Interrupt Skipping for Reducing Power Consumption of User Input Handling

Chetan Arvind Patil
Electrical Engineering & Computer Science Department
Northwestern University
Evanston, Illinois, USA
chetanpatil@u.northwestern.edu

Gokhan Memik
Electrical Engineering & Computer Science Department
Northwestern University
Evanston, Illinois, USA
g-memik@northwestern.edu

***Abstract* — Touch based smart phones and tablets have increased rapidly and have become primary mode of computing and communication. Users have to perform various types of touch inputs on these devices to interact with applications and manufacturers are always looking to improve the response time to satisfy users. However, performing touch gestures and making it more responsive has significant effect both on device battery and the microarchitecture. It is therefore important for computer architects and system designers to take into consideration the underlying effects of these touch gestures and user interactions at architectural level. Also, it's important to understand how efficient the touch architecture in today's smartphones are, and if there is any possibility to improve their efficiency.**

Therefore, there is a need to put forward ways to study these effects, and this technical report takes a look at touch interactions, their underlying architecture, and how our proposed implemented system helps improve efficiency at the system level. We call this new implementation, *Interrupt Skipping.*

Keywords— touch inputs; user interaction; android; smartphones

I. Introduction

Touch based devices, mainly smart phones and tablets, have seen rapid growth both in terms of use and technological development. Android OS [1] is leading all the way as preferred OS for mobile devices, with around billion activations and 50 billion of application installs [2]. In terms of hardware, ARM [3] is undoubtedly the leader in mobile devices and OEM's prefer to have ARM on the SoC's they built. Android is operated mainly using touch screen and this requires heavy user interactions with the help of touch inputs.

This presents an opportunity and need to study the effects of these interactions at system and hardware level. Same time, it is also very important to understand the how much more efficient these touch interactions can be. Even a few percentage of drop in battery current drawn from such task, which is mainly handle by drivers, can help improve efficiency of future smartphones. Thus, it's important to understand how touch interaction occur at the software-hardware level, and whether skipping these helps improve power consumption while keeping the user satisfied.

Touch devices have become an integral part of day to day life, and this has attracted research community's attention, and many groups have started looking into the details of how they can minimize the delay associated with every single touch. So far, most of them have concentrated on improving the response time, without much looking into the efficiency [4] [5]. Some of the study have also looked into the need of how fast the touch interaction should be [6]. Most recently, Alexander W. Min et al [7], came up with an adaptive touch sampling for mobile platforms. However, all these study lacks the real implementation, because they provide results based on bare devices, which don't run any operation system, applications or even use a SoC.

Currently, to the best of our knowledge, there hasn't been any study that looked into how touch processing can be handled in a very different manner, which in turn results in better efficient touch system. Thus, there is a need to put forward such implementation, and how that can make future smartphones less touch power hungry.

In this report, we show how interrupt which occurs during each touch interactions can be skipped without user perceiving any difference. Our implementation uses Android OS (AOSP), specifically we make use of CyanogenMod (CM) OS 11.0 [8] which is equivalent of Android KitKat 4.4. The reason to opt for a forked version of AOSP is due to the faster adoption both by hobbyist and OEM's. Also, since the underlying architecture is built on top of Linux Kernel, it gives us the same infrastructure to test with better build environment compared to AOSP. To validate our model, we make use of a widely popular Nexus 4 [9], and industry grade profiler Trepn [10].

In summary, this report put forwards:

- We show how current touch input systems in smartphones are power and CPU hungry.
- We developed, implemented, and tested a new concept called *Interrupt Skipping*, which helps in achieving better touch power efficiency.

The rest of this report is organized as follows. Section II describes the motivation to this research study on touch interactions. Section III shows how current underlying architecture works both at hardware and software level. Section IV describes the system developed. Section V talks about the experimental setups. Section VI shows the results we get from out implemented system. In section VII we share

related work, and conclude with conclusion and future work in section VIII and IX respectively.

II. Motivation

In this section, we describe the experiments we carried out in order to get power, CPU frequency, CPU load related touch interaction data. This in turn motivated us to look deeper into the touch architecture, and helped us in deciding how touches can be made more efficient.

A. *The Setup:*

The experiment setup for this uses swipe touches. To understand the power consumption between no touch interaction, heavy touch interaction, and low touch interaction, we simulate emulated swipe script for total of 80 seconds with Google Chrome Browser [11] loaded with a WiKi page. We make use of an open native utility called Orangutan [12], which allowed us to emulate near perfect touch events. We do away with using Android inbuilt utility *input* [13] to emulate same, due to the overhead it adds being written in Java. First 5 seconds, we don't emulate anything, then for next 20 seconds we emulate very fast 8 swipes per second touches. Again, we put the system to do away with emulating touch events for 5 seconds, with next 20 seconds emulating low 1 swipe per second before a 5 second delay, and then fast 8 swipes per second for 20 seconds. Finally, ending with 5 second of no interaction.

B. *Power Hungry:*

We carried out the explained setup to look into how change in frequency of swipe touch during a period of time on a Chrome Browser affects the battery current drawn.

