Computers, Materials & Continua
DOI:10.32604/cmc.2021.017872
Article

Tech Science Press

Mobile Memory Management System Based on User's Application Usage Patterns

Jaehwan Lee and Sangoh Park*

School of Computer Science and Engineering, Chung-Ang University, Dongjak-gu, Seoul, 06974, Korea
*Corresponding Author: Sangoh Park. Email: sopark@cau.ac.kr
Received: 15 February 2021; Accepted: 24 March 2021

Abstract: Currently, the number of functions to improve user convenience in smartphone applications is increasing. In addition, more mobile applications are being loaded into mobile operating system memory for faster launches, thus increasing the memory requirements for smartphones. The memory used by applications in mobile operating systems is managed using software; allocated memory is freed up by either considering the usage state of the application or terminating the least recently used (LRU) application. As LRU-based memory management schemes do not consider the application launch frequency in a low memory situation, currently used mobile operating systems can lead to the termination of a frequently executed application, thereby increasing its relaunch time. This study proposes a memory management system that can efficiently utilize the main memory space by analyzing the application usage information. The proposed system reduces the application launch time by leaving the most frequently used or likely to be run applications in the main memory for as long as possible. The performance evaluation conducted utilizing actual smartphone usage records showed that the proposed memory management system increases the number of times the applications resume from the main memory compared with the conventional memory management system, and that the average application execution time is reduced by approximately 17%.

Keywords: Mobile environment; memory management; machine learning; neural nets; user-centered design

1 Introduction

Various types of mobile applications are emerging with the development and widespread use of smartphones. Currently, the number of mobile applications registered in the application marketplace for mobile operating systems such as Android and iOS is approximately 2 million [1,2]. Smartphone users typically install dozens, and sometimes, even hundreds of applications on their devices; additional features are being included in applications to improve user convenience. Accordingly, requirements such as improvements in main memory capacity and computational

processing performance are increasing. The increasing demand for main memory can be accommodated with the use of hardware or software. The hardware approach involves expanding the main memory of the smartphone, whereas the software approach involves using the main memory efficiently by following carefully designed memory management policies.

Mobile operating systems, such as Android or iOS, implement application life cycle management [3], which partially frees up memory used by applications based on the execution state of the application and thus accelerates application launch requests under conditions of limited main memory capacity. Moreover, it caches as many applications as possible in the main memory so that they are loaded rapidly when switched or relaunched. To prevent main memory shortages, cached applications that are not frequently used are terminated. This prevents users from experiencing reduced system performance when they relaunch frequently used applications. Swap techniques that use secondary storage as part of the main memory space have been considered [4–6] to efficiently utilize the main memory of smartphones. However, owing to wear-out problems of NAND flash memory-based storage devices, ZRAM [7], which compresses memory space and uses it as swap space, or ZSWAP [8], which is used as swap cache, can be used for smartphone memory management. In existing memory management techniques applied to smartphones [9] it is difficult to distinguish between frequently and infrequently used applications, because these techniques consider that the most recently executed application is more likely to be executed again than other applications [10]; this problem is exacerbated as users install and use more applications.

In this paper, we propose a memory management system to efficiently utilize the main memory and swap space by analyzing the application usage patterns and application execution probabilities of the user. The proposed system reduces the execution time of applications by keeping in the main memory the applications that are frequently used or that present a high probability of being relaunched. In low-memory situations, the main memory space allocated to applications is reclaimed by terminating the least likely to be reused application. Consequently, frequently used applications remain in the memory.

The remainder of this article is organized as follows. In Section 2, we summarize the existing studies on memory management for mobile operating systems. In Section 3, we introduce the proposed memory management system for efficiently managing the main memory of mobile operating systems. In Section 4, we evaluate and analyze the application launch performance of the proposed and existing memory management systems. Finally, in Section 5, we present conclusions and future research directions.

2 Related Work

Unlike PC and HPC-based operating systems, mobile operating systems deployed in smartphones follow memory management policies that are suitable for mobile environments. Mobile applications can be cached in the main memory to reduce their launch time without the need for a high-performance processor or storage. The number of applications cached in the main memory can be increased with the use of compressed memory. Currently, users typically install dozens, and sometimes, even hundreds of applications on their smartphones [11]; some of these applications are frequently used, such as messengers and browsers, whereas others are rarely used [12]. Mobile operating systems allow users to switch screens by launching other applications, maintaining the previously used application in memory for as long as possible; this ensures that the application resumes from memory when launched again.

Android is an operating system that holds approximately 70% of the mobile operating system market share [13], making it one of the most installed operating systems on smart devices. It includes an operating system and middleware for mobile devices and operates based on Linux kernel. In addition to the existing functionalities for Linux kernel, Android specific functions such as power management and memory management, are also included in this system.

Android can host applications written in Java and Kotlin in separate processes on the Android runtime virtual machine. It implements an application life cycle management policy [3] that performs memory allocation and release based on the execution state of the applications. Running applications remain in the main memory with sufficient main memory unless explicitly terminated. The *application management service* (AMS) in the Android framework manages and tracks application status and records the status information in processes that host the application. When the system experiences memory shortage, Android's memory manager runs the *low memory killer* (LMK) [14–16] driver to terminate the applications, executing the least recently used (LRU) [9] method, thus reclaiming memory. This reclaiming task is repeated until sufficient free space has been reclaimed by the system.

The commonly used LRU-based memory management policies assume that recently used applications are likely to be launched again [12]. However, the effectiveness of these policies decreases with the increase in the number of applications that a user installs. Frequently used applications can be selected for termination if several infrequently but recently used applications are present. In this case, the application launch time will increase because frequently used applications are launched from the device storage.

Other software approaches for memory management feature swap techniques utilize a part of the main memory as compressed storage space. The swap techniques use a part of the free space from the secondary storage as swap area. This allows for the use of memory resources beyond the physical memory limit of the system. When the main memory space is not sufficient to allocate a page, the memory manager migrates a page in the main memory to the swap area. The swapped-out page is brought back to the main memory when a process refers the page again. Another page is then selected to be swapped out. The swap-in and out task is frequently performed if the main memory keeps running low. Therefore, NAND flash memory based mobile devices do not present a swap feature that employs secondary storage to prevent storage wear-outs. Main memory-based swap techniques, such as ZRAM or ZSWAP, are utilized to compress and store swap data using a part of the main memory space as a block device. However, these swap techniques present compression and decompression overhead; the overhead increases with the amount of swap out/in data to/from the compressed area.

In a study using ZRAM and storage as the swap area [17], the swap cost of a page was calculated to swap out frequently referenced pages into ZRAM while swapping out infrequently used pages to storage. Taking compression ratio [18] or application behavior [19] into consideration to estimate the cost of a page showed benefits for swapping out pages, extended storage lifespan and higher application launch speeds. In a study that proposed memory management using the average reuse distance [10,14], which indicates the number of times other applications were launched between the launch and relaunch of an application, it was considered that an application was more likely to be relaunched when its average reuse distance was smaller. Moreover, the total number of swaps was reduced by preventing the swap out of applications with a small average reuse distance and allowing swap out of applications with a high average reuse distance. A cloud-based memory expansion scheme [20], which is a device-reserved memory management scheme for mobile devices [21], was also investigated to accommodate the increasing memory

demands. However, existing memory management techniques present limitations when reflecting application usage patterns that are appropriate for the user. This is because they do not consider the application launch data and the correlation between applications.

