IMPROVING THE ENERGY EFFICIENCY AND THE PERFORMANCE OF AN ANDROID 7 CORE APPLICATION

Author: Olivier PHILIPPO, GREENSPECTOR
Date: Jan 10th, 2017
THANKS FOR DOWNLOADING!

Be sure to check out our others resources about software ecodesign:

YES, I WANT TO LEARN MORE!

+33 (0) 9 51 44 55 79
contact@greenspector.com
EXECUTIVE SUMMARY

The GREENSPECTOR team has been asked by a customer to help in optimizing an Android core application. The final goal was to reduce the energy consumption of the app, or to improve its performance, or both.

The application audited was the AOSP 7.0 core application “System UI” running on a Nexus smartphone. The project lasted 2 weeks in November 2016.

We first conducted a set of measures on the smartphone, using GREENSPECTOR metering features and a dedicated GREENSPECTOR energy probe. This showed that System UI had a substantial impact on the device, especially through Status Bar and Recent Apps features. Indeed, the energy consumption impact was measured to be between 2.2 and 2.9 times higher than the reference scenario.

During the audit, several issues were identified which correction could potentially help to decrease this impact:

- A high number of triggered events, which create unnecessary treatments and redraws. These treatments impact the platform resources even when in idle mode.
- A high frequency (and thus impact) of the animation and of the movement tracking feature. The animation performance is designed too high for the user to perceive its quality.
- A heavy layout, which creates a lot of consuming treatments and redraws enforced during animations.

In the second part of the audit, we modified the source code of System UI to apply some of these changes. We were able to obtain significant gains:

- Removal of unnecessary triggerings of redraws.
- Reduction from 250 ms to 150 ms of CPU treatments when showing the Status Bar.
- Reduction of the number of calls to several methods (up to 100 calls during sliding actions),
- Reduction of energy consumed during Show/Hide Status bar: - 28 μAh/s (- 9%).

The goal of this audit, which was to improve the application as per energy efficiency and performance standards, has been reached in a short time frame.

We spent 3 man.days on this code refactoring task, including some initial time necessary to understand the code. The overall audit duration was 9 days for 2 consultants. This is very positive, since we estimate that the gains could be more important with a better knowledge of the source code, and some more time to apply the corrections to the application.

We demonstrated with this case study that, provided you use the right method – which involves energy consumption measurement - you may try and optimize any application, be it a part of the Android core. Our approach based on software eco-design principles allowed us to identify areas of progress in a short time frame. The implementation of the key recommendations will permit not only a reduction of the energy consumption but also an increase of the hardware lifespan.
1. AUDIT DESCRIPTION

1.1. System description

The chosen platform was:
- Device: a Nexus Smartphone (Model: confidential)
- OS: Android 7.0 (Build AOSP on Angler - NRD90M)

1.2. Application

The application under test was a core Android application:
- System UI
- Version: from AOSP repository (1.0.3)

1.3. Use cases

The chosen use cases were features frequently used by the users:
- Open the status bar in minimal mode (use case name: ShowMiniStatusbar)
- Open the status bar entirely (ShowAllStatusBar)
- Hide the status bar (HideStatusBar)
- Show / Hide all the status Bar (ShowHideStatusbar)
- Open Recent Apps (OpenRecentApps)
- Clear All Recent Apps (ClearAllRecentApps)
- Show/ Hide Recent Apps (ShowHideRecentApps)

Another use case was used to measure the platform consumption in idle mode:
- Idle Mode (Reference)

All these test cases were automated using UIAutomator.
2. METHODOLOGY & TEST BENCH SETTING

2.1. Methodology

General description of the methodology

For this audit, we used a classical top-down approach. We first launched a set of measures to identify the most consuming features, then we went deeper into the analysis where there were “hotspots” of resources consumption. An optimization phase was then conducted, followed by the assessment of each progress done.

In the end, the hotspots having been corrected, the developers could switch to correcting their source code with respect to an eco-design set of rules.

Features measurement

The energy measurement allowed us to identify which features consume the most energy. Without that identification, we couldn’t focus the correction effort, and this could lead to working on some parts which have no or very little impact on energy consumption.

