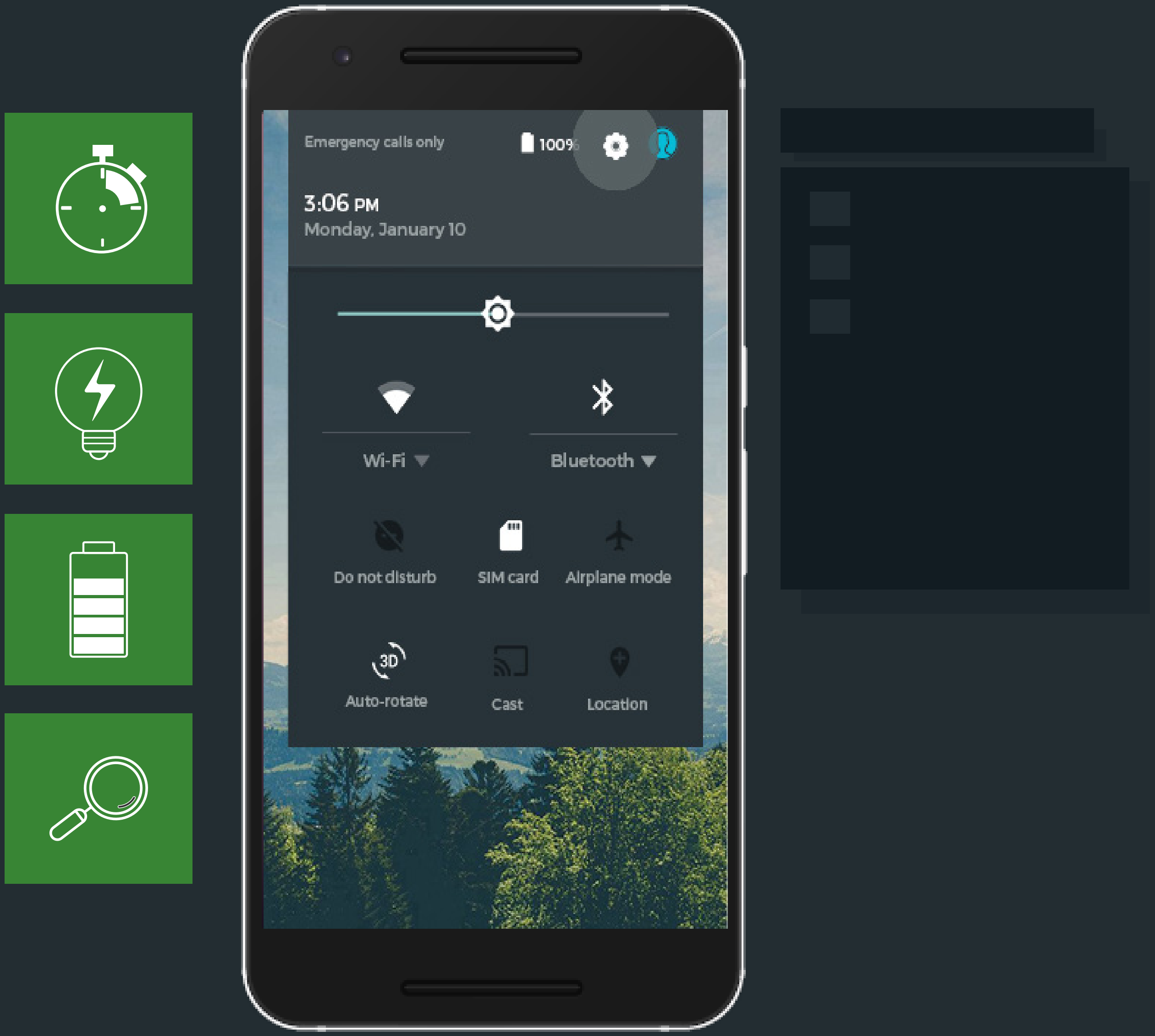
GREENSPECTOR

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



Author: Olivier PHILIPPOT, 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

GREENSPECTOR

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

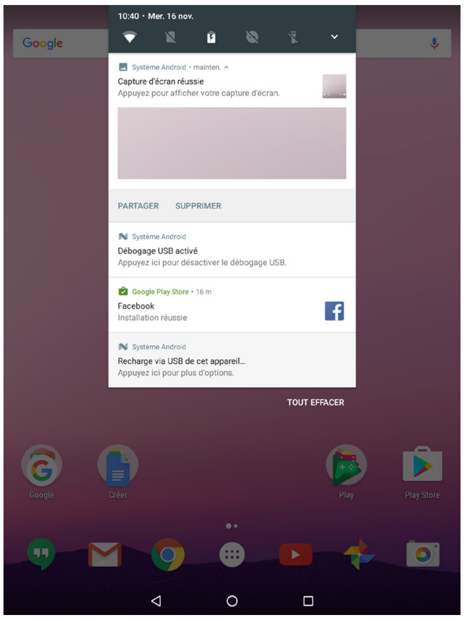
The application under test was a core Android application:
- System UI
- Version: from AOSP repository (1.0.3)

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.


10:40 · Mer. 16 nov.

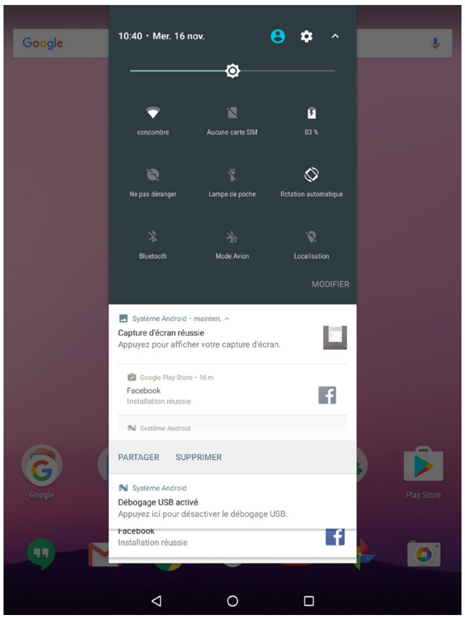
Système Android · mainten.
Capture d'écran réussie
Appuyez pour afficher votre capture d'écran.

Système Android
Débogage USB activé
Appuyez ici pour désactiver le débogage USB.

Google Play Store · 16 m
Facebook
Installation réussie

Système Android
Recharge via USB de cet appareil...
Appuyez ici pour plus d'options.

Illustration 1: Quick bar


10:40 · Mer. 16 nov.

Système Android · mainten.
Capture d'écran réussie
Appuyez pour afficher votre capture d'écran.

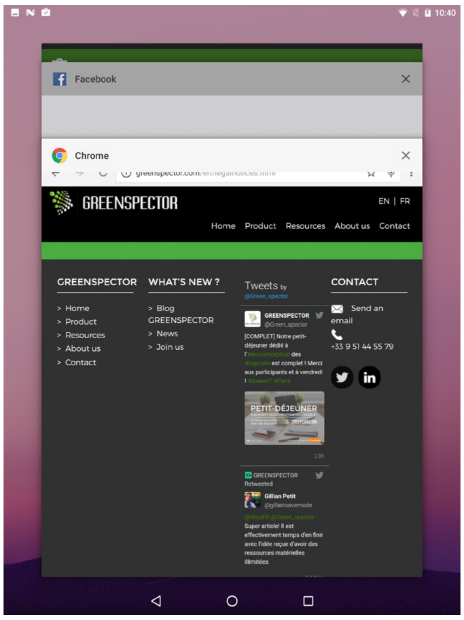
Google Play Store · 16 m
Facebook
Installation réussie

Système Android

Système Android
Débogage USB activé
Appuyez ici pour désactiver le débogage USB.

