MAPP: Predictive UI View Pre-caching for Improving the Responsiveness of Mobile Apps

Run Wang, Zach Herman, Marco Brocanelli, and Xiaorui Wang
The Ohio State University, Columbus, OH, USA
{*wang.10067, herman.453, brocanelli.1, wang.3596*}*@osu.edu*

***Abstract*—When mobile apps are used extensively in our daily lives, their responsiveness has become an important factor that can negatively impact the user experience. The long response time of a mobile app can be caused by a variety of reasons, including soft hang bugs or prolonged user interface APIs (UI-APIs). While hang bugs have been researched extensively before, our investigation on UI-APIs in today's mobile OS finds that the recursive construction of UI view hierarchy often can be time-consuming, due to the complexity of today's UI views. To accelerate UI processing, such complex views can be pre-processed and cached before the user even visits them. However, pre-caching every view in a mobile app is infeasible due to the incurred overheads on time, energy, and cache space.**

In this paper, we propose MAPP, a framework for Mobile App Predictive Pre-caching. MAPP has two main modules, 1) UI view prediction based on deep learning and 2) UI-API pre-caching, which coordinate to improve the responsiveness of mobile apps. MAPP adopts a per-user and per-app prediction model that is tailored based on the analysis of collected user traces, such as location, time, or the sequence of previously visited views. A dynamic feature ranking and model selection algorithm is designed to judiciously filter out less relevant features for improving the prediction accuracy with less computation overhead. MAPP is evaluated with 61 real-world traces from 18 volunteers over 30 days to show that it can shorten the response time of mobile apps by 59.84% on average with an average cache hit rate of 92.55%.

***Index Terms*—Mobile app, response time, caching, prediction.**

I. Introduction

In the past few decades, mobile apps have significantly changed the ways we communicate, work, shop, study, and entertain ourselves. As of 2023, there are estimated to be 8.93 million apps available worldwide [1], covering almost all aspects of our daily lives. As a result, based on recent statistics, American people spend almost 5 hours per day, on average, on various mobile apps, which, if aggregated, would be more than 70 days per year or 12 years over their lifespan [2]. When mobile apps are used so frequently, their responsiveness has become an important factor that can greatly impact the experience of mobile users. For example, it is recently reported that users would give up their interactions or even delete an app if its response time (between a user click and the expected UI updating) exceeds 2-3 seconds [3], [4]. Even apps that have delays on the order of just 100 milliseconds would receive poor response from the users [5]. Hence, the responsiveness problem not only impacts just end users but also hurts the reputation of an app through negative user reviews [6], [7].

The most straightforward solution to the responsiveness issue is to upgrade to a new mobile device with faster hardware. Unfortunately, such a solution can be costly and also may not be able to completely solve the problem, because many mobile apps have long response times due to software-related issues, called *soft hangs*. A soft hang is commonly defined as an app becomes unresponsive for a limited but perceivable time [8]. In general, a delay in UI refreshing can be perceived by the user if it is longer than 100ms [9]. Soft hangs can be mainly caused by two reasons: 1) hang bugs and 2) prolonged user interface APIs (UI-APIs). A hang bug is some blocking operation (e.g., networking or file reading/writing) executed on the app's main thread, which blocks the main thread from refreshing the UI views promptly. While hang bugs have been researched extensively and can be addressed by source code analysis [8], [10], [11], trace-based diagnosis [12], or runtime detection [13], [14], the other major reason, UI-APIs, has received much less attention, mainly because they are part of the mobile OSs and often overlooked by app developers. However, prolonged UI-APIs may cause more soft hangs than hang bugs. For example, in a recent study that tries to tell real hang bugs from prolonged UI-APIs [14], among the 114 investigated mobile apps that have soft hangs, only 34 are real hang bugs that are not caused by UI-APIs (as shown in Table 5 and discussed in Section 4.2 of [14]). In other words, the remaining 80 apps (i.e., 70.2%) have their soft hangs caused by prolonged UI-APIs.

In this paper, to our best knowledge, we make the first research effort to investigate such prolonged UI-APIs to find 1) what the most time-consuming part is in those UI-APIs, and 2) how to accelerate it. Most of today's mobile OSs (e.g., Android, iOS) create all the UI views for their apps programmatically with a hierarchical view layout file such as XML. For example, the most commonly used view creation function in Android is *inflate*, which involves several steps, including 1) loading an XML file, 2) parsing this XML file, 3) creating UI view objects based on the parsed XML layout and constructing the view hierarchy recursively, 4) rendering the view objects on the display. Among those steps, the third step, i.e., view object creation and hierarchy construction, needs recursive computation to finalize the layout and so takes

This work was supported, in part, by US NSF under grant CNS-2149533.

979-8-3315-4940-4/25/$31.00 ©2025 IEEE

about 60% of the API time (based on our measurements on several mobile devices) and is currently the bottleneck of most prolonged UI-APIs. Such a UI processing methodology is common to most of today's mobile OS, despite their different terminologies. Although a mobile OS often has its own cache system that can speed up the UI processing time the *second time* when the user visits the same view, the poor responsiveness of the first time, as a soft hang, could cause an unpleasant experience and a bad impression to users.

A natural way to accelerate a UI-API, even on its first invocation, is to *pre-cache* the required view hierarchy. When a user enters a UI, they typically spend several seconds to minutes browsing before clicking a UI component (e.g., button, link, drop box) to navigate to the next view. For instance, a news app displays today's breaking news on the main page, and a user may browse before selecting a story to read. Pre-caching can leverage this browsing time by pre-executing the initial steps of the UI-API for likely next views—excluding the final rendering step—and storing the resulting view objects in a cache. When the user eventually clicks, the view can be quickly rendered from the cache, bypassing the time-consuming recursive layout computation and improving responsiveness even for first-time visits.

However, pre-caching all potential next views is impractical due to significant time, energy, and memory overheads. For example, if pre-caching one link takes 200 ms, pre-caching 20 links would require 4 seconds—potentially exceeding the user's browsing time. Moreover, excessive background pre-caching can accelerate battery drain and consume substantial memory, possibly triggering the OS to kill other apps. Thus, only a select subset of views should be pre-cached. To achieve this, we observe that user behavior often follows predictable patterns. For example, a user may check stocks, weather, and traffic in the morning, order lunch during work hours, and chat or stream movies in the evening. By exploiting such patterns, the system can predict and pre-cache only the most likely next views, reducing overhead while significantly improving app responsiveness to avoid undesired soft hangs.

In this paper, we propose MAPP, a framework for Mobile App Predictive Pre-caching. MAPP has two main modules, 1) UI view prediction based on deep learning and 2) UI-API pre-caching, which coordinate to improve the responsiveness of mobile apps. Given that many factors (or features) can affect a user's app usage pattern, such as the user's location, time of the day, or the sequence of views that the user has visited previously in the same app, MAPP adopts a per-user and per-app prediction model that is tailored based on the analysis of collected personal data traces. There are two main challenges in the design of predictive pre-caching: 1) the pattern of each user may depend on different features, 2) the pattern can change over time even for the same user. To address those, we propose a dynamic feature ranking and model selection process that judiciously filters out less relevant features to 1) improve the rate of pre-caching the right views (i.e., cache hit rate), and 2) reduce the computation overheads to achieve the time constraint. Then, if any user pattern changes cause the cache hit rate to drop below a desired threshold, MAPP restarts the ranking process to flexibly adapt to the new user app pattern.