In this study, we present an approach for estimating the application launch probability by collecting users' application usage information and training a long short-term memory (LSTM) [22–24] model considering the association between the various usage patterns of the users. Based on the analysis of the usage pattern information, our proposed memory management system predicts the application to be launched, such that the system can determine the application that should be left in the main memory for a relaunch and that which should be terminated. Our approach utilizes the main memory and swap space more efficiently than existing memory management methods, when the least likely to be relaunched applications are terminated in low memory situations.

3 AMMS: An Application-Prediction-Based Memory Management System

In this paper, we propose *AMMS*, an application-prediction-based memory management system that analyzes the user's application usage patterns and launch probabilities. The overall architecture of *AMMS* is shown in Fig. 1. The *activity/process context generator* is a module added to the existing *activity manager service* to generate application launch information; it passes the generated information to the *context management* module. When the *application usage predictor* of the *context management* module receives the application launch information, it uses the LSTM network to predict the launch probability of the next application. The *application usage trainer* is a module that receives and stores application launch information and utilizes this information to train the LSTM network. In addition, the launch probability of each application is recorded as process information, which is managed by the *process management* module in the Linux kernel. The *AMMS interface* inside *file system management* is the interface module that transfers data from the Android framework to the Linux kernel. The *AMMS reclaimer* module of *memory management* searches for the process with the lowest launch probability when a main memory shortage is experienced; then, it terminates the process and frees up memory space. This immediately increases the resuming frequency of the most likely to be launched applications without removing them from memory. In summary, *AMMS* improves the application launch speed by utilizing the main memory more efficiently than existing methods.

3.1 Activity/Process Context Generator

The *activity/process context generator* generates contextual information when an application is launched, paused, and terminated in a mobile operating system, by collecting usage information such as the package name of the application, time of launch, process creation information generated for the application's execution, process ID, and application package name to which each process belongs. This module is located in the *activity manager service* that is responsible for launching and terminating applications. When an application is launched or a process is created, the *activity/process context generator* delivers application usage information to the *application usage trainer* and the *application usage predictor* in *context management*. In addition, when a process creation task for launching an application occurs, the relevant context information is transferred to the *application usage predictor*.

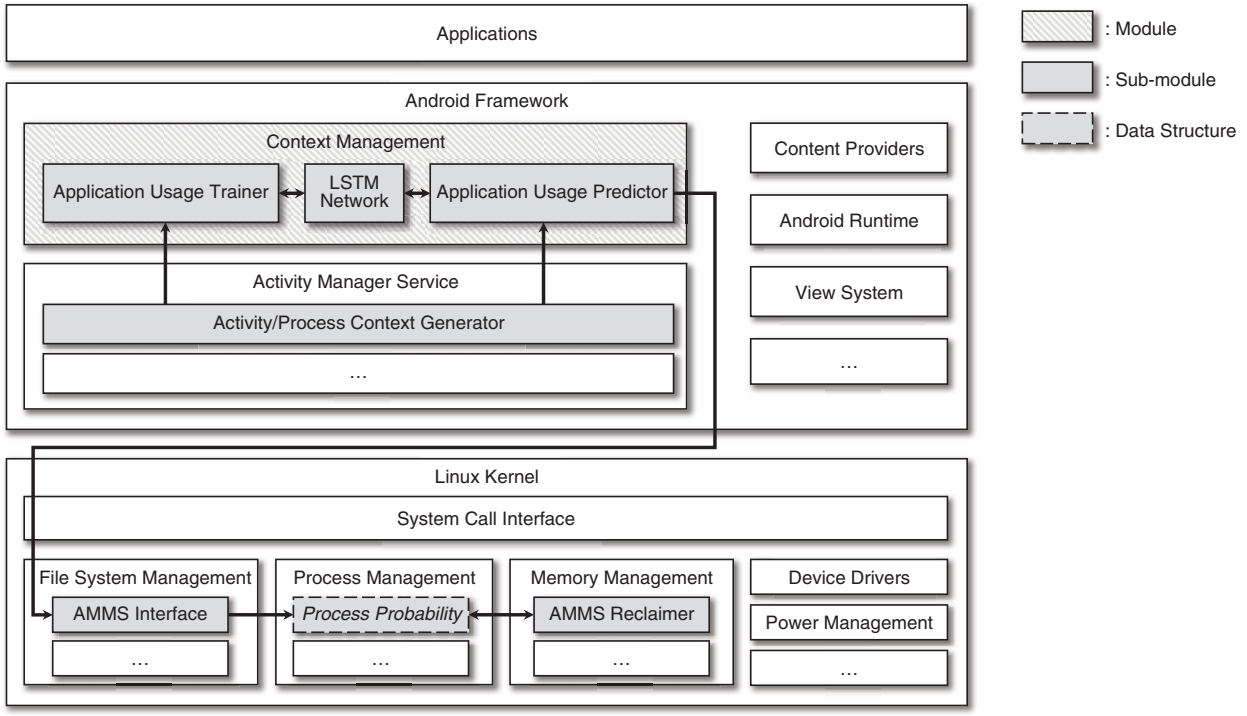

Figure 1: System architecture

3.2 LSTM Network

The LSTM network is located within the *context management* module. It learns the user's application usage pattern and predicts application launch probability. The parameters associated with the LSTM network are defined in Tab. 1. LSTM is designed to solve the problem of gradient vanishing in recurrent neural networks (RNNs) and learn the correlation between long-term and short-term data [25]. The functions forget gate f_t, input gate g_t, input gate i_t, memory cell c_t, and output gate o_t constituting the LSTM model are defined, as follows, in Eqs. (1)–(6).

$$f_t = \sigma(x_t w_{x\_f} + h_{t-1} w_{h\_f} + b_f) \tag{1}$$

$$g_t = tanh(x_t w_{x\_g} + h_{t-1} w_{h\_g} + b_g) \tag{2}$$

$$i_t = \sigma(x_t w_{x\_i} + h_{t-1} w_{h\_i} + b_i) \tag{3}$$

$$c_t = f_t \circ c_{t-1} + i_t \circ g_t \tag{4}$$

$$o_t = \sigma(x_t w_{x\_o} + h_{t-1} w_{h\_o} + b_o) \tag{5}$$

$$h_t = o_t \circ tanh(c_t) \tag{6}$$

The parameters used by the LSTM network for performing application launch probability prediction are defined in Tab. 2. The launch probability of application u_i at time $t+1$ with regard to application launch history v_t is defined as $P(u_{i\_t+1} \mid V_t)$. The data for the application launched at time t can be converted into the input of the LSTM model using Eq. (7). The function $lstm(x_t)$,

which performs operations according to the LSTM cell defined in Eqs. (1)–(6), can be expressed as in Eq. (8).

