通过快速浏览，进一步筛选。最后对筛选出的论文做详细阅读，记录下有帮助的部分，进行翻译、编辑或修改。绝不是拿来主义，一定是仔细筛选的。

- 3、通过对相似度较高的特性或技术的理解来理解当前特性，比如通过flashcache和dm-cache都是在设备映射器的下层，他们都要实现设备映射器提供的接口，都是通过设备映射器向上层提供服务，理解flashcache对理解dm-cache的帮助非常大。

期末报告文档—代码分析

- 代码分析
  - 1、去内核源码的Documentation文件夹下查找关于本特性的描述文件，这里面有关于特性的概述和分模块的介绍，还包含一些重要数据结构的描述，对理解源码帮助非常大。
  - 2、充分利用特性的commit页面。每个commit都是相对独立的功能模块，commit页面上会提供对本次commit的描述，提供commit涉及的文件及各文件变更的代码行数，以及代码的具体变更。
  - 3、利用一些源码分析工具。
  - 4、利用网络资源，使用百度、谷歌等搜索工具，查找网上对该特性的代码分析，仔细阅读，与源码进行印证，绝不照搬搜索结果，保证都是经过思考和印证的。
  - 5、查找及阅读Mailing List中关于patch的相关信息

期末报告文档——代码分析——函数调用分析

- 在函数调用关系图这一节中，画出特性的整体流程图并附加概要性说明
- 有时代码量很大，如Bcache特性的引入有超过一万行的代码改动，涉及数十个文件，则可以多层次绘制流程图。

期末报告文档——代码分析——函数调用分析

- 对于流程较为集中，结构清晰的代码，绘制其主线函数的函数流程图。这样流程图已经能表达特性主要逻辑结构，同时也能从中看出特性实现时考虑的细节。

- 对于代码量较大，逻辑较为分散的情况，可以绘制模块级别的调用逻辑和模块间的关联性。模块级的流程图与子系统视角的整体流程图相比，说明特性内部的逻辑和结构关系，同时又对代码逻辑做了抽象处理，避免代码中浩如烟海的逻辑细节。

- 对于新特性的代码分析，还可以进一步考虑特性的代码变化是以增量为主还是修改为主。 对于增量为主的代码，采用上述的方法；对于修改部分较多的特性，可以在流程图中对改变前后的逻辑流程进行对比，也在说明部分详细描述其中的差异。

期末报告文档——代码分析——关键函数分析

梳理关键函数的原则：

（1）实现某一特性的**入口函数**；

（2）**关键功能点**的实现所涉及的函数；

（3）该特性与其他模块**交互**所涉及的函数；

期末报告文档——代码分析——关键函数分析

在找出关键函数后，对其进行详细的分析，分析关键函数的步骤如下：

（1）首先总体把握关键函数实现的功能，并逐条列出这些**功能点**；

（2）其次对关键函数的具体实现给出详细的**注释**；

（3）最后对关键函数中**调用**的来实现其功能点的函数给出简要说明。

期末报告文档——代码分析——相关影响分析

在代码所影响的功能模块这一小节，主要写如何通过以下分析找到代码所影响的功能模块，并说明对这些模块有哪些影响。

- 从功能影响的角度：通过查阅书籍，查找作者论文论文、社区讨论等方式从架构上来理解其实现机制，帮助找到其可能影响的功能模块。
- 从代码级分析的角度：在理解机制的基础上，利用已有的源码分析工具，可找到其静态的调用关系，从而找到其影响到的功能模块；对于动态调用关系，通过阅读源代码、做实验的方式可以找到其指针的变动，从而找到特性所影响的功能模块，并分析其影响的内容。

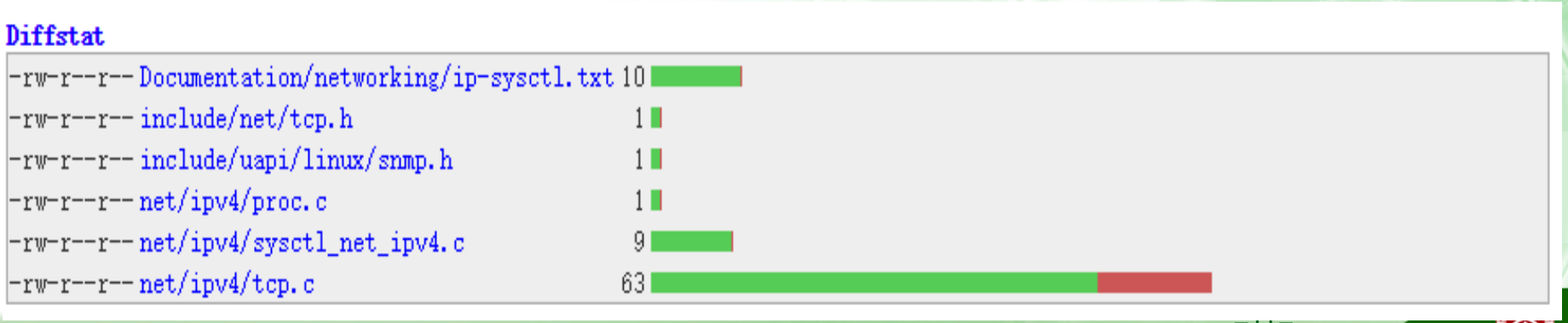
期末报告文档——特性分析文档

特性分析 - Autocorking

Autocorking特性介绍：

1. Autocorking是3.14内核引入的新特性，通过在tcp层合并小包提升系统性能。
2. Autocorking和Nagle都是通过合并小包提高性能，但是他们有所不同。
   - Nagle算法是在发送队列中有一个包没有收到ack的时候持续合并小包。
   - Autocorking是在发送队列中还有包未发送时在tcp层和并小包。

