Android Best Practices to Improve Battery Efficiency

Multi-core CPUs, motion sensors and multiple wireless radios all draw a significant amount of power which make a regular battery recharge a necessity. Applications, which extensively use the available hardware, reduce the battery runtime severely. Continuous motion sensor recording excessively stress a smartphone's CPU preventing it from entering a deep sleep state. Android 4.4 introduces a new batch-mode for sensors data to keep the CPU asleep for longer periods of time by delaying the reporting of new data from the sensors. The same technique is applied when writing to flash memory. When using a content provider to store data write amplification occurs, which affects writing performance negatively. The longer write times due to the reduced writing speed decrease the battery efficiency even more. A similar batch-oriented strategy reduces the occurrence of write amplification. In return, the reduced writing time improves the battery efficiency.

Chris Yereaztian, Jürg Luthiger | juerg.luthiger@fhnw.ch

Smartphones have evolved at such a rapid pace that they are capable of rivalling basic desktop computers regarding computational power. This trend greatly stimulates the development of mobile software, which in return promotes smartphone sales to an extent that they surpass PC sales [1]. However, battery technology has not caught up with the advancements in chip making. Dual-core CPUs are common amongst smartphones and the additional hardware packed into smartphones such as GPS, Wi-Fi, 3G and 4G radios all take its toll on the battery. A regular overnight recharge of the smartphone has become the norm due to the limited capacity of the batteries. Slow advancements in increasing the capacity combined with more powerful hardware can severely affect the battery efficiency. The applications running on a smartphone, which make extensive use of the hardware, are therefore at fault for the diminishing battery efficiency. In order to reduce the strain applications put on the hardware, developers need to understand the inherent restrictions of a mobile environment and the implications it has on their applications.

Battery Drain

In particular, applications using hardware features such as the motion sensors as well as the built-in flash storage significantly drain the battery. When an application requests access to the motion sensors it does so by using the sensor framework to subscribe to one of the sensors. In return, the sensor continuously publishes raw sensor data to the application, which keeps the CPU busy and prevents it from entering a deep sleep state. Writing the acquired raw data to the built-in storage leads to the effect of write amplification where more data needs to be written into the flash memory than actually specified due to the limited write capabilities of flash.

Since the use of the motion sensors is the underlying cause of the battery drain, on Android platforms a new batch-mode for listening to sensors data is introduced with version 4.4. It allows sensors to delay the reporting of new data to the application. The CPU is kept asleep for longer periods of time, which positively affects the battery efficiency.

A similar batch-oriented approach can also be applied to reduce the occurrence of write amplification. Writing a single record to a *Content Provider* in Android can trigger multiple writes on the flash storage that increase the power consumption. Instead of using one database transaction for writing each record, multiple records can be combined to a single transaction eliminating the transactional overhead. Storing data in larger junks less often compared to smaller ones more often improves the battery efficiency since the additional write operation necessary are kept at a minimum. As shown later in this article the GreenDAO framework supports this strategy when writing multiple records into a database.

Another aspect that has not been mentioned yet, but greatly influences the battery efficiency as well, is the wireless radio. Sending data to the Internet over UMTS consumes the most amount of power compared to other components as shown in Table 1. Applications that run services in the background, which access the Internet, keep the wireless radio awake for longer periods of time. Furthermore, the state machine of the wireless radio negatively amplifies the wake lock on the radio. When an application finishes a transmission the radio stays active for another 5 seconds before transitioning into a lower power state mode because re-activating the radio is bound to a delay. After another 12 seconds without any transmission the radio goes into a deep sleep state [2]. On the assumption that typically multiple applications are running in the background the radio

Action	Nexus 4 [mAh]	Nexus 5 [mAh]
UMTS Download	1339	1073
UMTS Upload	1410	1033
UMTS Call	983	637
UMTS Standby	18.3	10.4
WiFi Download	1158	549
WiFi Upload	475	488
GPS Searching	550	263
GPS Standby	0.4	0.7
NFC Standby	-	4
Sensors	751	487
Display (max)	310	567

Table 1: Power usage of different components measured with the TrepN profiler

is left active. Since not all applications transfer data to the Internet at the exact same moment in time the radio never enters the deep sleep state and therefore drains the battery.