$$x_t = \{x_{t\_0}, x_{t\_1}, \ldots, x_{t\_j}, \ldots, x_{t\_m-1},\}, \quad where\ x_{t\_j} = \begin{cases} 1, & if\ i = j \\ 0, & otherwise \end{cases} \quad for\ (0 \le j \le m-1) \tag{7}$$

$$h_t = lstm(x_t) \tag{8}$$

The *softmax* function [26] presented in (9) is used to transform the outputs of $lstm(x_t)$ to a probability for u_{i_t+1}.

$$P(u_{i\_t+1} \mid V_t) = \frac{e^{h_{t\_i}}}{\sum_{j=0}^{m-1} e^{h_{t\_j}}} \tag{9}$$

Table 1: Definition of parameters related to long short-term memory

Symbol	Definition
t	Time at which an input vector is fed to an LSTM
x_t	Input vector to an LSTM at time t
x_{t_k}	k-th element of an input vector x_t
h_t	Output vector of LSTM at time t
h_{t_l}	l-th element of an output vector h
σ	Sigmoid activation function
$tanh$	Tangent hyperbolic activation function
f_t	Forget gate function at time t to control the amount of discarded stored data
c_t	Memory cell to store data in LSTM at time t
g_t	Input gate function at time t to generate data to be stored in c_t
i_t	Input gate function at time t to generate data to be stored in c_t
o_t	Output gate function at time t to generate data to be stored in c_t
w_{x_f}, w_{x_g}, w_{x_i}, w_{x_o}	Vector weighted to x_t in functions f_t, g_t, i_t, o_t, respectively
w_{h_f}, w_{h_g}, w_{h_i}, w_{h_o}	Vector weighted to h_t in functions f_t, g_t, i_t, o_t, respectively
b_f, b_g, b_i, b_o	Bias constant for functions f_t, g_t, i_t, o_t, respectively

3.3 Application Usage Predictor

The *application usage predictor* module estimates the launch probability of an application using the LSTM network and delivers the obtained probability value to the Linux kernel; application usage information is passed through the *activity/process context generator*, following which the *application usage predictor* inputs the information to the LSTM network to obtain the application launch probability.

The *application usage predictor* reads the application mapping tables stored in permanent storage, using the format presented in Tab. 3 and then creates an application process information list called *AppInfo*, as described in Tab. 4. Application launch time and application process creation are different on mobile operating systems such as Android; therefore, the *application usage predictor* receives application launch and application process creation information from the *activity/process context generator*.

Table 2: Definition of parameters related to application usage prediction

Symbol	Definition
A	Set of all applications installed on a smartphone
S	Set of all applications that a user cannot launch
U	Set of applications for which $S \cup U = A$ and $S \cap U = \varnothing$
R	Set of cached applications that reside in system memory at time t
m	Number of elements of U
u_i	i-th application of U
u_{i_t}	Launched application u_i at time t
V_t	Set of application launch history at time t
v_{t_j}	j-th elementof V_t
$lstm(x_t)$	Fully connected LSTM network function
$P(\cdot)$	Probability of an event

Table 3: Mapping table of application name to ID

Name	ID
Full package name of the application	*Unique id of the application*

Table 4: AppInfo managed by application usage predictor

Field	Description
name	Full package name of the application
id	Unique id of the application
prob	Probability that the application is launched
pidlist	List of pids of processes that belong to the application

Algorithm 1: Executed when an application has been selected to run

procedure on Application Launched (u_{i_t})
  Generate x_t according to (7)
  $h_t \leftarrow lstm(x_t)$ according to (8)
  for each $u_i \in U$ **do**
    u_i's *AppInfo.prob* $\leftarrow P(u_{i\_t+1} \mid V_t)$ according to (9)
    for each *pid* in u_i's *AppInfo.pidlist* **do**
      if *pid* exists **then**
        $p \leftarrow u_i$'s *AppInfo.prob*
        $i \leftarrow \lfloor p \times maximum\_integer\_value \rfloor$
        write i through *procfs*
      end if
    end for
  end for
end procedure

Algorithm 1 describes the task performed at time t when the *application usage predictor* receives the application launch information. Application relaunch probability acquired from the LSTM network as shown in lines 2–3 is stored in the *prob* field by traversing the *AppInfo* list as shown in lines 4–5. If the application's process has already been created, the probability value is stored in the kernel's process data structure through *file system management*, as in lines 6–14. Since the Linux kernel does not support floating point operations, it converts the probability value to an integer and stores it as in line 10.

Algorithm 2: Executed when a process for an application has been created

```
1: procedure on Process Created (u_{i_t}, pid)
2:    add pid in u_i's AppInfo.pidlist
3:    for each pid in u_i's AppInfo.pidlist do
4:       if pid exists then
5:          p ← u_i's AppInfo.prob
6:          i ← ⌊p × maximum_integer_value⌋
7:          write i through procfs
8:       end if
9:    end for
10: end procedure
```

Algorithm 2 describes the task performed when the created application process information is received. When a process is created, the corresponding *pid* is stored in the *pidlist* of the application's *AppInfo* to which the process belongs.

3.4 Application Usage Trainer

The *application usage trainer* module trains the LSTM network using application usage information; it receives and stores application launch information from the *activity/process context generator*. The unique IDs corresponding to applications are stored chronologically. The LSTM network is unfolded according to the length of the data and becomes a feed-forward network. A backpropagation through time (BPTT) algorithm is used to update the weights of the network. The LSTM network is trained to minimize the mean squared error (MSE), which is defined as the difference between the predicted and actual values. MSE, in combination with the application launch probability, is defined in Eq. (10).

$$MSE = \frac{1}{|V_T|}\sum_{t=1}^{|V_T|}\sum_{i=0}^{m-1}(P\left(u_{i\_t} \mid V_{t-1}\right) - y_{t\_i})^2, \quad where\ y_{t\_i} = \begin{cases} 1, & if\ u_{i\_t} = v_{t\_|V_T|} \\ 0, & otherwise \end{cases} \tag{10}$$

The training of the LSTM network by the *application usage trainer* requires a significant number of computations. Therefore, it only trains the network when the smartphone is charging and not in use.

3.5 AMMS Interface

The *AMMS interface* provides an interface for information delivery so that the *application usage predictor* can provide application launch probability information to a process data structure inside the Linux kernel. The information delivery interface is provided by *procfs* [27]. System

information, such as process information, is presented in a file-like structure using *procfs* in Unix-like operating systems; *procfs* is used to change system parameters at runtime through a common file I/O interface. In the proposed system, the *application usage predictor* can record the launch probability of an application in the process data structure managed by process management via *procfs*.

3.6 Process Probability

Process probability is a data structure for recording the launch probability of an application's process. The *prob* field is added to the task data structure managed by *process management* as shown in Tab. 5. The *application usage predictor* only estimates the probability of user launched applications since the termination of a system process may cause system crash or failure. The *prob* value for system processes is initialized with a negative integer value to exclude them from memory reclaim candidates.

Table 5: Process information managed by process management

Field	Description
...	...
pid	Unique id of the application
prob	Integer probability value set by application usage predictor
...	...