特性更新与迭代情况：3.14版本时被加入主线内核。



特性分析 - Autocorking

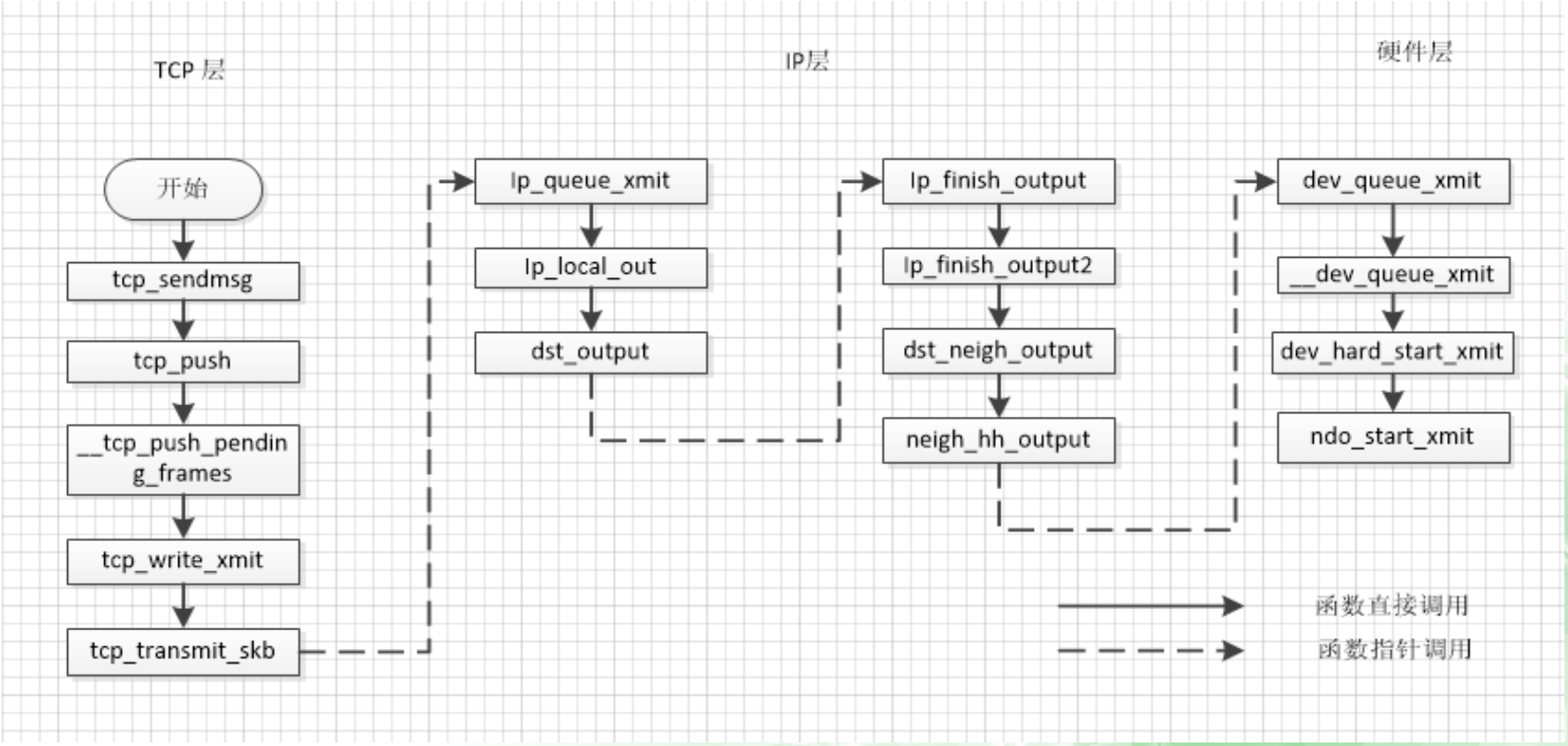
代码差异

1. 接口变化
   - 增加系统调用**tcp_autocorking**来使能此特性。
   - 增加**SNMP**计数器来统计调用次数。
   - 修改**tcp_push**函数，增加参数**size_goal**。
   - 增加**tcp_should_autocork**函数用来判断是否该合并小包。
2. 代码变化举例

```
+static bool tcp_should_autocork(struct sock *sk, struct sk_buff *skb,
+                                int size_goal)
+{
+       return skb->len < size_goal &&
+              sysctl_tcp_autocorking &&
+              atomic_read(&sk->sk_wmem_alloc) > skb->truesize;
+}
```

特性分析- Autocorking

网络系统调用流程



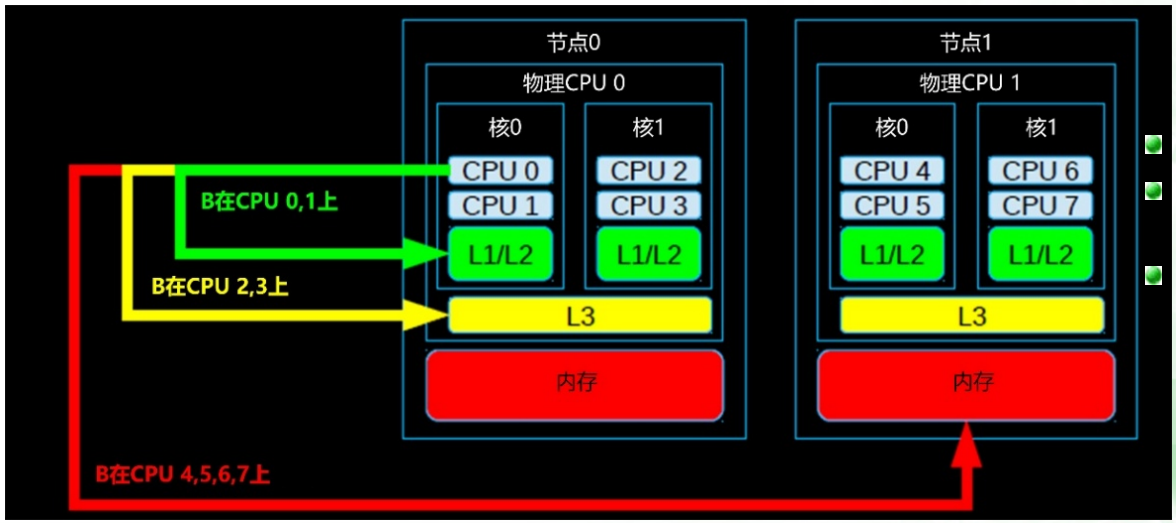
特性分析- Autocorking

Autocorking总结：

TCP Autocorking是在3.14内核中加入主线的特性之一，这个补丁通过在一定情况下阻塞TCP层的数据包向下一层发送，使得应用有更多的机会去合并小数据包，这样减少了信道中网络包的数量，并减少了系统发包的次数，对手机cpu性能和手机网络性能都有一定的提升。

特性分析--Wake affine新特性

- 相关术语
  - 唤醒（wakeup）：在运行队列中的进程唤醒不在运行队列中的进程
  - Waker：正在运行的进程，它会试图唤醒不在运行的进程
  - Wakee：不在运行的进程，等待被唤醒
  - 相邻CPU（neighbor CPU）：属于一块相同的物理CPU或者属于一块物理CPU上相同的核的逻辑CPU
  - 相关进程（related process）：有共享数据关系的进程
- Linux 2.6，提出了CPU wake affine的概念
  - 使得相关进程能够运行在相邻CPU上，进而能够高效的使用Cache，提升Cache的利用率，提高程序执行的速度



两个相关进程A和B，进程A在CPU 0上运行

进程B在物理CPU 1上运行时，进程B不能获得任何cache上的数据

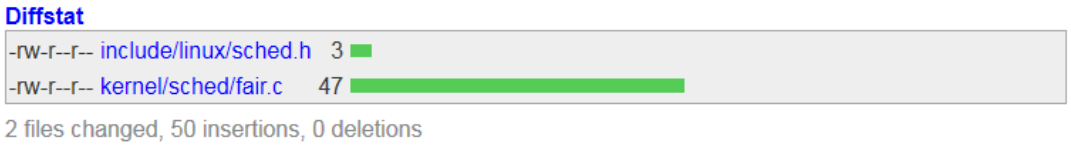
进程B运行在物理CPU 0上时，进程B会不同程度地命中cache中的数据，从而提高进程的运行速度。