Figure 1 shows the experimental set up for power analysis. To capture the battery current drawn, as noted above we make use of Trepn Profiler. Trepn being developed and distributed by Qualcomm makes for a good profiler due to the SoC in Nexus 4 is developed by Qualcomm only.

FIGURE 1: EMULATE SWIPES FOR POWER PROFILING

The data gathered during the analysis is shown in Figure 2. The results we got were very motivating, it clearly shows that with increase in touch interaction on an application in focus, results in tremendous amount of battery current being drawn. Since we make sure the screen brightness is lowest, all other wireless communication are turned off, and the data we got is near accurate. The 5 seconds of delay, where no touch interaction is happening, helps in showing the real difference.

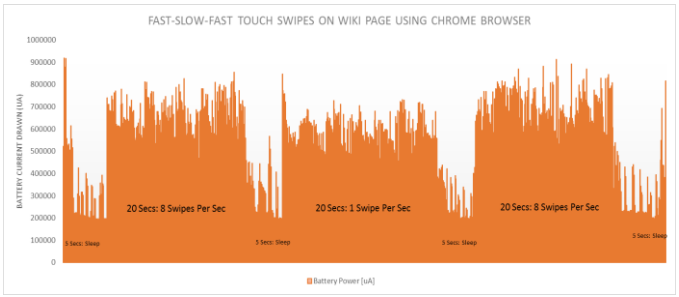

FIGURE 2: BATTERY CURRENT DRAWN FOR SWIPES

On an average, we see the difference of about 5% in battery current being drawn when there is no touch interaction, and when there is. We also note that between fast swipes of 8 swipes per second, and slow swipes of 1 swipe per second, there is around 2% of difference.

C. *CPU Hungry:*

To get more in depth understanding we also run the similar experiment to see how CPU frequency and CPU Load is affected with touch interactions. Nexus 4's SoC is a quad core one, and it honors core level voltage and frequency scaling. We wanted to see how different core react to our emulated touch swipes. For this experiment, we kept the CPU governor to on demand, this helps in giving the best performance and near accurate data. We again make use of same profiler we used for power analysis. Figure 3 illustrates the set up.

FIGURE 3: EMULATE SWIPES FOR CPU PROFILING

Figure 4 and 5 tells the similar story as told by Figure 2, i.e., touch interaction, specifically swipes affect CPU heavily, and thus the fact that touch interaction do have an effect at architectural level is proved here. Again, the 5 seconds no

touch interaction delay helps in distinguishing the fact that there is no other overhead.

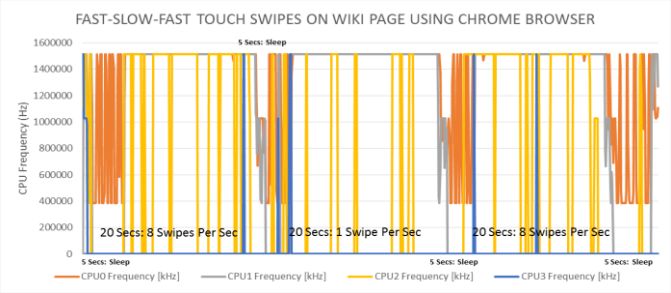

FIGURE 4: EMULATE SWIPES FOR CPU FREQUENCY PROFILING

On an average, the CPU frequency of core 0, 1 and 2 is between 1 GHz to 1.5 GHz, due to Qualcomm's in built core level governor, core 3 is almost never used. On the other hand, the CPU load is almost 60% during heavy touch interactions. With similar core level activity.

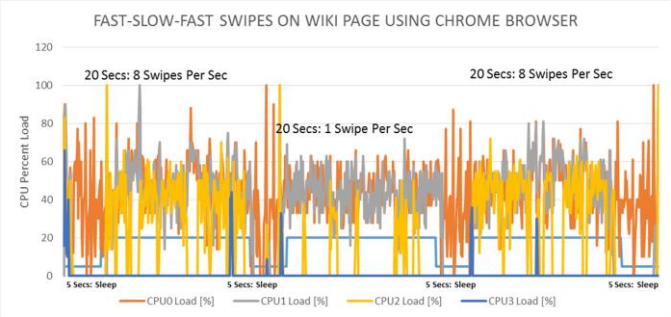

FIGURE 5: EMULATE SWIPES FOR CPU LOAD PROFILING

D. *What About Taps?*

It's very difficult to understand how the user is going interact with an Android smartphone. There are number of gestures which are available to user. The only way to distinguish the difference between these gestures is interaction time. If the touch is short, most likely the user is engaging for a tap events, else most likely drag or swipe. Above results motivated use to look into swipe touch events, but what about tap? Do we need to take them into consideration or not? To answer these questions, we carried out similar experiment exclusively for tap events.