Specifically, this paper makes the following contributions.

- While most existing research focuses on hang bug detection or resource management to improve the responsiveness of mobile apps, to our best knowledge, our work is the first one to investigate prolonged UI-APIs and identify the UI processing bottleneck in today's mobile OSs.
- We present a detailed study with 20 apps to show why UI-APIs can become soft hangs and how pre-caching can help accelerate their processing.
- To avoid the potential high overheads of pre-caching everything, we propose to predictively pre-cache just selected views with the highest probabilities. A dynamic feature ranking algorithm and a model selection process are designed to improve the cache hit rate and reduce prediction overheads.
- We evaluate MAPP with 61 real-world traces from 18 volunteers over a 30-day period to show that our solution can shorten the response time of mobile apps by 59.84% on average with an average cache hit rate of 92.55%.

The remainder of this paper is organized as follows. Section II reviews the related work. Sections III, IV, and V present the detailed designs of MAPP. Section VI evaluates MAPP with real-world traces. Finally, Section VII concludes the paper.

II. Related Work

Improving the responsiveness of mobile apps has been studied extensively before [8], [10]–[18].

Existing research on mobile user behavior prediction mainly predicts which apps will be launched next by the user [19]–[24]. Deep learning-based models have also been adopted for app launch prediction to pre-launch the predicted apps for shorter latency [25]–[28]. Different from those studies, our work is finer-grained and tries to predict which UI view the user is going to visit next within the same app. Lee et al. have proposed PathFinder [29], which predicts the next user click within a mobile app, but user clicks do not always generate new UI views. In contrast, we use a sequence of previously visited views and other factors, e.g., location, time, and sensor readings, to predict and pre-cache the next UI view based on deep learning.

Content prefetching has been proposed to enhance mobile app responsiveness by prefetching and caching selected data in advance [4], [30]–[33]. However, existing studies on prefetching mainly focus on downloading networking data ahead of time, while our work is the first one to investigate prolonged UI-APIs and pre-cache the time-consuming UI view computation. Our work can be integrated with these content prefetching techniques to provide a comprehensive solution for improving app responsiveness.

A recent study called Floo [34] proposes to automatically memorize (cache) app computations for faster app responsiveness. Our work is different because it focuses on UI-APIs, which are part of a mobile OS, while Floo tries to

cache the computations of an app itself. More importantly, we *pre-cache* the predicted UI views, while Floo is post-caching (memorization), which leads to compulsory delay the first time computation is executed.

III. Overview of MAPP

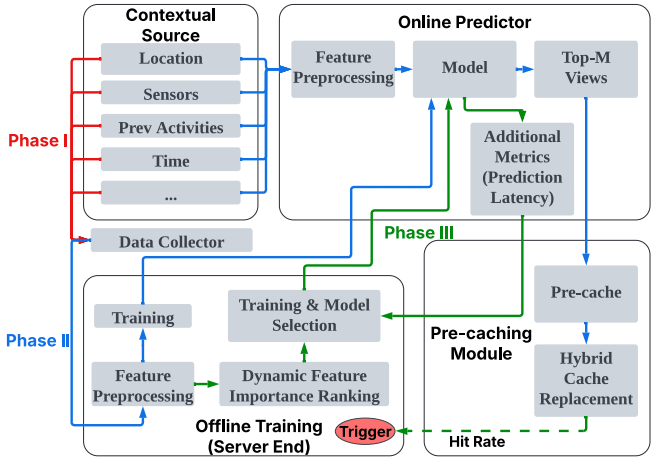


Fig. 1. High-level system architecture of MAPP, which includes three phases. Phase I (red links) collects user data. Phase II (blue links) trains an initial DNN model based on the collected data. Then, Phase II predicts which views to be visited next based on the trained model and pre-caches those views. Phase III (green links) performs dynamic feature importance ranking and model selection to optimize system performance under time constraint.

MAPP enhances the responsiveness of mobile apps to user actions by 1) *predicting* which UI views will be visited next based on a per-app and per-user Deep Neural Network (DNN) model, and 2) *pre-caching* the UI-APIs of the predicted views, which allows to considerably reduce the user-perceived latency upon cache hits, so that soft hangs can be avoided.

Figure 1 shows the high-level system architecture of MAPP, which operates in three phases. During Phase I, MAPP uses a data collector on the mobile device to gather user data, including location, a sequence of previously visited views in each app, time of the day, battery state of charge, and different sensor readings. After enough data has been collected (e.g., a few days), MAPP enters Phase II to send the collected data to a cloud server for training an initial DNN prediction model. The model is then uploaded to the mobile device for inference. Then, based on the data collected in real-time, MAPP starts predicting the *Top-M* views that are most likely to be visited next. This prediction is then used to pre-cache the related UI-APIs to reduce the UI response time. Phase III starts when a sufficiently large amount of data has been accumulated. In this phase, MAPP optimizes (on the server) the initial DNN model to reduce overhead. In particular, it 1) ranks features based on their impacts on the model accuracy, 2) trains various models wwith different feature combinations, and 3) evaluates their inference time to select the best model to redeploy on the mobile device. At this point, MAPP keeps monitoring the cache hit rate and triggers Phase III again if the rate falls below a certain threshold, which allows the DNN model to be re-trained based on the latest data collected.

In the following, we introduce the design and implementation of the prediction and pre-caching modules, respectively.

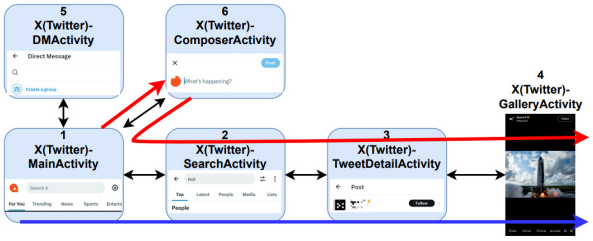


Fig. 2. An example sequence of visited UI views in the app Twitter.

IV. Design of MAPP's View Predictor

We first introduce what user data is collected for model training. We then discuss how to optimize the training prediction model to reduce overheads.

A. User Data Collection and Model Training

MAPP utilizes two types of input features. The first type is the sequential view controller usage history, which are consistently included throughout the training period. The second type is a group of contextual features, which can be personalized. They are always included during the initial deployment's training phase. However, in subsequent deployments, these personalized features are selectively incorporated based on feature importance rankings and model selection.

