iRAM: Sensing Memory Needs of My Smartphone

David T. Nguyen$^{\times}$, Hongyang Zhao$^{\times}$, Gang Zhou$^{\times}$, Ge Peng$^{\times}$, Guoliang Xing†
$^{\times}$College of William and Mary, USA;
†Michigan State University, USA
dunguk@gmail.com; {*hyzhao, gzhou, gpeng*}*@cs.wm.edu*; *glxing@cse.msu.edu*

***Abstract*—Our study reveals that facilitating warm launch of just five smartphone applications is extremely expensive, using up to 36 percent of memory. Further investigation of 20 popular applications indicates that rich multimedia applications have high heap usage and go above allowed boundaries, up to 5.63 times more heap than guaranteed by the system, and may cause crashes and erroneous behaviors. Therefore, we present iRAM, a personalized system that maintains optimal heap size limits to avoid crashes, efficiently maximizes free memory levels, and cleans low-priority processes to reduce application delays. The evaluation on memory hungry applications indicates that iRAM reduces application crashes by up to 14 percent, and reduces launch delays by up to 78.2 percent. In addition, the results confirm that iRAM increases free memory levels by up to 4.8 times. This performance gain comes with 3.5 percent of CPU overhead and 0.9 percent of power overhead.**

***Keywords*-Smartphone Memory; Application Crash; Application Launch**

I. Introduction

In a recent study [6], when consumers were asked if they had encountered a problem (app crashes, freezes, errors, or extremely slow launch) accessing a mobile app within the last six months, 56 percent said yes. Among those who have experienced a problem, 62 percent reported a crash, freeze or error; 47 percent experienced slow launch; and 40 percent reported an app that would not launch.

We think that many such performance issues are due to little available memory and its inefficient utilization. Our study reveals that after having launched five regular applications (Facebook, Instagram, G+, Angry Birds, and YouTube), the amount of free memory left is just around 4 percent. The low free memory level stays throughout the experiment, and does not recover to significantly larger values. This negatively affects the smartphone's overall application performance, and ultimately results in slow response times and crashes. In addition, many applications do not respect heap thresholds, and go far above their allowed heap usage, which also contributes to more erroneous behavior. Users quickly notice apps that are slow or likely to break (whether because of downtime, crashes, etc.), and this impairs both usage and brand perception. Users expect a mobile app to be fast and responsive; if it is not, it will get poor reviews, low ratings and low adoption numbers. While 79 percent of consumers would retry a mobile app only once or twice if it failed to work the first time, only 16 percent would give it more than two attempts [6]. Poor mobile app experience is likely to discourage users from using an app again.

Our paper addresses two key research questions towards achieving better application performance. (1) *How does memory affect smartphone application performance?* (2) *How can we improve smartphone performance through learning user application priorities and application memory behaviors?* In order to address the first research question, we study memory usage of several groups of applications. In particular, we identify the amount of memory available before and after their launch. Little available memory may result in delayed I/O operations or frequent communication with much slower flash disks, which essentially causes slow application response. Next, we investigate heap usage of applications. High heap usage of games and other rich multimedia apps may increase crash rates and likelihood of erroneous behaviors. To address the second research question, we design and implement a system prototype called iRAM on the Android platform. iRAM efficiently maximizes free memory levels, cleans low-priority processes, and maintains optimal heap size limits. The system learns which apps are of high priority for a particular user, and keeps them in the main memory. The launch of such apps is then much faster, since it corresponds to warm launch. iRAM also applies a prediction model to predict heap usage of a set of apps, and dynamically adjusts the heap size based on predicted values. With this set of simple optimizations, iRAM reduces application delays and decreases likelihood of erroneous behaviors.

In summary, the contributions of our paper are as follows:

- First, through a measurement study we find that facilitating warm launch of just five applications is extremely expensive, using up to 36 percent of memory. The resulting little memory left can be one of the main reasons causing slow application response due to delayed I/O operations or frequent communication with much slower flash disks. Therefore, in order to improve the application performance, we investigate how each application consumes the memory. Our heap usage study of 20 popular applications indicates that rich multimedia applications have high heap usage and go above allowed boundaries. This mainly applies to games that require up to 5.63 times more heap than guaranteed by the system, and may cause crashes and erroneous behaviors. Finally, further investigation reveals that limited heap may not only cause an app to crash, but may even prevent an app from launching. While on Samsung S4, all five games fail to launch until the heap size of 64MB, on Nexus 4, all five games fail to launch until the heap size of 128MB. Therefore, the heap size directly affects success or failure of application launch.

- Second, we design and implement iRAM, a system that maintains optimal heap size limits to avoid crashes, efficiently maximizes free memory levels, and cleans low-priority processes to reduce application delays.
- Third, our evaluation on memory hungry applications indicates that iRAM reduces application crashes by up to 14 percent. In addition, the results confirm that iRAM increases free memory levels by up to 4.8 times. The evaluation using 40 popular applications from four groups (games, streaming, miscellaneous, and sensing) also shows that iRAM reduces launch delays by up to 78.2 percent. This performance gain comes with 3.5 percent of CPU overhead and 0.9 percent of power overhead.

II. Measurement Study

In order to understand how memory affects smartphone application performance, we conduct a measurement study. First, we study memory usage of several regular applications. In particular, we measure the amount of memory available before and after their launch. Little available memory can be one of the main reasons causing slow application response due to delayed I/O operations or frequent communication with much slower flash disks. Next, we investigate heap usage of applications. High heap usage of games and other rich multimedia apps may increase crash rates and likelihood of erroneous behaviors. In our preliminary measurements, we utilize two devices: Samsung S4 (2GB RAM, 128MB heap size) with Android 4.3, and Nexus 4 (1GB RAM, 64MB heap size) with Android 4.2. Our Performance Evaluation section later on also studies more recent devices. The phones are normally used daily by the authors. During experiments, the devices have all radio communication disabled except for WiFi that is necessary to provide stable Internet connection required on most apps. The screen is set to stay-awake mode with constant brightness, and the screen auto-rotation is disabled. The cache is cleared before each measurement.