Figure 6 shows the experiment setup, here we opt to emulate tap using the same native utility, and log data with same profiler. Here, the frequency of taps for first 20 seconds are 10 tap inputs per second, and for next 20 seconds, it's 1 tap input per second. Total duration of simulation was around 60 seconds. Figure 7 shows the power analysis for tap inputs, and we do see the effect of these inputs on the battery current drawn. However, if we compare that against swipe touch events, then tap have minimal impact. Also, the interrupts generated compared to swipe events are very less, and skipping those will annoy user, which we have explained in detail in later sections.

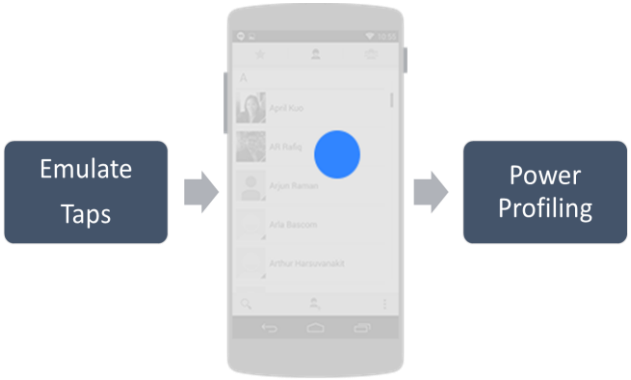

FIGURE 6: EMULATE TAP FOR CPU POWER PROFILING

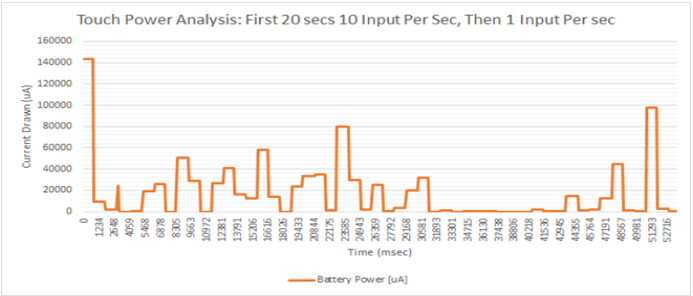

FIGURE 7: EMULATE TAP FOR CPU POWER PROFILING

To conclude, we observe that swipes are power, CPU hungry and considering that many of the leading applications on the Google Play like Facebook, Twitter, Chrome Browser, and other tools make heavy use of swipe interactions, we got a strong motivation to take this forward, and analyze the swipe touch system in depth, and see how they can be processed more efficiently.

III. Background

Touch devices have been in use since 1990s, and over the years it has found use in many more hardware devices. During this same era Linux Kernel started growing, and we saw more generic implementation of how touch screen work. All vendors has to do is to write a device driver specific to their hardware, and then hook it to Linux Input Subsystem [14], which is further processed by Android Input Subsystem [15] in case of Android OS or CM OS.

This section helps in understanding how the touch flow occurs in Nexus 4 which basically runs CM 11.0 OS built on top of Linux Kernel. Doing this analysis ultimately allowed us the ability to process the desired touch events, and also helped

in accessing the correct framework that resides within Android.

A. *Type of Inputs:*

Smartphones handle different types of input event and to understand the specific touch input driver we were interested in, we started looking into different types of input event that Nexus 4 can handle. Below snippet shows input types that Nexus 4 (codename: mako) supports, and it also means that the device driver will be processing all the raw data, which eventually is going to be thrown on to /dev/input/event2. Whether it is a tap, swipe or drag, all the events are going to be handled by this particular device.

```
shell@mako:/sdcard $ getevent
add device 1:  /dev/input/event0
name:     "pmic8xxx_pwrkey"
add device 2: /dev/input/event1
name:     "keypad_8064"
add device 3: /dev/input/event4
name:          "apq8064-tabla-snd-card
Headset Jack"
add device 4: /dev/input/event3
name:          "apq8064-tabla-snd-card
Button Jack"
add device 5: /dev/input/event5
name:     "hs_detect"
add device 6: /dev/input/event2
name:     "touch_dev"
```

The Input System comprises of two subsystems:

- Linux Input Subsystem
- Android Input Subsystem

B. *Linux Input Subsystem:*

Now that we have an understanding of the type of input events that can be handled in Nexus 4, let's take a look at how these are processed, and how many interrupts each of the swipe and tap produce. Linux input subsystem comprises of three main components:

- **Input Drivers:** This consists of the native code provided by the vendors whose hardware is being used. For touch screen this may come from Synaptics, Atmel etc. This helps in capturing raw data as soon as user touches the screen, and transfers them to the next section.

- **Input Core:** This is built in functionality provided by Linux Kernel, this specifically understands which type of touch driver is sending the data, and accordingly helps convert the raw data to more human readable form, and calls the correct handler responsible for handling such inputs.
- **Event Handlers:** After above decision of where to transfer the touch data, the final step is to understand which exact input the data should be given to. In last sub section, we saw that there are various input devices, to distinguish between different sub input devices, event handler is called, and it passed touch event to event2 in Nexus 4. Figure 8 illustrates the process.