Sequence of Previously Visited UI Views. MAPP employs sequential view browsing history as a core component, capturing the trace of previous user actions to understand and predict future user visits. In Android, the activity class is usually used as a view controller. An illustrative example is shown in Figure 2, where activities from the app Twitter are represented as nodes in a tree structure. By following the blue path, the sequence of views (i.e., activities) is [1, 2, 3, 4]. Alternatively, if we follow the red path, the sequence becomes [1, 6, 1, 2, 3, 4]. However, since the user returns to activity 1, the sequence is effectively restarted from the second occurrence of 1, resulting in the sequence [1, 2, 3, 4] once again.

Context Features. Following the sequential features, MAPP incorporates a set of features that are useful in specific scenarios. This includes sensor data from the accelerometer, gyroscope, and magnetometer, which serve as key numerical inputs in the model's training dataset. These sensors provide valuable insights into user movements, offering essential information for predicting the next action for users of sport-related and navigation-related apps.

Location may also be important to correlate the user actions with the environment. For example, a user interacting with Google Pay in a shopping area is more likely to trigger payment-related views for making purchases. In contrast, when the user is at other locations, such as at home or at work, these payment views are less likely to be visited.

Other features included in the data collection are time of day, battery level, signal strength, and duration of use. The context provided by the time feature may be important because user behavior patterns often vary significantly at different times. For instance, in a stock trading app, the user behavior is heavily influenced by the stock market's opening and closing

times. During market hours, users are more likely to engage in buying and selling views. Outside of these hours, users are more likely to be reviewing charts or analyzing details rather than initiating buy or sell actions. The battery level can also be important because the user may prefer using app functions that require less hardware resource utilization when battery levels are low, thereby affecting app usage predictions. Similarly, signal strength can impact the usability of network-dependent functionalities of an app. The duration of previous app usage sessions offers insights into user engagement levels, which affect the likelihood of transitioning to other consequent views.

All the above features are collected by MAPP directly on the user device until the dataset is large enough to initiate model training in Phase II. However, it is important to note that only a small set of features is collected at inference time to reduce runtime overheads.

Model Training. The model architecture includes distinct components for processing various types of input data. Sequential data is one-hot encoded, a method where categorical values are represented as binary vectors, allowing neural networks to interpret discrete inputs effectively. These encoded sequences are then processed using a Gated Recurrent Unit (GRU) [35]. Compared to Long Short-Term Memory (LSTM) networks, GRUs perform better on low-complexity sequences [36] and train faster [37]. Other contextual information, which can be selectively excluded in our system, is handled by a separate dense network layer employing a rectified linear unit (ReLU) as an activation function to capture complex patterns in the data. A dropout layer is included to prevent over-fitting. Additionally, the dataset is imbalanced, meaning some target classes appear more frequently than others. To address this, we apply class weighting, which assigns higher weights to underrepresented classes and lower weights to overrepresented ones. This helps the model learn meaningful patterns from all classes rather than being biased toward the majority classes.

B. Prediction Model Optimization

MAPP first analyzes the importance of features and then balances the trade-off between model performance and estimated end-to-end prediction latency to select an optimized model for use at runtime.

Dynamic Feature Importance Ranking. Empirical observations indicate that different users exhibit varying usage behavior patterns even within the same mobile app. Consequently, indiscriminately utilizing all contextual features for every user is not only redundant but also wastes computational resources and degrades prediction performance. To dynamically assess the importance of these features, we use a modified feature permutation importance algorithm [38]. Instead of permuting features individually, we analyze and group highly correlated features using the pairwise correlation matrix after collecting the data and sending it to the server. This collective permutation during the feature exclusion process ensures that the influence of correlated features is accurately assessed. We quantify each feature group's impact by comparing the baseline accuracy, Acc_{full}, obtained with the full feature set F_{full}, to the accuracy after excluding each group, $\text{Acc}_{\text{permute}}(g_i)$:

$$\text{Rank}(g_i) = \text{Acc}_{\text{full}}(F_{\text{full}}) - \text{Acc}_{\text{permute}}(g_i)$$

In our model, F_{full} represents the complete set of features used in Phase II to establish baseline performance. During the optimization phase, we assess the impact of excluding each feature group on the model's performance by permuting them one at a time. Groups whose removal results in less accuracy degradation—indicating lower importance—are identified as candidate groups g_{c_k}. These groups are then considered for exclusion in the final optimized feature set, which aims to reduce computational demand and enhance prediction accuracy.

Feature groups are categorized into:

- Proposed candidate groups $\{g_{c_k}\}$, identified for their minimal impact on model accuracy and considered for exclusion.
- Retained feature groups G_{r}, which are essential for maintaining model performance and thus are preserved in the optimized model.

The retraining feature set is refined by retaining crucial groups G_{r} and testing all combinations with subsets of the proposed candidate groups g_{c_k}:

$$\text{Combinations tested} = \{G_{\text{r}} \cup s : s \subseteq \{g_{c_k}\}\}$$

where s represents any subset of $\{g_{c_k}\}$, including the empty set. By selecting the top-2 less important feature groups, you would only need to train 2^2 models instead of 2^n, drastically lowering the number of combinations to test and optimize model performance efficiently.

Model Selection. After training a new set of models based on the results from the previous feature importance ranking, we proceed to evaluate their performance using test data collected during Phase II. MAPP then applies an objective function to select the final model, systematically integrating multiple performance metrics. This objective function takes into account both the F-measure and end-to-end inference latency to provide a balanced evaluation of each model's configuration. The objective function is defined as follows:

$$\text{Score}(M) = F\_\text{measure}(M) - \lambda \times \text{Time}(M)$$

where:

- $F\_\text{measure}(M)$ is the F-measure of the model M, computed as the harmonic mean of precision and recall for each class:
$$F\_\text{measure}(M) = \frac{1}{C}\sum_{i=1}^{C} 2 \times \frac{\text{Precision}_i \times \text{Recall}_i}{\text{Precision}_i + \text{Recall}_i}$$
- $\text{Time}(M)$ represents the end-to-end prediction latency associated with model M,
- λ are weighting factor that balance the trade-offs between end-to-end prediction latency and accuracy.

Precision and Recall for each class i are defined as:

$$\text{Precision}_i = \frac{TP_i}{TP_i + FP_i}, \quad \text{Recall}_i = \frac{TP_i}{TP_i + FN_i}$$

where TP_i, FP_i, and FN_i represent the true positives, false positives, and false negatives for class i, respectively. Models are ranked based on their scores, and the model with the highest score is selected for deployment.