Facebook
Installation réussie

Illustration 2: Show recent app


GREENSPECTOR @Green_spector
[COMPLET] Notre petit-déjeuner dédié à l'#écoconception des #logiciels est complet ! Merci aux participants et à vendredi ! #GreenIT #Paris

GREENSPECTOR Retweeted
Gillian Petit @gilliansavemode
@AtosFR @Green_spector Super article! Il est effectivement temps d'en finir avec l'idée reçue d'avoir des ressources matérielles illimitées

Illustration 3: All status bar

2. METHODOLOGY & TEST BENCH SETTING

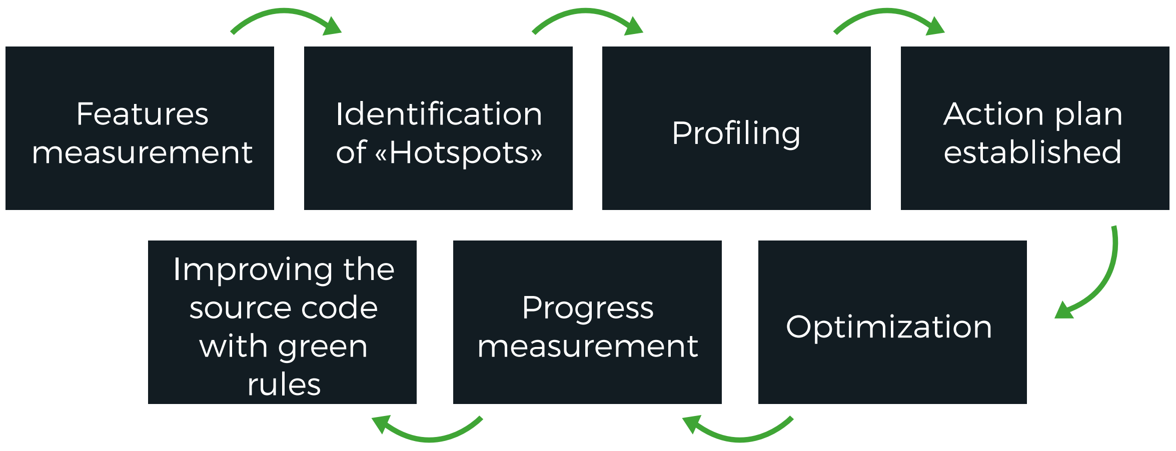
2.1. Methodology

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.

Drawing 1 : Synthetic methodology



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.

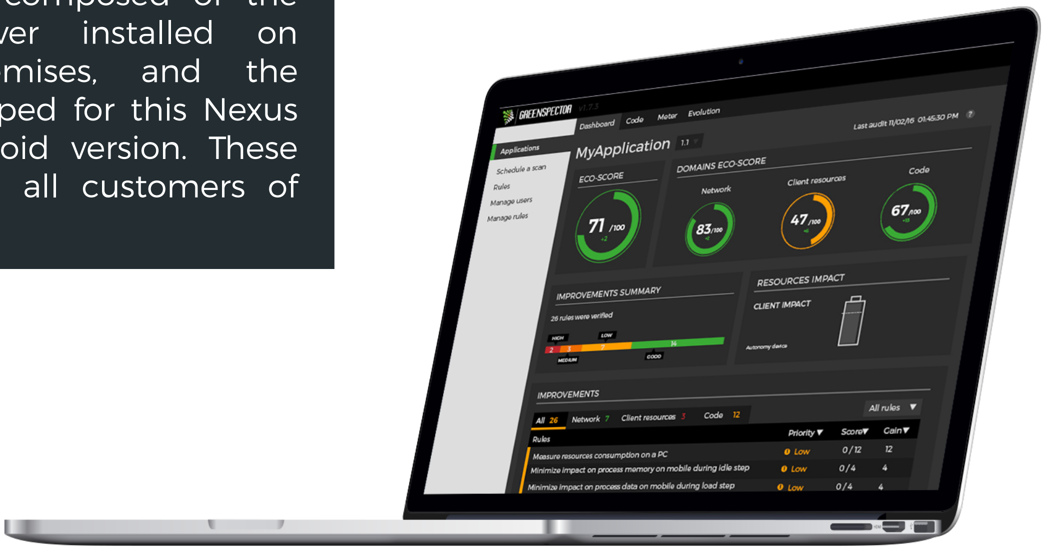
Source code Analysis

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.



Rules	Priority	Score	Gain
Measure resources consumption on a PC	Low	0 / 12	12
Minimize impact on process memory on mobile during idle step	Low	0 / 4	4
Minimize impact on process data on mobile during load step	Low	0 / 4	4

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:

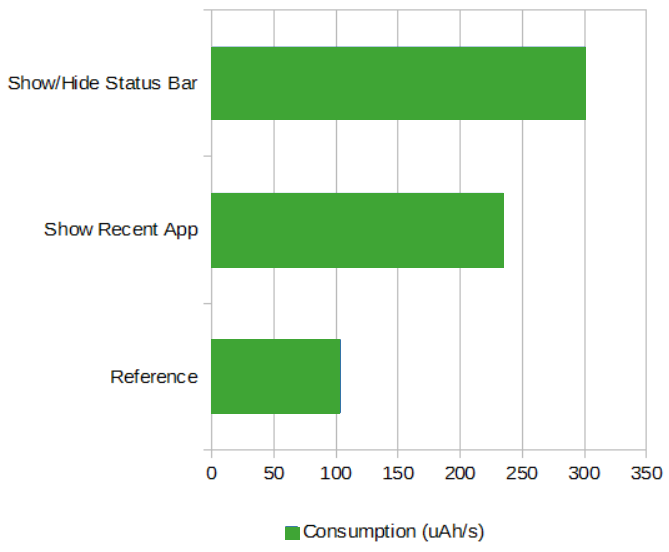


Illustration 5: Energy consumption of original version

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 :

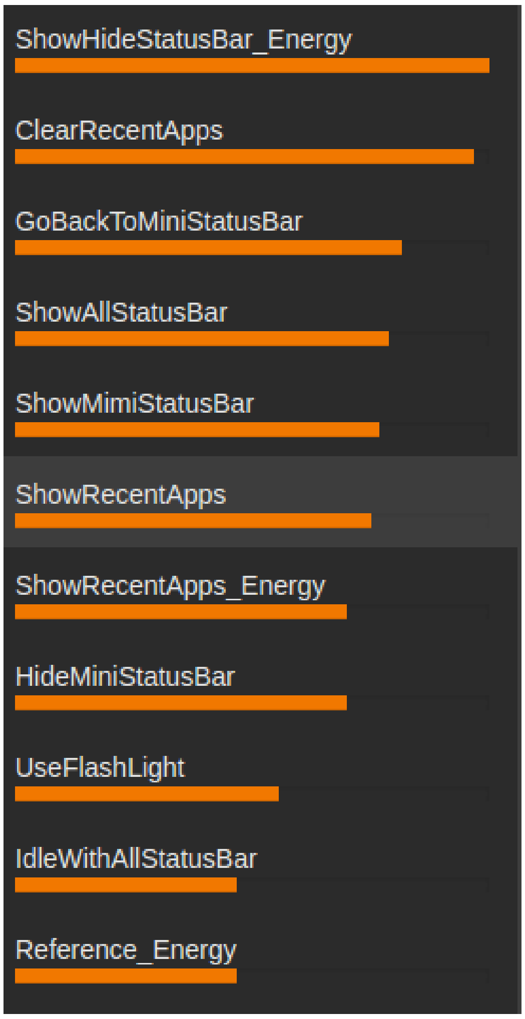


Illustration 6: CPU measure in Greenspector

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

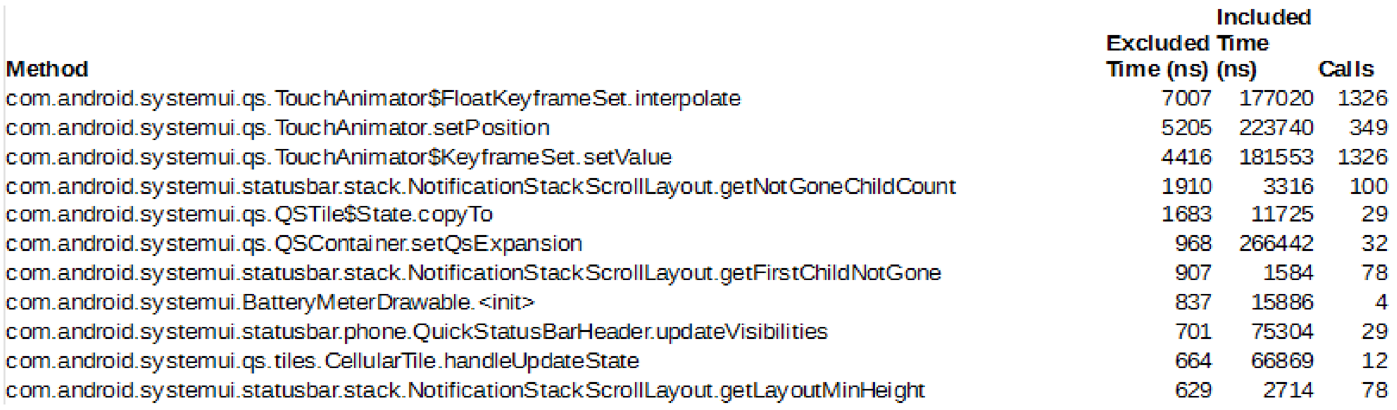
An optimization work is not an exhaustive and planned approach. We are in a constant search of balance between the hoped-for gains, and the workload that would be needed to obtain these gains.

> 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.

