User Impatience

Nick Dellorco, Keith Stebler

Introduction

Smart phones have become increasingly more and more utilized for various tasks. However, with increased usage, there is the result of increased stress on battery life. It is important to reduce this stress, but not at the cost of insufficient performance. However, how does one determine this balance, to improve performance when it is desired and to reduce performance when it is unnecessary? It could be at the behest of the user to determine this balance. However, how would the user know when performance is unneeded? If the device itself must make this distinction, how would it know when the user desires improved performance? The user and device should not be in conflict to determine balance, instead the device should learn from the desires of the user to cater balance to the expectations of the user.

Solution

The User Impatience application has a goal to cater the performance of an Android smartphone to the preferences of the user. The application acts as a CPU governor that attempts to minimize power consumption by learning where specific users become impatient with the performance. To accomplish this there must be interaction between the user and the phone by complaining when it becomes too slow. Meanwhile, the application must learn the users tendencies in order to minimize the frequency with

which they complain, while also improving the battery performance. To accomplish this the User Impatience application must record the CPU frequencies for each user complaint, specific to the app they are using. Then using that data we must effectively implement a predictive model to understand where the user is likely to complain next to decrease the resource allocation to an area where the user is more likely to complain. However, to accomplish some of our goals for the semester we first had to identify some of the flaws with the original state of the application.

Challenges

Approaching the project we understood that there would be a learning curve as only one of us had previous experience with any Android development and it was very little experience. Furthermore, the languages utilized within the project were Python, Kotlin, and C++. Only one of us knew Python, only one of us knew C++, and no one knew Kotlin. So there was some challenge in just understanding the project's code. Additionally, we had to learn about governors.

Background

A governor in Android controls how the CPU raises and lowers its frequency in response to the demands the user is placing on their device. Governors are especially important in smartphones because they have a large impact on the fluidity of the interface and the battery life of the device over a charge. Next, there were three major components of the app that we had to understand before doing any work. These were

the governor control, the scripts and the user impatience system. In the case of the governor control, you input into the command line the configuration that you want to set the device's governor to. The software then sends this data to the device as a series of ADB commands. The scripts send ADB commands, such as tap and swipe, to simulate a user utilizing an application on the device. Currently, there are five applications being used; Youtube, Gmail, Chrome, Spotify, and Photos. The last component, the impatience evaluation system, is while the scripts are running, the user is being evaluated for the potential for impatience for each task within a script. If the user is found to be impatient, the script is paused, a series of commands are sent to the phone to indicate impatience, and then the script is resumed.

Virtual User Evaluation System

The first major flaw that we identified and wanted to improve was the virtual user evaluation system. In its initial state the virtual user system was strictly probability based. Each user impatience level, denoted by a 1, 2 or 3, had specific values. If a randomly generated number fell below the user's "impatience threshold" they would complain. The largest problem that we quickly found was that there was equal probability of a complaint on the first command in the app as on the lowest possible frequency. This was not indicative of an actual user. For example, if we decide to test with a "patient" user we wanted to ensure that complaints would only occur at a reasonable level. In order to accomplish this we decided to base our complaints on the

user threshold with relation to the time each task took. So a user value, designated as a ‘time to wait’ value would be compared to the time that a task would take.

However, a direct comparison between task time and user time has the problem of ignoring what is reasonably possible within the testing system. Therefore, the first part of implementing a new virtual user system was to add a set up step. To do this, we would run the scripts with the governor in performance mode, which maximizes the CPU frequency. Then, the results for task completion time are logged in a text file. We, then, use these values as reference for the optimal time for each task. To utilize this functionality, run the C++ code with the setup parameter. It is important to note that this only needs to be run one time, unless the optimal task time or the number of tasks within a test changes.

With the optimal time, we are able to determine impatience results from our user impatience governor. If the resulting value of the time of a task minus the optimal time is lower than the time that a user is willing to wait, then the user would be ‘happy’ with the performance. However, if the resulting value is higher than the user’s value, then the user would be impatient with the performance and as such, would complain.