Wake affine特性迭代与更新

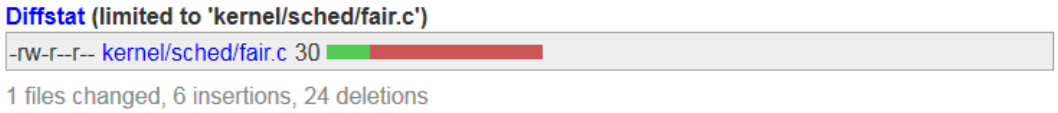
该特性共计修改4个文件，200行代码

12年8月：在select_task_rq_fair函数中恢复SD_WAKE_AFFINE选项，同时清理一些过时代码

13年7月份：实现更智能的wake affine逻辑，针对进程关系为1：N的情况进行了优化，提高了该情况下的系统性能

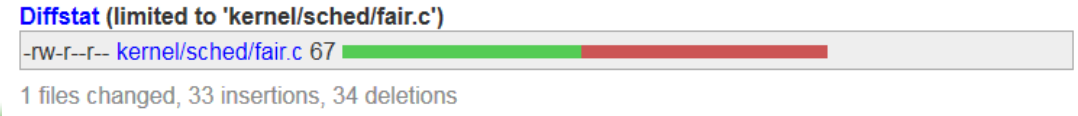


14年9月：删除wake affine中的一种条件判断，这些额外的情况并不会给系统带来任何益处。

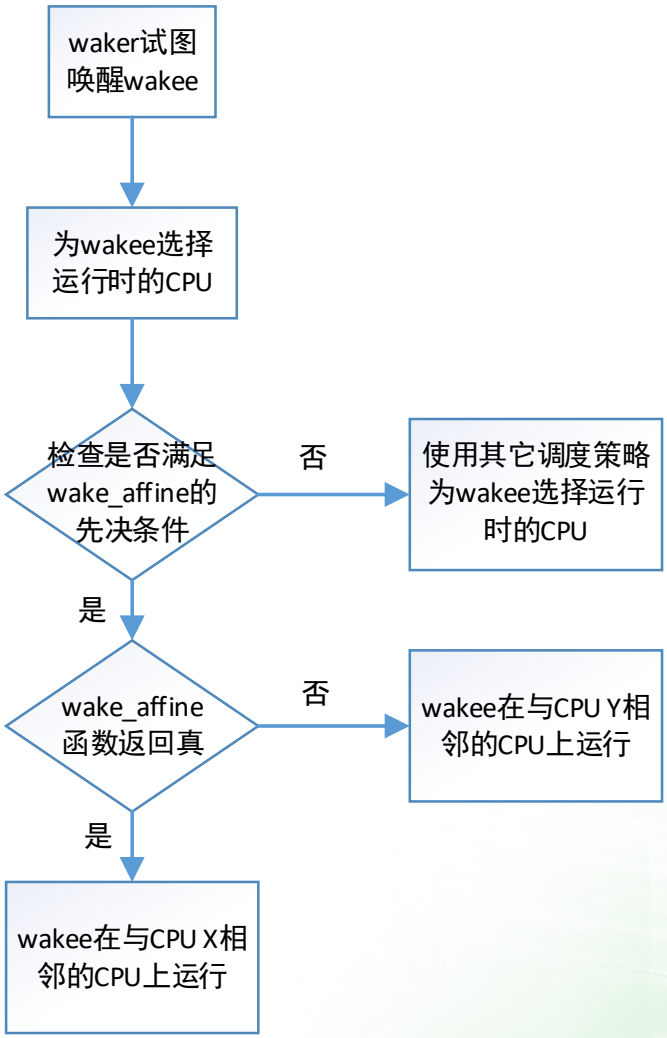


14年9月：在wake_affine中测试CPU的性能，当中断的数量或者花在中断上的时间很关键时，这个patch会使系统收益

15年8月：完善wake_wide()函数，改善进程关系为1：N时系统的性能



Wake affine功能及实现原理

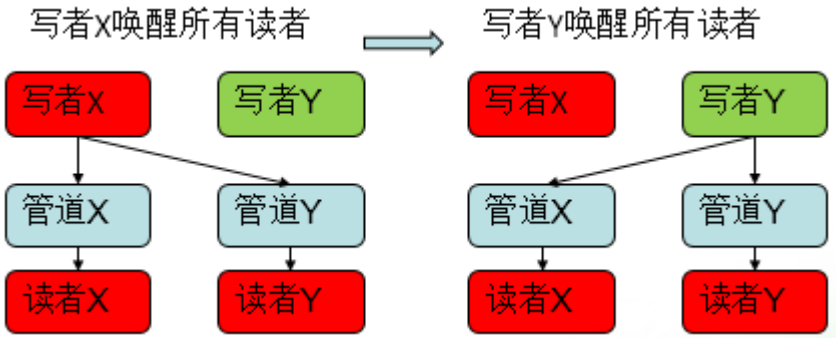


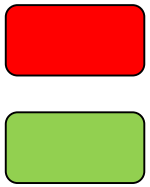
Wake affine特性生效的条件：

- Waker和wakee是相关进程
- Waker目前正在CPU X上运行
- Wakee上次是在CPU Y上运行
- Waker试图唤醒wakee，并为wakee选择运行时的CPU

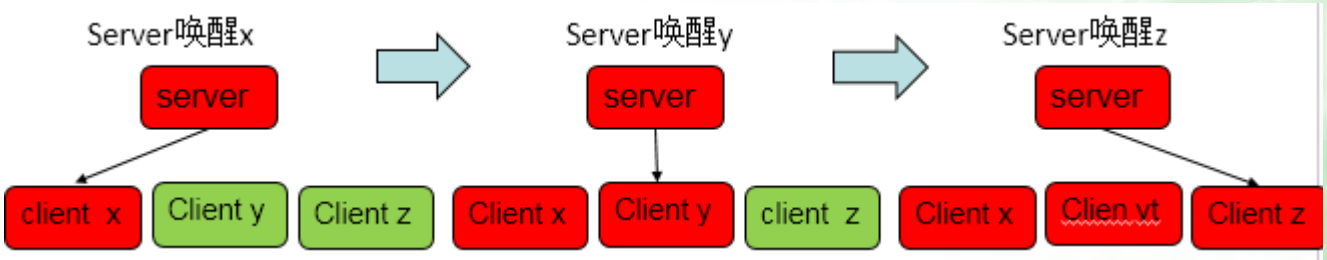
以前版本中Wake affine的缺陷

进程关系为M：N时：





进程关系为1：N时：



分析：

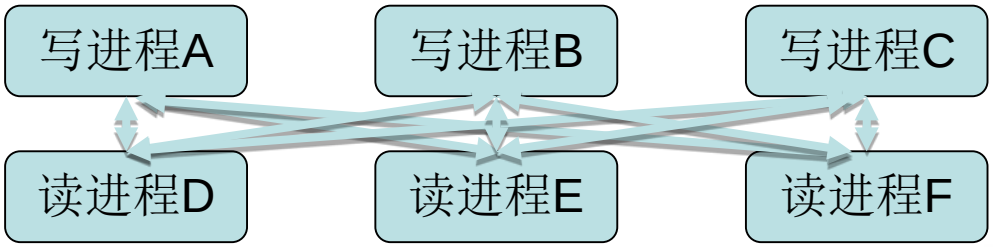
1：N的情况下使用wake affine可能导致进程饥饿

Server饥饿会导致它所有的client饥饿，从而影响系统的性能

改进后的Wake affine策略

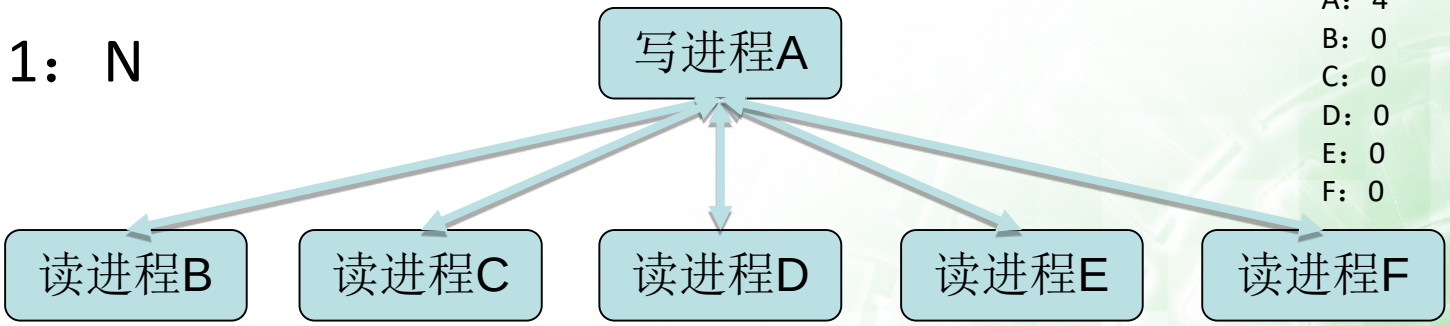
下图演示了在M：N与1：N 两种情况下进程切换次数的不同，依此为判断依据，对wake affine进行修改

N：N



一轮操作后的切换次数
A：2
B：2
C：2
D：2
E：2
F：2

1：N



一轮操作后的切换次数
A：4
B：0
C：0
D：0
E：0
F：0

该新特性主要由王赟（Michael Wang）提出，并正式加入到linux 3.12内核版本中。使用pgbench测试显示，性能可以提升40%，参考链接如下：
http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=62470419e993f8d9d93db0effd3af4296ecb79a5
http://www.infoq.com/cn/presentations/wake-affine-scheduler-properties