A. *Free Memory*

In this experiment, we study memory usage of five popular applications: Facebook, YouTube, CNN, Angry Birds, and Temple Run. In particular, we launch the five apps every ten minutes, and issue 100 user events via Android Monkey [11], which corresponds to using an app for a few seconds. Then we close each app with the Home button keyevent, assuring that an app does not get killed and stays in the background. We record free memory levels within one hour. To facilitate the measurement, we implement a shell script with the *free* [5] command to output free memory levels of the devices. All codes used in the paper are available for download on our GitHub page[1].

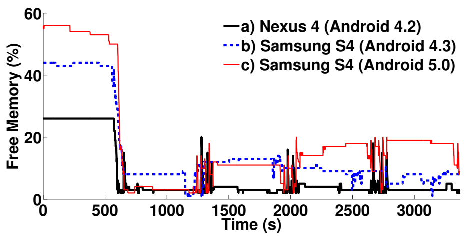


Figure 1. **Free Memory.**

The free memory levels during this experiment are illustrated in Figure 1. The results in a) and b) indicate that from the beginning, both devices have only less than half of memory available, since the Android OS already consumes a large portion. This is an expected behavior, and it may be interesting for Android OS developers to find ways to optimize memory consumption of the OS itself. When the apps are launched the first time, the memory level drops significantly. This is also expected, since the first launch corresponds to cold launch. Specifically, device a) has afterwards only three percent of memory left, while b) has eight percent left. When the apps are relaunched after 10 minutes, the levels drop again but not dramatically, which is because the apps are already in the background (warm launch). However, facilitating this warm launch is extremely expensive, using up 36 percent of memory on device b) and 23 percent on a). Notice that shortly after memory levels drop during launch, we observe slight memory growth and spikes due to Android's memory cleaning, often referred to as garbage collection. However, the Android's cleaning approach is not very efficient, and without the knowledge of user application priorities, it fails to recover more memory. This is observed both on older Android releases with Dalvik garbage collection and more recent ones with improved ART garbage collection [9] (Android 5.0 and newer). For comparison, we display memory levels of a Samsung S4 device with Android 5.0 in c). The results indicate that the newly introduced ART garbage collection is able to recover more memory than its predecessor. However, we still think that this cleaning can be more personalized and aggressive, and that we can remove more low-priority processes. For that we will need to learn which processes are of the top priority to a user. Notice also that device a) has mostly lower memory values throughout the measurement due to its smaller total RAM capacity. The above results reveal that simple usage of only five regular applications can already use up a significant amount of memory. To understand how each application consumes the memory, we study heap usage

[1]https://github.com/davidnguyenwm/iRAM

of several groups of popular applications in the following subsection.

B. Heap Usage

This experiment investigates heap usage of applications. To obtain a deeper understanding, we look into the heap behavior of 20 applications from four diverse groups: games (Angry Birds, Grand Theft Auto, Need for Speed, Temple Run, The Simpsons), streaming (CNN, Nightly News, ABC News, YouTube, Pandora), miscellaneous (Facebook, Twitter, Gmail, Maps, AccuWeather), and sensing apps (Accelerometer Monitor, Gyroscope Log, Proximity Sensor, Compass, Barometer Monitor). Each application is launched and run for a minute and gets issued a set of predefined user events via Android Monkey. Every time an app is running, there are no other apps in the background. To facilitate the measurement, we implement a shell script with the *dumpsys meminfo* [4] command to output the heap usage of the apps.

The results of the experiment are displayed in Figure 2. There are several key observations. Due to rich multimedia content, games in Figure 2 (a) use up much more heap than the heap size allowed by the OS. For instance, the Simpsons game reaches a maximum of 360MB, while the heap allowed is only 128MB on Samsung S4 (2.81 times difference), and 64MB on Nexus 4 (5.63 times difference), respectively. To make the figure less complex, we only display the larger heap size. The Android design allows apps to grow heap usage above the default heap size threshold, but such oversized usage does not have any performance guarantee, and apps may be killed unexpectedly by the system during run-time. This happens during the experiment with Need for Speed, and the app crashes unexpectedly at the fourth second, causing a sudden drop in heap usage. The app is subsequently relaunched at the sixth second to continue the experiment. Through a memory analysis tool named littleeye [10], we are able to see that right before the crash, there is an OutOfMemory error event. The streaming apps in Figure 2 (b) overall use up less heap than games, but still go above the threshold, especially CNN and ABC News with their rich user interface. The miscellaneous apps in Figure 2 (c) indicate a varying heap usage. Facebook has high heap usage due to its aggressive caching policy and advanced multimedia support (five posts ahead, auto video play, images pre-loading, etc.). Maps go over the heap threshold when a route across the whole U.S. is loaded. While Gmail, mainly text oriented, is always far below the threshold. Finally, the sensing apps with their simple user interface use little heap, and are the least memory hungry. The main takeaways of this subsection are following. Rich multimedia applications may have high heap usage and go above allowed boundaries. This mainly applies to games that require up to 5.63 times more heap than guaranteed by the system, and may cause crashes and erroneous behaviors.

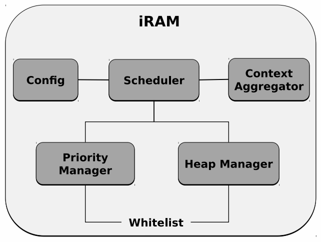


Figure 3. System Architecture.

III. System Architecture Overview

In order to improve application performance in smartphones through memory optimizations, we need to address the above challenges. Of the top priority is the finding that rich multimedia applications have high heap usage and are likely to crash. Next, users cannot tolerate slow application response caused by little available memory. Therefore, we present iRAM, a personalized system that maintains optimal heap size limits to avoid crashes, efficiently maximizes free memory levels, and cleans low-priority processes to reduce application delays. The architecture of iRAM is illustrated in Figure 3. It is fully located in the kernel space, and consists of several key modules: the Heap Manager, the Priority Manager, the Scheduler, the Config module, and the Context Aggregator. We elaborate each module and its functionality below. iRAM's complete source code is available for download on our mentioned GitHub page.