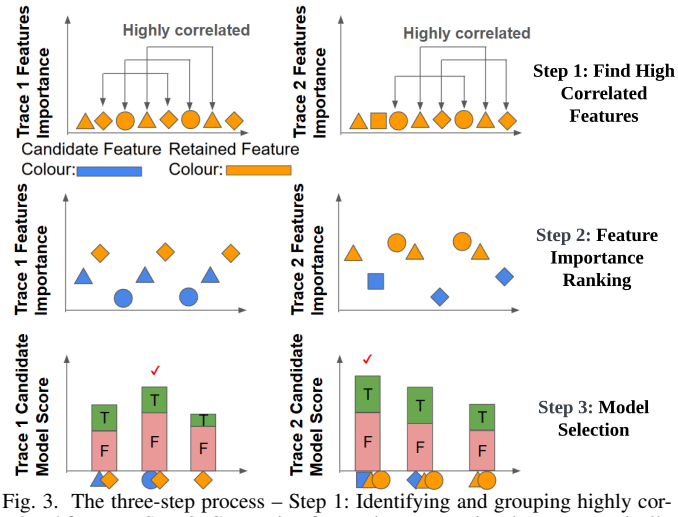


Fig. 3. The three-step process – Step 1: Identifying and grouping highly correlated features. Step 2: Computing feature importance involves systematically excluding each feature group one by one, with the top two groups having the lowest ranking scores identified as candidates. Step 3: Model selection, these candidate groups are evaluated based on model performance and inference overhead, leading to the selection of the optimal model for deployment. "T" represents the score reduction due to end-to-end prediction latency, and "F" represents F measure of the model.

For example, as shown in Figure 3, the first step is to group highly correlated features, where the same shape represents features belonging to the same group. After ranking feature importance, we identify two groups of less important features for each trace, which are represented by shapes colored in blue. Consequently, we train separate models for each group based on these findings. We evaluate the accuracy of each model using data collected while also considering end-to-end prediction latency, which can be profiled using pre-configured models. After analyzing the end-to-end prediction latency among the models, the predictor transmits these results to the server for model selection. Ultimately, the model with the highest overall score is selected. Generally, models using more features experience a larger deduction in overall scores.

V. Design and Implementation of Pre-Caching

We now introduce the design of the view cache used for pre-caching and its cache replacement rules. We then discuss how to implement pre-caching in Android as an example.

A. View Cache and Hybrid Cache Replacement

In Phase II of the overall architecture in Section III, when the user visits a new UI view in an app, the prediction module identifies *Top-M* UI views with the highest probabilities. Then, the pre-caching module starts processing the corresponding UI-APIs for those M predicted views. We pre-cache *M* views mainly to reduce the caching overheads, in terms of time, energy consumption, and memory space. Once the views are processed, they are placed into a customized view cache, ready for rendering onto the display. If the user indeed visits one of the pre-cached views later, the processed view will be loaded from the cache for rendering, which can bypass the time-consuming view object creation and so significantly reduce the user-perceived latency.

The size of the view cache is limited, so it is designed to be managed with the Least Recently Used (LRU) policy. Specifically, MAPP features a hybrid cache replacement rule that can be triggered by either 1) *pre-caching* with *Top-M* predicted views, or 2) *post-caching* with the actually visited view if it is not already one of the predicted *M* views. The post-caching rule is used to handle any unexpected user pattern variations. In particular, if MAPP detects that the cache hit rate stays below some threshold, it assumes that the user pattern has significantly changed. Thus, it stops pre-caching and performs only the LRU-based post-caching to promptly handle such sudden changes, before the prediction model can be re-trained and optimized on the cloud server.

The implementation of pre-caching in a mobile OS should be transparent to app developers. In other words, app developers should not need to do anything different for their apps to have MAPP shorten their UI response times. To this end, pre-caching can be implemented in two ways. The first way is to integrate MAPP as part of the mobile OS. For example, the Android framework currently has a low-level cache for holding XML blocks, to aid in the process of parsing XML files. The view cache in MAPP could be implemented in a similar manner for the best efficiency. The pre-caching module can be implemented as an OS service because all the user interactions are handled by the OS. In this way, all the UI-APIs can be easily modified to load the cached view first. The second way is to implement MAPP as a middleware service to run as separate threads in the background that monitor the user activities. It is more challenging to provide transparency for app developers in this way because the current UI-APIs are not designed to load cached views. However, it is still doable by modifying the parent classes that every view class has to inherit from, or by developing a tool that can automatically replace the related operations in UI-APIs in the app bytecode with the modified ones that load cached views instead. In the following, we use Android as an example OS to discuss how to prototype MAPP.

B. Implementation of Pre-Caching in Android

Most of today's apps in Android use the *inflate* UI-API to create UI views because it has been the primary method for view creation for over a decade. So, we use it to show how MAPP can be implemented in Android in the following, but our solution can be adapted for other view creation processes such as Jetpack Compose.

To implement MAPP's pre-caching module, we have created a background service app with a pre-caching class and a ViewCache class. When a UI view prediction is made, this service app executes the view creation process while the CPU is idle, in order to have it ready for rendering when the view is requested later. The ViewCache class has a Map data structure and four methods. A Map is essentially a set of objects, with a second data type used as a descriptor, typically referred to as a *key*. The Map in the ViewCache holds view objects with their keys being strings holding the class path for the view (i.e., activity), to which the view object belongs. The

ViewCache methods are to provide ways to add to the cache, take from the cache, check if the cache is empty, and check if the cache contains a specific view already. Below is the simplified pseudo-code that inflates and caches a target view.

```
function PRE-CACHE
    inflater ← getLayoutInflater()
    view ← inflater.inflate(R.layout.predicted_layout, null)
    ViewCache.add(view)
end function
```

As discussed before, for an app to utilize the ViewCache created by MAPP, the *onCreate* function of the app's activity class needs to be replaced with one that loads from the cache first. This can be done either by modifying the *onCreate* function of the parent class (which is part of the Android source code), or by replacing the function in the app's bytecode. After the modification or replacement, when an activity is initiated, the system first checks if the corresponding UI view is already cached. If so, and if the predicted activity class matches the current activity, the system bypasses the time-consuming view inflation process, directly setting the activity content to the pre-cached view. Below is the simplified pseudo-code that checks the ViewCache and renders the view if it is in the cache.

```
function ONCREATE
    view ← ViewCache.get(localClassName)
    if view ≠ null then
        setContentView(view)
    else
        inflater ← getLayoutInflater()
        view ← inflater.inflate(R.layout.layout_dummy, null)
        setContentView(view)
    end if
end function
```

There can be some other types of view in Android. For example, some Android apps have fragment view binding, which is a view object that has a separate reference for each view object within its hierarchy, allowing for direct access to a view's elements. The creation of view bindings is the same as views, with just one extra step to generate a Java class for the binding, which is automatically done by Android as view binding is enabled. The process for pre-caching view bindings is almost the same as for typical activity views because it just requires any references to the view object to be changed to the specific type of view binding object for that activity.

To demonstrate the feasibility of implementing MAPP in real Android apps, we have successfully applied the above method to 34 apps to improve their UI response time. Section VI-C presents the related results and discussion.

VI. Evaluation

In this section, we evaluate the performance of our MAPP framework. First, we present the experimental setup, then we evaluate the performance of MAPP's view/activity prediction, and finally, we examine the response time reduction achieved in several real-world apps through pre-caching.

A. Experimental Setup

Mobile Phones. For our pre-caching experiments, we use a Google Pixel 8 and a Samsung Galaxy A11 to test devices with different hardware configurations.