We focused on the following tests because they were designed for energy measurement (duration of tests > 1 minute):

- Show / Hide all the status Bar (ShowHideStatusBar)
- Show/ Hide Recent App (ShowHideRecentApps)
- Idle Mode (Reference)

Hotspot profiling

When a “hotspot” had been detected (which means that a highly consuming test had been identified), we then used classical profiling tools to get a better understanding about the underlying behaviour. In parallel we used GREENSPECTOR Code Analysis feature, to determine if some important code eco-design rules could be infringed and thus participate in causing the hotspot.
Action plan, optimization and measurement

When the auditing part was done, we set an action plan which aimed at reducing the energy consumption. The plan focused first on improving the “hotspots”, starting by the hotspot with the highest Impact ratio.

After each correction, we performed another profiling test, in order to check if the hotspot had been corrected or if it was still present. When a hotspot was suppressed, the next hotspot in the list became the next priority.

Please note that, given the very short time frame of this audit, the hotspot optimizations were applied as “quick and dirty modifications”, even if not fully functional, in order to see if such a modification was interesting. “Cleaner” modifications could easily be performed with the same principles, given some more development time.

We applied the modifications as increments, which allowed to check the gain of each improvement.

Source code Analysis

After the removal of the main hotspots, we focused on the correction of the source code, using the code eco-design rules for Android.

2.2. Testbench description

GREENSPECTOR tools

The test bench was composed of the GREENSPECTOR server installed on our customer’s premises, and the Android probe developed for this Nexus device and this Android version. These tools are available to all customers of GREENSPECTOR.

Illustration 4 : Greenspector dashboard
Testing environment

We used the following tools:

- Greenspector Server: to conduct the audit, gather and analyze the data.
- Greenspector Android Meter API: to link the smartphone probe to the test case run.
- Android tools: Traceview, Systrace and Layout Hierarchy.
- Git: to work on the refactory steps.

The testing protocol was the following:

- Charge the smartphone between 95% and 100% (to have the same energy behavior)
- Reboot the smartphone (to put it in a stable state)
- Unplug the USB cable or any other energy supply
- Run all the tests (always in the same order)
- Re-run the protocol x times to have stable measures.

3. AUDIT

3.1. Feature Measurement and identification of hotspots

We launched test runs for each of the functional cases that we had selected. As mentioned, prior to running the functional test cases, we ran a Reference (or “idle”) test case to establish the reference consumption for our platform.

The initial version of the application was measured with the methodology explained previously. For the energy we got the following results:

The impacts of Show/Hide Status bar and Show Recent App are significant. Their consumption ratio, as compared to the Reference test, are respectively 2.4 and 1.9.

For shorter use cases, the test durations were not long enough to get accurate energy measures on this Nexus device, which communicates its energy status only every 30 seconds. However, we were able to launch these tests and measure another key metric, which is the CPU consumption.

The relative impact of each feature in terms of CPU consumption was the following:
This first and simple step has allowed us:

- To compare the functional cases with respect to the consumption of the platform when idle (which is much more relevant than to try and assess absolute figures);
- To compare the functional cases between them (which makes it possible to prioritize the rest of the work).

3.2. Profiling

An optimization work is not an exhaustive and planned approach. We are in a constant search of balance between the hoped-for gains, and the workload that would be needed to obtain these gains.

In our case, the search for the big rocks had already begun: thanks to the measures carried out, we were able to target the most consuming test cases.

Hence, when we started deeper profiling with expert tools (Android Systrace, Android Traceview, HierarchyViewer…) we already knew where and how to use them. These tools being very accurate on narrow points and their understanding being rather arduous, the foremost step saved us a lot of time.

The energy measurement showed a hotspot on Show/Hide Status bar, so we began the profiling by this feature. The Systrace tool gave us a list of methods with the time spent. We analyzed and filtered this list to obtain the methods of system UI:

Illustration 6: CPU measure in Greenspector

"Hence, we used this good old 80/20 rule, or what we like to call "looking for the big rocks": if your road is blocked by a rock, you don’t have to mind the sand in your shoes for the moment.