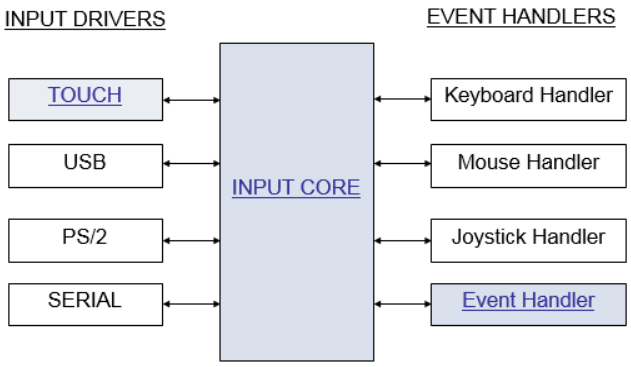

FIGURE 8: LINUX INPUT SUBSYSTEM

Figure 9 & 10 shows an interesting data that helped create the base for *Interrupt Skipping*. Figure 9 shows how many tap input events are captured and passed by the Linux Input Subsystem. The important thing here was to understand if we can do away with any of the inputs for the tap. But we will explain in later section why we didn't implement interrupt skipping for taps.

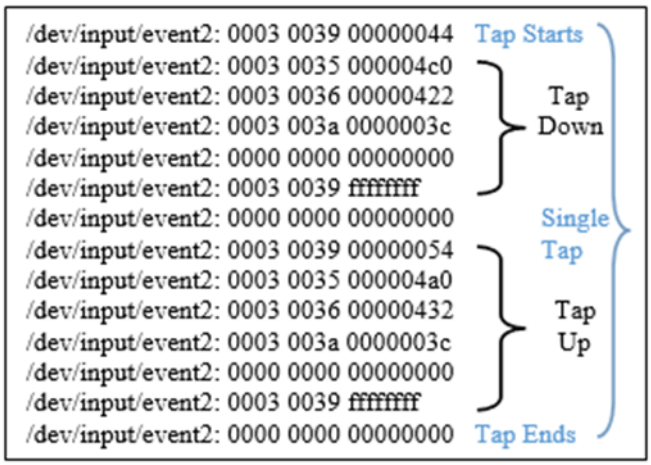

FIGURE 9: EVENTS WHEN TAP OCCURS

We did the same analysis for swipe events, and found that the number of touch events generated during this process are far more than that of taps. This pushed us into thinking if we can exploit this feature. Swipe event consists of *sync report* that is generated for each step, if an user is doing swipe from coordinate (x1,y1) to (x2,y2), and the distance between these is 100, then this 100 will be divide into equal steps. In nutshell, the swipe is nothing but a continuous tap occurring so fast that the user can't perceive this.

Figure 10 shows how many swipe input events are generated and passed on to the next subsystem i.e. Android Input Subsystem, which captures these inputs and process them in order to complete the full action on the application side.

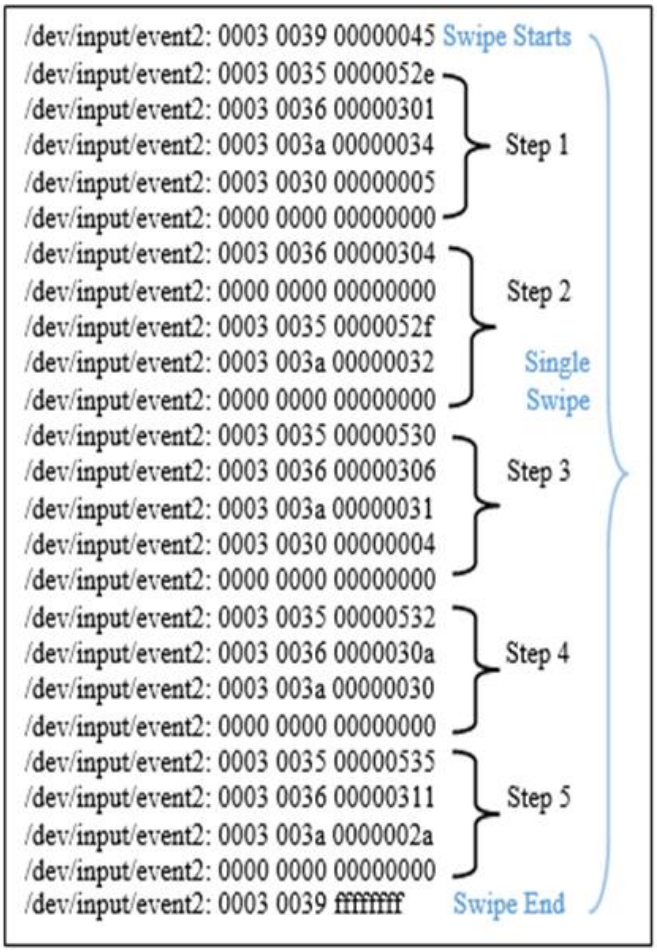

FIGURE 10: EVENTS WHEN SWIPE OCCURS

Next subsection describes how the Android Input Subsystem works, it's written in Java.

C. *Android Input Subsystem:*

After the input has been processed by the Linux Input Subsystem, Android Input Subsystem is called. This system is always pooling for events at any of the input device: /dev/input/event*, and as soon as it captures the data there, it is send for further processing. The Android system is designed such that it is able to understand the difference gestures for touch event based upon timing and other details.