TABLE I
EXAMPLES OF APP USAGE DATA

Activity Name	Time	Location	...	Battery Level	Signal Strength	Duration (ms)
Main	161553	39, -82	...	88	1	1133
Search	161600	39, -82	...	88	1	7161
Profile	161618	39, -82	...	87	1	17821
Gallery	161622	39, -82	...	87	1	3814

Interaction Traces. We have collected over 61 usage traces from 18 different volunteers to evaluate the performance of the proposed MAPP framework. Table I presents a truncated example of raw user activity records which include the sequence of activities performed by the user (Activity), the timestamp of each recorded activity, the geographical coordinates representing the user's approximate location, the device's battery level at the time of the activity, the signal strength of the connected network, and the duration of the previous activity. To better understand the dataset and the complexity of user interactions, we analyze metrics such as maximum activity depth and maximum degree distribution. For example, consider a user's interaction pattern with an app like Twitter, as depicted in Figure 4. Here, the maximum activity depth is 4, corresponding to the longest sequence path [1, 2, 3, 4] highlighted by a blue double arrow in the figure. The highest degree, indicated by a red dashed line, is observed in the main activity (Activity 1), which connects to three child activities (Activities 2, 5, and 6).

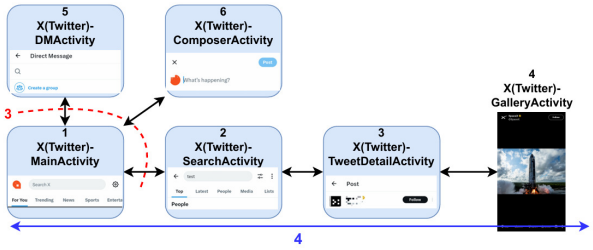


Fig. 4. Activity structure of a sample application with a maximum activity depth of 4 and maximum degree of 3.

Data Privacy. To protect user privacy, all collected data is anonymized, and any identity-related information is securely hashed. All experimental data is stored locally on our lab systems and is accessible only for research purposes. We acknowledge that data privacy remains an important aspect for future work. Various existing approaches, such as federated learning-based methods [39], could be applied to further enhance data protection and prevent data leakage.

Parameter Settings. Our analysis result shows that approximately 50% of traces have a maximum depth of four or greater. This suggests that users frequently engage in complex sequences of actions that span multiple levels within an app. Such depth indicates that user interactions can often be branching instead of being just linear, which means simple caching strategies may not work well because they typically manage only the most immediate data needs. In scenarios where user interactions create long, multi-step sequences without

repetition, simple caching may also fail to pre-cache relevant UI APIs effectively, leading to increased load times and a degraded user experience. Hence, we have set the length of the sequential activity features for training models to 4.

We have also analyzed the distribution of the maximum degree across all traces. For certain apps, the main activity's degree can exceed 20, indicating high connectivity among nodes. This suggests that a greedy method, such as caching all child activities connected to the current activity, is inefficient, particularly on mobile devices with limited memory. Notably, the median maximum degree among traces is 6. Therefore, in MAPP's pre-caching module, we have set the cache size to store view objects for only six activities.

Finally, we determine the number of top-M activities by considering both the MAPP processing latency and the minimum user interaction interval. Our data reveals a minimum user interaction interval of 1.12 seconds, establishing a hard constraint: the total latency—from receiving prediction results to completing precaching—must not exceed this threshold. On our test devices, the precaching process alone requires 353 milliseconds per view on average. For maximum prediction accuracy, this latency budget allows MAPP to precache 3 views ($3 \times 353\,\text{ms} \approx 1{,}059\,\text{ms}$) within the 1.12-second window. Thus, we select top-3 predictions and pre-cache these 3 views for our tested environment. Additional time is required for the prediction process itself, which can be further controlled and optimized by adjusting the parameter (λ) during model selection. This ensures that the overall latency remains under the user interaction interval constraint.

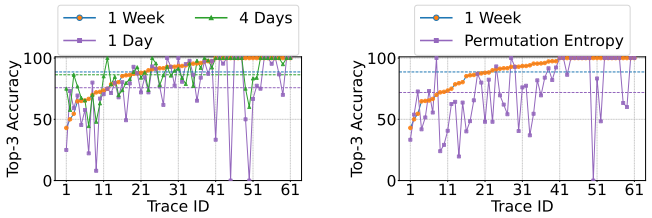
B. Performance of UI View/Activity Prediction

We assess the performance of MAPP's view prediction by testing heuristics for phase transitioning, evaluating the model optimization, and comparing MAPP with various baselines.

Phase Transitioning. The proposed MAPP framework transitions through three phases at runtime: Phase I—data collection, Phase II—initial training, and Phase III—optimization. First, we analyze the trade-off between improved responsiveness and the volume of user data collected. By comparing performance across one-week, four-day, and one-day data periods, we observe that certain apps achieve higher prediction accuracy with less data. This occurs because their usage patterns (e.g., weather forecast apps) exhibit minimal correlation with specific days of the week. However, the overall prediction accuracy using one week (7 days) of data outperforms shorter periods, as shown in Figure 5(a).

Next, we evaluate two heuristics for phase transitions from data collection to training: time-based and sequence-based. For the first method, one week represents the best cycle period for typical users, based on previous results. For the second method, we use Permutation Entropy [40] to focus on fluctuations of the collected view sequence, which means that each phase ends only when the entropy of the collected sequence has stabilized.

Figure 5(b) shows the top-3 prediction accuracy of MAPP using these two methods. While using quantified sequential



(a) (b)

Fig. 5. Effect of phase transitioning. (a) Comparison of Top-3 prediction accuracy comparison: one-week data (88.7%), four-day data (86.35%), and one-day data (75.80%). (b) Comparison of Top-3 prediction accuracy using one-week data collection (88.7%) versus using permutation entropy as an indicator (71.89%).

feature complexity as the indicator for ending data collection can yield satisfactory results on certain traces, particularly in applications not heavily influenced by contextual information, a fixed one-week data collection period generally performs better. This indicates that contextual information plays a crucial role in prediction accuracy. As a result, for all the remaining tests we use the time-based method to transition across phases with a one-week period for data collection.

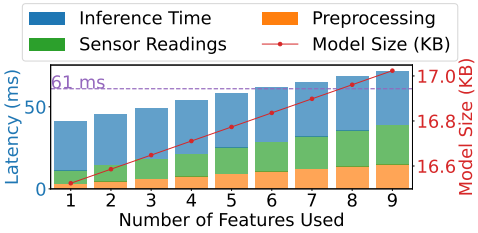
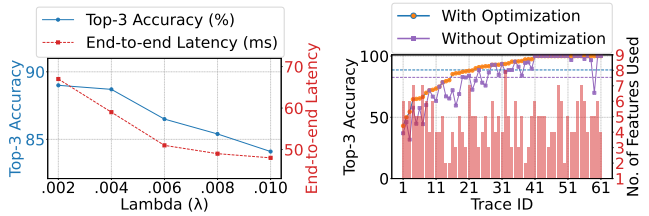


Fig. 6. End-to-end latency of the Predictor (including sensor reading, preprocessing, and inference time) as a function of the number of features used.