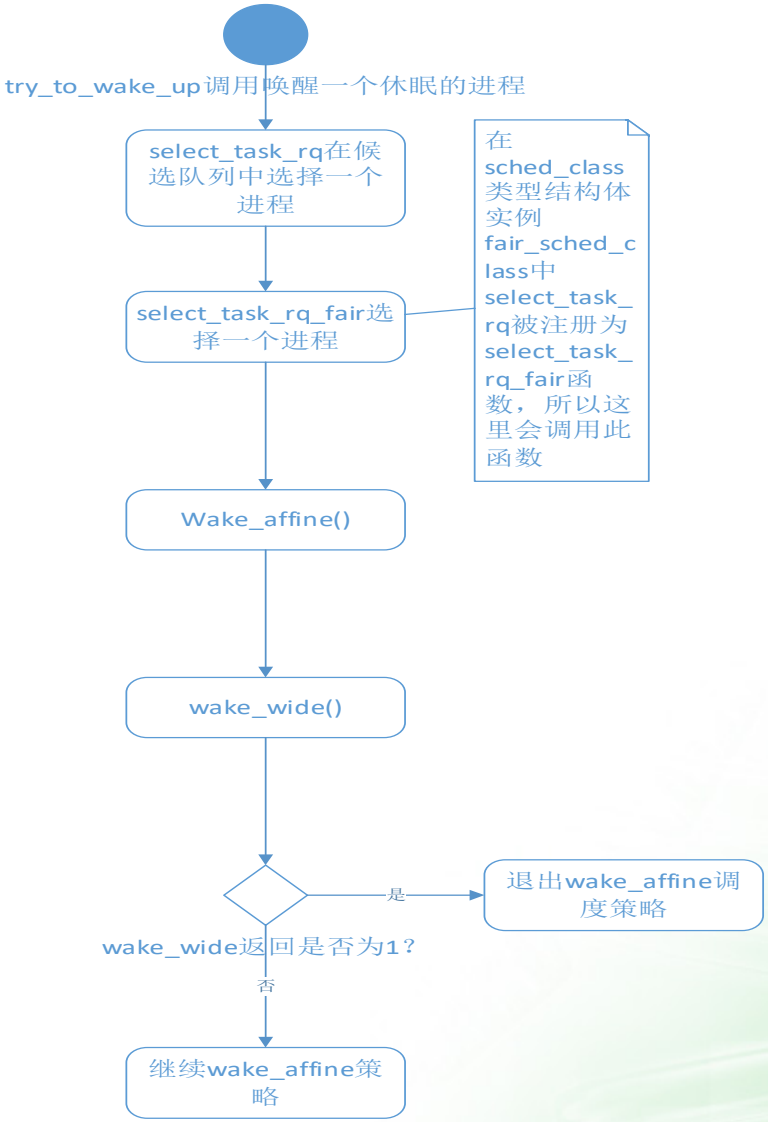
特性接口变化

进程的**task_struct结构体的修改**，在该结构体中增加了3个成员变量。该结构体位于include/linux/sched.h中，因为这种修改是基于SMP架构的，所以把该成员变量放在#ifdef CONFIG_SMP和#endif之间

```
struct task_struct {
#ifdef CONFIG_SMP
struct llist_node wake_entry;
int on_cpu;
+ struct task_struct *last_wakee;   /*记录上次唤醒该进程的进程*/
+ unsigned long wakee_flips;      /*记录进程状态切换的次数*/
+ unsigned long wakee_flip_decay_ts;
#endif
int on_rq;
```

- 在kernel/sched/fair.c中**增加函数record_wakee**，函数的功能是更新当前正在运行的进程的状态切换次数，上次被唤醒时的进程以及该进程状态切换衰退的时间
- 在kernel/sched/fair.c中增加函数wake_wide。函数的功能是判断该进程是否需要执行wake affine的逻辑。

函数流程图



在linux 3.12内核版本中，在wake_affine函数开始处添加wake_wide函数，根据wake_wide函数的返回结果判断要唤醒的进程是否要执行wake affine的逻辑。

关键函数分析

record_wakee函数：更新当前正在运行的进程的状态切换次数，上次被唤醒时的进程以及该进程状态切换衰退的时间。

```
+static void record_wakee(struct task_struct *p)
+{
+ /*jiffies为记录系统启动以来的时钟数，初始值为0，每次计时器中断，该变量加1
+*HZ：1秒钟有jiffies个HZ
+*如果当前时间大于当前进程状态切换的衰退时间加HZ，那么更新当前进程的状态切换次数以及状态切换衰退时间
+*/
+ if (jiffies > current->wakee_flip_decay_ts + HZ) {
+ current->wakee_flips = 0;
+ current->wakee_flip_decay_ts = jiffies;
+ }
+/*如果当前进程的上次被唤醒的进程跟当前要唤醒的进程不同，更新当前进
+*程的上次被被唤醒进程，同时当前进程的状态切换次数加1
+*/
+ if (current->last_wakee != p) {
+ current->last_wakee = p;
+ current->wakee_flips++;
+ }
+}
```

wake_wide函数：判断该进程是否需要执行wake affine的逻辑。

```
+static int wake_wide(struct task_struct *p)
+{
+/*factor为当前CPU的逻辑CPU个数*/
+ int factor = this_cpu_read(sd_llc_size);
+
+ /*
+ * 如果要唤醒的进程的状态切换次数多于当前CPU的逻辑CPU个数
+ */
+ if (p->wakee_flips > factor) {
+ /*
+*Wake也许调度很频繁，这时它需要一些CPU资源，因此，如果waker切换+*更频繁，这是wakee不再走wake affine的逻辑
+ */
+ if (current->wakee_flips > (factor * p->wakee_flips))
+ return 1;
+ }
+
+ return 0;
+}
+
```


Wake affine特性总结

- 在SMP系统架构中，使用wake affine这种调度策略能够使得有亲近关系的进程能够运行在同一处理器上，进而能够高效的使用Cache，提升Cache的利用率，加快程序执行的速度。
- 然而这种方式有一定缺陷，如果指定的CPU负载已经很忙，还给它增加负载时，将得不到很好的收效。这种情况在进程关系是1:N的情况下会更加明显。
- 该特性实现了一种更智能的wake affine策略，当进程关系为1：N时会对系统性能有10~40%的提升，同时，保留了进程关系为M：N时wake affine这一特性带来的10%的性能提升。
- 该特性总体代码修改情况比较集中，但是对多核系统的性能提升很多，从linux 3.12开始正式加入到内核主线中，为多核系统进程调度提供了一种很好的参考方案。

内核新功能升级补丁的移植、验证和优化效果测试

补丁验证实验

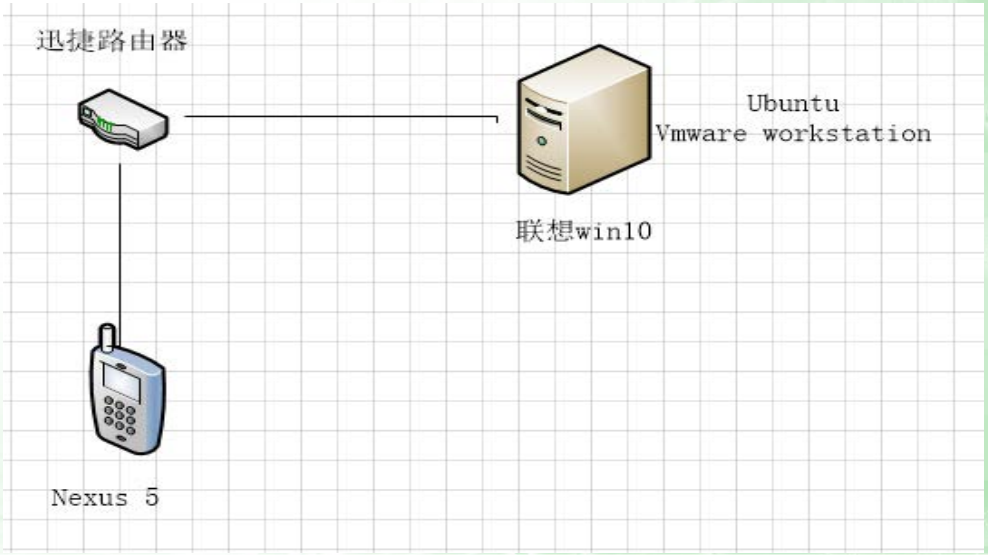
实验目的

实验的目的是在Android Nexus 5手机上打上Autocorking补丁，并验证这个补丁对手机在大量发送小包的情况下的cpu性能改善情况

实验环境

1. Android手机，型号：Nexus 5; Android版本：4.4.4；内核版本：3.4。
2. 虚拟机：Vmware Workstation 12.0.1 操作系统：Ubuntu-14.4.2
3. 联想Y40笔记本，win10系统
4. Fast迅捷FWR100 150M无线路由器。

实验环境部署图



补丁实验 -实验准备

SDK安装

1. 下载SDK的LINUX版本

2. 配置环境变量

Export PATH=$PATH:/path/to/sdk/platform-tools

3. 调用adb可验证sdk安装是否成功

补丁验证实验

补丁实验 – 实验步骤

1. 从以下网址获得**autocorking**补丁

   https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/patch/?id=f54b311142a92ea2e42598e347b84e1655caf8e3

2. 为内核打补丁

```
    SNMP_MIB_ITEM("TCPAutoCorking",LINUX_MIB_TCPAUTOCORKING),
```

```
Hunk #1 FAILED at 279.
1 out of 1 hunk FAILED -- saving rejects to file net/ipv4/proc.c.rej
patching file net/ipv4/sysctl_net_ipv4.c
Hunk #1 succeeded at 677 with fuzz 1 (offset -56 lines).
```

补丁实验 – 实验步骤

- Patch要求在tcp.h的第282行加入如下代码

```
extern int sysctl_tcp_autocorking
```

实际上应该在tcp.h的第254行增加此代码

- Patch要求在proc.c的第279行加入如下代码

```
SNMP_MIB_ITEM("TCPAutoCorking",LINUX_MIB_TCPAUTOCORKING),
```