In this article we discuss in more detail the battery drain caused by motions sensors and flash storage. The effects of wireless radio are omitted since the complexity of the interaction between the radio and the software legitimates its own analysis.

Sensor Batching

The built-in motion sensors in mobile phones such as the Nexus 4 and 5 draw a significant amount of power when the sensors are actively running, about half as much as the wireless radio (Table 1). The sensor framework provided by Android is built upon four distinctive classes and interfaces. The *SensorManager* provides access to the sensor services. It contains methods for directly querying the available sensors and registering sensor listeners. The *Sensor* class is used to create a specific instance of a sensor to determine its capabilities. The *SensorEvent* class is used by the system to create events, which include the raw, time stamped data of the sensor and the type of sensor that generated the event. The *SensorEventListener* interface defines two callback methods which the system calls when the values of a sensor or the accuracy of a sensor itself changes.

The best practice for accessing the motion sensors, before the introduction of the previously mentioned batch mode, used to be to limit the time window the application is actively subscribed to the sensors to a minimum. The activity class is a critical component of an application and transitions through multiple states in its lifecycle. When an application transitions to the state where it is actively in the foreground and visible to the user, the *onResume()* callback method of the corresponding activity is called. This is the recommended place to subscribe to the motion sensors. In contrast, when an application loses the focus or is partially covered due to another dialog, the activity is paused and the *onPause()* callback method is called. Even though the application may return to its running state the Android documentation recommends unsubscribing the sensor listeners to limit the unnecessary battery drain.

However, this approach cannot be applied to an application that is constantly recording the sensor data. For example, a pedometer requires constant access to the motion sensors to determine the amount of steps and detect possible false positives. With the existing method the applica-

Figure 1: left) Electronic interface to read out power consumption of a smartphone. right) Nexus 4 with an open back cover for connecting measurement equipment.

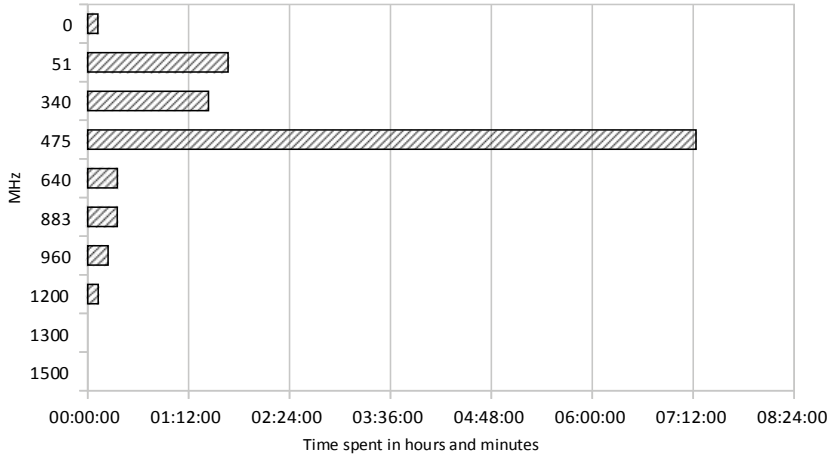

Figure 2: CPU usage of a Nexus 5 in non-batch mode

tion would severely drain the battery. The new batch-mode introduced with Android 4.4 permits constantly listening to the sensors without significantly increasing the battery consumption. The batching refers to the mechanism of bundling multiple values from the sensor before sending a new event to the sensor event listeners. The Android sensor framework support this by adding an additional parameter to the *registerListener()* method that allows the caller to specify the maximum latency in seconds before a batch needs to be send to the listeners [3]. If this parameter is set to zero, batch processing is completely disabled. The significant power savings, as seen in Figure 2, are based on preventing the *System on Chip* (SoC) of waking up for each receiving sensor event. Multiple events can be grouped and processed together while each of them retains their own individual timestamp. The batching is done in hardware using FIFO queues, which temporarily hold the sensor events before sending them through hardware abstraction layer to the system [4]. The oldest events in a queue will be dropped if there is not enough space available to accommodate new incoming events from the sensors.