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 :



Method	Excluded Time (ns)	Included Time (ns)	Calls
com.android.systemui.qs.TouchAnimator$FloatKeyframeSet.interpolate	7007	177020	1326
com.android.systemui.qs.TouchAnimator.setPosition	5205	223740	349
com.android.systemui.qs.TouchAnimator$KeyframeSet.setValue	4416	181553	1326
com.android.systemui.statusbar.stack.NotificationStackScrollLayout.getNotGoneChildCount	1910	3316	100
com.android.systemui.qs.QSTile$State.copyTo	1683	11725	29
com.android.systemui.qs.QSContainer.setQsExpansion	968	266442	32
com.android.systemui.statusbar.stack.NotificationStackScrollLayout.getFirstChildNotGone	907	1584	78
com.android.systemui.BatteryMeterDrawable.<init>	837	15886	4
com.android.systemui.statusbar.phone.QuickStatusBarHeader.updateVisibilities	701	75304	29
com.android.systemui.qs.tiles.CellularTile.handleUpdateState	664	66869	12
com.android.systemui.statusbar.stack.NotificationStackScrollLayout.getLayoutMinHeight	629	2714	78

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:

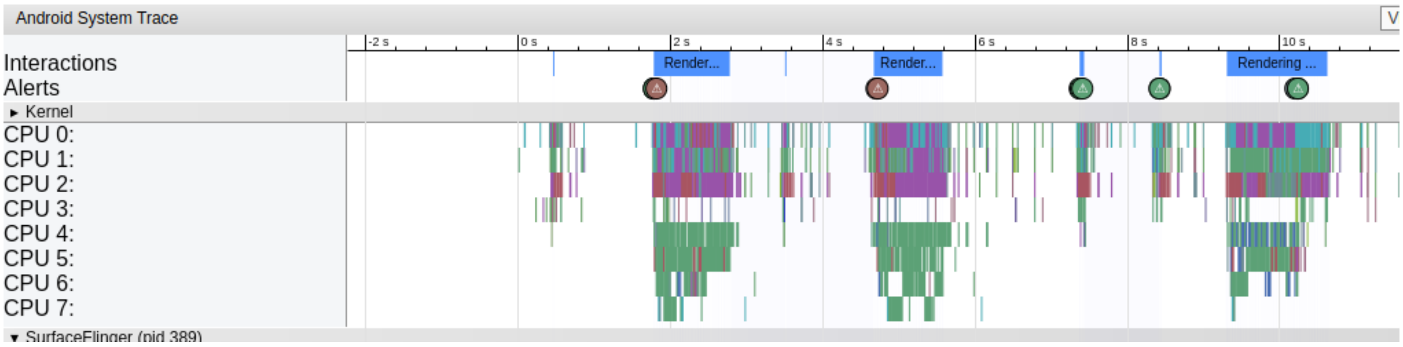
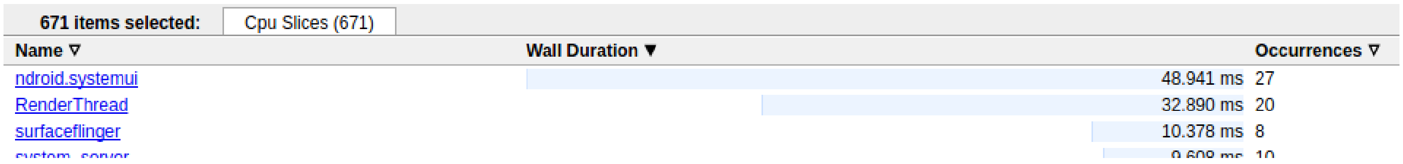


Illustration 8: Original Version - False triggering of redraw

Between user actions (show mini, show all and hide) we see 4 peaks which correspond to the triggering of the tiles redraws. 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 redraws.

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



671 items selected: Cpu Slices (671)

Name ▽	Wall Duration ▼	Occurrences ▽
ndroid.systemui	48.941 ms	27
RenderThread	32.890 ms	20
surfaceflinger	10.378 ms	8
system_server	9.608 ms	10

Illustration 9: Cost of refreshing the tiles

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 redraws. 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:

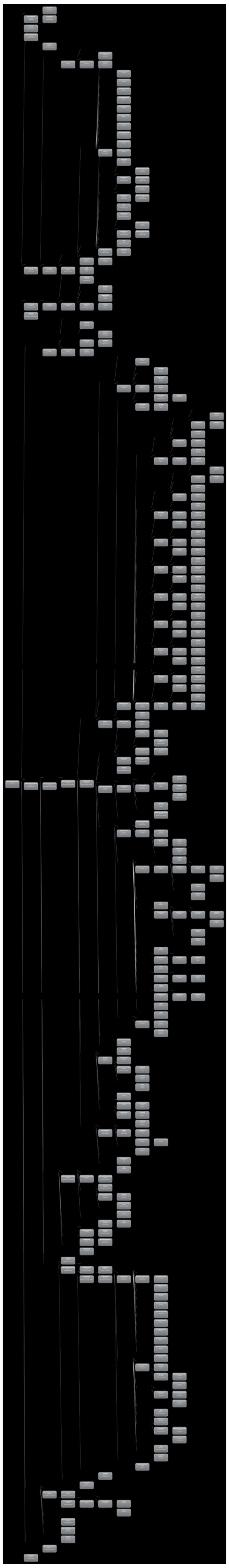


Illustration 10: Layout architecture

There are 12 levels, which is a lot, and all tiles are complex:

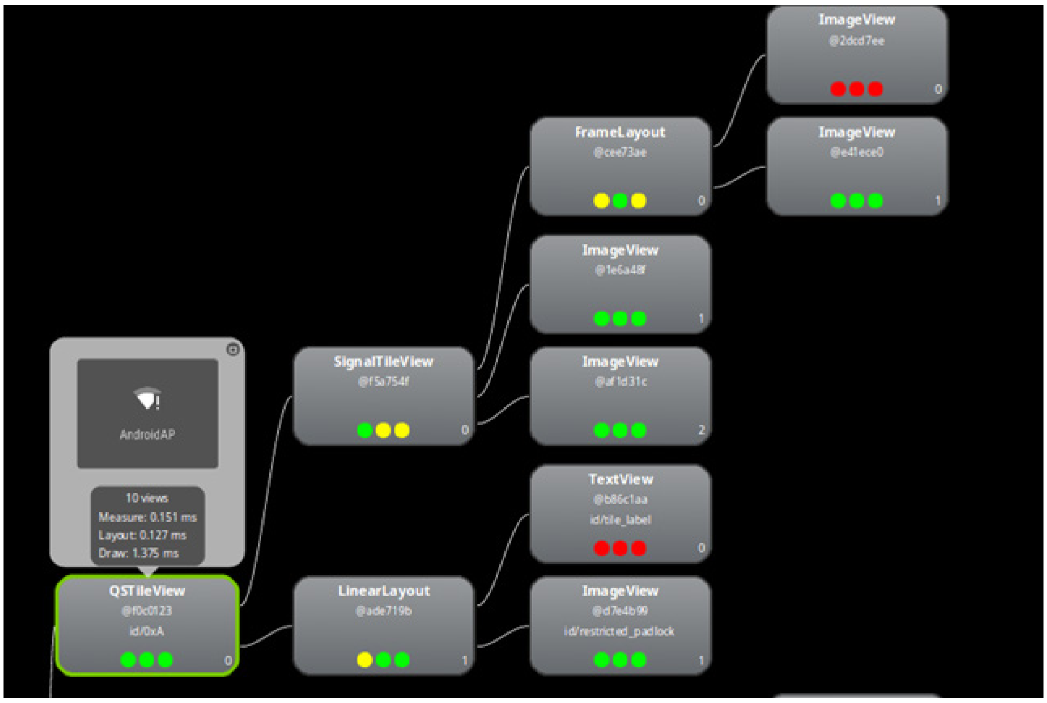


Illustration 11: Signal view layer hierarchy

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:

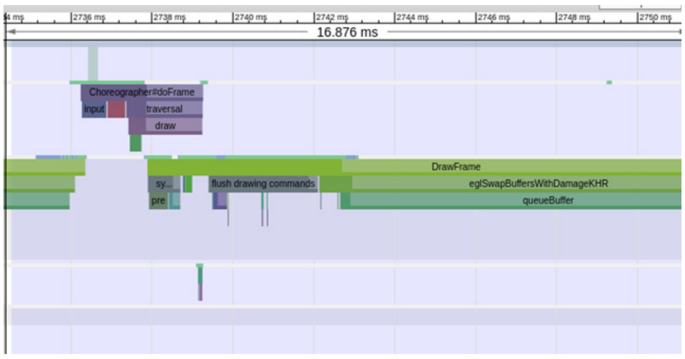


Illustration 12: Tiles drawing profilingy

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 :