Illustration 7: extraction of Traceview information
The cross analysis of this list and a peek at the code allowed us to conclude that several methods were called many times, and that 2 types of treatments were responsible of this behavior: Refreshing the tiles (icon and text of the status bar), and movement tracking/animations. The refreshing of tiles is done each time an event occurs. Therefore, it generates some false triggerings and thus a lot of treatments and redraws:

Between user actions (show mini, show all and hide) we see 4 peaks which correspond to the triggering of the tiles redrags. We don’t know here if they are really needed but we analyzed in the code and in the profiling that a lot of events with no real impact in the viewing were firing these redrags.

The cost of one peak is not negligible because of treatments and redrags:

Moreover, this treatments appear also during the sliding of the status bar, and not only during idle time.

For the animation, 2 big impacts were identified: treatments of movement (algorithm in Threads) and redrags. The redraw has a cost because of the size of the layout - which is big. The Hierarchy Viewer tool of Android permitted us to analyze the layout:
There are 12 levels, which is a lot, and all tiles are complex:

This layout has an impact on energy consumption, because lots of treatments are needed to update and draw the layer. In Systrace, we see that in the timing:

The frame rate is 60 fps (every 16 ms). The update and redraw of tiles take more that 16ms. This triggers warning in Systrace. For the quick bar, the layout has a lower impact:
### 3.3. Action Plan

**Code analysis (detailed analysis)**

The action plan followed the methodology by prioritizing the improvements with respect to their expected gains. We took action on the code by tackling first the most consuming methods:

**Code**

```
Method

Layout (all xml files)
  com.android.systemui as TouchAnimator
  com.android.systemui as TouchAnalyzer
  android.systemui as QSTileStateCopy?

com.android.systemui as QSTileStateCopy?

Layout (all xml files)
  com.android.systemui as TouchAnalyzer
  android.systemui as QSTileStateCopy?

com.android.systemui as QSTileStateCopy?

Terminal handlers
  com.android.systemui as Tile handlerUpdateState
  com.android.systemui as Tile handlerUpdateState

com.android.systemui as Tile handlerUpdateState
  com.android.systemui as CellularTile handlerUpdateState
  com.android.systemui as CellularTile handlerUpdateState

com.android.systemui as Tile handlerUpdateState
  com.android.systemui as CellularTile handlerUpdateState
  com.android.systemui as CellularTile handlerUpdateState

com.android.systemui as statusbarStackNotificationStackScrollLayout.get
  com.android.systemui as statusbarStackNotificationStackScrollLayout.get
  com.android.systemui as statusbarStackNotificationStackScrollLayout.get

com.android.systemui as statusbarStackNotificationStackScrollLayout.get

ShortcutView apply

Compress the icon to optimize the performance.

```

**Best practices**

- Optimize the layout (>10 level). Usage of relative layout will increase several method performance.
- Optimize inner class access
- Reduce call number (>1000)
- Don’t use first in interpolation
- Optimize inner class access
- Don’t use Objects.equals. Overhead of call and test to null (return a == b || (a != null && a.equals(b)))
- Don’t make copy if variable don’t change
- Reduce call number (>10). Call especially by handleRefreshState.

1 – Simplify the layout

The layout is complex (several levels, usage of linear layout…). There are a lot of treatments going on during every layout redraw and measure, especially during animation. Simplifying the layout would allow for an important reduction of the energy consumed.

2 – Reduce too many treatments and redraws with event messages reception

Lots of events are fired during the opening and closing of the status bar: wifi status modification, radio… No gathering of these events is done, so it creates a lot of unnecessary treatment.

3 – Reduce too many method calls and redraws with MotionEvent event

Animations fire a lot of events and treatments. The number of events is too high and gives a too high level of performance for the animations, because produced at a rate too high for the user’s perception. Decreasing the events numbers will permit to do less treatments and redraws. Also, the FPS is rated at 60, which is too high for System UI and can be decreased.

Illustration 14: Detail Action plan on code

This action plan could be summarized as follows:

1 - Simplify the layout

The layout is complex (several levels, usage of linear layout…). There are a lot of treatments going on during every layout redraw and measure, especially during animation. Simplifying the layout would allow for an important reduction of the energy consumed.

2 - Reduce too many treatments and redraws with event messages reception

Lots of events are fired during the opening and closing of the status bar: wifi status modification, radio… No gathering of these events is done, so it creates a lot of unnecessary treatment.