End-to-end MAPP Prediction Latency. MAPP first benchmarks the device's performance before proceeding with model selection. To achieve this, pre-configured models with different feature sets are tested to obtain performance metrics. Figure 6 illustrates the average end-to-end prediction latency on our devices using various model sizes, each differing in the number of features used. Additionally, we analyze the latency of each prediction step, including sensor reading, preprocessing, and inference time. Sensor reading and preprocessing time can be significantly reduced by using fewer features, the average sensor reading from 24 to 7 ms, and average preprocessing from 15 to 3 ms. We observe that end-to-end latency ranges from 41.5 ms when 1 feature is used to 74 ms when all 9 features are used. Meanwhile, reducing the number of features decreases the model size from 17 KB to 16.6 KB, providing a minor benefit in terms of bandwidth and storage.

Optimization vs Prediction Accuracy. On our test devices, we aim to precache up to three views, each taking an average of 353 ms. To ensure all three tasks complete within the total latency budget of 1.12 s (i.e., 1120 ms), the time allocated for end-to-end prediction must not exceed $1120 - 3 \times 353 = 61\,\text{ms}$. Accordingly, we test different λ values on the server side to determine the number of features used by the final model, allowing us to estimate the prediction latency and ensure it meets the time constraint. As shown in Figure 7(a), increasing λ reduces the average end-to-end prediction latency but also

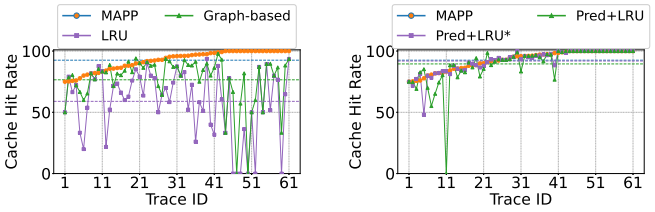


(a) (b)

Fig. 7. Effect of Optimization. (a) Optimization results using different lambda. (b) The feature ranking and model selection algorithm of MAPP improves the top-3 prediction accuracy from 82.61% to 88.7%, in addition to reducing the computation overheads.

lowers the model's top-3 accuracy. At $\lambda = 0.004$, the models on average use approximately five features. Based on performance results from our pre-configured models, the estimated prediction latency remains below 60 ms, satisfying the latency requirement while maximizing accuracy.

Finally, we evaluate the effect of MAPP's Phase III model optimization on prediction accuracy. After identifying the three least important feature groups, we test all candidate models and select the one that achieves the highest model score. As shown in Figure 7(b), we compare MAPP's top-3 accuracy with and without optimization, as well as the number of features used by the optimized model. The results indicate that the proposed dynamic feature importance ranking and model selection algorithm efficiently excludes less important features. On average, fewer than 5 features are selected in each case, thereby improving prediction accuracy and reducing latency.



(a) (b)

Fig. 8. Cache hit rate of our method MAPP against baselines. (a) The significantly higher cache hit rate performance of our method highlights the advantage of our predictive pre-caching strategy. (b) The proposed feature selection and hybrid cache logic also lead to performance improvements.

Cache Hit Rate. We use the collected traces to evaluate the cache hit rate of MAPP against two caching strategies: LRU and a graph-based method inspired by Floo. LRU is a traditional post-caching strategy that stores previously visited views without any optimization. The graph-based method, following the strategy of Floo [34], constructs a graph of activities based on usage patterns and performs pre-caching by memorizing the most frequently accessed views, without considering contextual information. For a fair comparison, the pre-cache module is cleared from the main memory of the mobile devices before each evaluation to ensure a cold cache.

Figure 8(a) shows that MAPP achieves a 92.55% cache hit rate on average across traces, significantly outperforming LRU and the graph-based method, which result in 58.95% and 76.55% average hit rates, respectively. Unlike LRU, which relies solely on recency, and the graph-based method, which pre-caches based on frequency without contextual understanding, MAPP leverages predictive pre-caching, hybrid cache replacement, and model optimization. These enhancements enable MAPP to achieve approximately 34% and 16% higher cache hit rates than LRU and the graph-based method, respectively.

We now evaluate the contribution of each component used in MAPP. Figure 8(b) illustrates the performance impact of selectively enabling different components of the MAPP system. Pred+LRU* utilizes the MAPP predictor along with its hybrid cache replacement policy but excludes feature optimization. Pred+LRU employs only the predictor, without hybrid cache replacement or feature optimization. On average, MAPP has a 92.55% cache hit rate, compared to the 91.87% of Pred+LRU* and the 89.51% of Pred+LRU. While this improvement may seem small, it is worth noticing that MAPP consistently shows a high cache hit rate across the traces, while the two baselines show higher variability in performance, despite their occasional better hit rates than MAPP (e.g., Trace 4). For example, for Trace 11, MAPP, and Pred+LRU* achieve a similar 83.34% cache hit rate, while Pred+LRU has a 0% hit rate. This highlights the importance of MAPP's hybrid cache replacement rule. As another example, in Trace 5, MAPP achieves a 77.78% hit rate, while Pred+LRU* has a lower 48.15% hit rate, which highlights the importance of feature optimization to improve the hit rate.

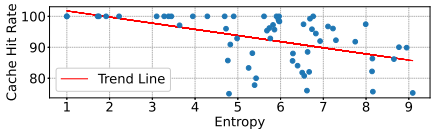


Fig. 9. The trend of cache hit rate versus Permutation Entropy.

Detailed Analysis of Cache Hit Rate. To understand factors affecting MAPP's performance, we analyze the correlation between permutation entropy and cache hit rates. We compute permutation entropy [40] using subsequences of length 8 with an interval of 1, applying a variant that directly handles categorical values. As shown in Figure 9, the red regression line suggests that traces with higher entropy generally exhibit lower cache hit rates, highlighting the difficulty of maintaining cache efficiency under complex interaction patterns. For example, traces with entropy between 1 and 4 achieve near 100% hit rates, whereas those with entropy between 5 and 9 show more variability (75%–100%). Although these hit rates are still relatively high, the results indicate the need for research on how to distill useful information from sequential interactions and further improve the prediction.

C. Responsiveness Improvement by Pre-Caching

Latency Reduction. Here we examine the latency of 24 real open-source mobile apps with and without MAPP's pre-caching module using two devices, the Google Pixel 8 and the Samsung Galaxy A11. Table II presents the results; due to space constraints, detailed analysis is shown for only 14 applications. The *before* and *after* columns refer to the latency of the *onCreate* method (which is the starting point of any activity in Android) before and after using MAPP. Across the 14 selected apps, the Google Pixel 8 and Samsung Galaxy A11 show average latency reductions of 59.57% and 64.53%,

TABLE II
THE UI RESPONSE TIMES OF 20 APPS BEFORE AND AFTER PRE-CACHING ON TWO MOBILE DEVICES.