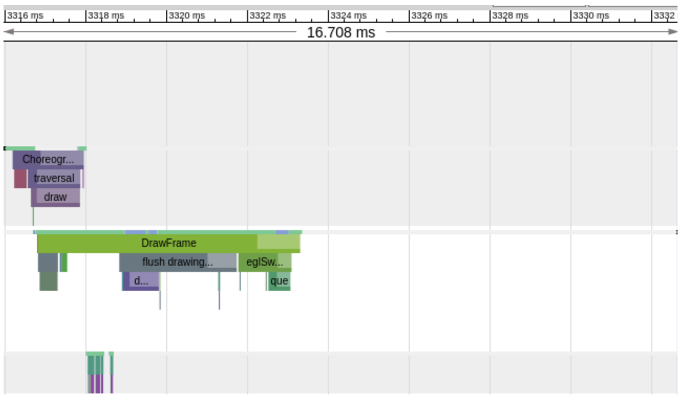
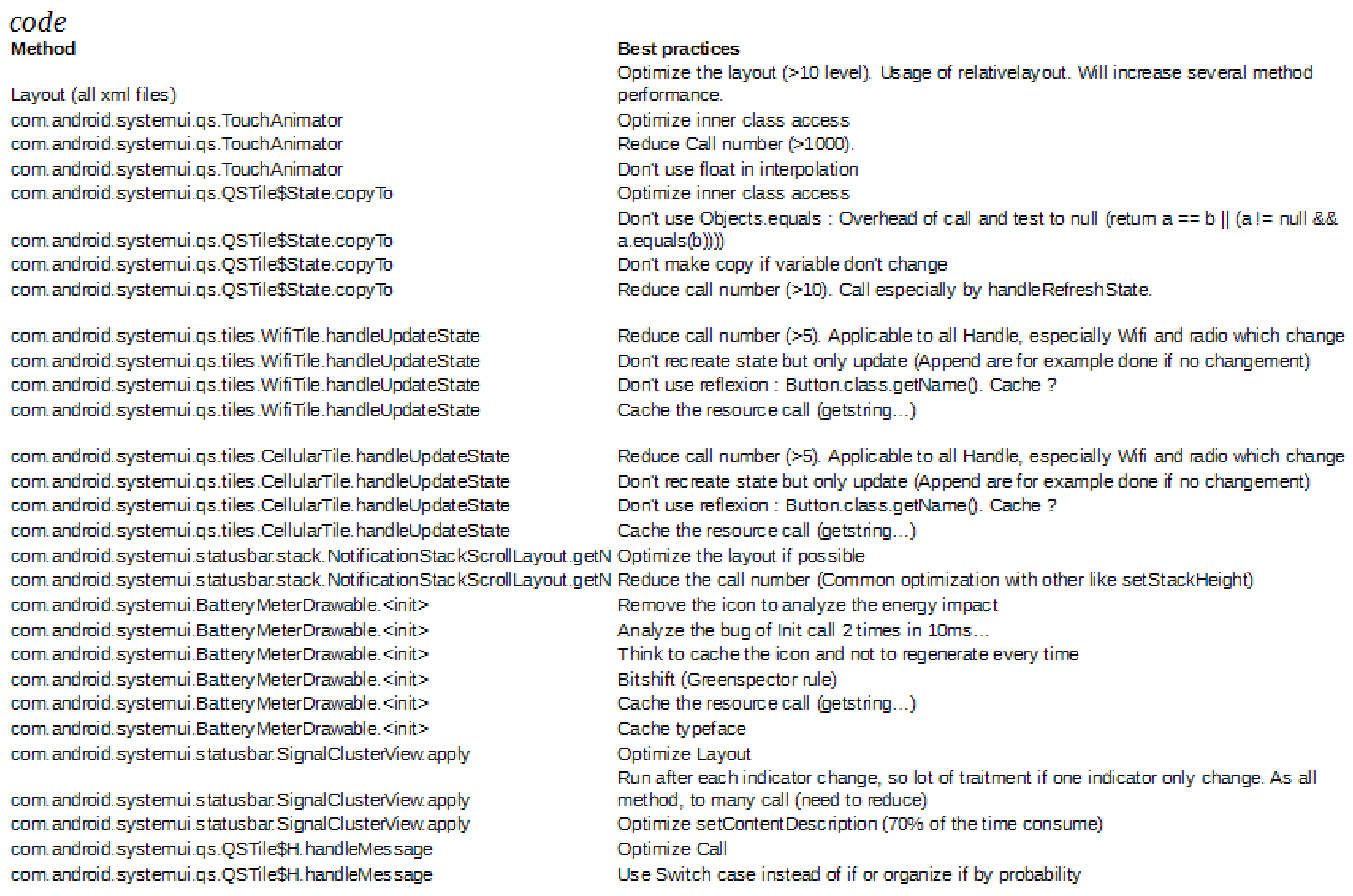


Illustration 13: Quick bar Drawing profiling

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	Best practices
Layout (all xml files)	Optimize the layout (>10 level). Usage of relativelayout. Will increase several method performance.
com.android.systemui.qs.TouchAnimator	Optimize inner class access
com.android.systemui.qs.TouchAnimator	Reduce Call number (>1000).
com.android.systemui.qs.TouchAnimator	Don't use float in interpolation
com.android.systemui.qs.QSTile$State.copyTo	Optimize inner class access
com.android.systemui.qs.QSTile$State.copyTo	Don't use Objects.equals : Overhead of call and test to null (return a == b || (a != null && a.equals(b)))
com.android.systemui.qs.QSTile$State.copyTo	Don't make copy if variable don't change
com.android.systemui.qs.QSTile$State.copyTo	Reduce call number (>10). Call especially by handleRefreshState.
com.android.systemui.qs.tiles.WifiTile.handleUpdateState	Reduce call number (>5). Applicable to all Handle, especially Wifi and radio which change
com.android.systemui.qs.tiles.WifiTile.handleUpdateState	Don't recreate state but only update (Append are for example done if no changement)
com.android.systemui.qs.tiles.WifiTile.handleUpdateState	Don't use reflexion : Button.class.getName(). Cache ?
com.android.systemui.qs.tiles.WifiTile.handleUpdateState	Cache the resource call (getstring...)
com.android.systemui.qs.tiles.CellularTile.handleUpdateState	Reduce call number (>5). Applicable to all Handle, especially Wifi and radio which change
com.android.systemui.qs.tiles.CellularTile.handleUpdateState	Don't recreate state but only update (Append are for example done if no changement)
com.android.systemui.qs.tiles.CellularTile.handleUpdateState	Don't use reflexion : Button.class.getName(). Cache ?
com.android.systemui.qs.tiles.CellularTile.handleUpdateState	Cache the resource call (getstring...)
com.android.systemui.statusbar.stack.NotificationStackScrollLayout.getN	Optimize the layout if possible
com.android.systemui.statusbar.stack.NotificationStackScrollLayout.getN	Reduce the call number (Common optimization with other like setStackHeight)
com.android.systemui.BatteryMeterDrawable.<init>	Remove the icon to analyze the energy impact
com.android.systemui.BatteryMeterDrawable.<init>	Analyze the bug of Init call 2 times in 10ms...
com.android.systemui.BatteryMeterDrawable.<init>	Think to cache the icon and not to regenerate every time
com.android.systemui.BatteryMeterDrawable.<init>	Bitshift (Greenspector rule)
com.android.systemui.BatteryMeterDrawable.<init>	Cache the resource call (getstring...)
com.android.systemui.BatteryMeterDrawable.<init>	Cache typeface
com.android.systemui.statusbar.SignalClusterView.apply	Optimize Layout
com.android.systemui.statusbar.SignalClusterView.apply	Run after each indicator change, so lot of traitment if one indicator only change. As all method, to many call (need to reduce)
com.android.systemui.statusbar.SignalClusterView.apply	Optimize setContentDescription (70% of the time consume)
com.android.systemui.qs.QSTile$H.handleMessage	Optimize Call
com.android.systemui.qs.QSTile$H.handleMessage	Use Switch case instead of if or organize if by probability

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.

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.

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.