3.7 AMMS Reclaimer

Algorithm 3: Executed for low memory event

```
1: procedure on Low Memory
2:     tgproc ← null
3:     minproc ← minimum_integer_value
4:     for each proc in running_process_list do
5:         if 0 ≤ proc.prob and minprob ≤ proc.prob then
6:             tgproc ← proc
7:             minprob ← proc.prob
8:         end if
9:     end for
10:    terminate tgproc and reclaim memory
11: end procedure
```

AMMS reclaimer is a module that terminates the process of the application, which is least likely to be launched next, when the available main memory starts becoming low. Algorithm 3 describes the memory reclamation process performed by *AMMS reclaimer*. The variables for designating the process with minimum probability are initialized as shown in lines 2–3. The process list is traversed as in line 4, followed by the selection of the process with minimum probability in lines 5–8.

4 Performance Evaluation

4.1 Design of LSTM Network

We designed an LSTM model for AMMS by changing its hyperparameters to identify the most efficient structure for predicting mobile application usage. To establish an efficient structure of the LSTM model, the number of computations as well as the prediction performance of the model should be considered. The model validation accuracy was determined by changing the number of layers of the LSTM network from 2 to 4 and the number of LSTM neurons per layer from 10 to 100. The number of application usage records varied from 100 to 10000 in this experiment that helped determine the robustness of the model prediction accuracy. The application usage records were randomly selected from LiveLab Research's real-world usage data [28].

The model validation accuracy and training loss for each number of layers and neurons is shown in Figs. 2 and 3, respectively. As shown in Fig. 2, the validation accuracy increased as the number of neurons increased. However, the accuracy decreased as the number of layers increased. The accuracy of layer 2 was the highest with the number of neurons from 20 to 100. The training loss for each number of neurons and layers was determined to identify if overfitting occurred. As shown in Fig. 3, the training loss increased as the number of layers increased. If overfitting had occurred, the training loss would have decreased along with the validation accuracy. Therefore, it can be concluded that overfitting did not occur. The experimental results also show that the deeper the LSTM network structure, the more difficult it will be to learn the application usage of a user.

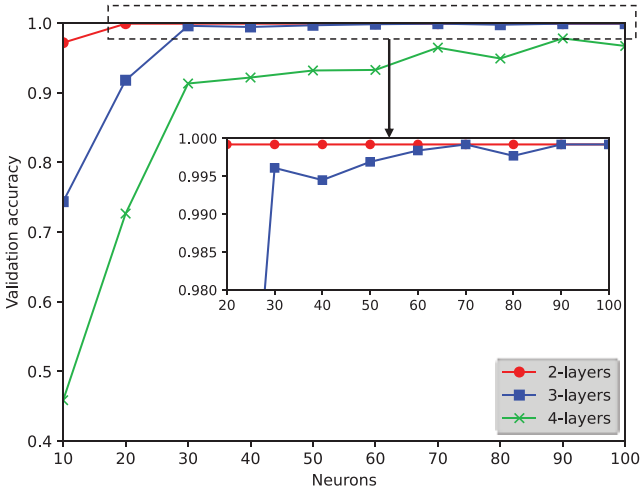

Figure 2: Comparison of validation accuracy

To consider a model for the mobile environment, it is necessary to identify the most computationally efficient model. We employ a method used in [29] to estimate the number of computations required to perform a single prediction task of an LSTM network. The number of computations is calculated in terms of L, N, m, which represent the number of layers, the number of LSTM neurons, and the number of elements of x_t, respectively.

Unlike the previous work, the number of application types in the usage dataset, represented by m, varies depending on the dataset. The number of applications according to the number of samples in a dataset is shown in Fig. 4. The number of applications increases with the number of

samples in a dataset. Furthermore, the accuracy varies as the number of applications changes. The validation accuracy according to the number of samples is collected for each number of layers and neurons. Note that models with 4 layers are excluded since they have been shown to be not appropriate for the application usage dataset. Figs. 2 and 5 show that the validation accuracy tends to increase as the number of neurons is increased; however, the same trend is not observed with increase in the number of layers. The accuracy is not always higher for one model compared to another one, e.g., for the models $l2n90$ and $l3n20$. The accuracy for model $l2n90$ is lower than that for $l3n20$ with the number of samples from 100 to 200, and it becomes higher for model $l2n90$ when the number of samples is larger than 600.