App Name	Downloads	Google Pixel 8			Samsung Galaxy A11		
		Before (µs)	After (µs)	Improvement	Before (µs)	After (µs)	Improvement
Wikipedia	50M+	46132	30676	33.50%	587626	314932	46.41%
WifiAnalyzer	10M+	64282	10436	83.77%	340997	30045	91.19%
Loop Habit Tracker	5M+	32035	11114	65.31%	228569	92896	59.36%
Infinity for Reddit	500k+	70087	50472	27.99%	197309	148984	24.49%
Noice	100k+	180650	17236	90.46%	1263607	95527	92.44%
NextPlayer	50k+	121806	65670	46.09%	1580223	299990	81.02%
Privacy Friendly Notes	10k+	16151	9203	43.02%	212321	102626	51.66%
Transportr	10k+	91730	50114	45.37%	180113	91730	49.07%
Fossify Music Player	5k+	28291	7005	75.24%	326431	98849	69.72%
SimpMusic	1740*	1547130	127463	91.76%	3897362	638771	83.61%
Fossify Notes	1k+	49646	10704	78.44%	718165	88871	87.63%
AudioAnchor	217*	47667	29974	37.12%	241972	187205	22.63%
Stealth for Reddit	110*	28502	7960	72.07%	244746	19966	91.84%
Brainf	100+	45027	25277	43.86%	577983	274984	52.42%
Average		169224	32379	59.57%	756959	177527	64.53%

** In addition to these 14 example apps, we have tested **10** more apps (downloads), including AntennaPod (1M+), Cloudstream (7k+*), Kotatsu (4.5k+*), Doodle (100k+), Omweather (332*), Markor (100k+), NewPipe (31k+*), Odyssey (235*), PocketPlan (1k+), and LibreTube (8.9k+*). The average improvement for all **24** apps is **56.54%** on Google Pixel 8 and **63.13%** on Samsung Galaxy A11, with an overall average improvement of 59.84%.*

respectively. Notably, on the Samsung Galaxy A11, all apps initially exceed the 100ms human-perceivable delay [9], but MAPP's pre-caching brings 7 of them below that threshold.

Detailed Latency Study. As Table II shows, some apps benefit more from pre-caching than others. The aspects of an application that will yield beneficial results are deep view hierarchies or complex view objects. NextPlayer for example, a media player application, has the third highest latency (∼122 ms on Pixel 8, 1,580 ms on the Galaxy A11) in the table. This high latency is due to the inclusion of a complex view object for displaying video media. SimpMusic, a music player app, has the highest latency overall (∼1,547 ms on Pixel 8, 3,900 ms on Galaxy A11) due to its extremely deep view hierarchy in the settings activity. The XML file for the corresponding view is nearly 1500 lines long. Comparatively, the XML layout of Noice is under 400 lines, which is still long compared to the other applications in the table. Performance of pre-caching is also affected by how developer implements their code. Brainf, a learning app, for the Samsung Galaxy A11, it starts with an execution time of 578 ms, which gets reduced to 275 ms after pre-caching. The residual latency stems from non-layout-related operations in onCreate(), such as initializing a Markdown interpreter, which pre-caching cannot optimize.

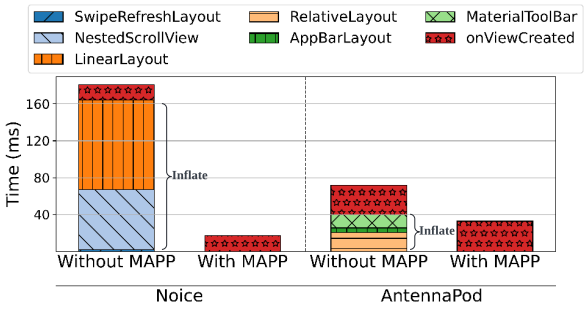


Fig. 10. Execution time breakdown for the onCreate method in two different cases. The first case is for the account fragment in the Noice application. The second is for the search fragment in the AntennaPod application.

To gain insights into how much time view creation within the UI-APIs contributes to latency, we have analyzed the breakdown of the onCreate methods across various apps. Figure 10 shows the latency characteristics of two representative apps, Noice and AntennaPod, on Pixel 8. Specifically, the figure breaks down the latency, which includes the creation of six different views using the *inflate* UI-API (SwipeRefreshLayout, NestedScrollView, LinearLayout, RelativeLayout, AppBarLayout, and MaterialToolBar). Additional setup methods are involved, which are listed as "onViewCreated" in the figure, which is the name of the method where these are usually found in the case of view fragments. Noice contains a small series of views that each hold several smaller view objects such as buttons or text-boxes, and their repetitive creation takes up the majority of the total latency. With MAPP's pre-caching, this latency, along with the other view objects, is reduced below 20ms. AntennaPod, on the other hand, does not have such complex view objects and shows a lower latency of ∼79ms. In this case, pre-caching is able to drop ∼50ms from the total latency.

D. Overall Improvement by MAPP

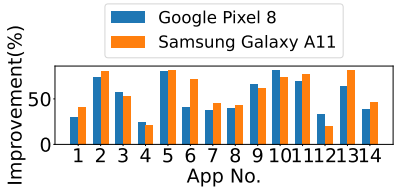


Fig. 11. Overall responsiveness improvement observed after integrating the pre-caching module and online prediction module with 14 open-source apps.

We evaluate the overall improvement of MAPP by integrating it with 14 open-source apps listed in Table II. Since each interaction with MAPP can either successfully predict and pre-cache or fail, we calculate the overall response time as $(1 - c) \times$ response time before pre-caching $+ c \times$

response time after pre-caching, where c represents the cache hit rate. As illustrated in Figure 11, the average overall improvements are 53% for the Google Pixel 8 and 57% for the Samsung Galaxy A11. These results suggest that MAPP can be more beneficial for less powerful devices.

E. MAPP Overhead

We systematically evaluate the overhead of MAPP in terms of power consumption and memory utilization. Power measurements are conducted using AccuBattery [41], a validated tool for fine-grained energy profiling in mobile systems [25]. On average, MAPP consumes 34 mW, increasing total device power consumption by 3.27%. Additionally, MAPP raises memory utilization by an average of 2.64% (less than 100 MB) on test devices.

VII. Conclusion

In this paper, we have presented MAPP, a framework for Mobile App Predictive Pre-caching. MAPP has two main modules, UI view predictor based on deep learning and UI-API pre-caching, which coordinate to improve the responsiveness of mobile apps. MAPP adopts a per-user and per-app prediction model that is tailored based on the analysis of collected user traces, such as location, time, or the sequence of previously visited views. A dynamic feature ranking algorithm is designed to judiciously filter out less relevant features to improve the prediction accuracy with less computation overhead. MAPP is evaluated with 61 real-world traces from 18 volunteers over a 30-day period to show that it can shorten the response time of mobile apps by 59.84% on average with an average cache hit rate of 92.55%.