Figure 11, shows the system described in detail by Andrew S. Hughes [16]. It shows how the native events are captured, and there on processed as InputReader thread and InputDispatcher thread. After fully processing based on various criteria's like timing, sync report, device ID etc, the final processed event is dispatched to the frame buffer, where the action is final taken on the user side, the application. Thus completing the full cycle of touch event.

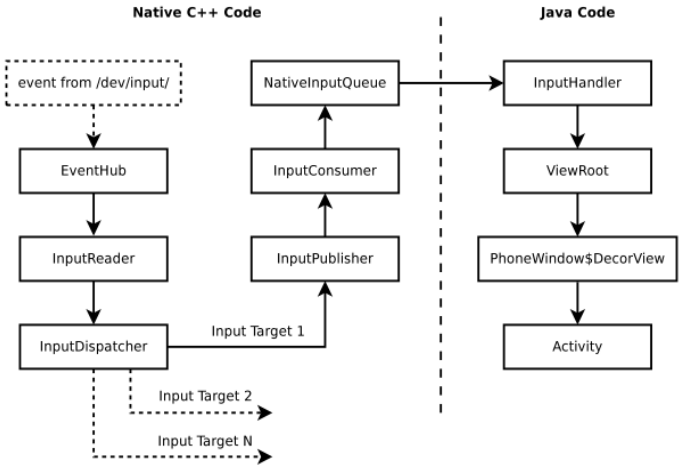

FIGURE 11: ANDROID INPUT SUBSYSTEM. IMAGE COURTESY [16]

D. *Interrupt Skipping In Input Subsystem:*

As established in the last two subsections, the input system produces lots of events particularly in swipe touches. If we take a closer look at these input events for swipe, then we see the possibility of skipping few of them. Doing so will not allow these to be processed fully on the Android Input Subsystem side. We can't opt for interrupt skipping on Linux Input Subsystem, as we do need some form of input data to come to a conclusion that whether some of these can be skipped or not.

Figure 10 presents the number of steps a single swipe takes to complete the full swipe, what is we started to skip 50% of these? What if we skip 20%? There is a strong need to understand these, and that is where the concept we are introducing of *interrupt skipping* needs to be tested. For this we developed algorithmic approach on the Android Input Subsystem, and thoroughly tested it in order to see how the system reacts, and whether there is possibility for saving power. We describer all this in next section.

IV. System Design

Carrying from the previous section, here we describe the system we have designed, and how that has been incorporated in the current operating system, and the modified internal Android Input Subsystem.

A. *The Current Model:*

Figure 12 shows how the current model handle the inputs, specifically the touch inputs. All the inputs as received are processed first by the Linux Input Subsystem, and there on

passed on to the Android Input Subsystem. All these goes on to affect the power consumption.

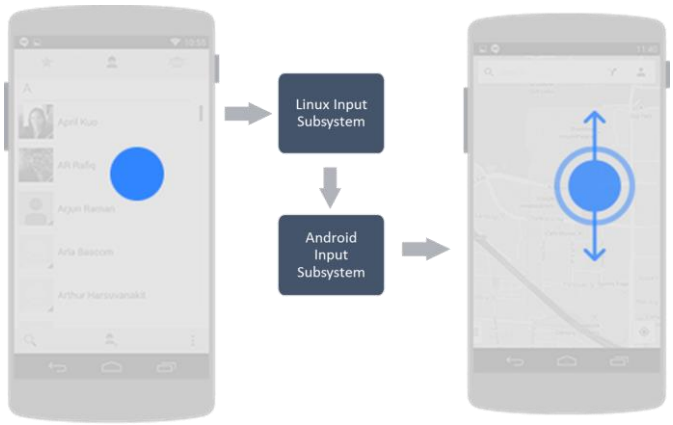

FIGURE 12: CURRENT MODEL OF TOUCH PROCESSING

This model doesn't take into consideration the user. Each and every user is different, and is able to adopt the system as per the needs. After looking into the inner working of the Android Input Subsystem, we decided to modify it as per the user. The important argument here is that the holistic approach required to process the system should also be considered. The next subsection describes how the new model works.

B. The Interrupt Skipping Model:

As stated in the last few sections, we have seen that swipes are event hungry, and they in turn become power and CPU hungry. To make them more efficient we need to model the current subsystem as per the user needs. As of now the implemented version is not adaptive, but it surely understands the activities that are being done by the users. Let's consider various scenarios:

1. **Opening of an application:** When a user unlocks the devices and starts looking through the application he she wants to open up, he expects that the icon clicked should respond as fast as possible. Our interrupt skipping model, takes this into consideration, and allows all the tap events to process without any hassle.

2. **Using the application:** After the user has opened the application, there are many things he she might be able to do depending upon the application. For emails, first thing would be to scroll through. Usually, user will scroll slowly, and since these scroll events are swipes lots of events will be generated, this is where our model kicks in, and tries to understand whether the scroll is too slow or fast. For fast, to keep user satisfied, the model will let go all the things, but for slow swipes, since the user expects page to move slowly, it will start dropping events. If the user changes back to editing mode while in the email application, the model is still active, but won't skip any events as while editing any test, user expects things to move fast, and there on keeping user satisfaction high.