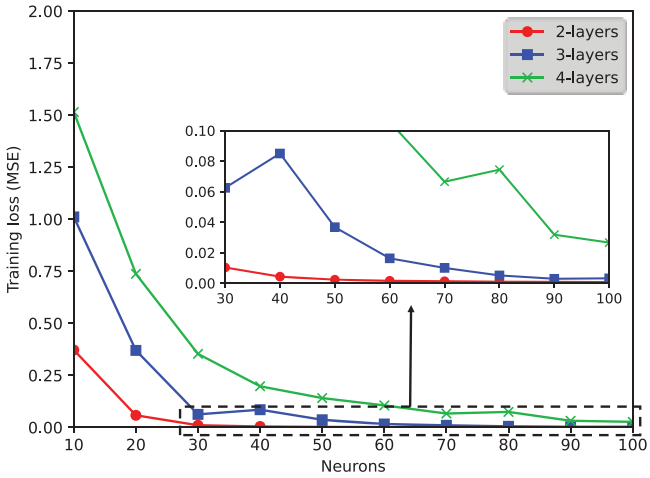

Figure 3: Comparison of training loss

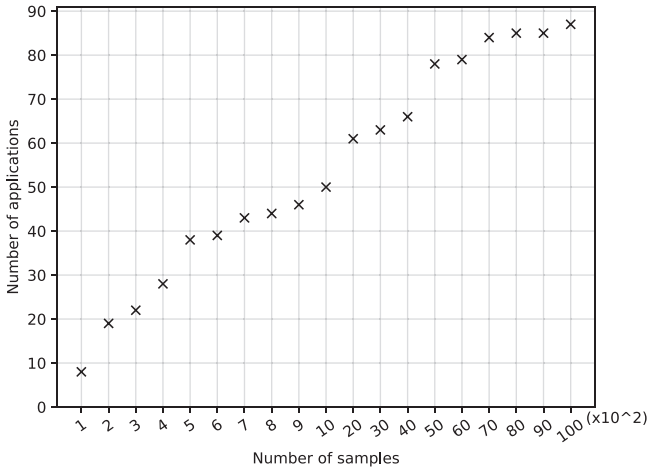

Figure 4: Number of applications *vs*. the number of samples in the dataset

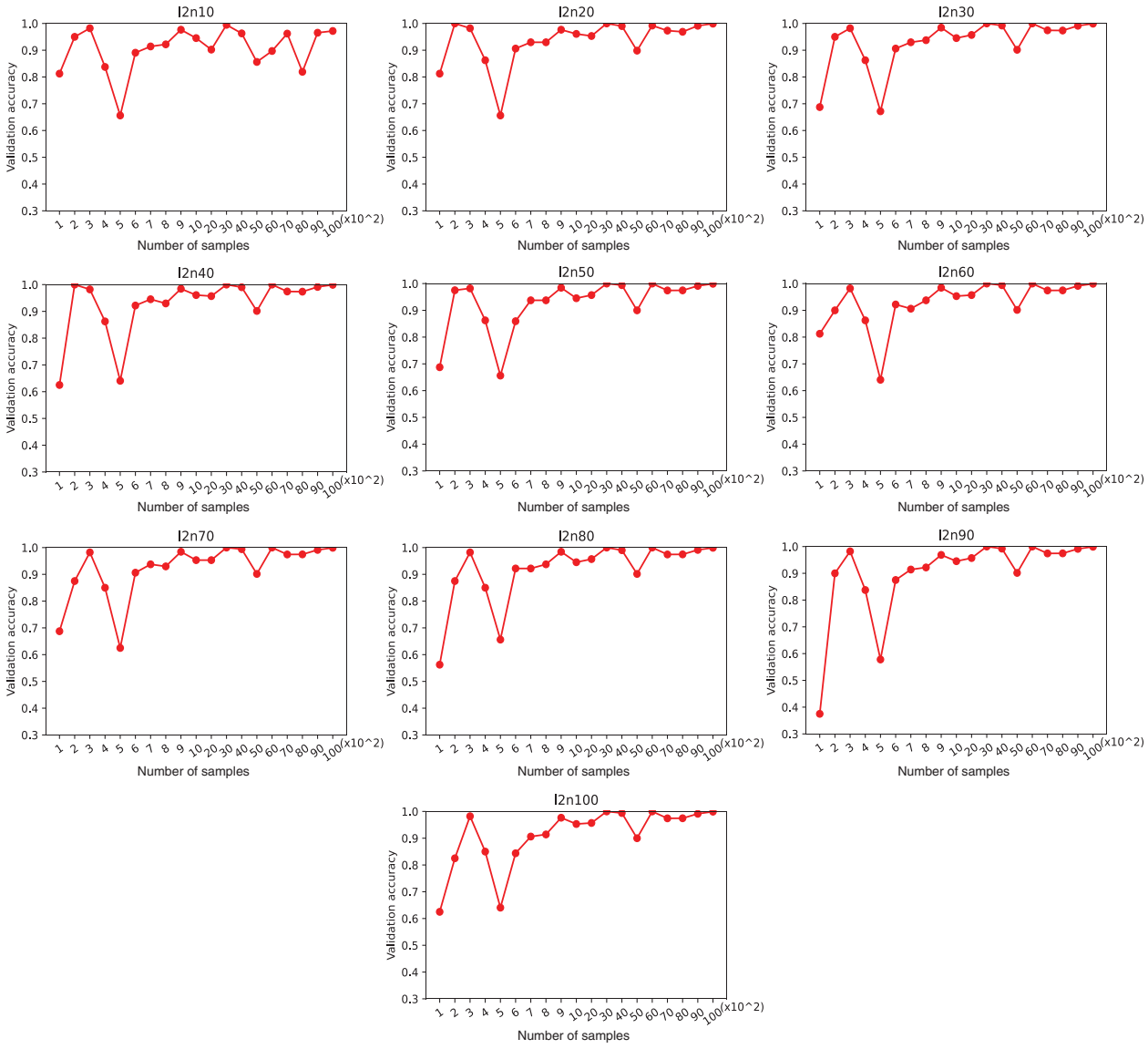

Figure 5: Validation accuracy of 2-layer model *vs*. the number of samples

Based on the experimental results presented in Figs. 2, 5, and 6, it is concluded that a model should not be chosen unconditionally although the accuracy is higher for a specific sample range, nor the number of computations is smaller. Therefore, we propose an evaluation method to identify a model that provides high efficiency and good performance. The number of computations required to perform a single prediction task of an LSTM network is determined using Eq. (11) [29].

$$p = L\left(4N^2 + 5Nm + 8N\right) + 2m \tag{11}$$

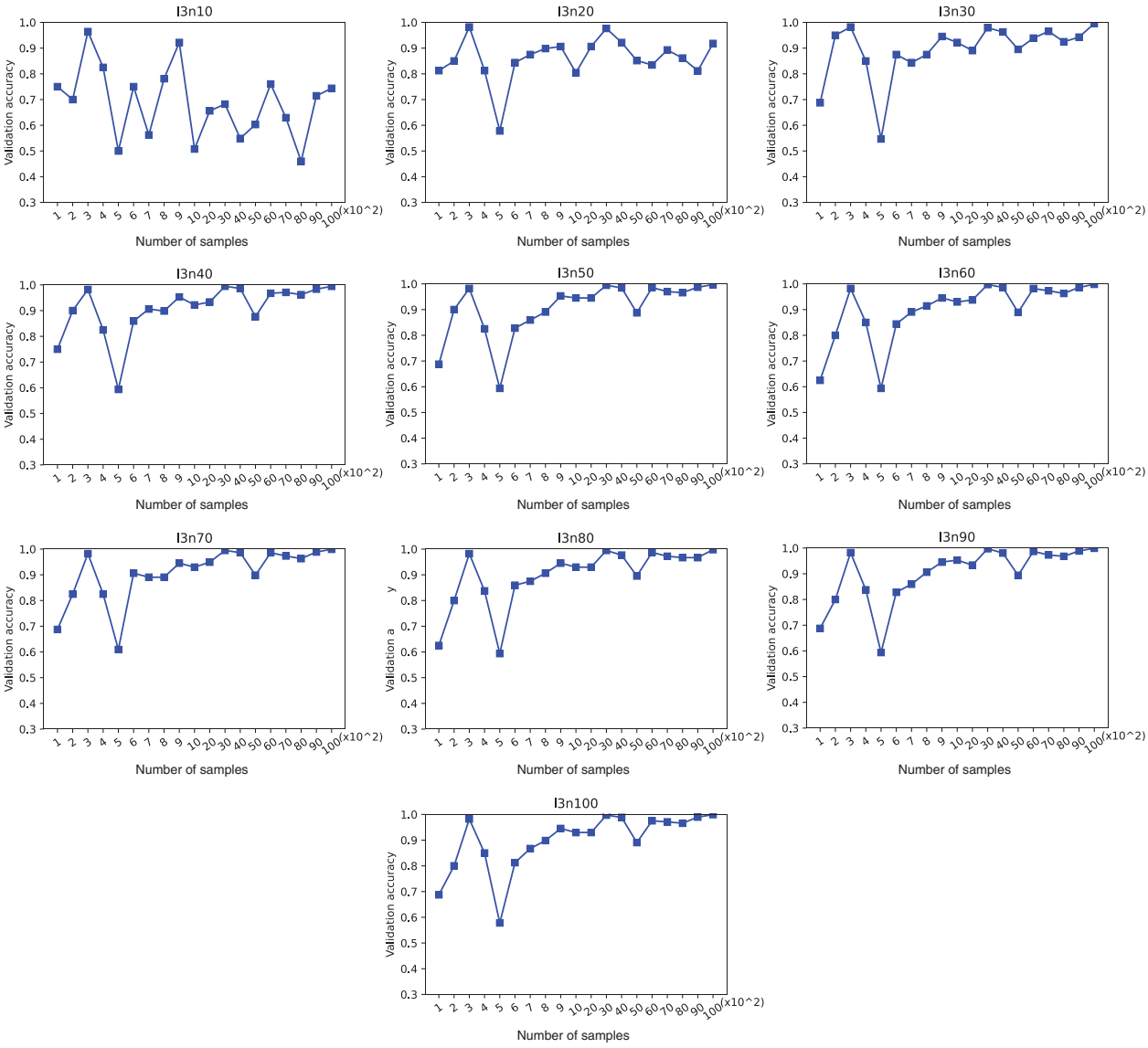

Figure 6: Validation accuracy of a 3-layer model *vs*. the number of samples

Algorithm 4: Determination of model score

1: **procedure** get Model Score (l', n')
2: $score \leftarrow 0$
3: **for each** $\forall\, l \in Layers, \forall\, n \in Neurons, \forall\, s \in Samples$ **do**
4: $m \leftarrow$ the number of apps from Fig. 4 with s
5: $comp_{this} \leftarrow l' \left(4n'^2 + 5n'm + 8n'\right) + 2m$
6: $acc_{this} \leftarrow$ validation accuracy of a model with l', n'