实际上应该在文件的261行加入上述代码

补丁验证实验

补丁实验 -实验步骤

3. 需要修改部分代码才能让编译通过

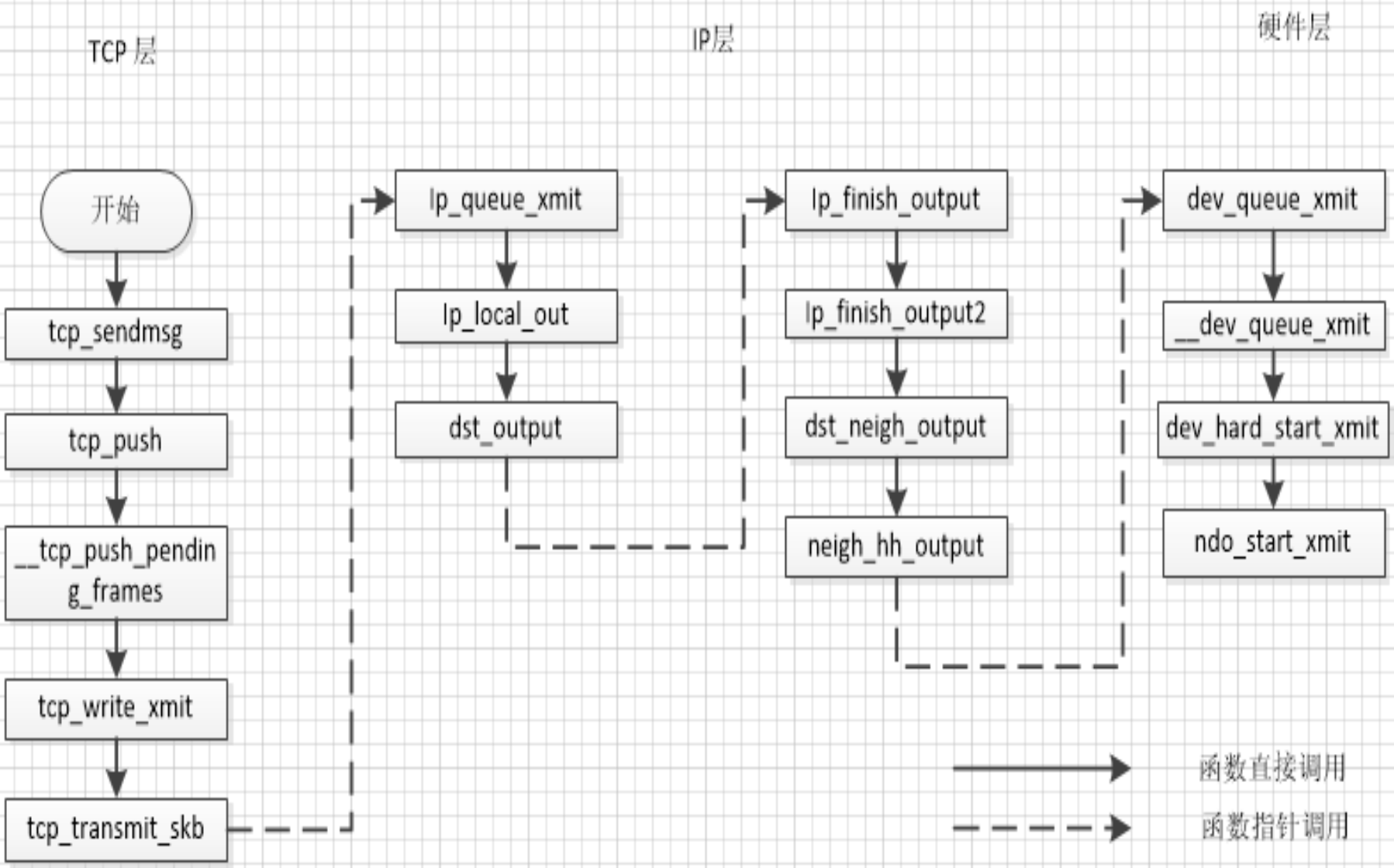
```
net/ipv4/tcp.c:607:31: error: 'TSQ_THROTTLED' undeclared (first use in this function)
net/ipv4/tcp.c:607:31: note: each undeclared identifier is reported only once for each function it appears in
net/ipv4/tcp.c:607:49: error: 'struct tcp_sock' has no member named 'tsq_flags'
```

查看源码发现，这个判断里面会判断tp的某个标志位，并调用NET_INC_STATS对autocorking的调用次数进行了统计，而在android系统和其后验证过程中，并不需要用到这个统计值，也没有相关子系统。所以经过研究之后，将这一段代码删掉，即可成功编译。

补丁实验 -实验步骤

4. 需要修改部分代码才能让补丁使能

修改/net/core/dev.c文件中的dev_hard_start_xmit函数，删除对skb->orphan的调用，此补丁才能正常发挥作用。分析代码可以发现在dev_hard_start_xmit函数中将具体的网络包向具体的网络设备传输之前，会调用skb->orphan清空sk数据结构和skb数据结构的联系，这样tcp层就无法通过sk知道发送队列中是否还有网络包。因此删除这个调用，才能正确使用autocorking特性。在3.14中，此调用已经删除。


TCP 层
IP层
硬件层
开始
tcp_sendmsg
tcp_push
__tcp_push_pending_frames
tcp_write_xmit
tcp_transmit_skb
Ip_queue_xmit
Ip_local_out
dst_output
Ip_finish_output
Ip_finish_output2
dst_neigh_output
neigh_hh_output
dev_queue_xmit
__dev_queue_xmit
dev_hard_start_xmit
ndo_start_xmit
函数直接调用
函数指针调用

补丁验证实验

补丁实验 -实验步骤

5. 设置交叉编译环境变量

- 由于**android**是**arm**架构，和常用的**x86**指令集不同，我们需要交叉编译之后才能获得在**arm**架构下能够正确运行的内核。
- 按照华为内核中 README_Kernel.txt的要求，从googlesource.com上下载交叉编译环境 aarch64-linux-android-gcc-4.9。设定交叉编译时的环境变量 $PATH 和 $CROSS_COMPILE:

```
export PATH=$PWD/bin:$PATH
export ARCH=arm
export SUBARCH=arm
export CROSS_COMPILE=arm-eabi-
```

补丁实验 -实验步骤

6. 编译

运 行 **make hammerhead_defconfig** 指 定 目 标 **android**机型

调用**make**命令对内核进行编译

内核新功能升级补丁构建

6. 编译——问题

1. 缺少头文件 /include/linux/netfilter/xt_mark.h。头文件在3.10主线内核中不存在，最后一次出现在3.5版本主线内核中。加入了3.5内核中的 xt_mark.h

2. 缺少文件 /net/netfilter/xt_hl.c。目录下有文件xt_HL.c，将其改名后继续编译

补丁验证实验

补丁实验 -实验步骤

7. 包装内核文件

先将官方的**boot.img**解压，得到**bootimg.cfg, zImage-dtb, initrd.img**三个文件。将**zImage-dtb**替换成编译之后的内核，然后调用

abooting –create boot.img –f bootimg.cfg –k zImage-dtb –r initrd.img

命令将其重新打包成**boot.img**

补丁实验 -实验步骤

8. 刷机

运行**adb reboot bootloader**让手机进入**bootloader**模式
调用**fastboot boot boot.img**将内核刷入手机

补丁验证实验

补丁实验 -实验步骤

9. 验证内核是否能够运行

需要验证手机是否能够正常开机

10. 验证是否调用了**autocorking**特性

在 **tcp_push** 函数中增加 **printk** 函数，如果 **tcp_should_autocork**判断为真，就调用**printk**打印信息。

printk(“<0>””guoziao++++++++ autocorking autocorking autocorking at last !);

补丁验证实验

补丁实验 -实验结果与分析

```
<0>[ 1381.767928] guoziao++++++++  in tcp_push skb->len is 57 size goal is 1448
<0>[ 1381.768120] guoziao++++++++  in tcp_push sk_wmem_alloc is 2241 skb->truesize
<0>[ 1381.768266] guoziao++++++++  autocorking autocorking autocorking at last !
<0>[ 1381.768456] guoziao++ tcp_write_xmit before while 1
<0>[ 1381.768573] guoziao++ tcp_write_xmit before tcp_transmit_skb 1
<0>[ 1381.768739] guoziao++ tcp_transmit_skb before icsk->icsk_af 2241
```