3. **Switching to another application:** When the user switches to other application, model may or may not kick in depending upon the action. If the user is scrolling slowly looking for another app, he she may feel the slight change in response, but as per our rigorous test these are negligible.

4. **Closing the application:** The user will stop the application, and there on close the screen. While he is doing this, only for the first activity the model will jump in, for the second one, even though it's an input event, our model ignores it, as this is not something we skip, else the user will be annoyed beyond extent.

Figure 13 shows our interrupt skipping models, and the working described above is represented there too. This model is capable of working against any application because of the implementation at the system side. Also, if we look at the application like Facebook, Google+, Twitter, LinkedIn, or event any gaming application, swipes are used more that 50% of the time to engage with these applications. Considering this, our model adds a significant argument of using it across such apps.

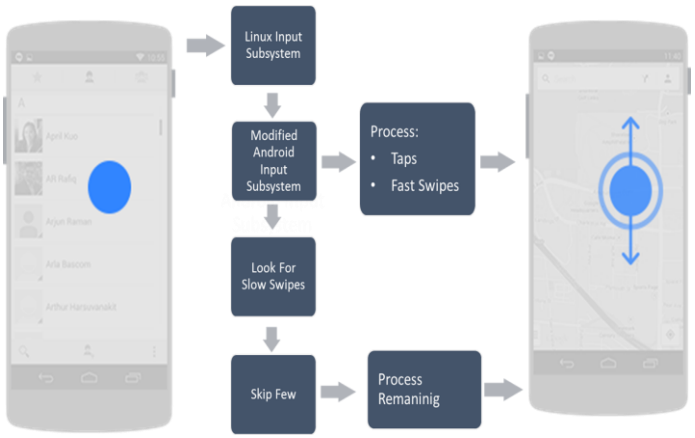

FIGURE 13: INTERRUPT SKIPPING MODEL OF TOUCH PROCESSING

C. Incorporating Interrupt Skipping Model:

The system designed can be easily applied to any Android smartphone. Since, this process is happening at the system level, it doesn't matter which platform the Android OS is running on. Our model can be easily attached to the current Android Input Subsystem, and can be tested across devices. We are also able to decide the percentage of swipe events we wish to skip, this gives the model more flexibility to adapt and decide. We have carried extensive analysis for this, and we describe the findings of our power analysis in next few sections.

V. Experimental Setup

In this section, we describe the experimental setup used to test *interrupt skipping*. This experimental setup was used to gather power performance with and without i*nterrupt skipping*. At the time of writing this technical report, we were in process to reach out to users in order to carry out user study.

For this experiment to collect and present power results, we concentrated only on single heavily used application, the Google Chrome Browser. This was decided because of the heavy use of browsers on smartphones, and also that the Chrome Browser is very reliable when it comes to parallel processing, and makes good uses of multithreading. We simulated swipes events with the frequency of 1 swipe per second to make sure that our model is used when we are testing it. We will be able to test it fully with any frequency of touch during the user study. The power profiling setup consists:

- LG Nexus 4:
    a. CyanogenMod 11.0 a.k.a Android Kitkat 4.4
    b. Kernel 3.4.0
    c. CPU: 1.512 GHz Quad Core Krait
    d. SoC: Qualcomm Snapdragon S4 Pro APQ8064
- Trepn Profiler to gather power data.
- Google Chrome browser application.
- Orangutan to emulate swipes.

Since interrupt skipping is capable of skipping any percentage of swipe events, we collected the power data for 40%, 50%, 60%, 70%, and 80% drop of swipe events.

VI. Results

In this section, we describe and compare the results for current input model stacked against the implemented *interrupt skipping*. As said in last section, we vary the number of swipe events that are being dropped, and then run the power analysis to understand the effect on the battery current. Figure 14 shows how the current model works when only 40% of the swipe events are skipped under *interrupt skipping* model. There is significant drop in battery current drawn. We also take into consideration different brightness level, as these are important technical aspects which are related to touchscreen.