(Continued)

```
 7:      comp_target ← l(4n^2 + 5nm + 8n) + 2m
 8:      acc_target ← validation accuracy of a model with l, n
 9:      if comp_this < comp_target then
10:        if acc_this ≥ acc_target then
11:          score ← score + 1
12:        end if
13:      end if
14:    end for
15:    return score
16: end procedure
```

Algorithm 4 depicts the proposed method to evaluate an LSTM model considering its performance and computational efficiency simultaneously. To evaluate a model with l' layers and n' neurons, its validation accuracy for each number of samples s is compared to that of all other models. A model's score indicates how many other models with higher complexity and lower accuracy exist. Line 3 is executed to search through all other models with l layers, n neurons, and s number of samples. The computation amount of the model for which we want to calculate the score is expressed as $comp_{this}$ in line 4. The accuracy of the model is expressed as acc_{this} in line 5. The computation amount and the accuracy of a model that needs to be compared with are expressed as $comp_{target}$ and acc_{target}, respectively, in lines 7–8. As shown in lines 9–13, if there is a model for which the number of computations is larger than $comp_{this}$ and the accuracy is equal to or lower than acc_{this}, the score count is increased by 1. For example, with 200 samples, if acc_{this} of $l2n10$ is higher than acc_{target} of $l3n10$ and $comp_{this}$ of $l2n10$ is lower than $comp_{target}$ of $l3n10$, the score of $l2n10$ is increased by 1.

Fig. 7 shows the score of each model for different numbers of layers and neurons. Models with 2 layers outperformed other models and the $l2n20$ model was rated a score of 261, the highest score among all models. Score 261 means that there were 261 cases in which $l2n20$ performed more accurately and was computationally more efficient. In the case of $l2n10$, since it involves the smallest number of computations, there were 167 cases in which the model performed better. The score decreased as the number of computations increased. Therefore, the proposed AMMS was implemented with the $l2n10$ model based on the model evaluation results.

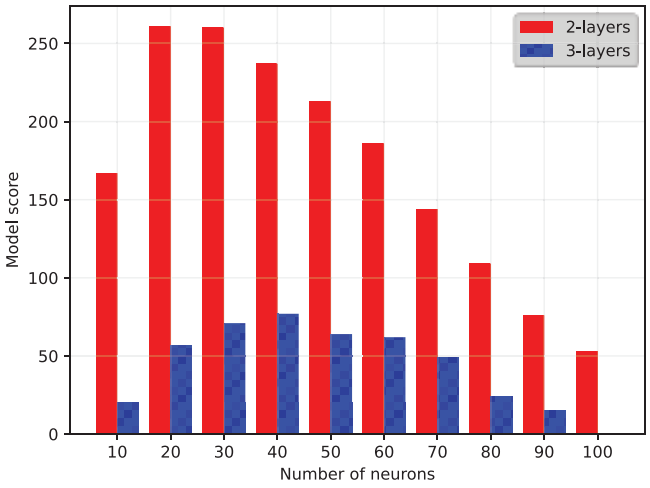

Figure 7: Comparison of the number of computations

4.2 Evaluation of AMMS

We implemented *AMMS* on an Android smartphone to evaluate the performance of the proposed system. The launch time and characteristics of mobile applications were collected and analyzed. The device used for performance evaluation was a Google Nexus 6P smartphone [30] with a Snapdragon 810 processor, as shown in Tab. 6.

Table 6: Target device specification

Model	Google Nexus 6P
Processor	Qualcomm Snapdragon 810 MSM8994 SoC ARM Cortex-A57 2 GHz + ARM Cortex-A53 1.55 GHz
Memory	3 GB LPDDR4 SDRAM
Display	WQHD (2560 × 1440) AMOLED touch screen display
Storage	32 GB flash storage
OS	Android 7.1.2 (Nougat) with Linux Kernel 3.10.73

Algorithm 5: Determination of application launch type

```
1: procedure get Launch Type (u_{i_t})
2:     type ← WarmLaunch
3:     time ← u_{i_t}'s launch time from Android framework
4:     if time is null then
5:         type ← HotLaunch
6:     else if u_{i_t}'s AppInfo.pidlist is empty then
7:         type ← ColdLaunch
8:     end if
9:     return type
10: end procedure
```

There are three types of application launches: cold launch, warm launch, and hot launch [31]. Hot launch occurs when all the application processes have been already created and there is no need to create new ones to execute the application. In warm launch, some processes have been created, but additional ones need to be created to execute the application. In cold launch, processes have not been created yet, so all the necessary processes need to be created. The performance measurement data were randomly selected from the actual application usage data collected by LiveLab Research [28]. The total number of applications in the application usage history was 35 (see Tab. 7). Ten-fold cross validation was performed with 90% (4500) of the 5000 application usage records for training and 10% (500) of the records for validation.

Android logs the time interval from application launch to application screen rendering, i.e., it provides cold and warm launch information. The time is not logged for hot launches because the process is already created, and the application is already rendered. Therefore, we compared the proposed and existing methods in terms of the application launch time information provided by Android to measure the number of launch types. Algorithm 5 describes the manner in which application launch type is determined. Hot launch occurs when the Android framework does not provide the launch time of an application, as shown in lines 3–5. If the

framework provides the launch time but no data is left in the main memory, cold launch occurs, as shown in lines 6–7. Otherwise, warm launch occurs. The existing memory management system is labeled $Android + ZRAM + LRU$ to indicate that the system is built from the source code of the Android platform and its kernel, and the proposed memory management system is labeled $Android + ZRAM + AMMS$.