The sensors in a smartphone are divided into two categories: the wake-up sensors and the non-wake-up sensors. Sensor events from wake-up sensors are stored separately in a wake-up sensor queue to ensure that the data is delivered regardless of the SoC's current state. The driver of the wake-up sensors achieves this by keeping a wake-lock for at least 200 ms to ensure that the new event is delivered to an application. As a result the wake-up sensors, as their name implies, will wake-up the SoC to deliver the events before the specified maximum allowed reporting latency has elapsed [5].

In contrast, non-wake-up sensors do not prevent the SoC from entering in its sleep state and more importantly will not wake up the SoC to report new sensor events. The driver of the non-wake-up sensors does not hold a wake-lock. Therefore, the application is responsible for keeping a partial wake-lock if it wants to receive new events from non-wake-up sensors while the screen is off. The events in the FIFO queues will be delivered to the application as soon as the SoC returns from its sleep state.

The batching cannot be emulated in software and needs to be implemented in hardware. Since

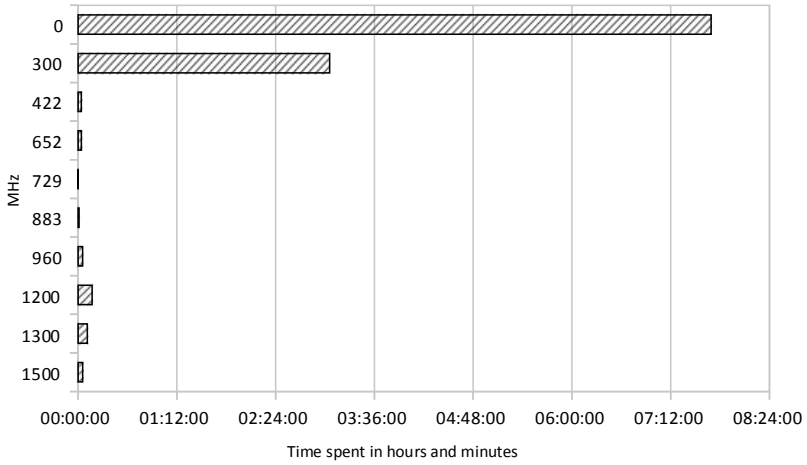

Figure 3: CPU usage of a Nexus 5 in batch-mode

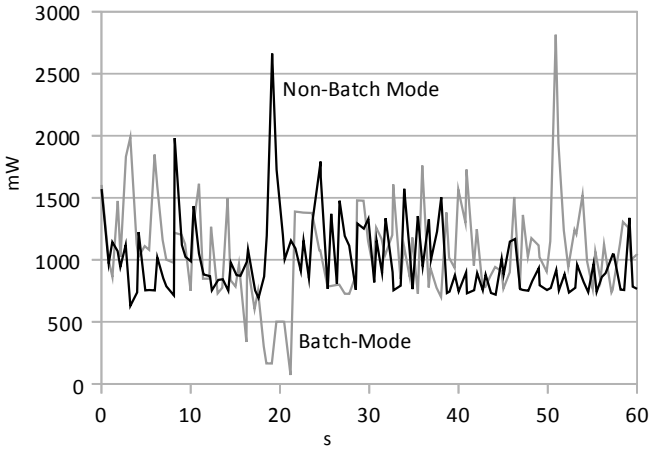

Figure 4: Battery usage of a Nexus 4 in non-batch and batch-mode

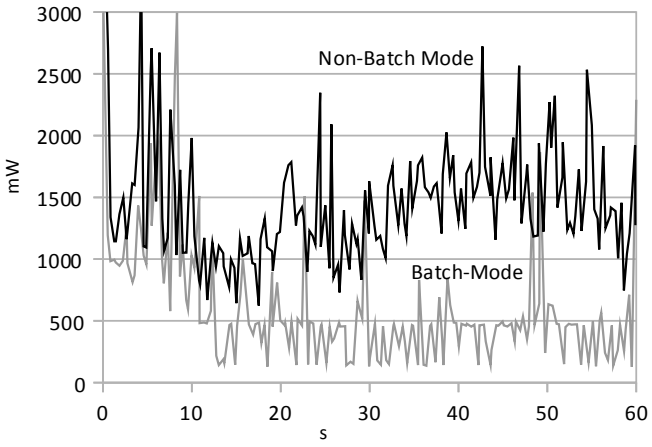

Figure 5: Battery usage of a Nexus 5 in non-batch and batch-mode

the goal of batching is too reduce the battery drain caused by sensors waking up the SoC, it definitely cannot be implemented using the SoC itself. Therefore, a separate hardware chip that provides and manages the FIFO queues is required to keep the SoC in suspend mode during batching. Older devices such as the Nexus 4 are not able to take advantage of this new feature in Android 4.4 as they lack the necessary hardware chip for the queue. Registering a sensor listener with the *maxBatchReportLatency* set to other than zero will silently be ignored. The sensor listener will receive the events as if it had registered the listener with batching disabled.