While it is important for test results to have an identifiable reason for the results, a problem that was brought up was a lack of variability in those results. A patient user can, at times, be impatient. An impatient user can, at times, be patient. So to apply variability, the probability system was added back in. However, the influence of probability was greatly diminished. The original system utilized percentage values of ten, thirty, and forty for a user to demonstrate impatience. Additionally, the new system

is not restricted to a limited set of possible users. Since the user is defined by a time and that time is directly inputted as the impatience value, there are as many users as there are possible times that can be inputted. Furthermore, in the old system, a higher user value meant a higher chance for impatience. In the new system, a higher user value means a lower chance for impatience since that user is willing to wait for a longer period of time. So with these things understood, the new probability system utilizes an algorithm. So that a higher user value correlates to a lower chance of complaining, the inverse of that user is taken. The value is then multiplied by one hundred to have decimal place accuracy of one and the probability range is out of one thousand. So the probability equation is 100/'user value'. For example, a user value of one would have a ten percent chance to demonstrate impatience, as well as, demonstrating impatience if a task takes one second longer than expected.

Testing the Virtual User System

The testing process of the virtual user evaluation system is synonymous with running the system. The first process that was tested was evaluating task time. This was done with the old probability system in place and running the test system, evaluating for each task's time. Success was indicated by the output of valid time values.

The second process that was tested was the setup process. The C++ code was run with the setup parameter. Successful completion of the process was indicated that

at the end of the program there would be additional text files within the program's directory.

The next process that was tested was the updated user system. The first part of this system was to evaluate that the user would complain appropriately within the test system. This was indicated by output comparison. At first, this system was only tested with the time system. It was not until later that it would be tested with the addition of the probability system.

Test Scripts

Another area that was improved upon was the testing scripts. A problem that was discovered with the testing system was that three applications, out of the five, were insufficient in testing user impatience. These applications were Photos, Spotify, and Youtube. The problem with these three applications is that during the time that the application was playing a fifty second video or song, the user was not being checked for impatience. The testing system would simply go into sleep for the expected amount of time that the song or video would play and then from there the only thing left for the script was to quit out of the application. This resulted in the governor having the opportunity to decrease the CPU frequency multiple times, depending on the time interval set, but only one opportunity for the user to complain. In the case of Youtube and Spotify, additional tasks were added during the playing of the song or video, to give the user additional opportunities to complain. In the case of Youtube, a user will scroll

down to look at the comments. It is important to note that there was a problem that came up with this. If an advertisement popped up in the beginning of the video, there was a possibility that the advertisement would load a different Youtube page while the advertisement was playing. The comment section for the expected video would be unavailable until after the advertisement was over and the expected video loaded. So, a period of sleep time is still necessary, before the user is able to do any tasks. In multiple tests, the maximum amount of time for an advertisement was thirty seconds. So the test waits thirty seconds before attempting to scroll down to the comments. In the case of Spotify, the user looks at the song's album. There were no problems that came up with this method. However, in the case of the Photos application, while a video is playing there was no discernible way to add tasks while the video was playing. So, adding additional tasks, would add additional time to the Photos application test.

Testing the Scripts

The updated scripts were tested by running each script in performance mode. It was unnecessary to run it in userspace mode since functionality only depended on successful completion of a script. It is worth noting that while Spotify gave little problems, Youtube was problematic. The process of testing Youtube required independent running of the application to evaluate the extent of advertisement interference. Then, the script had to be evaluated multiple times to ensure that it successfully completed with the variability in advertisements.

Problems with CPU Frequency Adjustments

Another major issue that we identified early in the project was the static values for changing CPU frequency. This dramatically hinders performance because there is too much time spent at higher CPU frequencies where we don't expect a user to complain. Therefore, to improve performance we needed to decrease frequency more rapidly to minimize wasted resources. In order to do this we needed to create a predictive model to output the next predicted complaint level in order to minimize the current gradual descent. Another aspect we are now able to improve is how frequently we will decrease the CPU frequency, which is currently set to a static time amount (in seconds). This is another important aspect to optimize because the further away from the expected complaint we are, the less time we want to use before decreasing next. Then as we approach an area where the user is more likely to complain, we can raise this value to give the user more time at those levels. In order to further understand how these values should be set we had to first create the prediction model.

We have a model that will predict the user's next complaint based on prior data to understand how we can minimize both average CPU frequency and the number of times a user complains. The biggest reason that this helps us improve system performance is because previously the CPU frequency would drop by a fixed rate in fixed time intervals. However, if we know around when the next complaint will happen, we can drop the frequency much quicker until we reach a range close to the expected complaint level. This minimizes unnecessary CPU usage at frequencies where the user is unlikely to complain based on their prior trends. Our biggest challenges with this

aspect of the system have been testing and incorporating it. Currently the prediction model runs in the python code that initializes the whole test. However, we are missing the component of the testing environment that allows the python code to run so we are unable to test this for now. However, we are working to incorporate it into the phone itself as well as having it run on the computer. The idea with this is that there are steps to developing a prediction model; gather data, use that data to train the model, and then use the trained model. For the second step, the computer and by extension the python code is necessary to help keep the strain of both the testing environment and training system at a minimum. For the third step, the trained prediction model will then need to function without the help of the computer. For the first step, logging is necessary to gather information.