Heap Manager. Our system prototype follows the implications from the previous experimental study. Since high heap usage of games and other rich multimedia applications may cause crashes and erroneous behaviors, the Heap Manager dynamically configures global heap thresholds such that they are always higher than application heap requirements. To achieve this, we employ an autoregression model with exogenous inputs (ARX) to predict future heap usage, and then dynamically update the global heap size to avoid crashes. Specifically, we combine past data from the same run and historical data from previous runs together to predict future heap usage with the use of the ARX model. The prediction of future heap usage using past heap usage forms the autoregressive portion of the model, while the historical heap usage data serves as exogenous inputs. Finally, based on the predicted heap usage, we dynamically adjust the global heap size threshold to avoid crashes.

Priority Manager. The Priority Manager module maximizes free memory levels, and cleans low-priority processes to reduce application delays. In particular, the Priority Manager finds candidate processes to be killed based on Android's importance hierarchy, from the lowest to the highest importance. Next, if there are any high priority user processes among the candidate processes, they are filtered out. Finally, the processes left in the candidate list are killed. Motivated by [28], we obtain high priority user processes through a

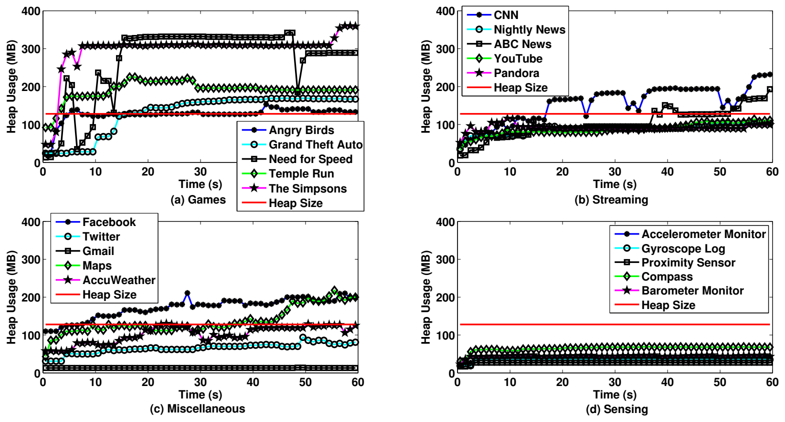


Figure 2. Heap Usage.

simple prediction method. Assuming the next application to be used by the user has the highest user priority, we apply prediction by partial matching (PPM). Therefore, predicting next app based on previous usage pattern corresponds to predicting next letter based on probability distribution occurrence of previous letters. iRAM predicts 9 next applications and places them in the Whitelist. If a candidate process to be killed is also listed in the Whitelist, such process is filtered out and is not killed.

Scheduler. The Scheduler triggers Priority Manager and memory cleaning if the free memory level is below a threshold defined in the Config module. The Scheduler checks this memory level each time period, which is also configurable in the Config module. By default, we set the period to be 20 seconds when the device screen is on, and 60 seconds when the screen is off. The rationale behind this is the assumption that when it is off, there are not many user activities, and hence no frequent cleaning is required. While when the screen is on, the user is actively using the device, and likely many memory operations are going on.

Config Module. Config includes several global parameters. *AGGRESSION_LEVEL* defines how aggressively the system should proceed during memory cleaning. There are three levels, roughly 1 includes all background processes, 2 includes background processes and system caches, 3 includes background processes and foreground processes and system caches. Details are discussed in the Priority Manager Design section. Finally, *MIN_MEM* is a memory threshold below which the system should proceed with cleaning, and its default value is set to be 60 percent, implying that a relatively high free memory level is required.

Context Aggregator. Context Aggregator collects information about the user and device. Such information is, for example, whether the device is being used or in the sleep mode, how long the device is being used or how long the screen is off. The context information is utilized by the

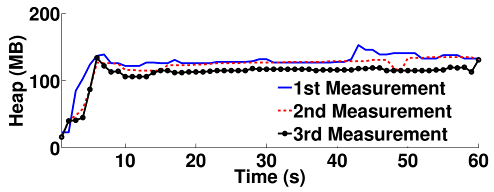


Figure 4. Angry Birds' Heap Usage.

Scheduler module to manage memory cleaning process.

IV. Heap Manager Design

In this section, we elaborate the Heap Manager's system design previously introduced in System Architecture Overview.

A. Heap Usage Prediction with ARX

To predict future heap usage, a simple approach is applying past data, which refers to the heap usage data obtained from the moment an app is launched till the current time stamp. The accuracy of such method is based on high autocorrelation between the predicted heap usage and past heap usage. If predicted and past heap usage data are highly correlated, then this method can achieve satisfiable prediction result. Figure 4 displays three measurements of Angry Birds' application launch. The figure indicates that after 10 seconds, heap usage remains relatively stable. During this stable heap period, if we use past stable heap usage data to predict future usage, the prediction accuracy will be high. However, heap usage increases rapidly at the beginning of the application launch. Therefore, if we only predict heap usage based on past data obtained during launch, the predicted value will be lower than the real heap usage. Hence, setting the global heap size threshold based on the predicted value may lead to an application crash, since the real heap usage will exceed the threshold. Another method to predict heap usage is based on historical data, which refers to heap usage traces collected in the past. Figure 4 indicates that the launch curves from different measurements are similar at the beginning (approx. first 10 seconds). The correlation coefficients between the first and second, second and third, third and first measurements in the first 10 seconds are 0.907, 0.892, and 0.987, respectively. However, once the app runs stably, heap usage from different measurements shows about 20MB difference. Therefore, after the app runs stably, prediction only based on historical data will not be accurate.