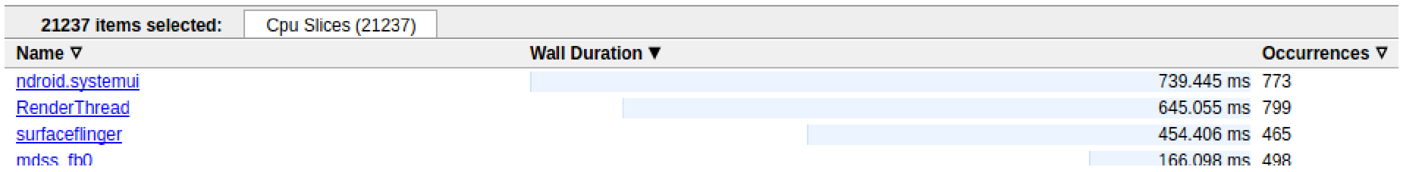
To fix this problem, we looked at the code of each tiles that are updated too many times like BatteryTile for example. In the original code, the method on BatteryLevelChanged was called many times, even if the battery level was the same than for the last call. No check was done on this for this method. 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

```
@Override
public void setIsAirplaneMode(IconState icon) {
        // Greenspector-UpdateReduce: We have to check if the airplane mode has changed
        if (mInfo.airplaneModeEnabled == icon.visible) return;
        mInfo.airplaneModeEnabled = icon.visible;
        refreshState(mInfo);
}
```

card in the phone.
We called the version resulting from theses changes “version UpdateReduce”.

Profiling

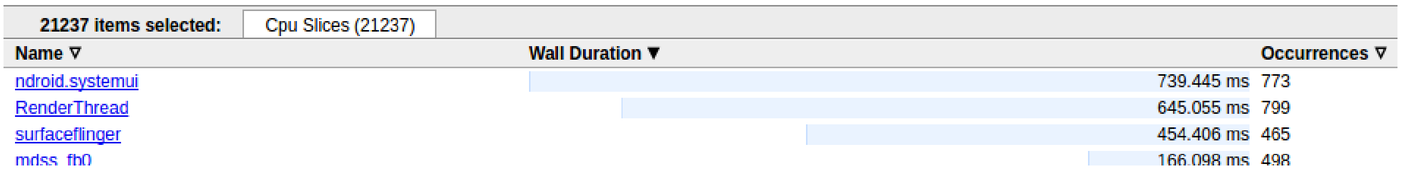
The profiling with Systrace gave us the slices number and the time taken by showing the QuickBar:



21237 items selected: Cpu Slices (21237)

Name ∇	Wall Duration ▼	Occurrences ∇
ndroid.systemui	739.445 ms	773
RenderThread	645.055 ms	799
surfaceflinger	454.406 ms	465
mdss_fb0	166.098 ms	498

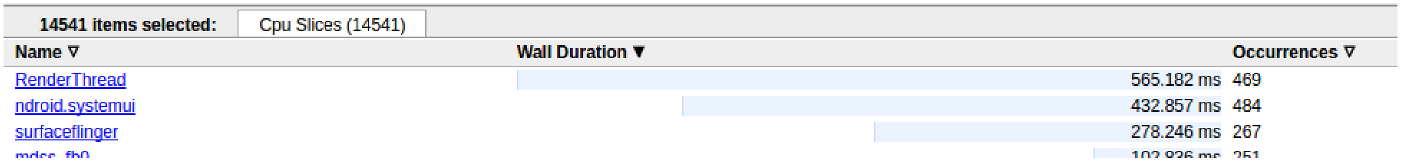
For the UpdateReduce version, we have: For the all show/Hide, the original:



21237 items selected: Cpu Slices (21237)

Name ∇	Wall Duration ▼	Occurrences ∇
ndroid.systemui	739.445 ms	773
RenderThread	645.055 ms	799
surfaceflinger	454.406 ms	465
mdss_fb0	166.098 ms	498

And the UpdateReduce:



14541 items selected: Cpu Slices (14541)

Name ∇	Wall Duration ▼	Occurrences ∇
RenderThread	565.182 ms	469
ndroid.systemui	432.857 ms	484
surfaceflinger	278.246 ms	267
mdss_fb0	102.836 ms	251

The results are as follows:

TestName	Metrics	Original	Optimized	Gain
ShowMinStatudBar	Number of Slice	6667	5812	12.8%
	com.android.systemUI timing (ms)	264	168	36.4%
ShowHideStatudBar	Number of Slice	21237	14541	31.5%
	com.android.systemUI timing (ms)	739	432	41.5%

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):

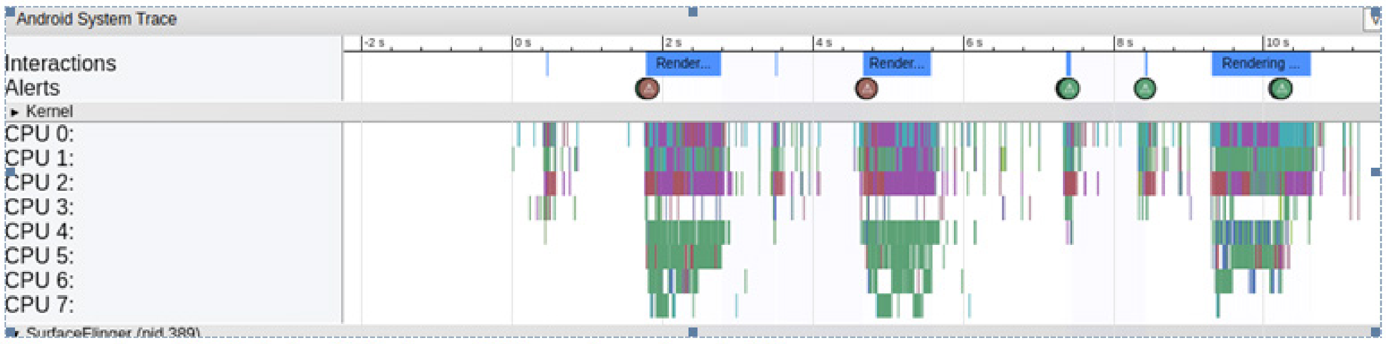


Illustration 15: Original Version - False triggering in idle

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.

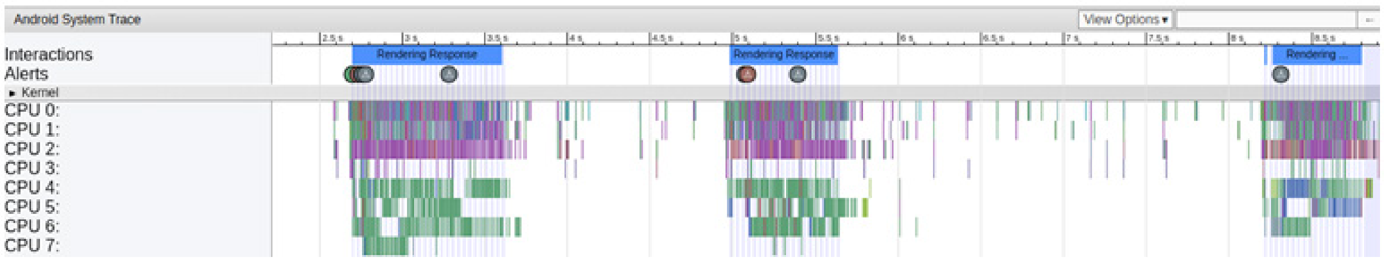
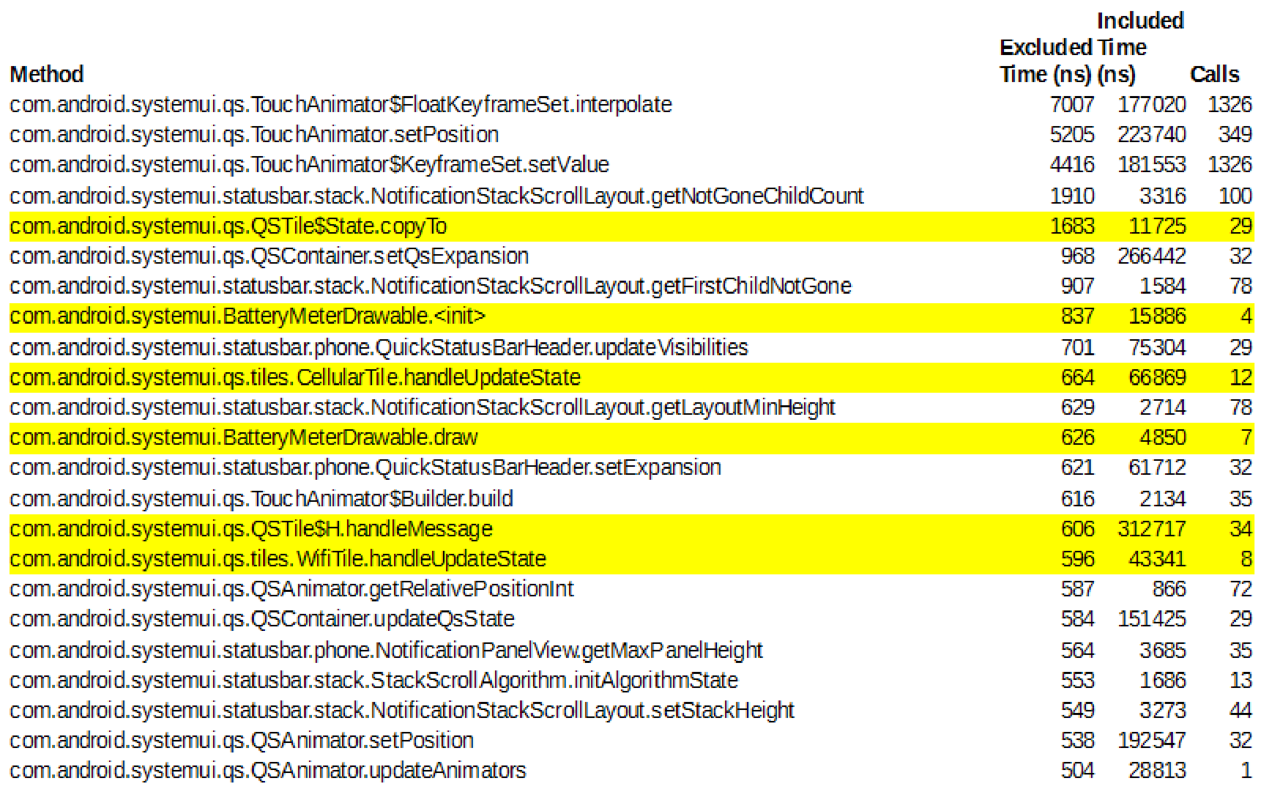


