Android kernel level power management

Systems Engineering - Project
AY 2014/2015
Peter Plant - Stefan Kofler

Introduction:

Smartphones have become a central aspect of our daily life. In the last 10 years mobile computing has become a multi billion dollar business which is continuously growing. Manufacturers try to raise their sellings by producing high end mobile phones containing sophisticated hardware. Unfortunately computing with high performance is strongly correlated to a high battery consumption therefore new power management methodologies are needed. The so-called green computing has become a growing research field. Researchers try to preserve battery charge by investigating different approaches. This approaches range from high level applications running on the user space to low level customization of the operating system kernel.
The app stores for the different mobile operating system platforms provide already a variety of different power management applications trying to extend the battery life. Since this kind of applications a running all the time on the user space already the power management app it self is consuming battery charge.

Motivation:

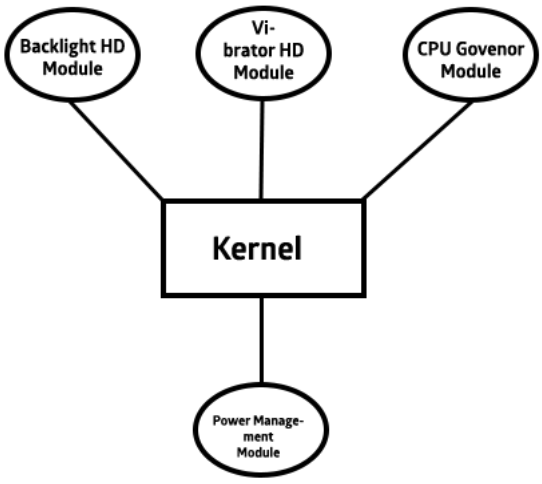
An average smartphone's battery charge lasts depending on its usage around 24 hours. Compared to cell phones used in the past the battery life has drastically decreased. State of the art hardware including quad core processors, 5 inch high definition touch screens, high performing graphical processing units are setting the user experience of such devices to new standards but at the same time such components are draining the battery. Therefore the importance of sophisticated power management tools has increased.
Such tools allow to reduce the cpu load, switch of the screen when not used. Some mobile operating system manufacturers have a built in power manager on the kernel level and as already stated previously there exist a lot of apps which enforce the power saving. Unfortunately such kind of tools are running on the user space all the time which is draining the battery too.

Proposed solution:

Since there exist already a large set of power saver applications running on the user space we propose a progressive power manager which is running on the kernel level. Since we have to access hardware drivers, modify kernel code in this project the focus relies on the Android kernel which is open source. The proposed power management tool will change different kernel parameters according to the current battery level which help to preserve battery charge.
There are many advantages of running a power management tool directly on the kernel level instead of running it as an application on the application layer. A kernel level power management tool can directly talk to the different hardware drivers without the need of sending messages through the whole layer stack.
Running a kernel module has a smaller battery consumption footprint than running an application on the user space. In previous Android versions (< 4.4) every application was running in its own instance of a Dalvik Virtual Machine. Every time an application has been launched the bytecode of the app needed to be converted to native machine code. Newer versions of Android discarded this concept and replaced it with the Android Runtime (ART).

Architecture:

The Android kernel is build upon the Linux kernel. Certain components have been added, changed and removed in order to run on a mobile computing environment. Both kernels implement the concept of modules. Modules are pieces of code which extend the functionality of the kernel.



They can be loaded and unloaded into the kernel at runtime without the need of rebooting the system. In Android the major part of hardware drivers are build as modules. This allows to efficiently configure the kernel depending on the device and the different hardware components within the device.
The power management tool is a module which can be loaded and unloaded at runtime. This allows to do small changes on the module without the need of recompiling the whole kernel which takes several minutes. Once the tool gets loaded a timer starts which periodically checks the battery charge (see code snippet below).

```
int result = 0;
union power_supply_propval capacity_val;
result = psy->get_property(psy, POWER_SUPPLY_PROP_CAPACITY,&capacity_val);
if(!result)
{
int capacity = capacity_val.intval;
//TODO stuff here with capacity
}
```

According to the current capacity a power class is selected. Each power class is related to a certain range and when activated different actions are performed in order to reduce the overall power consumption. The lower the battery charge is the more aggressive the power preserving settings are going to be. A power class is identified by an id and relies in an own C file. Every power class has the following basic structure.

```
#include "include/power_class.h"
#include "include/powerm.h"
#include <linux/kernel.h>

static void activate(void)
{
    printk(KERN_INFO POWER_TAG": Activated power class X\n");
}
const struct power_class_ops *POWER_OPS_X = &((struct power_class_ops){
activate });
```

If a new power class is selected the activate function is called which will start the procedure to change kernel settings/power_classes in other modules.
This function gets called by a “kind of” polymorphism pattern from the main C file.