Based on the above observations, we combine past data from the same run and historical data from previous runs together to predict future heap usage with the use of the ARX model. The prediction of future heap usage using past heap usage forms the autoregressive portion of the model, while the historical heap usage data serves as exogenous inputs:

$$y\left(t\right)=\sum_{i=1}^{p}a_i y\left(t-i\right)+\sum_{j=0}^{q-1}b_j u\left(t-j\right)+e\left(t\right), \qquad (1)$$

where t indexes time, $y\left(t\right)$ denotes the heap usage at time t, $u\left(t\right)$ represents historical heap usage at time t, a_i and b_j are coefficients, and $e\left(t\right)$ is a sequence of independent random variables. The objective of the model is to provide timely prediction of future heap usage. In order to do this, we have to estimate the coefficients, a_i, b_j. In addition, the model orders p and q are also unknown and have to be estimated. Parameters a_i and a_j can be estimated using the least-squares method. For orders p and q, in our experiment, we vary p from 0 to 3 and q from 0 to 4 in Equation 1 to obtain the optimal values of p and q for prediction. Intuitively, this answers the question of how much of past and historical heap usage data should be used to predict heap usage in the future. Within the ranges examined, $p=0$ or $q=0$ represent models where there is no past data or historical data. Also, if $p=0$ and $q=1$, we have a linear regression between current heap usage and historical heap usage. If $q=0$, we have standard autoregression model (AR).

B. ARX Parameters

Based on the ARX model in Equation 1, we adopt K-fold cross-validation approach to compute the optimal combination of p and q. In a typical K-fold cross-validation scheme, one dataset is equally divided into K subsets. At each step of the scheme, one subset is selected as a test set, while other subsets function as a training set in order to estimate the model parameters. In the experiment, we collect eleven one-minute-long heap usage traces for each app as the dataset in K-fold cross-validation scheme. Heap usage prediction in the ARX model depends both on past heap usage and historical heap usage. Therefore, we randomly select one trace as a historical trace, and combine it with any other trace into one sample. This way the dataset is divided into K subsets, where each subset contains two traces, one current running and one historical trace.

	q=0	q=1	q=2	q=3	q=4
p =0	null	11.07	9.49	8.50	8.45
p =1	7.68	5.97	6.16	6.29	5.98
p =2	7.63	**5.95**	6.15	6.38	6.07
p =3	7.63	6.17	6.26	6.33	6.14

Table I
ANGRY BIRDS' RMSE.

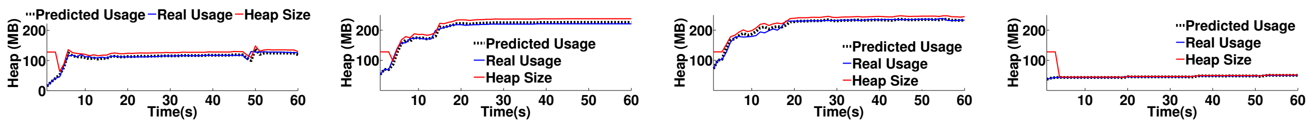
By studying the launch curves of all applications, we find a common property that it takes three seconds for apps to use up the default heap size (128MB). This means the default heap size helps avoiding crashes during the first three seconds, but does not help avoiding crashes afterwards. Therefore, we can apply the default heap size for the first three seconds, and use the ARX model to predict heap usage for the time period after those three seconds. Notice that $y\left(t\right)$ is computed based on past data, $y\left(t-1\right),\cdots,y\left(t-p\right)$, and historical data, $u\left(t\right),\cdots,u\left(t-q+1\right)$. Therefore, the first data value that can be predicted is $y\left(\max\left(p+1,q\right)\right)$. As we use the ARX model after three seconds, the first prediction time $\max\left(p+1,q\right)$ should be less than or equal to four seconds. Therefore, the data range for p is $0,1,2,3$, and q is $0,1,2,3,4$.

Considering the above constraints, our K-fold validation testing procedure is as follows. For each app and for each (p,q) pair from $p=0,1,2,3$ and $q=0,1,2,3,4$, repeat the following steps:

1) Divide the dataset into K subsets $\{S_1,S_2,\ldots,S_K\}$.
2) For each $S_k, k=1,\ldots,K$, compute the parameters a_i and b_j using all the other subsets with the least squares methods. Based on the estimated model parameters and associated prediction model in Equation 1, predict the heap usage value of each member of S_k.
3) Compare the predicted heap usage result with the real heap usage data using the root mean-squared error (RMSE):

$$\varepsilon=\sqrt{\frac{1}{T}\sum_{t}\left(y(t)_{predicted}-y(t)_{real}\right)^2} \qquad (2)$$

We choose $K=10$. Using the above 10-fold cross-validation, we compute the root mean-squared error for all the apps under different (p,q) pairs. The (p,q) pair that gives the smallest RMSE is selected as the optimal model orders. Table I shows the RMSE for Angry Birds. The table indicates that the ARX model achieves the smallest RMSE under $p=2$ and $q=1$. This means that to predict heap usage for Angry Birds at time t, two heap usage data values at time $(t-1)$ and $(t-2)$, and one historical heap usage data value at time t are the most effective data points in prediction. We also observe that when using past heap usage data alone $(q=0)$ or using historical heap usage data alone $(p=0)$, the RMSE values are relatively high (larger than 7.6MB and 8.4MB in corresponding cases). When past and historical data are combined together $(p\neq 0, q\neq 0)$, we get lower RMSE values. This indicates that both past heap usage data and historical heap usage data are useful in prediction.



(a) Angry Birds (b) CNN (c) Facebook (d) Accelerometer Monitor

Figure 5. Heap Usage Prediction.