Illustration 16: ReduceUpdate Version - No False triggering in idle

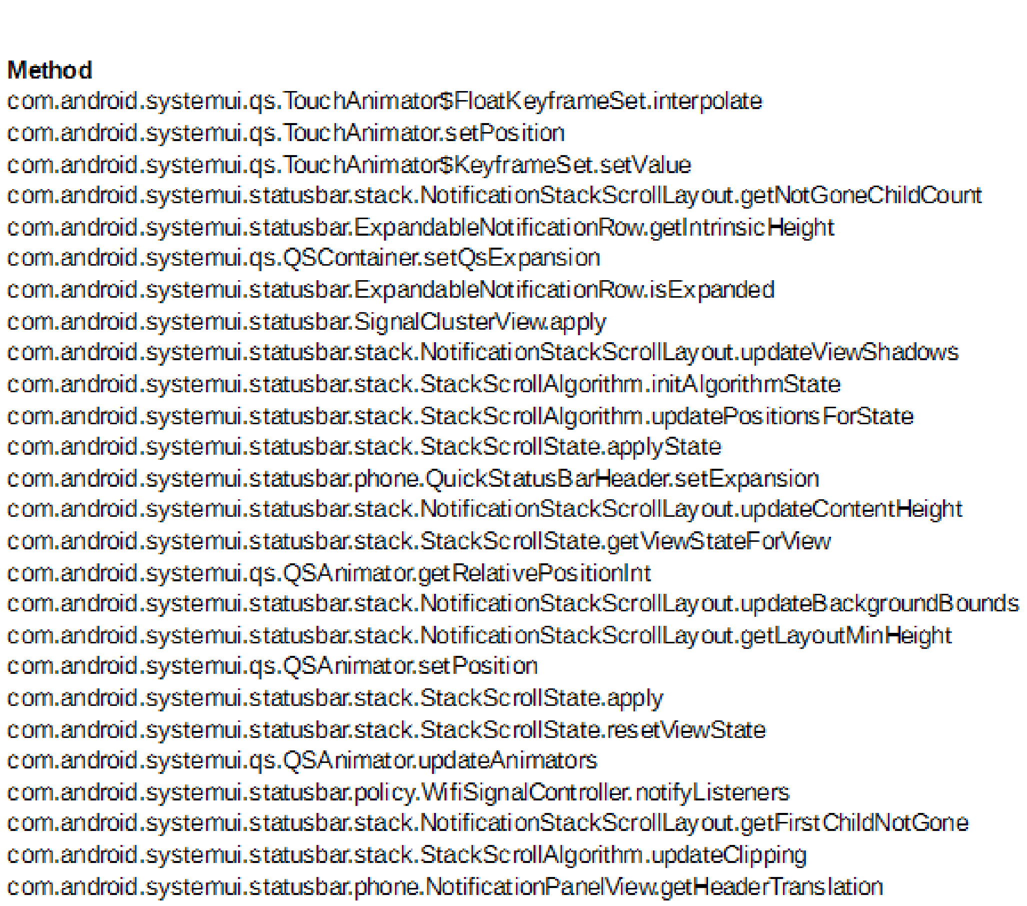
In the UpdateReduce version, there is no such peak anymore. If a real update of the tiles is needed, refresh will be done (and a peak will happen).
We can confirm this with Systrace, on the methods which have an excluded time greater that 0.5 ms:



Method	Excluded Time (ns)	Included Time (ns)	Calls
com.android.systemui.qs.TouchAnimator$FloatKeyframeSet.interpolate	7007	177020	1326
com.android.systemui.qs.TouchAnimator.setPosition	5205	223740	349
com.android.systemui.qs.TouchAnimator$KeyframeSet.setValue	4416	181553	1326
com.android.systemui.statusbar.stack.NotificationStackScrollLayout.getNotGoneChildCount	1910	3316	100
com.android.systemui.qs.QSTile$State.copyTo	1683	11725	29
com.android.systemui.qs.QSContainer.setQsExpansion	968	266442	32
com.android.systemui.statusbar.stack.NotificationStackScrollLayout.getFirstChildNotGone	907	1584	78
com.android.systemui.BatteryMeterDrawable.<init>	837	15886	4
com.android.systemui.statusbar.phone.QuickStatusBarHeader.updateVisibilities	701	75304	29
com.android.systemui.qs.tiles.CellularTile.handleUpdateState	664	66869	12
com.android.systemui.statusbar.stack.NotificationStackScrollLayout.getLayoutMinHeight	629	2714	78
com.android.systemui.BatteryMeterDrawable.draw	626	4850	7
com.android.systemui.statusbar.phone.QuickStatusBarHeader.setExpansion	621	61712	32
com.android.systemui.qs.TouchAnimator$Builder.build	616	2134	35
com.android.systemui.qs.QSTile$H.handleMessage	606	312717	34
com.android.systemui.qs.tiles.WifiTile.handleUpdateState	596	43341	8
com.android.systemui.qs.QSAnimator.getRelativePositionInt	587	866	72
com.android.systemui.qs.QSContainer.updateQsState	584	151425	29
com.android.systemui.statusbar.phone.NotificationPanelView.getMaxPanelHeight	564	3685	35
com.android.systemui.statusbar.stack.StackScrollAlgorithm.initAlgorithmState	553	1686	13
com.android.systemui.statusbar.stack.NotificationStackScrollLayout.setStackHeight	549	3273	44
com.android.systemui.qs.QSAnimator.setPosition	538	192547	32
com.android.systemui.qs.QSAnimator.updateAnimators	504	28813	1

Illustration 17: Original Version - Top consuming methods

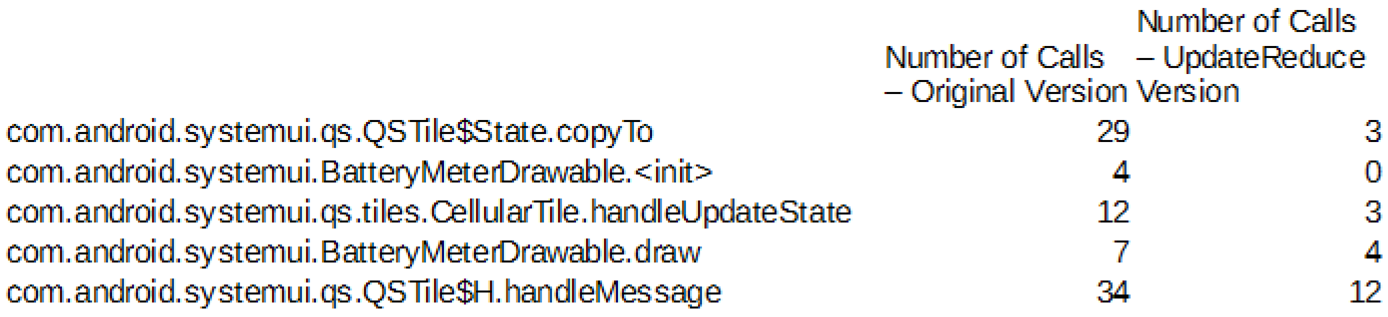
In yellow, we can see the method linked with the refresh event.