Furthermore, by logging past complaints we can improve two other aspects of the overall application. Similarly to the previously static decrease CPU frequency option, the amount to increase and the time in between decreases were also both static values. The importance of logging our user complaint for the increasing frequency is that we have a better idea of the range in which a user may complain. Therefore, we can reduce the amount we increase on each complaint, which will in turn improve the efficiency of the application as a whole. For example, we can increase to just above the highest previous complaint for a user so they would either create a new higher complaint range or more likely will be within a range with which they were previously satisfied.

Logging

As such CPU frequency logging was added to the test system. The goal of this logging was to determine at which frequency a user complains at to acquire information for the prediction model. Initially, frequency logging was added onto the desktop side. This effort was fairly simple as part of the functionality had already been utilized during the process of testing. It is a command line argument that utilizes ADB to output the CPU frequency of the first core on the device. The governor adjusts all of the cores together, so only the first core needs to be checked. To make the process automatic, so that the testing system has logging, the command is passed as a parameter via a call to the popen function in the C++ code. However, there were two problems that came up after implementing this functionality. The first problem is that the prediction model required the values of the CPU frequency position, not the frequency itself. The frequency position is a range of values from zero to some number depending on the CPU. This range is associated with the range of frequencies that the CPU can run at. In the case of the Nexus 6, the range is from zero to seventeen, and is associated with the frequency range of three hundred thousand hertz to about two billion six hundred million hertz. The other problem is that the ultimate goal of the prediction model, or more specifically, the governor, is to function without the need to be connected to a desktop system. The purpose of being connected is to test and improve the functionality of the governor, but eventually the governor will need to function without the training wheels. So, because of these concerns, logging was added to the device as well. The value of

the frequency position is already utilized within the governor application. Whenever a user complains, the value is collected within a vector for the application that the user complained within. When the governor is deactivated, the values in any non-empty vectors are written to text files, named according to the tested application. These files are then stored within the governor application's data directory on the device.

Prediction Model

The prediction model itself is a two layer neural network that predicts the value that the next complaint will occur by using the mean and standard deviation of the past twenty-five complaints as inputs. This allows the model to be flexible with changes in user behavior although we may have to test with other amounts of prior data to see what represents the user best. Using the standard deviation as an input allows us to make predictions using the uncertainty of the user. If the user complains at the exact same point every time, then the model will predict that point.

Testing the Model

The initial stages of testing and analyzing this neural network was to create various sets of data for training and testing the model. The datasets were generated using a random normal distribution, which I thought would be appropriate to approximate a user's behavior (most data within 1 standard deviation of the mean and

almost all observations within 3). From there I started evaluating what configurations would work best for the given sets. The initial model had one input, the running average. I tested it with various versions of the dataset including different standard deviations for the random normal generation and with anywhere from twenty to thousands of observations. This is where a trade-off became evident and may still need to be re-evaluated in further testing. This trade-off was how many observations to account for when making a prediction. When training with hundreds or thousands of observations, the predictions all became far too close to the mean because as there were more observations the average converged towards the mean. If we used too few observations, one outlier point would have too much of an impact. Therefore, the current setting is at 25 observations but this may need to be variable in the future to accomodate for users that are more variable or unexpected in their complaints. To alleviate some of my concerns I decided to add a second input, the standard deviation, which would account for some of the variability in complaints among users. By doing this there is an additional parameter considered by the model which focuses on the unpredictability of users, which is important because users that tend to complain within a small range are far easier to predict than those with high uncertainty.

Conclusion

Moving the prediction model into the phone will allow for us to test without using the python code and hopefully with real users. Then we would be to test with the full

environment and collect enough data to further optimize the whole system and hopefully this will lead to improved performance compared to other governor options that are implemented. The next major step for the project will be to move the entire system onto the phone so it can be tested by actual users. This process would mean moving everything into the kernel, which we won't be able to do this semester, but that will likely be the next. Once the app is functional in the phone we hope to prove that our concept is an improvement over the existing governors and then test on real users.