C. Global Heap Threshold

Based on the predicted heap usage, we update the global heap size threshold as follows:

$$y(t)_{global} = y(t)_{predicted} + \alpha \cdot \varepsilon, \quad (3)$$

where $y(t)_{global}$ is the global heap size we need to set, $y(t)_{predicted}$ is the predicted heap usage, α is a constant, and ε is root mean-squared error. We vary α from 0 to 5 to compute the global heap threshold, and compute the percentage of time that the threshold is larger than real heap usage under different α. When α is set to 0, 1, 2, 3, 4, we get the following numbers: 45.47%, 81.14%, 96.05%, 98.39%. The larger the α is, the larger the heap threshold, and the lower probability the app crashes. However, if the heap threshold is set to be a very large value, the system may need to distribute a large amount of memory to an app. In our system, we select α as 2. In this case, the real heap usage does not exceed the global heap threshold for 96% of time. To dynamically adjust the heap size, we utilize Android command *setprop dalvik.vm.heapsize*.

We plot heap usage prediction for Angry Birds in Figure 5(a) ($p = 2$ and $q = 1$). The figure confirms that the Predicted Usage curve and Real Usage curve are in proximity with the maximum error of 29.20MB and RMSE of 5.33MB. The global heap threshold (Heap Size) is set equal to the default threshold provided by the device manufacturer (128MB) for the first three seconds. Then the threshold is updated by Equation 3. The figure indicates that real usage data only exceeds the heap size at the time stamp of the 49th second by 18.55MB, which demonstrates the effectiveness of the Heap Manager module. We also display the results for a sample streaming app (CNN) in Figure 5(b), a miscellaneous app (Facebook) in Figure 5(c), and a sensing app (Accelerometer Monitor) in Figure 5(d), with RMSE values of 5.47MB, 5.34MB, and 0.48MB, respectively. This confirms that our heap prediction methodology allows efficient heap utilization by setting the heap size reasonably close to its real usage.

V. Priority Manager Design

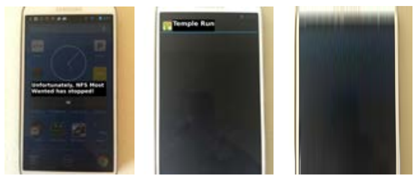
In this section, we elaborate the Priority Manager's system design. The Priority Manager efficiently maximizes free memory levels, and cleans low-priority processes to reduce application delays. Processes with the lowest importance are obtained based on Android's importance hierarchy [8].

Based on the *AGGRESSION_LEVEL*, iRAM selects processes to kill from the lowest to highest importance. If it equals 1, iRAM populates the candidate list with Empty and Background processes. If it equals 2, iRAM populates the candidate list with Empty processes, Background processes, and system caches. If it equals 3, iRAM populates the candidate list with Empty, Background, Service, Perceptible, Visible, Foreground processes, and system caches.

Next, if there are any high priority user processes among the candidate processes, they are filtered out. We define high priority user processes as applications that have the highest probability of being used next by the user. We obtain high priority user processes through app prediction by partial matching, a variant of an existing text compression method called PPM [15]. PPM generates a probability distribution for the prediction of the next character in a sequence. Consider the alphabet of lower case English characters and the input sequence "abracadabra". Assume that each character corresponds to an application used by the user. For each character in this string, PPM needs to create a probability distribution representing how likely the character is to occur. However, the only information it has to work with is the record of previous characters in the sequence. For the first character in the sequence, there is no prior information about what character is likely to occur, so assigning a uniform distribution is the optimal strategy. For the second character in the sequence, 'a' can be assigned a slightly higher probability because it has been observed once in the input history.

Consider the task of predicting the next character after the sequence "abracadabra". One way to go about this prediction is to find the longest match in the input history which matches the most recent input. The most recent input is the character furthest to the right and the oldest input is the character furthest to the left. In this case, the longest match is "abra" which occurs in the first and eighth positions. The string "dabra" is a longer context from the most recent input, but it does not match any other position in the input history. Based on the longest match, a good prediction for the next character in the sequence is simply the character immediately after the match in the input history. In this case, after the string "abra" was the character 'c' in the fifth position. Hence, 'c' is a good prediction for the next character.

Therefore, predicting next app based on previous usage pattern corresponds to predicting next letter based on probability distribution occurrence of previous letters. Since we want to predict at each moment nine next applications to



(a) NFS (b) TR (c) TS

Figure 7. **Crash Behaviors.** (a) Need for Speed app displaying error message “Unfortunately, NFS Most Wanted has stopped!”; (b) Temple Run app displaying its logo and a black screen; (c) The Simpsons app displaying a black screen

be used by the user, we want to know nine next characters instead of one next character. The choice of nine apps will be explained in the evaluation. Our code is also available on our mentioned GitHub account. As proven by the previous work [28], this prediction approach outperforms all previous solutions that also incorporate user context such as location and time.

VI. Performance Evaluation

This section evaluates iRAM, and answers the following questions. *(1) How does iRAM contribute to reducing erroneous application behaviors?* We address this by performing a crash rate test on memory hungry applications. *(2) How does iRAM improve application performance?* This is addressed by evaluating how free memory levels are improved by iRAM, and how well high priority applications are predicted. We also record the launch delay of 40 popular apps from Google Play with and without iRAM. In addition, we conduct an experiment on the Facebook application to determine the user-perceived performance improvement of our solution. *(3) Does iRAM incur any performance penalties or cost?* This is determined by evaluating CPU and power overhead.

The evaluation utilizes the same devices as in the measurement study: Samsung S4 (2GB RAM) with Android 4.3 and Nexus 4 (1GB RAM) with Android 4.2. When necessary, we discuss results of both devices, otherwise Samsung S4 is the main phone. As mentioned, the phones are normally used daily by the authors. During experiments, the devices have identical setup as in the Measurement Study.

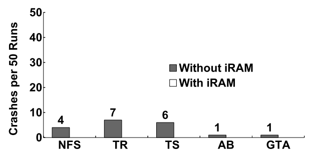
A. Crash Rate

First, we investigate how iRAM contributes to reducing erroneous behaviors of the five games (Need for Speed, Temple Run, The Simpsons, Angry Birds, and Grand Theft Auto). Specifically, we launch each game every ten minutes, and record crash statistics, with and without iRAM. Each time we launch a game, there are four common apps in the background (Facebook, YouTube, Accelerometer Monitor, and Heads Up), while the game being launched is not currently in the background. This is to create typical memory pressure on the device. Having four apps in the background is reasonable, since current Android OS allows users to have up to 17 background apps. Each game is launched 50 times throughout 500 minutes.