带有补丁的内核已经构造成功，并且**autocorking**特性已经成功发挥作用。

补丁验证实验

性能实验

实验环境

1.Perf：perf是linux进行性能测试的常用工具，为了使perf能够在Android内核上使用，我们需要进行交叉编译。

a.下载NDK 9c版本，并设置环境变量

```
export NDK=/path/to/android-ndk
export NDK_TOOLCHAIN=${NDK}/toolchains/arm-linux-androideabi-4.6/prebuilt/linux-x86/bin/arm-linux-androideabi
export NDK_SYSROOT=${NDK}/platforms/android-9/arch-arm
```

补丁验证实验

性能实验

实验环境

1.Perf： perf是linux进行性能测试的常用工具

b.使用如下命令编译perf工具

```
make ARCH=arm
CROSS_COMPILE=${NDK_TOOLCHAIN}
CFLAGS=”—sysroot=${NDK_SYSROOT}”
```

c. 将perf安装到手机上

```
adb push perf /data/perf
```

设置环境变量**PERF_PAGER=cat**即可使用**perf**

补丁验证实验

性能实验

实验环境

2. netperf： netperf是linux上测试网络的工具之一，要运行此工具需要有一个服务端和客户端。在电脑上运行netperf服务端软件作为服务器。之后在adb shell中使用./netperf -t TCP_STREAM -H IP -- -m 128指定向服务端发送分组大小为128的tcp包，发送时长为10秒。

补丁验证实验

性能实验

实验环境

3. test_perf ：test_perf是实验人员自己编写的测试脚本，为了增大实验压力，我们同时创建了5个子进程调用netperf客户端对服务器发送tcp包进行测试。

```
#!/system/bin/sh

i=6
#read i < test
while [ $i -gt 1 ]
do
{
    i=$(($i-1))
}
{
    echo $i

    ./netperf -t TCP_STREAM -H 172.27.35.1 -- -m 128
}&
done
wait
echo 1
```

补丁验证实验

性能实验 -实验步骤

1. 在电脑上运行netperf的服务器端软件netserver.exe作为服务器，并修改手机上的test_perf中服务器的ip地址。

2. 将没有补丁的内核/有补丁的内核分别刷入手机中

3. 进入adb shell中，进入data目录，运行命令./perf stat ./test_perf进行实验，十秒之后我们会观测到实验数据，其中会观测到cpu使用率。

4. 重复上述实验，获得统计结果

补丁验证实验

性能实验 -实验结果与分析

在本次实验中，我们以**cpu**使用率作为**cpu**性能的主要指标。下面是某次实验的测试结果，其中**0.077 CPUs**代表**cpu**的使用率，是通过**task-clock-msecs**的值除以**seconds time elapsed**得到的比值。

```
Performance counter stats for './test_perf':

        903.151726  task-clock-msecs         #      0.077 CPUs
              1118  context-switches         #      0.001 M/sec
                 2  CPU-migrations           #      0.000 M/sec
              2342  page-faults              #      0.003 M/sec
         587020076  cycles                   #    649.968 M/sec
         309268530  instructions             #      0.527 IPC
          36999450  branches                 #     40.967 M/sec
           1246100  branch-misses            #      3.368 %
     <not counted>  cache-references
     <not counted>  cache-misses

      11.715969787  seconds time elapsed
```

补丁验证实验

性能实验 -实验结果与分析

补丁前	补丁后	差值	比值
0.089	0.085	0.004	4.5%
0.093	0.079	0.014	15.1%
0.083	0.070	0.013	15.7%
0.079	0.085	-0.006	-7.6%
0.050	0.078	-0.028	-56.0%
0.082	0.062	0.020	23.4%
0.094	0.082	0.012	12.8%
0.111	0.079	0.032	28.8%
0.097	0.080	0.017	17.5%
0.110	0.096	0.014	12.7%

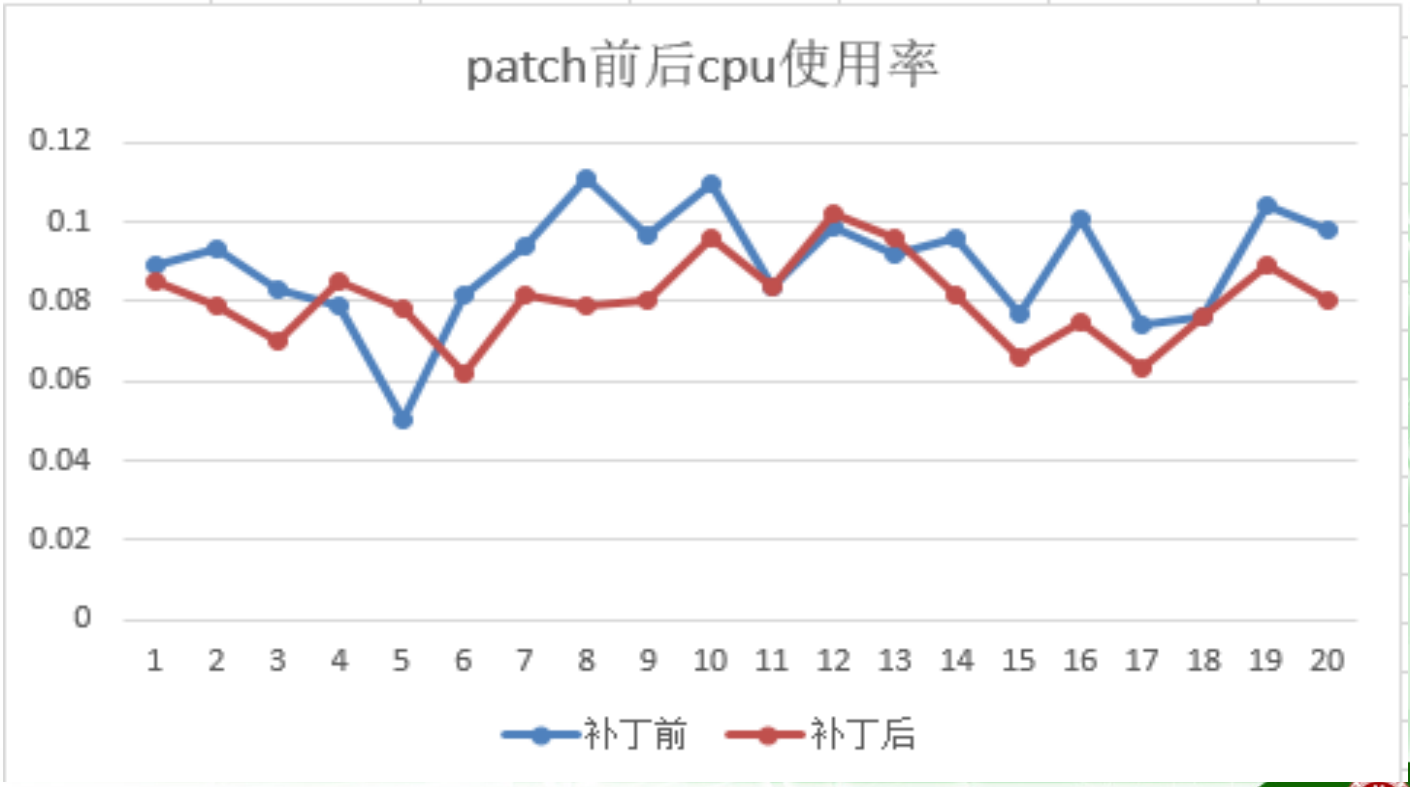
补丁前	补丁后	差值	比值
0.084	0.084	0.000	0.0%
0.081	0.042	0.039	3.0%
0.099	0.102	-0.003	48.1%
0.092	0.096	-0.004	4.3%
0.096	0.082	0.014	14.6%
0.077	0.066	0.011	14.3%
0.101	0.075	0.026	25.7%
0.074	0.063	0.011	14.9%
0.076	0.076	0.000	0%
0.104	0.089	0.015	14.4%
0.098	0.080	0.018	18.4%
0.053	0.109	-0.056	105.7%

第一列是补丁前的cpu使用情况，第二列是补丁后cpu的使用情况。我们发现，打补丁前，cpu的平均使用率是0.087，打补丁之后，cpu的平均使用率是0.080。可知在平均情况下，打完patch之后，cpu的使用率下降了大约8.5%。