Power_class /Actions	**Max Screen Brightness (0-255)**	**Vibration (%)**	**Screen of timeout**	**Auto switch off 2G/3G/ wifi**	**Max CPU frequency (GHz)**	**Power save bias (‰)**	**Max GPU frequency (0-6)**
P 9	default	Default	Default	Not active	default	50	0
P 8	150	Default	80%	Not active	1958,4	50	1
P 7	100	Default	60%	Suspended state	1728,0	100	1
P 6	40	30%	60%	Suspended state	1574,4	100	2
P 5	30	30%	50%	Suspended state	1267,2	100	3
P 4	20	30%	50%	Suspended state	1190,4	200	3
P 3	10	0%	50%	Suspended state	1190,4	200	4
P 2	5	0%	50%	Suspended state	1036,8	300	4
P 1	3	0%	20%	Suspended state	1036,8	300	5
P 0	1	0%	20%	Suspended state	960,0	300	5

(All values are subject of change, since they have not tested in a long term experiment)

The concept behind the power class id’s is rather simple the current battery level (shown in percentage) is divided by 10 the result returns in the current power class (e.g 50/10=5 results in 5th power class). There exists a special case the power class X which falls in the range of 100 to 90 battery level. For this range no settings on the kernel are done.

- **Reducing the screen brightness:** In order to set the screen brightness within the power management module we need to do some changes within the Android Kernel source code. Since the backlight driver provides no interface to read the

lcd backlight current level directly we needed to add a function which allows us to set it.
It was important to change the driver to avoid that the user could switch the backlight back to the old state or higher than the current allowed maximum.
In addition to this we also reduced the minimum possible brightness since the actual minimum seemed to be quite high. We achieved this by changing the minimum backlight brightness in the configuration file.

- **Reducing the vibration strength:** The vibration driver within the Android kernel was missing an interface in order to control it from another module. Therefore we implemented an interface that allows us to control the vibration strength from our power management module.

```
void msm8974_pwm_vibrator_gain(int gain)
{
      forced_gain = gain;
}
EXPORT_SYMBOL(msm8974_pwm_vibrator_gain);
```

- **Max CPU frequency:** We decided to implement the CPU underclocking over the cpu governor instead of statically setting the cpu clock frequencies. A static setting could lead to situations where the actual frequency is higher than currently needed. The Android kernel contains a set of different CPU governors containing different policies. We decided to not change the CPU governor and to use the default one (OnDemand). We added functionalities to change the governors policy in order to be able to set dynamically the maximum frequency the CPU can clock up. With lowering battery capacity also the maximum frequencies will be reduced. Especially when running on the low power classes a limited maximum frequency will has negative effect on the device performance therefore those frequency are still subject to change.

- **Powersave Bias:** This parameter takes a value between 0 to 1000. It defines the per mille value that will be removed from the target frequency. For example:
    - Powersave Bias set to 200
    - OnDemand target frequency 1500MHz

$$1500 - \frac{20 \times 1500}{100} = 1200MHz$$

By default this value is set to 10 and we set it to increase progressively as shown in the table above.

- **GPU max power level:** A static underclocking would be in many cases inefficient since especially when the smartphone is idling the GPU would use more resources than needed therefore we decided to change the maximum allowed power level inside the gpu. The power level scale starts at 0 and ends at 6 whereas 0 is the highest performing level. Every power level corresponds to a certain clock speed. By default the maximum power level is set to 0 which means the GPU increases its clock speed to a maximum when stressed. By reducing the limit the GPU can not clock up so high which results in a performance loss and battery efficiency gain. In addition to this we decided also to change the clock speeds of the different power levels, such that on the lower power levels (5 and 6) the clock speed is even slower. This changes have been done in the following configuration file:

  arch/arm/boot/dts/msm8974-gpu.dtsi

- **Screen off timeout: (not implemented)** After researching on the internal power management of the Android kernel it seems that the kernel is not aware of when to switch off the screen. The kernel only executes incoming wake locks.

- **Auto switch off Wifi/2G/3G: (not implemented)** The documentation and searches on different forums showed that the baseband interface/drivers are not open source which means it is impossible to access them to switch of either 2G, 3G or both.