Table 7: Tested package set for target platform

Original package name	Tested package name	Original package name	Tested package name
com.apple.Preferences	com.android.settings	com.apple.MobileSMS	com.android.messaging
com.apple.mobilephone	com.android.dialer	com.apple.compass	com.vincentlee.compass
com.apple.weather	com.weather.Weather	com.apple.mobilemail	com.google.android.gm
com.apple.mobilesafari	com.android.chrome	com.apple.mobileipod-MediaPlayer	com.mxtech.videoplayer.ad
com.apple.calculator	com.android.calculator2	com.apple.mobiletimer	com.android.deskclock
com.apple.VoiceMemos	com.splendapps.vox	com.apple.stocks	com.yahoo.mobile.client.android.finance
com.apple.mobileslideshow-Photos	com.android.gallery3d	com.apple.mobileslideshow-Camera	com.android.camera2
com.apple.youtube	com.google.android.youtube	com.apple.AppStore	com.android.vending
com.apple.MobileStore	com.google.android.music	com.apple.MobileAddress-Book	com.android.contacts
com.apple.mobilenotes	com.socialnmobile.dictapps.notepad.color.note	com.apple.mobilecal	com.google.android.calendar
com.apple.Maps	com.google.android.apps.maps	com.apptomic.bugspray	com.alphapps.antiflysound
com.gothamwave.SubwayMap	net.orizinal.subway	com.myapps.NYSubwayApp	uk.co.mxdata.newyorksub
com.google.GoogleMobile	com.google.android.googlequicksearchbox	com.google.b612	com.linecorp.b612.android
com.davaconsulting.ruler	org.nixgame.ruler	com.ihandysoft.carpenter.level	com.ihandysoft.carpenter.level
com.johnhaney.Flashlight	com.surpax.ledflashlight.panel	com.oishan.DriversEd	com.driversed.driversed
com.srividya.nycway	com.ulmon.android.play-newyork	com.pandora	tunein.player
com.rhapsody.iphone.Rhapsody	com.nhn.android.music	com.trancreative.Speller	com.nounplus.grammarcheckerenglish
com.Epocrates.Rx	com.Epocrates.Rx	com.espn.ScoreCenter	com.espn.score_center

In order to evaluate application launch performance, the validation dataset of 500 application records was executed in order, and the number of hot, warm, and cold launch occurrences was measured for the existing and proposed systems. Tab. 8 shows the average number of launches of each type. The average number of cold launches was 76.4 for $Android + ZRAM + AMMS$, which was approximately 10% lower than the average of 85 for $Android + ZRAM + LRU$. Moreover, $Android + ZRAM + AMMS$ reduced the average number of warm launches, in which application data and processes remained in the main memory, by approximately 11% (from 14.4 to 12.8), in comparison with $Android + ZRAM + LRU$. These results indicate that our proposed memory

management system predicts the application likely to be launched next, more accurately than the existing system.

Table 8: Average launch time for each type

	Hot launch	Warm launch	Cold launch
$Android + ZRAM + LRU$	400.6	14.4	85.0
$Android + ZRAM + AMMS$	410.8	12.8	76.4

Next, we measured application launch time, which is the time required to completely load an application, create processes and activities, and render the initial screen; we evaluated how this parameter is affected by the application prediction accuracy of the existing and proposed systems. Application launch time was measured using the measurement tool provided by the Android framework [30]. Since the measurement tool provides the launch time for cold and warm launches, hot launches correspond to the data and processes cached in main memory; hot launch time was not measured in this study and thus treated as 0 ms.

Tab. 9 shows the cumulative average launch times for cold and warm launches obtained using the aforementioned validation dataset with 500 application records. The cumulative average launch time for warm launches decreased by about 11% (from 7095.8 ms for the existing system to 6302.8 ms for the proposed system). In addition, the launch time for cold launches reduced by about 18% (from 106128.6 ms for the existing system to 86810 ms for the proposed system). The total cumulative launch time (including warm and cold launches) for the proposed system was approximately 17% less than that for the existing system. Since warm and cold launches take more time than hot launches [30], a decrease in the launch time for cold and warm launches implies a decrease in the total launch time.

Table 9: Average launch time in milliseconds

	Hot launch	Warm launch	Cold launch
$Android + ZRAM + LRU$	7095.8	106128.6	113224.4
$Android + ZRAM + AMMS$	6302.8	86810.0	93112.8

The results shown in Tab. 9 are graphically compared in Fig. 8a. An increase in the cumulative launch time implies that warm or cold launches have occurred; otherwise, it implies that hot launches have occurred. Fig. 8b depicts the difference of cumulative average launch times between the existing and proposed systems. In Fig. 8b, there are sections where the cumulative launch time increased rapidly, e.g., around the launch of the 100th application or the 200th application. According to the results, application launch time improved in sections where warm and cold launches occurred frequently. Applications that are likely to be launched again remained in memory as much as possible on $AMMS$; warm and cold launches were turned into hot launches so that the cumulative launch time decreased. However, there are sections where the cumulative launch time for $AMMS$ was more than that for the existing system. Since the memory reclamation policy of $AMMS$ is different from the existing systems' LRU policy, a cold launch occurred in the section where a hot launch occurred in the existing system. The increased launch time in these sections is negligible compared to the increased time in other sections.

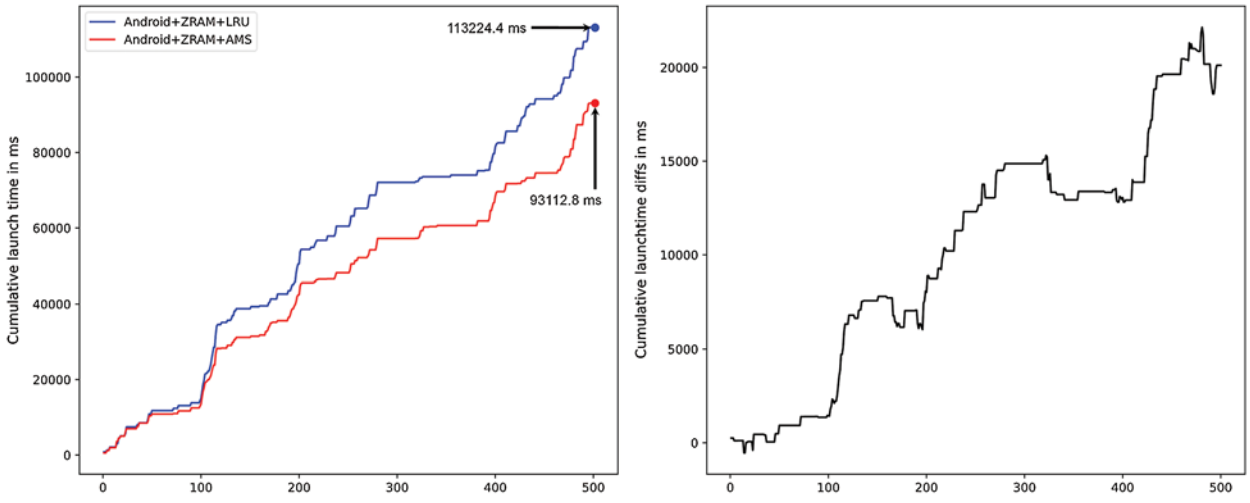

Figure 8: Comparison of cumulative launch time performances

In summary, the proposed system reclaims the memory used by the applications with the lowest launch probability in a low-memory situation, thus increasing the launch speed of more frequently used applications. Consequently, the proposed system utilizes the main memory more efficiently than the existing system.

5 Conclusion

As smartphone users install and use performance-demanding applications, significant amount memory is needed. Techniques such as ZRAM swapping or swapping using storage were developed to accommodate main memory demands; however, these techniques present limitations such as compression and decompression overhead for ZRAM and NAND flash memory wear-out problems for memory swapping using storage, thereby rendering them relatively slower than the main memory. Moreover, the performance of the existing application memory management system using LRU decreases when the number of applications used increases.