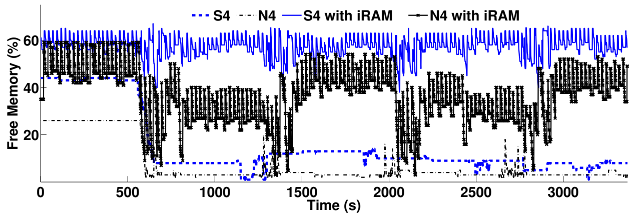
The experiment has several interesting findings. First, the crashes of the applications have varying behaviors, as illustrated in Figure 7. Need for Speed displays an error message indicating the app has stopped. While Temple Run crashes by showing its logo and a black screen, The Simpsons game displays a completely black screen. Finally, Angry Birds and Grand Theft Auto freeze on their launch screen. For the sake of space, we display the first three cases. All these crash behaviors are unacceptable, and impair both usage and brand perception. Users expect a mobile app to be fast and responsive, but most of all, users expect an app to work. Next, Figure 6 (a) illustrates the amount of crashes over 50 runs for each application. Without iRAM, Temple Run has seven crashes over 50 runs, corresponding to 14 percent crash rate. Compared to other games, Temple Run has the highest heap usage during the first two seconds of its launch, which explains this result. The Simpsons game has six crashes, and Need for Speed four crashes. These two games have the steepest growth of heap usage, and hence the high crash rates are not surprising. Finally, Angry Birds and Grand Theft Auto have the lowest crash rate. Both games have mild heap usage at the beginning, and their maximal values are much smaller than those of others. Note that with iRAM, there are no crashes recorded. This is credited to the fact that iRAM predicts future heap usage, and then dynamically updates the global heap size thresholds such that they are always higher than application heap requirements.

Is the crash rate of up to 14 percent of the state-of-the-art significant? We think it is. Bad application performance means losing users. Users will not tolerate a problematic mobile app, and will abandon it after only one or two failed attempts. According to a recent study [6], 79 percent of users would retry a mobile app only once or twice if it failed to work the first time. If dissatisfied with the performance of a mobile app, 48 percent of users would be less likely to use the app again.

So what are the main reasons causing app crashes? There are no statistics covering applications across the board, but based on error submissions from Facebook’s user base, 62 percent of app crashes are due to OutOfMemory errors, while others are due to various system instability (CPU, GPU, network, storage, etc.). The results are based on 1.385 billion Facebook’s mobile monthly active users across all mobile platforms (Android, iOS, Windows Mobile, etc.). According to our analysis, most OutOfMemory errors are caused by bad programming habits of application developers. iRAM can minimize consequences of poor memory management, but only developers can eliminate root causes of app crashes. To avoid memory leaks, developers may follow this simple guidance:



(a) Crash Rate (NFS: Need for Speed; TR: Temple Run; TS: The Simpsons; AB: Angry Birds; GTA: Grand Theft Auto)



(b) Free Memory

Figure 6. Crash Rate and Free Memory.

1) Strictly apply pairs based on owner lifecycle
   - onResume $->$ onPause
   - onCreate $->$ onDestroy
   - onAttachToWindow $->$ onDetachFromWindow
2) Use only what is needed
   - plan standard memory usage, and specify in the app's manifest if higher usage is expected
   - monitor usage (use tools such as dumpsys, littleeye, Omura, etc.)

B. Free Memory

In this experiment, we study memory usage of the five popular applications used in Measurement Study: Facebook, YouTube, CNN, Angry Birds, and Temple Run. In particular, we launch the five apps every ten minutes and issue 100 user events via Android Monkey [11], which corresponds to using an app for a few seconds. Then we close each app with the Home button keyevent, assuring that an app does not get killed and stays in the background. We record free memory levels within one hour. This is measured both on Samsung S4 and Nexus 4, with and without iRAM.

The free memory levels during this experiment are illustrated in Figure 6 (b). The results indicate that without iRAM, Samsung S4 has an average of 15.1 percent of free memory, and an average of 57.1 percent with iRAM (3.8 times more). Nexus 4 has an average of 7.5 percent of free memory without iRAM, and 35.9 percent of free memory with iRAM (4.8 times more). This large amount of memory on both devices is made available thanks to the Priority Manager that periodically finds candidate processes to be killed based on Android's importance hierarchy, from the lowest to the highest importance. However, if there are any high priority user processes among the candidate processes, they are filtered out and hence not killed. High priority user processes are obtained through a prediction method, and we will find out in the following subsection how well such prediction works.

C. Prediction Accuracy

To evaluate prediction accuracy, we run the prediction on a dataset from real-world iPhone usage of 34 students [30] in the period of one year. The trace with applications run by users includes entries with user IDs, names of applications, and time stamps. The prediction accuracy results are displayed in Figure 8. As seen, the more applications in the Whitelist, the higher likelihood it includes an app to be used next. In other words, the more apps in the Whitelist, the better the prediction. However, the accuracy growth slows down significantly when close to 5-6 apps, and there are almost no changes when close to 9-10 apps. Since we aim for high accuracy, but also not having too many apps to potentially pollute memory, nine applications seem reasonable. That is also what iRAM applies. Therefore, an app to be used next by the user exists in the Whitelist with the probability of 91 percent.

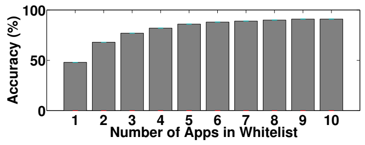


Figure 8. **Prediction Accuracy.**

Since iRAM has 91 percent of prediction accuracy, in order to evaluate launch delay of an application, we can assume the app is in the Whitelist. This is discussed in the following subsection.