Used technologies:

For this project we chose to use the Google Nexus 5 as it runs a version of Android that is up-to-date and free of third-party software. This device runs Android 5.0 which is based on the Linux kernel 3.4.
Up until now Android can only be build on either a Linux or a Mac OS system. We used Ubuntu 14.04 which comes with a lot of required software pre installed, like Python 2.6/2.7, GNU make 3.81/3.82, the OpenJDK 7, and the build-essential package. The kernel source code was pulled from the official android git repositories and to actually build the kernel we used the arm-eabi-4.6 toolchain.
To make changes to the kernel we made use of text editors like Sublime and Vim as most of the changes were made to C files. We created git patch files for each specific change so we could share the kernel versions easily without having to share the whole kernel source.
To deploy our work, we flashed the modified kernel using fastboot, then pushed the kernel module to the device and activated it through the Android Debugging Bridge (ADB).

Issues and problems with solutions:

- **Scarce documentation and misleading search results :** When starting with the implementation of our power management tool we recognized that the code is not documented at all. There exist some little documentation within the kernel where some of the hardware drivers are partially documented. However only a small fraction of the different hardware drivers were documented. The lack of documentation made it pretty hard to understand the code and its behavior especially when we needed to add pieces of code in order to induce some settings. When searching for tutorials and documentation using a well known search engine results where often related to the user space and not to the kernel level in which we were interested. The solution for this problem was either a sophisticated search until we found something we could use or the hard way understanding the code without documentation.

- **Proprietary modules:** Even if the Android Kernel is open source it doesn't mean that all the code running within the kernel is accessible. Some modules such as the baseband drivers are proprietary. This fact makes it impossible to induce any changes within these components.

- **Sysfs not accessible from kernel level:** Sysfs is a virtual file system that is already available in the Linux kernel. It acts as an interface between the hardware and the user space by using virtual files. It cannot only be used to fetch hardware related information but also to allow low level modifications from user space. In this way the Applications can modify information about hardware and its drivers (kernel modules) such as the Hardware Abstraction Layer (HAL). Our problem was that this interface is not available from kernel modules. To change system settings like Screen Brightness or CPU governor parameters we had to add such interfaces to each module in the kernel source code.

- **CPU adaptations:** Reducing the energy consumption through CPU usage adaptations can be quite difficult to foresee and even more to prove. The facts are:
    - the lower the voltage the lower energy consumption and,
    - lower frequencies allow lower voltages (without affecting the devices stability)

  However, even though the user might not notice the lowering of frequencies the device might be taking more time to perform a certain calculation, which in turn leads to a higher energy consumption. Finding the right settings requires long lasting testing and measurement cycles as explained in "Energy-Aware Performance Evaluation of Android Custom Kernels[1]".
  Furthermore, in CPU governors, we can find many parameters, like up_threshold and sampling_rate that can be changed, however these parameters influence each other making it impossible to prove their exact impact on energy consumption and performance.
  Now, in contrast to the other parameters, changing the powersave_bias allows to adapt the frequencies dynamically and therefore reduce the required energy uptake.

Future work:

The next step of this project will be the empirical investigation of our power management tool. There are several aspects which will be taken into consideration during the evaluation of the tool. Since we are progressively reducing the CPU performance some benchmark test will be carried out for each power class to determine the level of trade off in terms of performance a user has to face.

[1] "EN-ACT - Publications." 2015. 28 Jun. 2015 <http://www.en-act.eu/index.php/publications-list>

The next step will be to analyze the effects of the power save settings on the user experience. The goal is to collect data coming from a survey a set of users will complete after treatment with the power manager running and without using the power management.
The last step will be the probing of the power consumption which is probably the hardest part of the whole empirical investigation.

References:

https://wiki.archlinux.org/index.php/kernel_modules

http://marcin.jabrzyk.eu/posts/2014/05/building-and-booting-nexus-5-kernel

http://stackoverflow.com/questions/16920238/reliability-of-linux-kernel-add-timer-at-resolution-of-one-jiffy

http://stackoverflow.com/questions/1225844/what-techniques-strategies-do-people-use-for-building-objects-in-c-not-c

https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt

https://code.google.com/p/milestone-overclock/wiki/SmashingTheAndroidkernel

http://www.dailyphonetricks.com/lock-adreno-gpu-clock/

http://forum.xda-developers.com/galaxy-s2/general/ref-kernel-governors-modules-o-t1369817