Moreover, the treatments fired with these events are heavy (update and redraw even if there is no change of state). It is necessary to reduce the number of calls, by reducing the frequency of treatment, by making smarter event firing, and so on.
4 - Analyze the impact of BatteryMeterDrawable by removing it
The battery Meter is a heavy object and its generation is called several times. Removing it will permit us to understand its energy bug and to know if it is necessary to optimize it.

5 - Optimize the redraw (Global redraw and poor caching of lazy update)
After the optimization of the number of calls (cf. actions 2 and 4), the remaining calls can be optimized. Indeed, the items are cached but the update is not optimized.

6 - Analyze a potential Bug
During the test runs, we detected a potential bug: The energy and the memory increased during the test period. It is a potential memory / energy leak, to be investigated.

7 - Optimize source code as per eco-design code rules
After optimizing the main hotspots, focus can switch to improving the code with respect to code eco-design rules. These eco-design rules are those included in the GREENSPECTOR code scan tool for Java/Android language.

4. REFACTORING

4.1. Action 1 – Layout optimization

After discussion with the customer team, this best practice, although it was a big improvement on energy, was too difficult to apply (need to redesign the layout, impact on several parts of the code...) and we decided not to implement it. However, it can still be applied later to improve the energy efficiency, or in new projects.

4.2. Action 2 - Reduction of the number of refresh events

Modification
For the original version, some problems on tiles update have been detected. In fact, after every opening of the StatusBar, every tile was redrawn at least once even if its value had not changed. Then, the WifiTile was redrawn once more, the CellularTile was redrawn three times, the BluetoothTile was redrawn once and finally the BatteryTile was redrawn three times at every battery level update.

As the methodology explained, we suppressed several refreshes which made the application not fully functional, but which permitted to confirm the reduction in the number of method calls.

To fix this problem, we looked at the code of each tiles that are updated too many times like BatteryTile for example. In the original code, the method on BatteryLevelChanged was called many times, even if the battery level was the same than for the last call. No check was done on this for this method. Therefore, every call of this function implied a redraw of the BatteryTile. So we added a condition at the first line of this method to check if the data had really changed, and if not, just stop the treatment here because a redraw would be useless.

For the other tiles, the problem was really similar. For example in the CellularTile, we added a condition to stop the method setNoSims if the tile already knows if there is or not a SIM...
We called the version resulting from these changes “version UpdateReduce”.

**Profiling**

The profiling with Systrace gave us the slices number and the time taken by showing the QuickBar:

For the UpdateReduce version, we have:

For the all show/Hide, the original:

And the UpdateReduce:

The results are as follows:

<table>
<thead>
<tr>
<th>Test Name</th>
<th>Metrics</th>
<th>Original</th>
<th>Optimized</th>
<th>Gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>ShowMinStatusBar</td>
<td>Number of Slice</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>com.android.systemUI timing (ms)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ShowHideStatusBar</td>
<td>Number of Slice</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>com.android.systemUI timing (ms)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

com.android.systemUI was not the first consumer anymore but it was the RenderThread, which manages the rendering of the status bar during the animation. One other interesting indirect result was the suppression of false triggering of refreshes (redraw and treatment event if there is no visual modification):
One other interesting indirect result was the suppression of false triggering of refreshes (redraw and treatment event if there is no visual modification):

Illustration 15: Original Version - False triggering in idle

Between actions (show mini, show all and hide) remember that we could see 4 peaks which corresponded to the triggering of the refreshing of the tiles.

Illustration 16: ReduceUpdate Version - No False triggering in idle

In the UpdateReduce version, there is no such peak anymore. If a real update of the tiles is needed, refresh will be done (and a peak will happen). We can confirm this with Systrace, on the methods which have an excluded time greater than 0.5 ms:

Illustration 17: Original Version - Top consuming methods
In yellow, we can see the method linked with the refresh event.