D. Launch Delay

To address the second question on how iRAM improves application performance, we measure launch delay of 40 popular apps (10 games, 10 streaming, 10 miscellaneous, and 10 sensing) from Google Play, with and without iRAM. In order to evaluate application launch with iRAM, we insert each tested app in the Whitelist. During the experiment, only one app runs at a time. This is to achieve a fair comparison between the two cases: with iRAM, and without iRAM.

The Android Monkey tool [11] is utilized to trigger the launch process of each app. The application *launch delay* starts when the launch process is triggered, and ends when the process completes. The launch delay includes three components. We use the *time* command [1] to output the three time components: the time taken by the app in the user mode (*user*), the time taken by the app in the kernel mode (*system*), and the time the app spends waiting for the disk and network I/Os to complete (*totalIO*). The storage I/O delay is obtained by dividing the total number of I/Os completed *(kBread + kBwrtn)* over the total rate of I/Os completed *(kBreadRate + kBwrtnRate)* in a flash block device. The network I/O delay is then calculated as the total I/O delay *(totalIO)* subtracted by the storage I/O delay *(diskIOdelay)*.

The launch delay of the 40 apps with and without iRAM

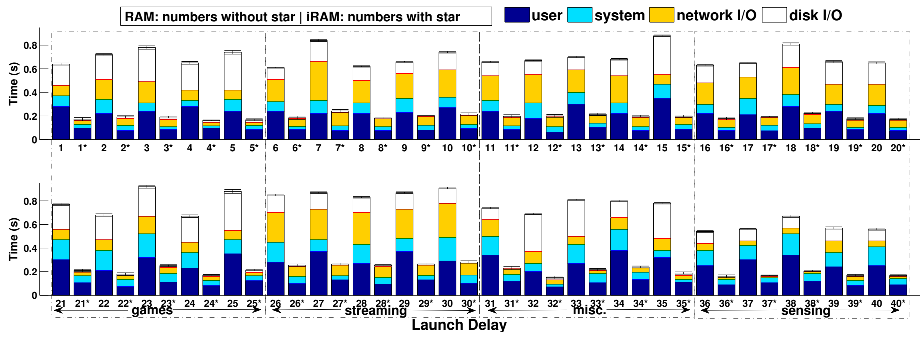
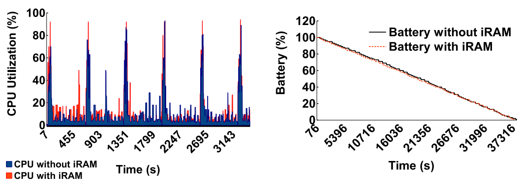


Figure 9. **Launch Delay.** **1:**Angry Birds; **2:**GTA; **3:**Need for Speed; **4:**Temple Run; **5:**The Simpsons; **6:**CNN; **7:**Nightly News; **8:**ABC News; **9:**YouTube; **10:**Pandora; **11:**Facebook; **12:**Twitter; **13:**Gmail; **14:**Google Maps; **15:**ZArchiver; **16:**Accelerometer M.; **17:**Gyroscope Log; **18:**Proximity Sensor; **19:**Compass; **20:**Barometer; **21:**2048 Puzzle; **22:**Pet Rescue Saga; **23:**Pou; **24:**Solitaire; **25:**Words; **26:**CT 24; **27:**Live Extra; **28:**VEVO; **29:**VOYO.cz; **30:**WATCH ABC; **31:**Instagram; **32:**File Commander; **33:**RAR for Android; **34:**Dropbox; **35:**File Manager; **36:**Physics Toolbox; **37:**Sensor Kinetics; **38:**Android Sensor Box; **39:**Sensor Music Player; **40:**Sensor Mouse.



(a) CPU Overhead. (b) Power Overhead.
Figure 10. Launch Delay and Overhead.

is illustrated in Figure 9. The figure includes 10 games (1-5, 21-25), 10 streaming apps (6-10, 26-30), 10 miscellaneous apps (11-15, 31-35), and 10 sensing apps (16-20, 36-40). Applications running with iRAM are denoted with a star (*). The reduction in launch delays with iRAM ranges from 68.8 percent (Instagram) to 78.2 percent (File Commander) as compared to delays without iRAM. The launch delay with iRAM enabled for all the 40 apps is on average 71.9 percent faster than with iRAM disabled. These results are expected. The app launch is I/O intensive, and includes a lot of I/O activities involving the flash disk. However, thanks to iRAM, an application being launched is already in the background, and most I/Os only involve the main memory, which is much faster than the flash disk. The speed of main memory on our device is 400Mbps, while the speed of the flash disk is only 24Mbps, which makes the main memory 16.7 times faster than the flash disk.

E. Overhead

To answer the last question whether iRAM incurs any performance penalties, we evaluate iRAM's CPU and power overhead. We study the CPU overhead of the five popular applications used in Measurement Study: Facebook, YouTube, CNN, Angry Birds, and Temple Run. In particular, we launch the five apps every ten minutes and issue 100 user events via Android Monkey [11], which corresponds to using an app for a few seconds. We record CPU utilization within one hour, with and without iRAM, and the results are illustrated in Figure 10(a). As observed, CPU utilization with iRAM is on average 3.5 percent higher than the case without iRAM. iRAM's CPU utilization peeks during the launch time of the five apps, but since each optimization period lasts only 0.13 second, the average overhead is acceptable. While improving the application performance is important, having solid power efficiency is equally important. To evaluate power overhead, we launch the above five apps every ten minutes and issue 100 user events via Android Monkey, with and without iRAM. The battery on Samsung S4 is fully charged at the beginning. We record how long the battery lasts for each case. The results are presented in Figure 10(b). While without iRAM, the battery lasts 10.54 hours (10 hours 32 minutes). With iRAM, the battery lasts 10.44 hours (10 hours 26 minutes). This implies that the power overhead of iRAM is 0.9 percent, which is acceptable.