References

[1] Ishan Gupta. Mobile app industry statistics 2023: Trends and insights you shouldn't ignore. https://ripenapps.com/blog/mobile-app-industry-statistics/, 2023.

[2] Harmony Healthcare IT. Black mirror or black hole? american phone screen time statistics. https://www.harmonyhit.com/phone-screen-time-statistics/, 2024.

[3] Dimensional Research. Failing to meet mobile app user expectations: A mobile app user survey. https://techbeacon.com/sites/default/files/gated_asset/mobile-app-user-survey-failing-meet-user-expectations.pdf, 2015.

[4] Murali Ramanujam, Harsha V. Madhyastha, and Ravi Netravali. Marauder: synergized caching and prefetching for low-risk mobile app acceleration. In *MobiSys*, 2021.

[5] Android Developers. Keeping your app responsive. https://developer.android.com/training/articles/perfanr#Reinforcing, 2020.

[6] Apigee. Users Reveal Top Frustrations That Lead to Bad Mobile App Reviews. http://apigee.com/about/press-release, 2014.

[7] X. Ma et al. eDoctor: automatically diagnosing abnormal battery drain issues on smartphones. In *Proceedings of 10th USENIX Symposium on Networked Systems Design and Implementation*, 2013.

[8] Xi Wang, Zhenyu Guo, Xuezheng Liu, and Zhilei Xu. Hang analysis: Fighting responsiveness bugs. In *Eurosys*, 2008.

[9] Brad Fitzpatrick. Writing zippy android apps. In *Google I/O Developers Conference*, 2010.

[10] Yepang Liu, Chang Xu, and Shing-Chi Cheung. Characterizing and detecting performance bugs for smartphone applications. In *ICSE*, 2014.

[11] Shengqian Yang, Dacon Yan, and Atanas Rountev. Testing for poor responsiveness in android applications. In *MOBS*, 2013.

[12] Rui Ding, Hucheng Zhou, Jian-Guang Lou, Hongyu Zhang, Qingwei Lin, Qiang Fu, Dongmei Zhang, and Tao Xie. Log2: A cost-aware logging mechanism for performance diagnosis. In *USENIX ATC*, 2015.

[13] Milan Jovic, Andrea Adamoli, and Matthias Hauswirth. Catch me if you can: performance bug detection in the wild. *ACM SIGPLAN Notices*, 46, 2011.

[14] Marco Brocanelli and Xiaorui Wang. Hang doctor: runtime detection and diagnosis of soft hangs for smartphone apps. In *EuroSys*, 2018.

[15] Olga Chukhno, Gurtaj Singh, Claudia Campolo, Antonella Molinaro, and Carla Fabiana Chiasserini. Machine learning performance at the edge: When to offload an inference task. In *NET4us*, 2023.

[16] Mohammad Reza Golzari Oskoui and Brunilde Sansò. Online dependency-aware task offloading in cloudlet-based edge computing networks. In *MobiWac*, 2023.

[17] Marco Brocanelli and Xiaorui Wang. Surf: Supervisory control of user-perceived performance for mobile device energy savings. In *ICDCS*, 2018.

[18] He et al. HangFix: automatically fixing software hang bugs for production cloud systems. In *SoCC*, 2020.

[19] Tingxin Yan, David Chu, Deepak Ganesan, Aman Kansal, and Jie Liu. Fast app launching for mobile devices using predictive user context. In *MobiSys*, 2012.

[20] Hong Cao and Miao Lin. Mining smartphone data for app usage prediction and recommendations: A survey. *Pervasive and Mobile Computing*, 2017.

[21] Choonsung Shin, Jin-Hyuk Hong, and Anind K. Dey. Understanding and prediction of mobile application usage for smart phones. In *UbiComp*, 2012.

[22] Huang et al. Predicting mobile application usage using contextual information. In *UbiComp*, 2012.

[23] Chen et al. Cap: Context-aware app usage prediction with heterogeneous graph embedding. *IMWUT*, 2019.

[24] Yingzi Wang, Nicholas Jing Yuan, Yu Sun, Fuzheng Zhang, Xing Xie, Qi Liu, and Enhong Chen. A contextual collaborative approach for app usage forecasting. In *UbiComp*, 2016.

[25] Shen et al. Deepapp: A deep reinforcement learning framework for mobile application usage prediction. *IEEE Transactions on Mobile Computing*, 2023.

[26] Zhihao Shen, Xi Zhao, and Jianhua Zou. GinApp: An inductive graph learning based framework for mobile application usage prediction. In *INFOCOM - IEEE Conference on Computer Communications*, 2023.

[27] Kang Yang, Xi Zhao, Jianhua Zou, and Wan Du. ATPP: A mobile app prediction system based on deep marked temporal point processes. In *DCOSS*, 2021.

[28] Fang et al. BERT-based semantic-aware heterogeneous graph embedding method for enhancing app usage prediction accuracy. *IEEE Transactions on Human-Machine Systems*, 2024.

[29] Lee et al. Click sequence prediction in android mobile applications. *IEEE Transactions on Human-Machine Systems*, 2019.

[30] Paul Baumann and Silvia Santini. Every byte counts: Selective prefetching for mobile applications. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, 2017.

[31] Khalloufi Issam and El Beqqali Omar. Real-time data prefetching in mobile computing. In *AICCSA*, 2015.

[32] Zhao et al. Leveraging program analysis to reduce user-perceived latency in mobile applications. In *ICSE*, 2018.

[33] Zhengquan Li, Summit Shrestha, Zheng Song, and Eli Tilevich. Edge cache on WiFi access points: Millisecond-level app latency almost for free. In *ICDCS*, 2024.

[34] Murali Ramanujam, Helen Chen, Shaghayegh Mardani, and Ravi Netravali. Floo: automatic, lightweight memoization for faster mobile apps. In *MobiSys*, 2022.

[35] Cho et al. On the properties of neural machine translation: Encoder-decoder approaches. In *arXiv*. arXiv, 2014.

[36] Roberto Cahuantzi, Xinye Chen, and Stefan Güttel. *A Comparison of LSTM and GRU Networks for Learning Symbolic Sequences*, pages 771–785. Springer Nature Switzerland, 2023.

[37] Shudong Yang, Xueying Yu, and Ying Zhou. LSTM and GRU neural network performance comparison study: Taking yelp review dataset as an example. In *IWECAI*, 2020.

[38] Leo Breiman. Random forests. *Machine Learning*, 45(1), 2001.

[39] Wei Yang Bryan Lim, Nguyen Cong Luong, Dinh Thai Hoang, Yutao Jiao, Ying-Chang Liang, Qiang Yang, Dusit Niyato, and Chunyan Miao. Federated learning in mobile edge networks: A comprehensive survey. 22(3):2031–2063.

[40] Christoph Bandt and Bernd Pompe. Permutation entropy: A natural complexity measure for time series. *Physical Review Letters*, 88, 2002.

[41] Accubattery. [online]. available:. https://www.accubatteryapp.com/.