<table>
<thead>
<tr>
<th>Method</th>
<th>Included Time (ns)</th>
<th>Excluded Time (ns)</th>
<th>Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.android.systemui.qs.TouchAnimator$FloatKeyframeSet.interpolate</td>
<td>6868</td>
<td>18437</td>
<td>1375</td>
</tr>
<tr>
<td>com.android.systemui.qs.TouchAnimator.setPosition</td>
<td>5181</td>
<td>221621</td>
<td>349</td>
</tr>
<tr>
<td>com.android.systemui.qs.TouchAnimator.setTimeValue</td>
<td>4461</td>
<td>184838</td>
<td>1375</td>
</tr>
<tr>
<td>com.android.systemui.statusbar.stack.NotificationStackScrollLayout.getNotGoneChildCount</td>
<td>2113</td>
<td>3697</td>
<td>80</td>
</tr>
<tr>
<td>com.android.systemui.statusbar.expandableNotificationRow.get intrinsicHeight</td>
<td>1235</td>
<td>3282</td>
<td>127</td>
</tr>
<tr>
<td>com.android.systemui.statusbar.expandableNotificationRow.isExpanded</td>
<td>1101</td>
<td>27320</td>
<td>33</td>
</tr>
<tr>
<td>com.android.systemui.statusbar.signal.ClusterView.apply</td>
<td>853</td>
<td>30650</td>
<td>10</td>
</tr>
<tr>
<td>com.android.systemui.statusbar.stack.NotificationStackScrollLayout.updateViewShadows</td>
<td>853</td>
<td>4141</td>
<td>15</td>
</tr>
<tr>
<td>com.android.systemui.statusbar.stack.StackScrollAlgorithm.initAlgorithmState</td>
<td>809</td>
<td>3245</td>
<td>15</td>
</tr>
<tr>
<td>com.android.systemui.statusbar.stack.StackScrollAlgorithm.updatePositionsForState</td>
<td>783</td>
<td>4827</td>
<td>15</td>
</tr>
<tr>
<td>com.android.systemui.statusbar.stack.StackScrollState.applyState</td>
<td>733</td>
<td>15383</td>
<td>65</td>
</tr>
<tr>
<td>com.android.systemui.statusbar.phone.QuickStatusBarHeader.setsetPosition</td>
<td>711</td>
<td>73522</td>
<td>83</td>
</tr>
<tr>
<td>com.android.systemui.statusbar.stack.NotificationStackScrollLayout.updateContentHeight</td>
<td>701</td>
<td>2831</td>
<td>16</td>
</tr>
<tr>
<td>com.android.systemui.statusbar.stack.StackScrollState.getViewStateForView</td>
<td>679</td>
<td>8285</td>
<td>200</td>
</tr>
<tr>
<td>com.android.systemui.qs.QSAnimator.onRelativePositionInt</td>
<td>652</td>
<td>980</td>
<td>95</td>
</tr>
<tr>
<td>com.android.systemui.statusbar.stack.NotificationStackScrollLayout.updateBackgroundBounds</td>
<td>644</td>
<td>41</td>
<td>16</td>
</tr>
<tr>
<td>com.android.systemui.statusbar.stack.NotificationStackScrollLayout.getLayoutMinHeight</td>
<td>606</td>
<td>3645</td>
<td>73</td>
</tr>
<tr>
<td>com.android.systemui.qs.QSAnimator.onsetPosition</td>
<td>588</td>
<td>187069</td>
<td>34</td>
</tr>
<tr>
<td>com.android.systemui.statusbar.stack.StackScrollState.apply</td>
<td>581</td>
<td>18959</td>
<td>15</td>
</tr>
<tr>
<td>com.android.systemui.statusbar.stack.StackScrollState.resetViewState</td>
<td>565</td>
<td>4039</td>
<td>65</td>
</tr>
<tr>
<td>com.android.systemui.qs.QSAnimator.updateAnimators</td>
<td>561</td>
<td>2005</td>
<td>1</td>
</tr>
<tr>
<td>com.android.systemui.statusbar.policy.WiFiSignalController.notifyListeners</td>
<td>555</td>
<td>6830</td>
<td>4</td>
</tr>
<tr>
<td>com.android.systemui.statusbar.stack.NotificationStackScrollLayout.getFirstChildNotGone</td>
<td>529</td>
<td>900</td>
<td>73</td>
</tr>
<tr>
<td>com.android.systemui.statusbar.stack.StackScrollAlgorithm.updateClipping</td>
<td>526</td>
<td>2213</td>
<td>16</td>
</tr>
<tr>
<td>com.android.systemui.statusbar.phone.NotificationPanelView.getHeaderTranslation</td>
<td>513</td>
<td>2876</td>
<td>33</td>
</tr>
</tbody>
</table>

Illustration 18: UpdateReduce Version - Top consuming methods

There are no more methods related to event. We reduced drastically the number of calls!