Sensor Batching Tests

The effectiveness of the proposed practices is shown with a series of tests on two different Nexus smartphones, Nexus 4 and 5. Both devices are based on nearly the same Qualcomm Snapdragon platform and equipped with same model of sensors. However, the Nexus 5 also includes the required separate hardware chip to manage the formerly discussed FIFO queue. In order to create an identical testing environment on both devices, the latest stock Android 4.4 image (latest at the time of writing) is used without any modifications to the kernel. Since Android supports running applications concurrently in the background, the same number of applications is installed on both devices. This ensures that none of the running background processes will influence the measured results because if one of process would indeed affect the battery drain, it would do so on both devices.

Diagnosing the battery drain requires direct access to the power management on a smartphone. On all Qualcomm Snapdragon based smartphones such as the Nexus 4 and 5 the kernel provides access to the power management. The *TrepN Profiler* [6] is a plugin for Eclipse that uses kernel functions to read out the statistics from the internal power management chip. This enables the accurate profiling of the overall CPU usage and frequency, memory as well as network usage. To measure the overall power consumption of a Nexus device the back cover needs to be opened because the battery with the connectors (marked in Figure 1b) is sealed inside the body. The electronic interface (Fig. 1a) connects to a debug port on the battery connector (4-pin socket instead of the usual 2-pin plus and minus pole socket). To minimize the effect of measuring errors on our test results, each test is run three times and the average of all three runs is taken as the actual measurement value. If one measurement of those three runs differs from all the others by more than 10% then that specific run is repeated and the current measurement is discarded.

As already mentioned the hardware-backed FIFO queues require a separate chip that takes over the management of the queues. Our test of continuous recording sensors shows a clear difference in CPU usage between batching disabled and enabled on a Nexus 5. The CPU in the Nexus 5 is kept asleep for longer periods since no interrupts from the sensor are waking up the SoC. The 0 MHz bar in the charts (Fig. 2 and 3) represents the CPU in deep sleep state (the 0 MHz is just the interpretation of the *TrepN* profiler). The time axis represents the time the CPU was running at the specific frequency. With batching disabled the CPU spends most of its time running at 475 MHz for processing sensor data even if the application is idling, as seen in Figure 2. In contrast, Figure 3 shows that enabling batching keeps the CPU asleep for longer periods of time (Figure 2 and Figure 3 use different scaling since the CPUs dynamically scale the clock up and down depending on the workload). Figure 4 and 5 confirm that the different CPU states affect the battery drain. The battery usage on a Nexus 4 shows no difference between measuring sensors in non-batch and batch mode in Figure 4. Since the Nexus 4 does not have the required hardware chip the Android's sensor framework will automatically fall back to non-batch mode. In comparison, the Nex-

us 5 in Figure 5 does show that the battery usage in batch-mode is drastically cut by more than one third. Logically, due to the lower CPU usage and the fewer interrupts caused by wake-ups from sensor events, the overall battery consumption is reduced.

Flash Memory

So far we have shown that accessing the sensors contribute significantly to the battery drain. The presented solution of batching sensor events together has proven to positively affect the battery drain issue. However, the sensors are not the only component in a smartphone influencing the power consumption in a significant manner. Depending on its use, the flash storage increases the battery drain as well. The typically used flash memory (eMMC) in mobile phones suffers from an unwelcome effect of write amplification where more data is written to the flash than the application actually committed to [7]. This is primarily due to the nature of flash memory where data cannot be directly overwritten compared to traditional hard disk drives [8]. The severity of write amplification is linked to the storage provider used in an application. A storage provider on a mobile phone should support fast reading and writing of data while using a minimal amount of power. Furthermore, it should facilitate the processing (searching, sorting, and filtering) and sharing of data between applications on the phone while making the files unattainable for direct modification from the outside to secure their integrity.