VII. Related Work

The previous work can be classified into 3 categories: smartphone storage, application delay, and enterprise solutions.

Smartphone Storage. Kim et al. [20] present an analysis of storage performance on Android smartphones and external flash storage devices. Nguyen et al. [22], [23] study the impact of flash storage on smartphone energy efficiency. In their more recent work [25], [24], [26], the authors also research storage I/O behaviors, and design a system that improves application response times by prioritizing reads over writes. Our work does not study internal or external flash storage devices, but instead focuses on smartphone's memory component that is also referred to as dynamic random-access memory (DRAM), or often simply random-access memory (RAM) [7]. Finally, Guo et al. [18] propose a system that modifies page swapping and employs several flash-aware techniques. However, a major disadvantage of swapping is longer memory access time that negatively impacts application responsiveness.

Application Delay. Yan et al. [32] and Parate et al. [28] propose systems to reduce the launch delay via prelaunching. Our proposed solution (iRAM) learns which apps are of high priority for a particular user, and keeps them in the main memory. Hence, there is no complex prelaunching involved as in the previous work. In [31], [33], the authors develop applications that present icons for the most probable apps on the main screen of the phone and highlight the most probable one. Li [21] presents an on-device service that predicts actions the user is likely to perform at a given time. In contrast, iRAM predicts heap usage of applications, and dynamically adjusts the heap size based on predicted values. With this set of simple optimizations, in addition to reduced application delays, iRAM also contributes to smaller likelihood of crash behaviors.

Enterprise Solutions. DRAM technology has been recognized in enterprise systems. Over the years, DRAM has been used to improve performance in main-memory database systems [16], [17], and large-scale web applications have rekindled interest in DRAM-based storage in recent years. In addition to special-purpose systems like web search engines [12], general-purpose storage systems [3], [19], [14], [27], [29] also keep part or all of their data in memory to maximize performance. Inspired by these works, we believe that we are among the first to study smartphone application performance from the main-memory perspective.

VIII. Conclusion and Future Work

This paper presents iRAM, a personalized system that maintains optimal heap size limits to avoid crashes, efficiently maximizes free memory levels, and cleans low-priority processes to reduce application delays. In future work we plan to evaluate the impact of other stages of the life cycle on application performance such as install, update, switch, and uninstall, and quantify their effects on everyday phone usage.

Acknowledgment

This work was supported in part by CIT grant MF15-032-IT.

References

[1] Time man page. http://goo.gl/dEKuxs. (2014).

[2] Android Known Issues. https://code.google.com/p/android-developer-preview/wiki/KnownIssues. (2015).

[3] A distributed memory object caching system. http://memcached.org/. (2015).

[4] Dumpsys Meminfo. http://goo.gl/i6qCvt. (2015).

[5] Linux Free Command. http://linux.die.net/man/1/free. (2015).

[6] Mobile Apps: What Consumers Really Need and Want. http://goo.gl/VcE59W. (2015).

[7] Mobile DDR. http://goo.gl/lFEaBk. (2015).

[8] Processes and Threads. http://goo.gl/KraoUH. (2015).

[9] ART and Dalvik. https://source.android.com/devices/tech/dalvik/. (2016).

[10] Little Eye Labs. http://www.littleeye.co/. (2016).

[11] Monkey. http://goo.gl/F14hW. (2016).

[12] Luiz André Barroso el al.. Web search for a planet: The Google cluster architecture. *IEEE Micro*, 2 (2003).

[13] Stuart K Card el al. The information visualizer, an information workspace. *ACM SIGCHI*.

[14] Fay Chang el al. Bigtable: A distributed storage system for structured data. *ACM TOCS* (2008).

[15] John G Cleary et al. Data compression using adaptive coding and partial string matching. *IEEE ToC*.

[16] David J DeWitt et al. *Implementation techniques for main memory database systems*.

[17] Hector Garcia-Molina et al. Main memory database systems: An overview. *IEEE TKDE* (1992).

[18] W. Guo et al. MARS: Mobile Application Relaunching Speed-up through Flash-Aware Page Swapping. *IEEE TC* (2015).

[19] Robert Kallman et al. H-store: a high-performance, distributed main memory transaction processing system. *VLDB Endowment* (2008).

[20] Hyojun Kim et al. Revisiting storage for smartphones. *ACM ToS* (2012).

[21] Yang Li. Reflection: Enabling Event Prediction As an On-device Service for Mobile Interaction. *ACM UIST 2014*.

[22] David T Nguyen et al. Storage-aware smartphone energy savings. *ACM UbiComp 2013*.

[23] David T Nguyen et al. Evaluating Impact of Storage on Smartphone Energy Efficiency. *ACM UbiComp 2013*.

[24] David T. Nguyen Improving Smartphone Responsiveness through I/O Optimizations. *ACM UbiComp 2014*.

[25] David T. Nguyen et al. Reducing Smartphone Application Delay through Read/Write Isolation. *ACM MobiSys 2015*.

[26] David T. Nguyen et al. Study of Storage Impact on Smartphone Application Delay. *ACM MobiSys 2014*.

[27] John Ousterhout et al. The case for RAMClouds: scalable high-performance storage entirely in DRAM. *ACM SIGOPS* (2010).

[28] Abhinav Parate et al. Practical prediction and prefetch for faster access to applications on mobile phones. *ACM UbiComp 2013*.

[29] Stephen M Rumble et al. Log-structured memory for DRAM-based storage.. *FAST 2014*.

[30] Clayton Shepard et al. LiveLab: measuring wireless networks and smartphone users in the field. *ACM SIGMETRICS* (2011).

[31] Choonsung Shin et al. Understanding and Prediction of Mobile Application Usage for Smart Phones. *(ACM UbiComp 2012)*.

[32] Tingxin Yan et al. Fast app launching for mobile devices using predictive user context. *ACM MobiSys*.

[33] Chunhui Zhang et al. Nihao: A predictive smartphone application launcher. *Mobile Computing*. Springer.