<table>
<thead>
<tr>
<th>Method</th>
<th>Number of Calls</th>
<th>Number of Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.android.systemui.qs.QS_tileState.copyTo</td>
<td>29</td>
<td>3</td>
</tr>
<tr>
<td>com.android.systemui.BatteryMeterDrawable.&lt;init&gt;</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>com.android.systemui.qs.tiles.CellularTile.handleUpdateState</td>
<td>12</td>
<td>3</td>
</tr>
<tr>
<td>com.android.systemui.BatteryMeterDrawable.draw</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>com.android.systemui.qs.QS_tile$h.handleMessage</td>
<td>34</td>
<td>12</td>
</tr>
</tbody>
</table>

Illustration 19: Number of calls of refresh methods

Measurement
The measurements are the following for the ShowHideStatusBar:

Illustration 20: Comparison with original for ShowHideStatusBar
(screenshot from GREENSPECTOR “Evolution” tab)
The gain in energy consumption is -10.75 μAh/s (-3.56 %). This is good, but we were expecting better results, judging upon the profiling aspects. One explanation is that the ShowHideStatusBar energy test has been extended to include an idle period, which lowers the mean consumption during the test. More tests could be conducted on other Android devices with more accurate energy probes, this flavor of the Nexus showing its limitations here.

When we checked the gain of all the tests, we had:

<table>
<thead>
<tr>
<th>Platform CPU</th>
<th>11.35 %</th>
<th>11.17 %</th>
<th>-0.18 %</th>
<th>-1.61 %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process Data</td>
<td>0 B</td>
<td>0 B</td>
<td>0 B</td>
<td>0 %</td>
</tr>
<tr>
<td>Process Memory</td>
<td>120.74 MB</td>
<td>118.38 MB</td>
<td>-2.36 MB</td>
<td>-1.96 %</td>
</tr>
</tbody>
</table>

Illustration 21: Global gain of all test for CPU and Memory

We had some gains in CPU and Memory. We decreased the RAM consumption by 2 MB. However it is interesting to note that we also slightly decreased the pressure on the CPU.

4.3. Action 3 – Animation and redraw optimization

Modification
To reduce the number of animation treatments, one simple modification was to reduce the amount of input events. For that, in dispatchTouchEvent in com.android.systemui.statusbar.phone.StatusBarWindowView we took one action to delete one event out of 3. The result is not visible for the user. If we wanted to keep the same performance (and not to loose events), then the same gain (and even more) could be obtained by optimizing the treatment of animations and the layout.

We added the modification to the modifications of Action 2 (incremental modification).

Profiling
The Systrace analysis of the optimized version gives the following metrics for Showing the status bar:

<table>
<thead>
<tr>
<th>Name</th>
<th>Wall Duration</th>
<th>Occurrences</th>
</tr>
</thead>
<tbody>
<tr>
<td>android_systemui</td>
<td>264,650 ms</td>
<td>330</td>
</tr>
<tr>
<td>surfaceflinger</td>
<td>157,982 ms</td>
<td>150</td>
</tr>
<tr>
<td>RenderThread</td>
<td>157,793 ms</td>
<td>250</td>
</tr>
</tbody>
</table>

The metrics for original version was:

The result :
The global gain for the test ShowHide Status bar is the following:
We got a gain of -28 μAh/s (-9.3%)!

We also reduced the pressure on the memory management, as shown by the 2 graphs below. On the original version, the garbage collector runs every 1 minute and 20 seconds. On the optimized version, it runs every 2 minutes 30. There are less objects to destroy, so the GC is not called as much as before.
4.4. **Action 4 - BatteryMeterDrawable optimization**

After the action 2, we had reduced and suppressed a lot of calls. The BatteryMeterDrawable object was still heavy, but there were no more calls on Show/Hide of status bar. This action was deemed as not relevant for the sort duration of the audit.

4.5. **Action 5 – Optimize the redraw (Global redraw and poor caching of lazy update)**

During the audit, we did not have enough time to work on this action. However, decreasing the triggering of refreshes had made this action less important (although still necessary).