The Android storage framework provides the following mechanism to store data in an application: *Shared Preferences* represent a key-value pair store, which holds primitive data types. It is designed to store internal application preferences rather than user object-data. A *Content Provider* is a mechanism that manages the access to a set of structured user data. The provider encapsulates the data by providing a common CRUD interface that allows easy access to the data from within the application as well as third party applications. The Android framework also provides read/write access to the underlying file system provided by the Linux kernel. Applications can use the file system to store user data and preferences in files regardless of data structure and format being employed.

Shared Preferences are designed for persisting preferences that can be retrieved and set using a string as a key to identify the associated value. The critical issue with *Shared Preferences* is that they are application specific and therefore cannot be used to share data. Files suffer from the same problem. All applications on Android run in a sandboxed environment where each application is run with a separate user account. The associated files are kept private and no other user respectively application can access these files [9].

The *Content Provider* mechanism is a set of methods and structures designed to separate the raw data from the aggregated complex data at runtime. For example, the *Contact Provider* combines information about the contacts from multiple sources such as a SIM card, contacts from a Google account, application specific contacts etc. The provider itself can implement the usage of the storage in two ways:

- Firstly, a file-centric approach where the data normally goes into files which are stored in the application's private space.
- Secondly, a structured data approach where the data is placed into a database, an array or a similar structure by mapping the data into a set that is compatible with rows and columns. In general, any type of storage can be used, but a common way to store this type of structured data is to use an *SQLite* database, because Android offers built-in support for *SQLite*. A substantial part of the writing process is hidden from the developer when a single record is put into a database. The commit call for the database immediately returns and the developer assumes that the data has already been written to the disk and moves on.

It is inherently difficult to determine which components of the process are the least efficient and affects a phone's battery the most. To identify the factors that influence the power consumption, a detailed understanding of how embedded Multi-Media-Card (eMMC) storage works is necessary.

Write Amplification

General flash-based storage found for example in solid state drives separate the NAND flash chips from the controller. The controller is a critical component as it handles the mapping between disk-based track and sector geometry and the flash-based cell geometry. Due to the small device requirements of smartphones, the controller and NAND flash memory are contained within one package [10].

When writing data to flash storage, an undesirable but unavoidable issue is the occurrence of write amplification where the actual data written to the flash is a multiple of the data that the host requested to be written. The root cause of write amplification is that individual pages of memory can be written to empty flash memory cells but the pages can only be erased in larger units, called blocks. If a block contains invalid and valid pages, then the eMMC controller must first read in all the valid pages of a block it wants to write new data into. After caching those valid pages it will invalidate all pages in that specific block and finally write both the old and updated pages back to the block.

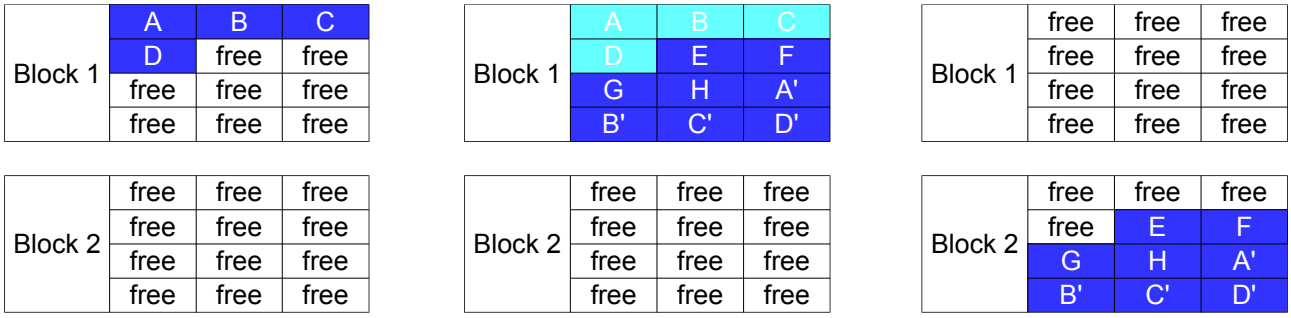

Figure 6: left column) Pages written to free memory cells in a block. middle column) Invalidating old pages (stale data) and replacing them with new ones in a block. right column) Cleaning up stale data and writing old as well as new data back to the block.