Method	Excluded Time (ns)	Included Time (ns)	Calls
com.android.systemui.qs.TouchAnimator$FloatKeyframeSet.interpolate	6868	180437	1375
com.android.systemui.qs.TouchAnimator.setPosition	5181	221681	349
com.android.systemui.qs.TouchAnimator$KeyframeSet.setValue	4461	184898	1375
com.android.systemui.statusbar.stack.NotificationStackScrollLayout.getNotGoneChildCount	2113	3687	80
com.android.systemui.statusbar.ExpandableNotificationRow.getIntrinsicHeight	1235	3282	127
com.android.systemui.qs.QSContainer.setQsExpansion	1101	273280	33
com.android.systemui.statusbar.ExpandableNotificationRow.isExpanded	1015	2399	254
com.android.systemui.statusbar.SignalClusterView.apply	863	30660	10
com.android.systemui.statusbar.stack.NotificationStackScrollLayout.updateViewShadows	863	4141	16
com.android.systemui.statusbar.stack.StackScrollAlgorithm.initAlgorithmState	809	3245	16
com.android.systemui.statusbar.stack.StackScrollAlgorithm.updatePositionsForState	793	4827	16
com.android.systemui.statusbar.stack.StackScrollState.applyState	793	15383	65
com.android.systemui.statusbar.phone.QuickStatusBarHeader.setExpansion	711	73522	33
com.android.systemui.statusbar.stack.NotificationStackScrollLayout.updateContentHeight	701	2831	16
com.android.systemui.statusbar.stack.StackScrollState.getViewStateForView	679	8261	200
com.android.systemui.qs.QSAnimator.getRelativePositionInt	662	981	96
com.android.systemui.statusbar.stack.NotificationStackScrollLayout.updateBackgroundBounds	644	3160	16
com.android.systemui.statusbar.stack.NotificationStackScrollLayout.getLayoutMinHeight	606	3645	73
com.android.systemui.qs.QSAnimator.setPosition	588	187069	34
com.android.systemui.statusbar.stack.StackScrollState.apply	581	18959	16
com.android.systemui.statusbar.stack.StackScrollState.resetViewState	565	4039	65
com.android.systemui.qs.QSAnimator.updateAnimators	561	29055	1
com.android.systemui.statusbar.policy.WifiSignalController.notifyListeners	535	6890	4
com.android.systemui.statusbar.stack.NotificationStackScrollLayout.getFirstChildNotGone	529	900	73
com.android.systemui.statusbar.stack.StackScrollAlgorithm.updateClipping	526	2213	16
com.android.systemui.statusbar.phone.NotificationPanelView.getHeaderTranslation	513	2876	33

Illustration 18: UpdateReduce Version - Top consuming methods

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

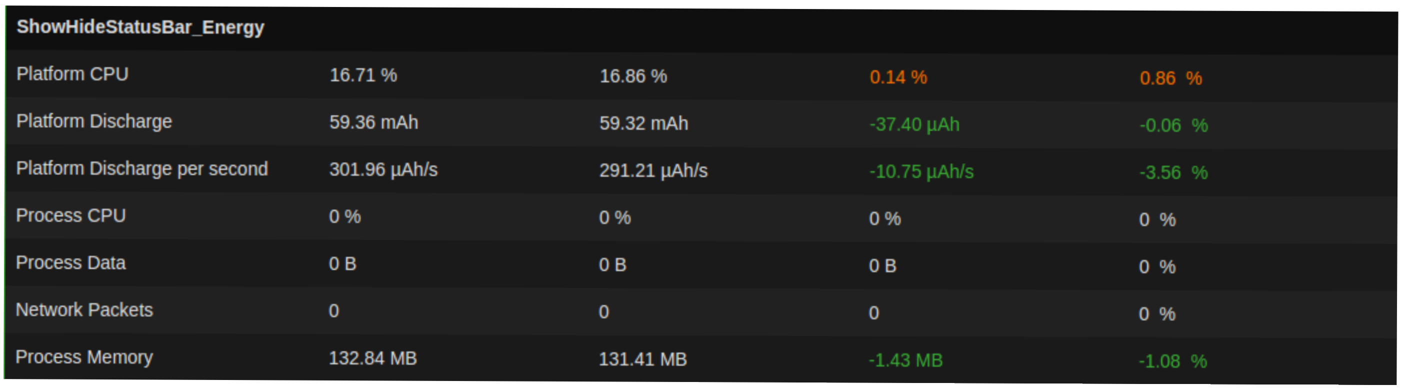


	Number of Calls – Original Version	Number of Calls – UpdateReduce Version
com.android.systemui.qs.QSTile$State.copyTo	29	3
com.android.systemui.BatteryMeterDrawable.<init>	4	0
com.android.systemui.qs.tiles.CellularTile.handleUpdateState	12	3
com.android.systemui.BatteryMeterDrawable.draw	7	4
com.android.systemui.qs.QSTile$H.handleMessage	34	12

Illustration 19: Number of calls of refresh methods

Measurement

The measurements are the following for the ShowHideStatusBar:



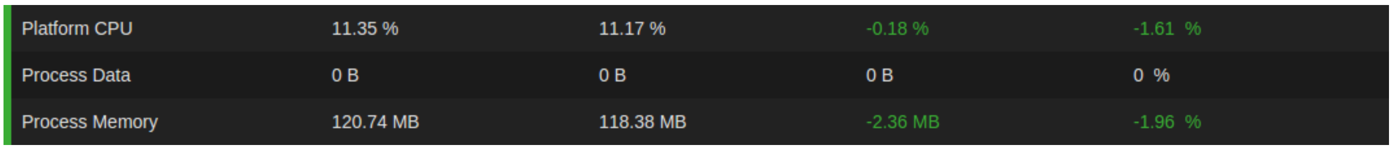
ShowHideStatusBar_Energy				
Platform CPU	16.71 %	16.86 %	0.14 %	0.86 %
Platform Discharge	59.36 mAh	59.32 mAh	-37.40 µAh	-0.06 %
Platform Discharge per second	301.96 µAh/s	291.21 µAh/s	-10.75 µAh/s	-3.56 %
Process CPU	0 %	0 %	0 %	0 %
Process Data	0 B	0 B	0 B	0 %
Network Packets	0	0	0	0 %
Process Memory	132.84 MB	131.41 MB	-1.43 MB	-1.08 %

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 expeting 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.

-10.75 µAh/s

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



Platform CPU	11.35 %	11.17 %	-0.18 %	-1.61 %
Process Data	0 B	0 B	0 B	0 %
Process Memory	120.74 MB	118.38 MB	-2.36 MB	-1.96 %

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.

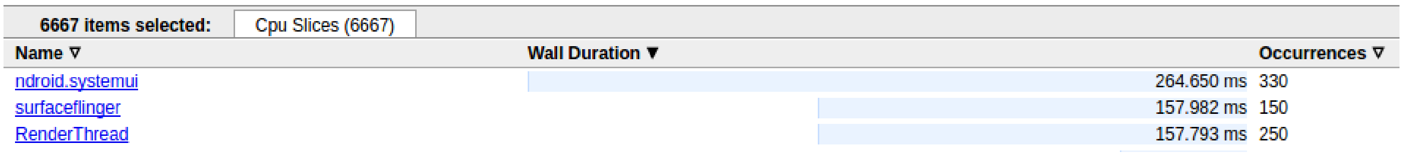
4.3. Action 3 – Animation and redraw optimization

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

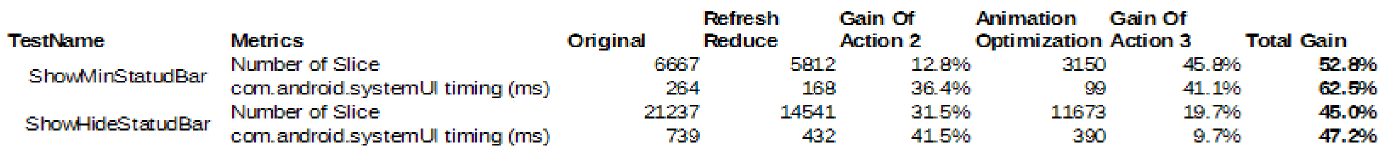
The Systrace analysis of the optimized version gives the following metrics for Showing the status bar:



6667 items selected: Cpu Slices (6667)