4.6. **Action 6 – Analysis of Energy leak bug**

We identified two potentials bugs:
- There is a memory leak. Even if we see that the garbage collector does its job, the memory is continuously increasing from test run to test run. The energy of all tests also continuously increases. We had identified this during the measurement step, and we had adapted our protocol of measurement not to suffer from that. Let it live, and the idle consumption goes from 85μAh/s to 150μAh/s!

Illustration 26: Evolution of reference energy consumption for original version (showing a leak)

- When the status bar is expanded and the user locks the screen, it seems that the listener which permits to update the tiles stays open. This creates unnecessary treatments and Redraws which happen even though the screen is off.

4.7. **Action 7 – Improve the code as per code eco-design rules**

During the audit, we lacked the necessary time to conduct this action. We agreed with the Customer team that this one will be done by its development teams. Indeed, the use of GREENSPECTOR tools (including the Eclipse plug-in and Android Studio plug-in) allows any developer to easily scan their code and apply the “green” rules.

**Note:** The scanning had shown that no “high priority” green rules had been infringed. Green rules are interesting with a mid and long term vision. Like maintainability and other best practices which improve the code quality, it will improve the energy consumption of software as coding goes along.
5. CONCLUSIONS

5.1. Performance is not enough. Watch your efficiency.

Mobile hardware has become more and more powerful. The smartphone used during this audit boasted as many as 8 CPUs and one GPU. Indeed, the application uses all the available CPUs and there is no performance problem. This meets Wirth's Law: "Software is getting slower more rapidly than hardware becomes faster." ([https://en.wikipedia.org/wiki/Wirth's_law](https://en.wikipedia.org/wiki/Wirth's_law)).

Hardware provides more and more power and resources to the software, hence the software uses all the resources. The only limitations for the software are defined by two factors: the user's perception of performance (aka speed), and the available hardware resources.

Getting back to this audit, we saw that the performance level of the SystemUI app was quite good. But this was done at the cost of using all the available resources without limitation, leading to an app consuming way more battery than it could, or should. We showed that the same performance level may be achieved with a lower consumption of energy, thus granting the user a longer battery life.

One of our usual proposals in order to reduce the consumption of resources is to limit it with a budget. Therefore we introduce a third limitation factor, which will permit to better control the behavior of the software. This concept is already applied in the performance domain with the RAIL model. But the current performance models need to be improved. For example, there is this reflection on the RAIL model proposed by Paul Irish and Paul Lewis: Add B (for battery) and an M (for memory), turning into BLAIMR, PRIMAL. Just as we have a performance budget, we need an energy consumption budget. Set your own target, like “this software should not double the discharge rate of the battery”, or “this software should not increases by more than 10% the discharge rate of the battery when in Idle mode…”, and so on.

Timing or Speed performance is not the only solution to improve the software efficiency. This leads to over-consuming software and to empty batteries. Performance models need to be improved and to integrate energy and resources consumptions.

5.2. Measure, Measure, Measure

Our goal was to reduce the power consumption of SystemUI, an Android core application. We have succeeded in a very short time frame, and above all we have shown that it was possible to go way further.

We managed to do it because we had both:
- A good method: look for the big stones, proceed by elimination, and above all MEA-SU-RE. It’s by measurement that knowledge arrives, and by measurement that progress is evaluated. Once the big stones are identified, you spend your time and your expertise much more efficiently.
- A good tool: since you have to measure, let it be easy to do, and let the findings be relevant. We have shown that GREENSPECTOR’s API offered a nice versatility for in-house Android developers, and that GREENSPECTOR’s interface allowed to easily follow the findings and progresses.
Still some work to do

The SystemUI app can still be improved in many ways. Event programming is overused and has some drawbacks: no clear management of the impact of event triggering, redraws happening way too often… The performance of the UI is also too high: maintaining a 60 FPS is over-quality.

We managed to improve the energy consumption by 2 quick actions. The memory was improved also. But the impact on the system is still too high. An impact less that 2 times the reference consumption would be more acceptable (See? This is an energy consumption budget!). Improvements can be continued in order to reach this goal…

Want to learn more about software ecodesign?

www.greenspector.com

+33 (0) 9 51 44 55 79
contact@greenspector.com