补丁验证实验

性能实验

实验结果与分析
去掉了补丁前后方差较大的数据项12项和24项。制作出折线图



补丁验证实验

性能实验

实验结果与分析

可以发现，打完补丁之前，**cpu**平均使用率为**0.089**，打完补丁之后**cpu**的平均使用率为**0.080**，性能大概提高**10%**。在某些特殊情况下，此补丁甚至可以降低大约**50%**的**cpu**使用率。

补丁验证实验

性能实验

实验结果与分析

我们发现作者的测试结果为cpu的使用率降低了大概12%左右，和我们的测试结果基本一致。

```
lpq83:~# echo 1 >/proc/sys/net/ipv4/tcp_autocorking
lpq83:~# perf stat ./super_netperf 4 -t TCP_STREAM -H lpq84 -- -m 128
9410.39

 Performance counter stats for './super_netperf 4 -t TCP_STREAM -H lpq84 — -m 128':

      35209.439626 task-clock                #    2.901 CPUs utilized
             2,294 context-switches          #    0.065 K/sec
               101 CPU-migrations            #    0.003 K/sec
             4,079 page-faults               #    0.116 K/sec
    97,923,241,298 cycles                    #    2.781 GHz                      [83.31%]
    51,832,908,236 stalled-cycles-frontend   #   52.93% frontend cycles idle     [83.30%]
    25,697,986,603 stalled-cycles-backend    #   26.24% backend  cycles idle     [66.70%]
   102,225,978,536 instructions              #    1.04  insns per cycle
                                             #    0.51  stalled cycles per insn [83.38%]
    18,657,696,819 branches                  #  529.906 M/sec                    [83.29%]
        91,679,646 branch-misses             #    0.49% of all branches          [83.40%]

     12.136204899 seconds time elapsed

lpq83:~# echo 0 >/proc/sys/net/ipv4/tcp_autocorking
lpq83:~# perf stat ./super_netperf 4 -t TCP_STREAM -H lpq84 -- -m 128
6624.89

 Performance counter stats for './super_netperf 4 -t TCP_STREAM -H lpq84 — -m 128':
      40045.864494 task-clock                #    3.301 CPUs utilized
               171 context-switches          #    0.004 K/sec
                53 CPU-migrations            #    0.001 K/sec
             4,080 page-faults               #    0.102 K/sec
   111,340,458,645 cycles                    #    2.780 GHz                      [83.34%]
    61,778,039,277 stalled-cycles-frontend   #   55.49% frontend cycles idle     [83.31%]
    29,295,522,759 stalled-cycles-backend    #   26.31% backend  cycles idle     [66.67%]
   108,654,349,355 instructions              #    0.98  insns per cycle
                                             #    0.57  stalled cycles per insn [83.34%]
    19,552,170,748 branches                  #  488.244 M/sec                    [83.34%]
       157,875,417 branch-misses             #    0.81% of all branches          [83.34%]

     12.130267788 seconds time elapsed
```

补丁验证实验

实验总结

我们已经成功在android 3.4的内核版本中构建好了Autocorking补丁，并且发现Autocorking补丁能在连续发送小包的情况下提高手机性能，平均情况下会降低10%的cpu的使用率，在特殊情况下，可能降低大概50%的cpu使用率。

Wake affine补丁准备工作

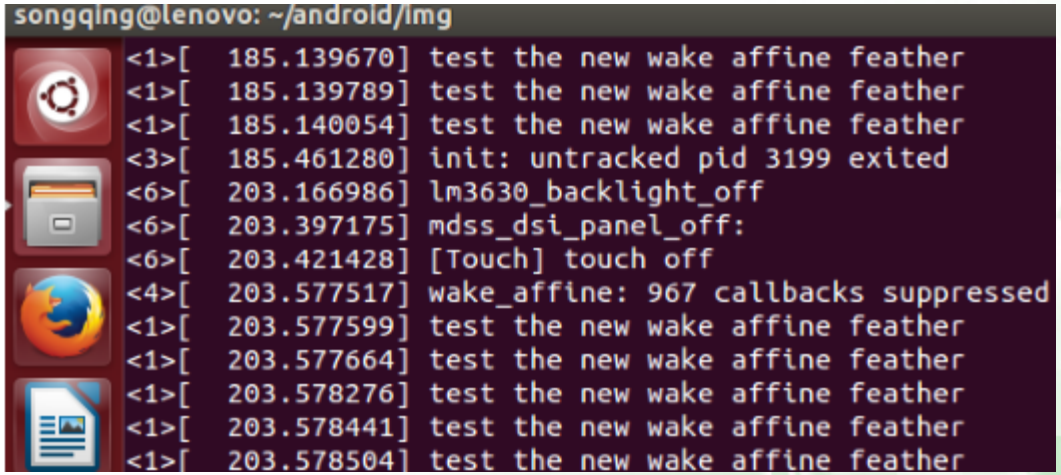
- 补丁介绍
  - 在Google Nexus 5手机的内核（3.4.0）上打补丁，验证新内核对手机性能的提升情况
- Patch来源
  - Linux Kernel Newbies网站（http://kernelnewbies.org/）
  - XDA开发者论坛（http://forum.xda-developers.com/google-nexus-5/orig-development ）
  - Google 安卓源码官网（ https://android.googlesource.com/kernel/common/+/android-3.4 ）

Wake affine patch选取

- 选取过程
  - 通过Linux Kernel Newbies网站发现该patch，出现在3.12内核版本中
  - 查找3.4与3.12之间所有与wake affine相关的patch，依次打到3.4内核上，形成最终patch
- 遇到的问题
  - 由于patch的版本与3.4内核版本跨度较大，因此，出现打某些patch时代码无法对应的问题
- 解决方案
  - 从3.4内核版本开始查找commit的记录中与该特性相关的patch，每次打上一个patch编译验证通过后，再继续打下一个patch

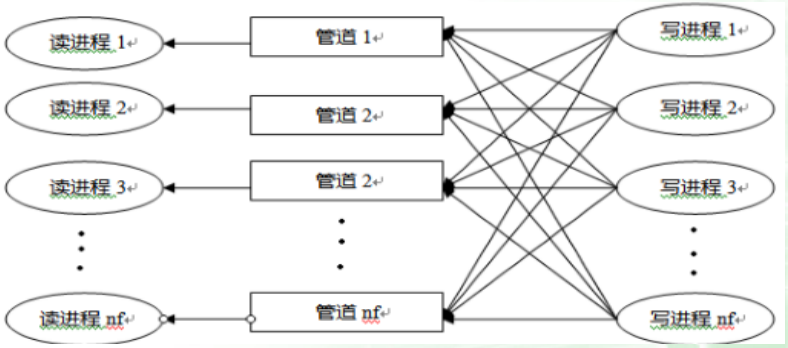
Wake affine patch验证

- 验证patch是否生效
  - 使用printk打印日志的方式
- 遇到的问题
  - Patch打上后导致手机无法正常开机，原因是系统大量调用wake affine这一特性，导致系统日志ring buffer溢出，解决方案：在printk函数之前调用printk_ratelimit函数，控制打印速率
- 验证结果
  - 成功看到printk打印的日志，系统开机后表现正常



Wake affine 性能实验

- 实验测试工具
  - 该特性主要是提高了进程调度时进程间关系为1：N时系统的性能
  - 原作者使用pgbench在linux测试性能，但是pgbench 是对 PostgreSQL 这种特定的数据库进行压力测试的，在安卓手机上很难移植
  - 考虑到linux下常用hackbench测试进程调度器的性能、开销以及可伸缩性，但是hackbench模拟的是进程间N：N的关系，因此，决定对hackbench进行修改，使之能够模拟1：N的场景
- Hackbench简介
  - Linux下的c语言程序
  - 通过模拟多组C/S模式下的客户端进程和服务器端进程的通信来测试Linux进程调度器的性能、开销和可伸缩性



Wake affine 实验过程

- Hackbench程序使用NDK（android 原生开发包）编译后生成的可执行程序可以在android的adb shell环境下运行
- 减少hackbench程序中写进程的数目，从而达到模拟进程间1：N关系的效果

- 遇到的问题
  - 原程序中读进程、写进程以及管道的数目相等且用同一个变量num_fds存储，修改过程中对不同地方出现的num_fds含义不清，导致程序运行时崩溃，解决思路：研读程序代码，弄懂hackbench程序的逻辑
  - 当程序创建进程数目超过1000时程序会崩溃退出，原因是linux系统默认的最大文件句柄数为1024，使用ulimit -HSn 4096命令进行修改