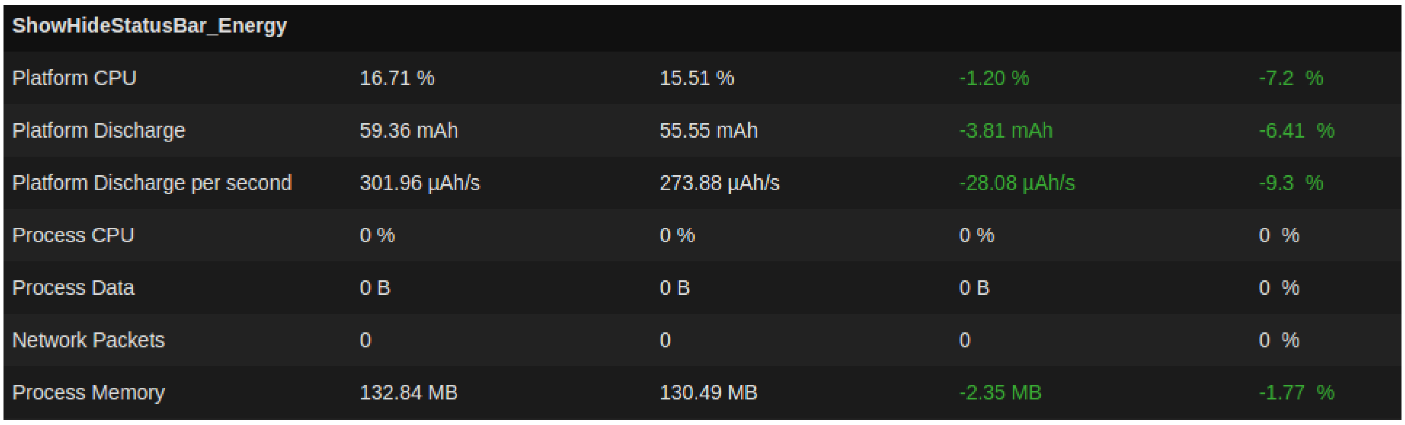
Name ∇	Wall Duration ▼	Occurrences ∇
ndroid.systemui	264.650 ms	330
surfaceflinger	157.982 ms	150
RenderThread	157.793 ms	250

The metrics for original version was:
The result :



TestName	Metrics	Original	Refresh Reduce	Gain Of Action 2	Animation Optimization	Gain Of Action 3	Total Gain
ShowMinStatudBar	Number of Slice	6667	5812	12.8%	3150	45.8%	**52.8%**
	com.android.systemUI timing (ms)	264	168	36.4%	99	41.1%	**62.5%**
ShowHideStatudBar	Number of Slice	21237	14541	31.5%	11673	19.7%	**45.0%**
	com.android.systemUI timing (ms)	739	432	41.5%	390	9.7%	**47.2%**

Illustration 22: Gain in term of method call



ShowHideStatusBar_Energy				
Platform CPU	16.71 %	15.51 %	-1.20 %	-7.2 %
Platform Discharge	59.36 mAh	55.55 mAh	-3.81 mAh	-6.41 %
Platform Discharge per second	301.96 µAh/s	273.88 µAh/s	-28.08 µAh/s	-9.3 %
Process CPU	0 %	0 %	0 %	0 %
Process Data	0 B	0 B	0 B	0 %
Network Packets	0	0	0	0 %
Process Memory	132.84 MB	130.49 MB	-2.35 MB	-1.77 %

Illustration 23: ShowHide Status bar improvement between original version (column 1) and optimized version (col 2), absolute difference (col 3) and relative difference (col 4) (screenshot from GREENSPECTOR interface)

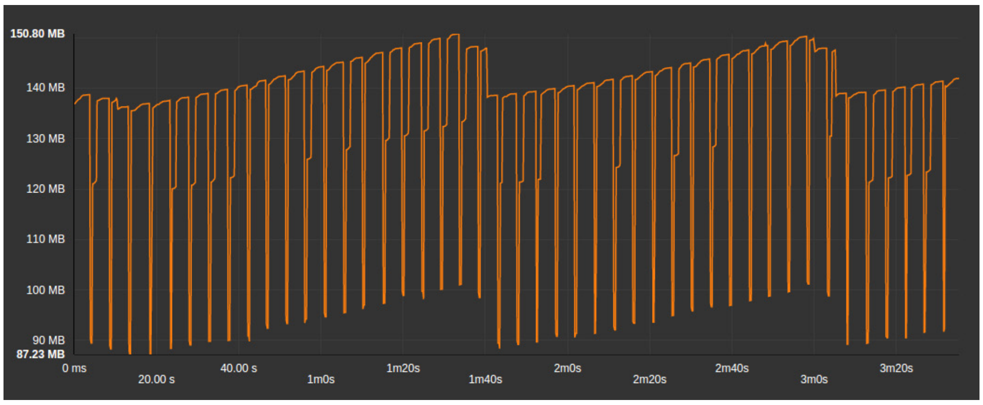
The global gain for the test ShowHideStatus bar is the following:
We got a gain of -28 µAh/s (- 9.3%) !

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.

We got a gain of **-28 µAh/s** (- 9.3%)

Reduction of the memory management pressure

The garbage collector runs every **2mn30**



llustration 24: RAM / initial version

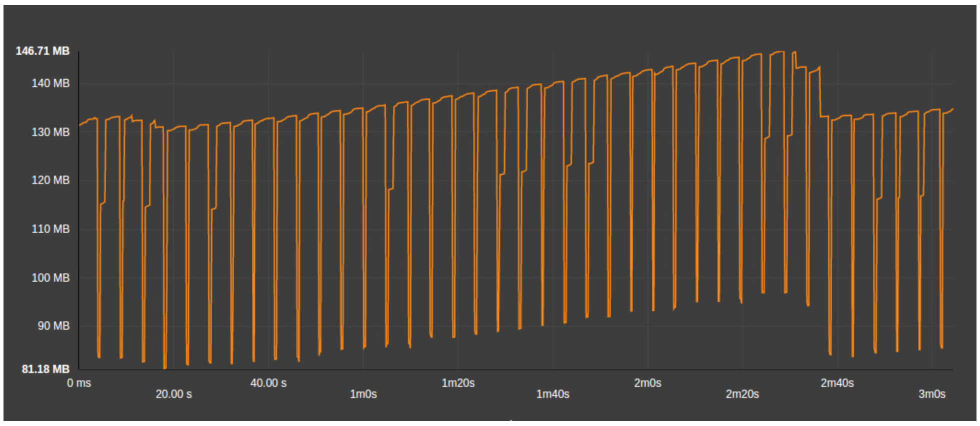


Illustration 25: RAM / Optimized version with the ShowHideStatusBar test

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!

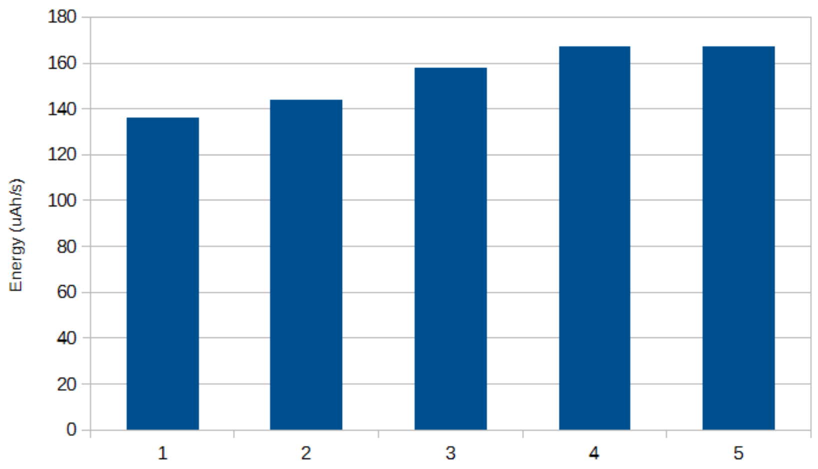


Illustration 26: Evolution of reference energy consumption for original version (showing a leak)

- 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 aa 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).

Hardware provides more and more power and resources to the software, hence the software use all the resources. The only limitations for the software are defined by two factors: the user’s perception of performance (aka speed), and t he 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 reflexion 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.

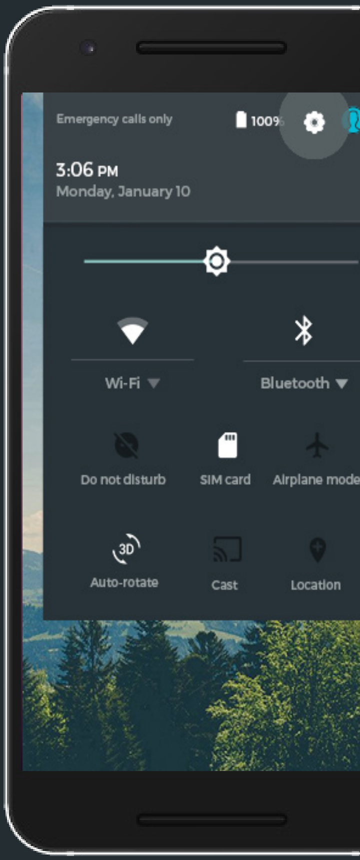
5.2. Measure, Measure, Measure

“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

GREENSPECTOR