The example depicted in Figure 6 illustrates the issue quite well:

1. In the left column of Figure 6 four pages (A–D) are written to block 1. Individual pages can be written at any time if they are currently free respectively erased.
2. In middle column of Figure 6 four new pages (E–H) and four replacement pages (A'–D') replacing (A–D) are written to block 1. The original pages (A–D) are invalidated (so called stale data) and cannot be overwritten with new data until the whole block has been erased.
3. In order to overwrite pages with stale data (A–D) the remaining valid pages in block 1 (A'–D' and E–H) are read, cached within the controller or directly written to another block (right column in Figure 6). Now, the controller is able to completely erase block 1 which resets the memory cells so new data can be written into it.

The undesired increased number of writes occupies bandwidth to the flash storage and hampers the random write performance severely. When the storage is relatively empty write amplification doesn't occur since there are enough fresh empty cells the controller can use. However, when a large amount of data has already been written to the flash memory and only few empty cells are available, write amplification occurs. The controller needs to shift valid pages to other blocks in order to be able to clear all invalid pages, as seen in the former example. The I/O performance of Google's Nexus 7 (2012 Edition) suffers from this issue where the device's subjective performance slows down after months of use, which leads to an inconsistent user experience.

Since Android 4.3 the negative effect of write-amplification has been alleviated, because Google has enabled the support for *TRIM*, a mechanism that trims blocks that are not used in the file system. Briefly summarizing, *TRIM* allows the OS to tell the eMMC controller that a block is no longer in use and ready for garbage collection. It is important to mention that deleting a file or a record in a database is not actually communicated to the eMMC controller. Even though the space is freed up in the file system the controller still treats the block with the pages as containing valid data that cannot be purged. The controller is forced to move pages that are invalid but still treated as valid pages in the block and hence needs additional power. Exactly quantifying the additional battery drain is difficult, because file system operations are handled by the Linux kernel and applications do not have a direct way of managing those from user space to observe when exactly write amplification occurs. However, when writing records into a *Content Provider* backed by a database the hypothesis that a correlation between battery consumption and the occurrences of write amplification exists can be made. The higher the write amplification the more power the smartphone will draw from the batteries since it requires more time to complete the writing process. To test this hypothesis 10000 objects consisting of sensor event values are inserted into a *Content Provider* while the execution time and battery consumption are measured. Three distinctive approaches for inserting the records into the database are used:

- Each single record is inserted separately using the methods provided by a *Content Provider*. No optimizations are done in a *Content Provider* or the insertion call itself.
- A batch-oriented approach is applied. Similar to the solution of the sensors where multiple events are bundled together, multiple records are bundled to a batch for inserting them at once into the database. The idea is to eliminate the transactional overhead as well as preventing possible write amplifications that might occur in the first approach due to repeated writings of smaller data junks.
- GreenDAO, an object-relational mapping (ORM) framework for Android is used for storing the sensor event objects into a database [11]. It promises to put its focus on maximum performance. GreenDAO takes over the responsibility of a *Content Provider* and offers methods to persist objects directly.

In order to minimize measurement errors, again each test is repeated three times and the average execution time of the three runs is used for the comparison. The test results depicted in Figure 7