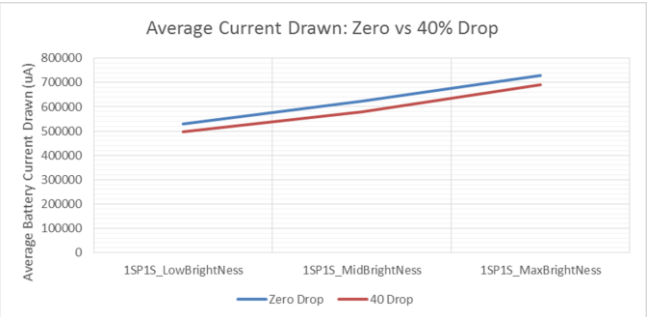

FIGURE 14: CURRENT MODEL vs 40% INTERRUPT SKIPPING MODEL

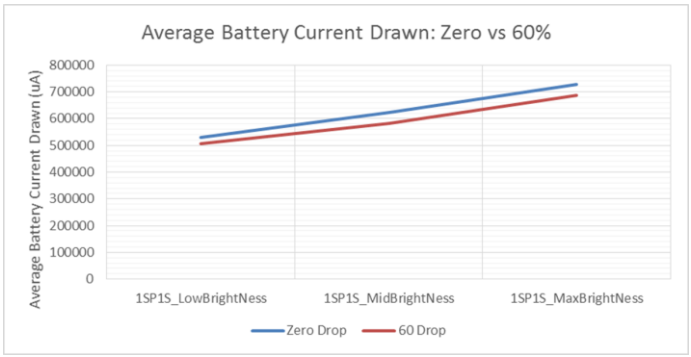

FIGURE 15: CURRENT MODEL vs 60% INTERRUPT SKIPPING MODEL

Figure 15 shows the same analysis which has been carried out with only 60% of swipes being dropped. We see consistent power savings, when the interrupt skipping is in place. We again observe similar activity for 70% of the swipe events dropped as shown in figure 16.

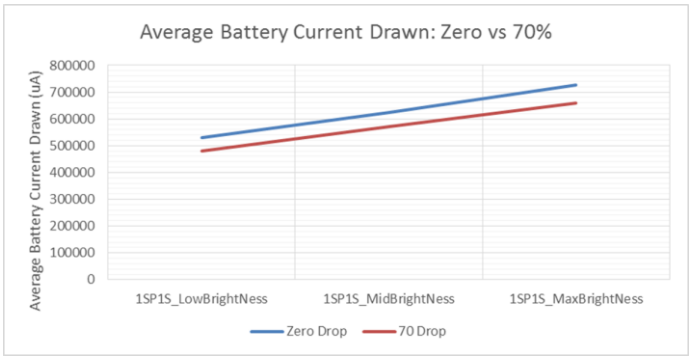

FIGURE 16: CURRENT MODEL vs 70% INTERRUPT SKIPPING MODEL

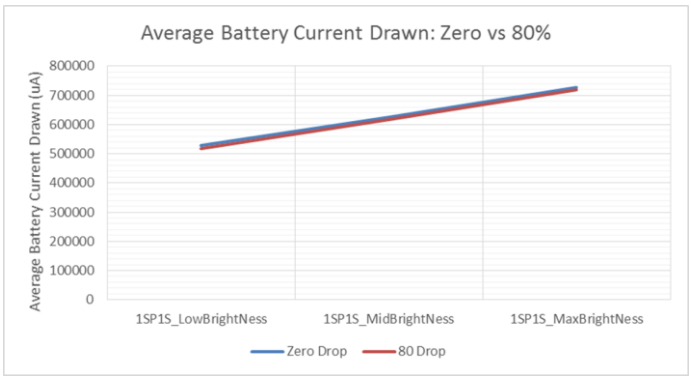

FIGURE 17: CURRENT MODEL vs 80% INTERRUPT SKIPPING MODEL

We do have one result where the interrupt skipping didn't had any effect. Figure 17 shows that for 80% skipping, the model is missing out on skipping many swipe events, and that lead to less impact on the power savings. We come to the conclusion that the best case scenario to skip interrupts is with 50%, where we see satisfactory power savings of around 5% as shown in Figure 18.

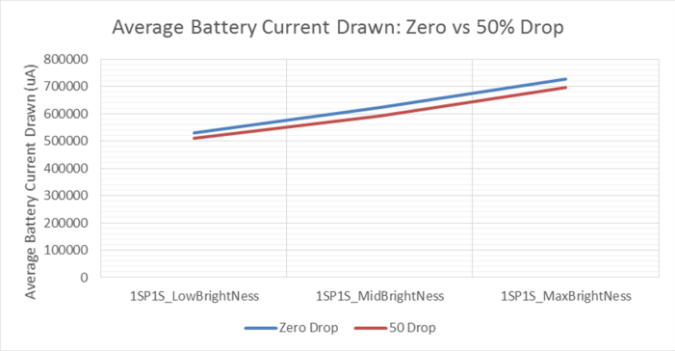

FIGURE 18: CURRENT MODEL vs 50% INTERRUPT SKIPPING MODEL

In figure 19, we compare all the varying interrupt skipping against the original model, and see impact of the new proposed model.