In this paper, we proposed a memory management system—*AMMS*—that utilizes the main memory more efficiently. The proposed system predicts application launch probability more accurately by collecting and analyzing the user's application usage information. Frequently used applications reside in the memory for as long as possible, thereby reducing the average launch time of the applications. Performance evaluation using actual smartphone usage records showed that the proposed system increases the number of launches from main memory (hot launches) by approximately 10%, while reducing the average launch time of applications by approximately 17%. This indicates that the proposed system is superior to the existing system at resuming frequently used applications from the main memory. In future, we plan to study an online learning version of the prediction model and a generalized prediction model using application categories to improve prediction performance.

Funding Statement: This work was supported by the National Research Foundation of Korea (NRF) Grant funded by the Korea Government (MSIT) under Grant 2020R1A2C1005265.

Conflicts of Interest: The authors declare that they have no conflicts of interest to report regarding the present study.

References

[1] AppBrain, "Number of android apps on google play," *AppBrain*, 2018. [Online]. Available: https://www.appbrain.com/stats/number-of-android-apps.

[2] E. Price, "The number of apps in Apple's app store declined for the first time last year," *Fortune*, 2018. [Online]. Available: http://fortune.com/2018/04/06/apple-app-store-iphone-apps.

[3] Google, *Understand the Activity Lifecycle*. Mountain View, CA, USA: Google, 2018. [Online]. Available: https://developer.android.com/guide/components/activities/activity-lifecycle.

[4] W. Guo, K. Chen, H. Feng, Y. Wu, R. Zhang *et al.,* "Mobile application relaunching speed-up through flash-aware page swapping," *IEEE Transactions on Computers*, vol. 65, no. 3, pp. 916–928, 2016.

[5] J. Kim and H. Bahn, "Maintaining application context of smartphones by selectively supporting swap and kill," *IEEE Access*, vol. 8, pp. 85140–85153, 2020.

[6] K. Zhong, T. Wang, X. Zhu, L. Long, D. Liu *et al.,* "Building high-performance smartphones via non-volatile memory: The swap approach," in *Proc. EMSOFT*, New Delhi, India, pp. 1–10, 2014.

[7] N. Gupta, "ZRAM: Generic RAM based compressed R/W block devices," *LWN*, 2010. [Online]. Available: https://lwn.net/Articles/3888897N.

[8] S. Jennings, "The ZSWAP compressed swap cache," *LWN*, 2013. [Online]. Available: https://lwn.net/.

[9] P. Jelenkovic and A. Radovanovic, "Least-recently-used caching with dependent requests," *Theoretical Computer Science*, vol. 326, no. 1–3, pp. 293–328, 2004.

[10] S.-H. Kim, J. Jeong and J.-S. Kim, "Application-aware swapping for mobile systems," *ACM Transactions on Embedded Computing Systems*, vol. 16, no. 5s, pp. 182, 2017.

[11] Google, *How People Discover, Use, and Stay Engaged with Apps*. Mountain View, CA, USA:: Google, 2016. [Online]. Available: https://www.thinkwithgoogle.com/_qs/documents/331/how-users-discover-use-apps-google-research.pdf.

[12] T. Yan, D. Chu, D. Ganesan, A. Kansal and J. Liu, "Fast app launching for mobile devices using predictive user context," in *Proc. MobiSys*, Ambleside, UK, pp. 113–126, 2012.

[13] Statcounter, *2019 Mobile Operating System Market Share Worldwide Worldwide*. Dublin, Ireland: Statcounter, 2019. [Online]. Available: https://gs.statcounter.com/os-market-share/mobile/worldwide/2019.

[14] S.-H. Kim, J. Jeong and J.-S. Kim, "SmartLMK: A memory reclamation scheme for improving user-perceived app launch time," *ACM Transactions on Embedded Computing Systems*, vol. 15, no. 3, pp. 47, 2016.

[15] J. Lee, K. Lee, E. Jeong, J. Jo and N. Shroff, "Cas: Context-aware background application scheduling in interactive mobile systems," *IEEE Journal on Selected Areas in Communications*, vol. 35, no. 5, pp. 1013–1029, 2017.

[16] M. Ju, H. Kim, M. Kang and S. Kim, "Efficient memory reclaiming for mitigating sluggish response in mobile devices," in *Proc. ICCE-Berlin*, Berlin, DE, pp. 232–236, 2015.

[17] J. Han, S. Kim, S. Lee, J. Lee and S. Kim, "A hybrid swapping scheme based on per-process reclaim for performance improvement of android smartphones," *IEEE Access*, vol. 6, pp. 56099–56108, 2018.

[18] J. Kim, C. Kim and E. Seo, "ezswap: Enhanced compressed swap scheme for mobile devices," *IEEE Access*, vol. 7, pp. 139678–139691, 2019.

[19] J. Kim and H. Bahn, "Maintaining application context of smartphones by selectively supporting swap and kill," *IEEE Access*, vol. 8, pp. 85140–85153, 2020.

[20] A. Pamboris and P. Pietzuch, "C-RAM: Breaking mobile device memory barriers using the cloud," *IEEE Transactions on Mobile Computing*, vol. 15, no. 11, pp. 2692–2705, 2016.

[21] J. Jeong, H. Kim and J. Lee, "Transparently exploiting device-reserved memory for application performance in mobile systems," *IEEE Transactions on Mobile Computing*, vol. 15, no. 11, pp. 2878–2891, 2016.

[22] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[23] G. K. Durbhaka, B. Selvaraj, M. Mittal, T. Saba, A. Rehman *et al.,* "Swarm-LSTM: Condition monitoring of gearbox fault diagnosis based on hybrid LSTM deep neural network optimized by swarm intelligence algorithms," *Computers, Materials & Continua*, vol. 66, no. 2, pp. 2041–2059, 2021.

[24] D. Zhu, H. Du, Y. Sun, X. Li, R. Qu *et al.,* "Massive files prefetching model based on LSTM neural network with cache transaction strategy," *Computers, Materials & Continua*, vol. 63, no. 2, pp. 979–993, 2020.

[25] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink and J. Schmidhuber, "LSTM: A search space odyssey," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 28, no. 10, pp. 2222–2232, 2017.

[26] S. Lim and D. Lee, "Stable improved softmax using constant normalisation," *Electronics Letters*, vol. 53, no. 23, pp. 1504–1506, 2017.

[27] B. Wang, B. Wang and Q. Xiong, "The comparison of communication methods between user and Kernel space in embedded Linux," in *Proc. ICCP*, Lijiang, CN, pp. 234–237, 2010.

[28] C. Shepard, A. Rahmati, C. Tossell, L. Zhong and P. Kortum, "LiveLab: Measuring wireless networks and smartphone users in the field," *ACM SIGMETRICS Performance Evaluation Review*, vol. 38, no. 3, pp. 15–20, 2011.

[29] J. Lee, S. Nam and S. Park, "Energy-efficient control of mobile processors based on long short-term memory," *IEEE Access*, vol. 7, pp. 80552–80560, 2019.

[30] GSMArena, *Huawei Nexus 6P Full Phone Specifications*. Vilnius, Lithuania: GSMArena, 2015. [Online]. Available: http://www.gsmarena.com/huawei_nexus_6p-7588.php.

[31] Google, *App Startup Time*. Mountain View, CA, USA: Google, 2021. [Online]. Available: https://developer.android. com/topic/performance/vitals/launch-time#internals.