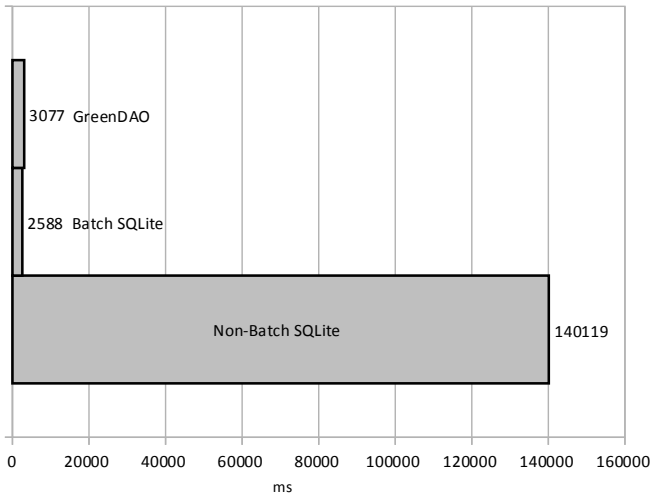

Figure 7: Execution time for write operation into a database

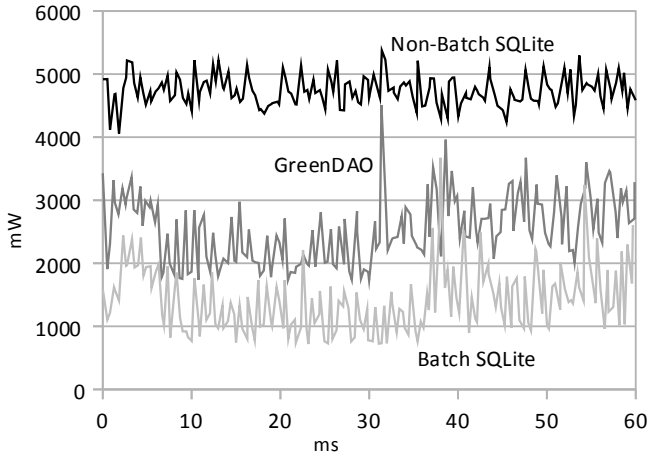

Figure 8: Comparison of battery usage between different storage providers

show a distinctive difference between the batch and non-batch approach of inserting the records into the database. Using a single transaction for inserting 10000 objects takes 2.5 seconds. It is relatively fast in comparison with the non-batch approach that requires 14 seconds. The negligible difference between the ORM framework GreenDAO and batch-insertion is due to the overhead of object-relational mapping. The measured battery usage in Figure 8 reflects the same result. There is a considerable difference between batch and non-batch approaches. The drain is about twice as high for not batching the records. In contrast, the battery consumption only slightly increases from manual batching to using GreenDAO. Since the CPU usage between all three approaches is nearly identical, the correlation can be made that the increased consumption is caused by the amplified writes to the flash memory. The less time a smartphone is spending on data persisting, the more battery can be saved.

Summary

Application development on mobile platforms is still more difficult compared to desktop systems because of the limited resources available on mobile devices. Applying specific techniques considerably improves the battery efficiency as shown in the presented examples. Compared to desktop systems where the abundance of resources eliminates any incentive of optimizing an application, small changes in mobile applications lead to noticeable improvements. Bundling operations in batches has proven to be an effective technique in general. Applied to the sensor framework as well as the persistence mechanism it drastically reduces the power consumption.

References

[1] Rob van der Meulen, Janessa Rivera. Gartner Newsroom Press Release, 2014. https://www.gartner.com/newsroom/id/2791017

[2] Oliver Spatscheck, Alexandre Gerber, Subhabrata Sen. A call for more energy-efficient apps, 2011. http://www.research.att.com/articles/featuredstories/201103/201102Energyefficient?fbid=pvldc4jPUQE.

[3] M. Hidaka. Android 4.4 sensor batching. 2013. http://techbooster.org/android/device/16666/

[4] Android Developers Documentation. Android Sensor Batching. 2014. https://source.android.com/devices/sensors/batching.html

[5] Android Developers Documentation. Suspend mode – Wake-up Sensors, 2014. https://source.android.com/devices/sensors/suspend-mode.html#wake-up_sensors

[6] Qualcomm Technologies. Increase app performance with TrepN profiler, 2013. https://developer.qualcomm.com/mobile-development/increase-app-performance/trepn-profiler

[7] Xiao-Yu Hu, Evangelos Eleftheriou, Robert Haas, Ilias Iliadis, Roman Pletka. Write Amplification Analysis in Flash-Based Solid State Drives. IBM Zurich Research Laboratory.

[8] Cameron Crandall. SSD: Flash Memory and Write Amplification. Kingston Technology. http://www.kingston.com/us/community/articledetail?articleid=17

[9] Nikolay Elenkov. Android Security Internals: An In-Depth Guide to Android's Security Architecture. No Starch Press, p12, p50-59, 2014.

[10] Datalight Technologies. What is eMMC? http://www.datalight.com/solutions/technologies/emmc/what-is-emmc

[11] Vivien Dollinger, Markus Jungiger. greenDAO – Android ORM for SQLite, 2013. http://greendao-orm.com/features/