Wake affine 实验截图—替换手机内核

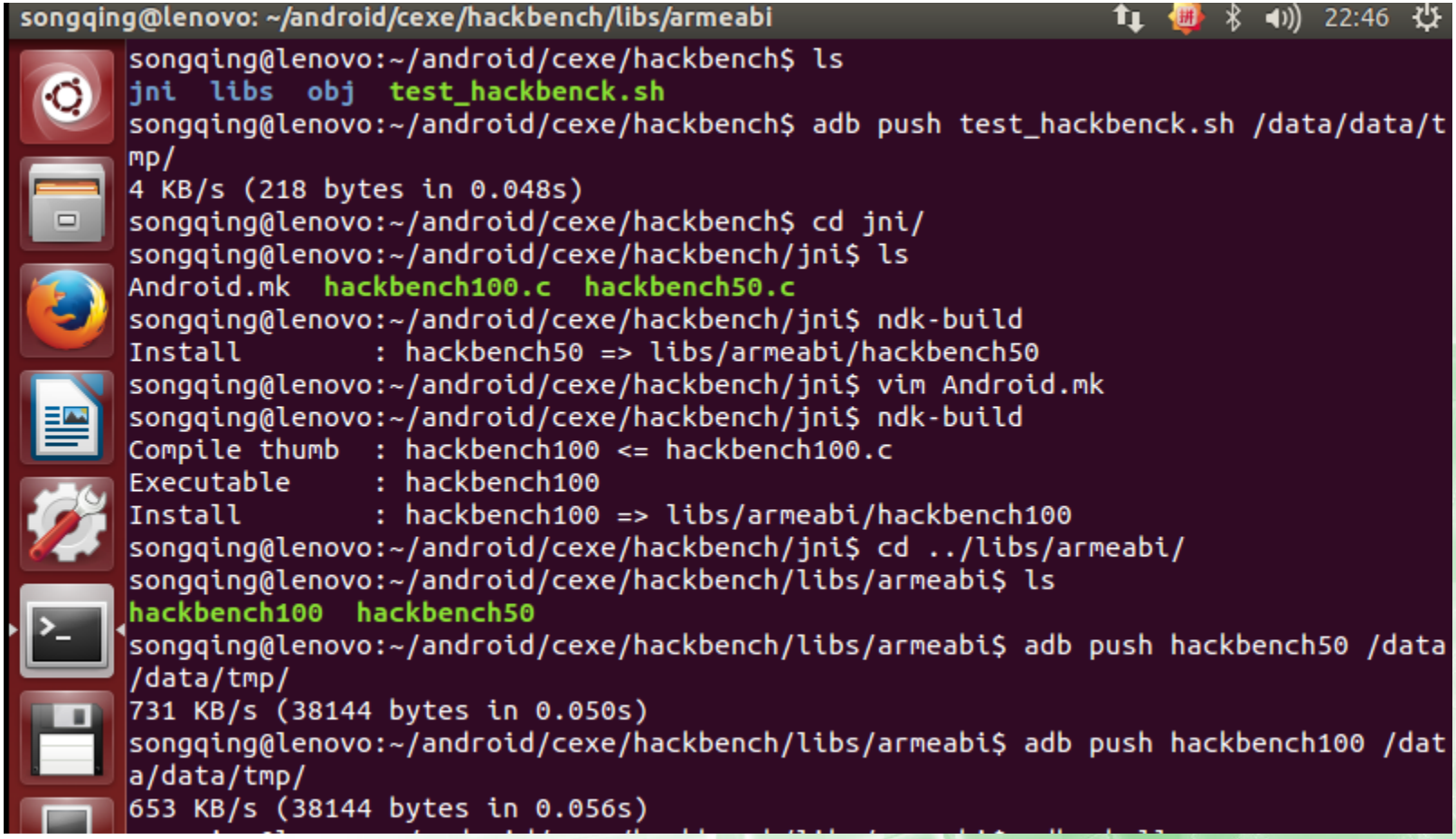
将打上patch之后的新内核刷到手机上，
安卓上层环境不变

```
songqing@lenovo:~/android/img$ ls
boot.img
songqing@lenovo:~/android/img$ abootimg -x boot.img
writing boot image config in bootimg.cfg
extracting kernel in zImage
extracting ramdisk in initrd.img
songqing@lenovo:~/android/img$ ls
boot.img  bootimg.cfg  initrd.img  zImage
songqing@lenovo:~/android/img$ abootimg --create oldboot.img -f bootimg.cfg -k ..
reading config file bootimg.cfg
reading kernel from ../msm/arch/arm/boot/zImage-dtb
reading ramdisk from initrd.img
Writing Boot Image oldboot.img
songqing@lenovo:~/android/img$ ls
boot.img  bootimg.cfg  initrd.img  oldboot.img  zImage
songqing@lenovo:~/android/img$ adb reboot bootloader
songqing@lenovo:~/android/img$ fastboot flash boot oldboot.img
sending 'boot' (8700 KB)...
OKAY [  1.266s]
writing 'boot'...
OKAY [  0.750s]
finished. total time: 2.017s
songqing@lenovo:~/android/img$ adb devices
List of devices attached
0670dc1f0ac6cb21        device

songqing@lenovo:~/android/img$ adb shell
```

Wake affine 实验过程——测试程序

- 修改hackbench程序
- 使用ndk进行编译
- 将可执行程序push到安卓手机上



Wake affine 实验截图——执行测试程序

- 下图为简单的脚本程序，通过使hackbench可执行程序执行多次，最后取平均值
- 右图为脚本执行时，输出的每次hackbench完成时所用的时间

```
1 #!/system/bin/sh
2 for i in 1 2 3 4 5 6 7 8 9 10
3 do
4     ./hackbench550 10
5     sleep 5
6 done
```

```
songqing@lenovo:~/android/img$ adb devices
List of devices attached
0670dc1f0ac6cb21        device

songqing@lenovo:~/android/img$ adb shell
root@hammerhead:/ # cd data/data/tmp/
root@hammerhead:/data/data/tmp # ls
hackbench100
hackbench50
test_hackbenck.sh
root@hammerhead:/data/data/tmp # ./test_hackbenck.sh
Running with 10*(50+5) (== 550) tasks.
Time: 1.161
Running with 10*(50+5) (== 550) tasks.
Time: 1.067
Running with 10*(50+5) (== 550) tasks.
Time: 1.186
Running with 10*(50+5) (== 550) tasks.
Time: 1.025
Running with 10*(50+5) (== 550) tasks.
Time: 1.059
Running with 10*(50+5) (== 550) tasks.
Time: 1.041
Running with 10*(50+5) (== 550) tasks.
Time: 1.167
Running with 10*(50+5) (== 550) tasks.
Time: 1.170
Running with 20*(50+5) (== 1100) tasks.
回收站
20*(50+5) (== 1100) tasks.
```

Wake affine 实验结果分析

写者：读者,测试组数为10，时间为s				
	patch前	patch后	p前-p后	性能提升
5：50	1.116	1.006		
5：50	1.18	0.971		
5：50	1.219	1.132		
5：50	1.201	1.013		
5：50	1.194	1.026		
5：50	1.186	0.976		
5：50	1.197	1.046		
5：50	1.144	1.089		
5：50	1.167	1.106		
5：50	1.242	1.001		
5：50 平均	1.1846	1.0366	0.148	12.49%
5:100	3.003	2.791		
5:100	3.185	2.785		
5:100	3.073	2.726		
5:100	2.951	2.817		
5:100	3.284	2.838		
5:100	3.24	2.753		
5:100	3.298	2.833		
5:100	3.463	2.83		
5:100	3.303	2.731		
5:100	3.937	2.811		
5:100 平均	3.2737	2.7915	0.4822	14.73%

- 实验中通过模拟写进程与读进程不同的数量比例进行测试，每种场景通过测试多次取平均值来进行比较
- 测试发现同一组测试数据数值变化不大，大致相等，因此，程序结果有说服力
- 测试结果：新内核会对系统的性能提升10~20%

Wake affine 实验总结

- 通过修改hackbench程序模拟进程间1：N的关系，可以用来验证该patch对系统性能的提升情况
- 测试发现进程调度时进程间关系为1：N时，会对系统的性能提高10~20%，进程数目越多，进程唤醒状态切换越频繁时，性能提升越明显
- 由于该特性属于进程调度中一种基本的调度策略，因此，系统中的众多模块会对该特性有所依赖，该特性的性能提升会提高CPU的处理能力，从而从总体上提升系统的性能

谢　　谢！