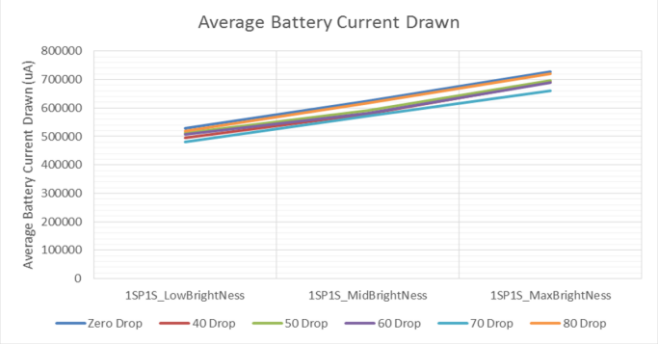

FIGURE 19: CURRENT MODEL vs VARYING INTERRUPT SKIPPING MODEL

VII. Related Work

Since, Android OS started taking over the smartphone domain, there has been surge in research activities related to power efficiency for Android smartphones. Research has been carried out on full analysis of power consumption in a smartphone [17]. Such study show that here is need to make smartphones more efficient, and less power hungry. The best of best smartphones don't last more than a day, and with less improvement in battery technology, there is need to put efficient system architecture. Our approach of implementing novel touch interrupt skipping is the first one to the best of our knowledge. The nearest research on touch efficiency [7] doesn't even take into consideration the full system and don't event provide future approach. Few research has targeted the CPU governor to make use of user perceived response time [18], but these framework are targeted towards power management, rather than efficient touch management.

All the power related research on smartphones haven't taken the approach we took i.e. the efficient touch processing, and that has been our focus in this research.

VIII. Conclusion

After running power analysis for varying touch interrupt skipping we come to the conclusion that there is significant need to improve the current touch processing system. Our approach shows a path of how skipping the interrupts will lead to better power efficiency without impacting the working of the smartphones. We are able to shown an improvement of around 5%, and that can be significantly improved further if tested in a real environment.

Future smartphones not only have to be smart in terms of processing, they need to be smart in conserving power too. Upcoming smartphones are going to evolve around Linux Kernel, and Android OS, which also raises the question to go through working of legacy framework, and understand how that can help improve the efficiency of a smartphone.

Interrupt skipping for reducing power consumption can also be extended to other internal frameworks in Android where lots of interrupts are generated, and it would be interesting to understand whether all of these need to be processed, and how much impact they have at architectural level.

IX. FUTURE WORK

This study has established the fact that touch events can be skipped to gain better power consumption. Our main target now is to understand, and prove that fast is not always better. To do this, we need to go to the user, and understand what they feel about the implemented model, and how it can be improved further.

It would be interesting to look into more details about similar simulation being carried out on an architectural simulator like gem5, and whether there is scope of improving the framework further depending upon the data gathered for the architecture simulated. Also, same study can be extended to other Android OS version and also to different Instruction Set Architectures.

This will allow computer architects and system designers to study the effects of touch inputs across different operating systems and future mobile processor architectures.

X. Acknowledgment

We would like to thank Matthew Schuchhardt for his continuous guidance on Android system. Prof. Peter Dinda for important advice on system level architecture. Emphatic Systems Project and Intel group for their valuable inputs.

Android phone images used in figure 1, 3, 6, 12 and 13 are provided by Google Inc.'s Android Open Source Project under Creative Commons Attribution 2.5.

XI. References

[1] Android Operating System: http://www.android.com/
[2] Google I/O by the numbers: 900 million Android activations: http://www.zdnet.com/google-io-by-the-numbers-900-million-android-activations-7000015432/

[3] ARM: http://www.arm.com/

[4] A. Ng, J. Lepinski, and D. Wigdor, “Designing for low-latency direct-touch input,” Proc. 25th …, pp. 453–464, 2012.

[5] H. Xia, R. Jota, B. Mccanny, Z. Yu, C. Forlines, K. Singh, and D. Wigdor, “Zero - Latency Tapping : Using Hover Information to Predict Touch Locations and Eliminate Touchdown Latency,” pp. 205–214, 2014.

[6] R. Jota, A. Ng, P. Dietz, and D. Wigdor, “How fast is fast enough?,” Proc. SIGCHI Conf. Hum. Factors Comput. Syst. - CHI ’13, no. 1, p. 2291, 2013.

[7] A. W. Min, K. Han, D. Hong, and Y. Park, “Adaptive Touch Sampling for Energy-Efficient Mobile Platforms.”

[8] Cyanogen Mod: http://www.cyanogenmod.org/

[9] Nexus 4: http://www.google.com/intl/ALL/nexus/4/

[10] Trepn: https://developer.qualcomm.com/mobile-development/increase-app-performance/trepn-profiler

[11] Chrome Browser: http://www.google.com/chrome/

[12] Orangutan: https://github.com/wlach/orangutan

[13] Input: http://developer.android.com/reference/android/view/KeyEvent.html

[14] Linux Input Subsystem: https://www.kernel.org/doc/Documentation/input/input-programming.txt

[15] Android Input Subsystem: https://source.android.com/devices/input/overview.html

[16] A. Hughes, “Active Pen Input and the Android Input Framework,” Thesis, no. June, 2011.

[17] A. Carroll and G. Heiser, “An analysis of power consumption in a smartphone,” Proc. 2010 USENIX Conf. USENIX Annu. Tech. Conf., pp. 21–21, 2010.

[18] W. Song, N. Sung, B.-G. Chun, and J. Kim, “Reducing energy consumption of smartphones using user-perceived response time analysis,” Proc. 15th Work. Mob. Comput. Syst. Appl. - HotMobile ’14, pp. 1–6, 2014.