Practical Prediction and Prefetch for Faster Access to Applications on Mobile phones

Abhinav Parate
University of Massachusetts
Amherst, MA, USA
aparate@cs.umass.edu

Matthias Böhmer
DFKI GmbH
Saarbrücken, Germany
matthias.boehmer@dfki.de

David Chu
Microsoft Reasearch
Redmond, WA, USA
davidchu@microsoft.com

Deepak Ganesan
University of Massachusetts
Amherst, MA, USA
dganesan@cs.umass.edu

Benjamin M. Marlin
University of Massachusetts
Amherst, MA, USA
marlin@cs.umass.edu

ABSTRACT

Mobile phones have evolved from communication devices to indispensable accessories with access to real-time content. The increasing reliance on dynamic content comes at the cost of increased latency to pull the content from the Internet before the user can start using it. While prior work has explored parts of this problem, they ignore the bandwidth costs of prefetching, incur significant training overhead, need several sensors to be turned on, and do not consider practical systems issues that arise from the limited background processing capability supported by mobile operating systems. In this paper, we make app prefetch practical on mobile phones. Our contributions are two-fold. First, we design an app prediction algorithm, *APPM*, that requires no prior training, adapts to usage dynamics, predicts not only which app will be used next but also when it will be used, and provides high accuracy without requiring additional sensor context. Second, we perform parallel prefetch on screen unlock, a mechanism that leverages the benefits of prediction while operating within the constraints of mobile operating systems. Our experiments are conducted on long-term traces, live deployments on the Android Play Market, and user studies, and show that we outperform prior approaches to predicting app usage, while also providing practical ways to prefetch application content on mobile phones.

Author Keywords

App prediction, Prefetch, Mobile computing

ACM Classification Keywords

C.5.3 Computer System Implementation: Microcomputers—*Portable devices*; D.4.8 Operating Systems: [Performance Modeling and prediction]

UbiComp'13, September 8–12, 2013, Zurich, Switzerland.

http://dx.doi.org/10.1145/2493432.2493490

INTRODUCTION

The success of smartphones has resulted in an explosive increase in the number of apps available in app marketplaces. Currently, Apple's iOS app store and Google's play store both have more than 700,000 apps available while the Window Phone app store is catching up with more than 150,000 available apps. Among the most popular apps in the marketplace are real-time content-driven applications such as News, Email, Facebook, Twitter, and others that provide timely information to users.

While the utility provided by installed apps has made the smartphone an indispensable companion to users, it comes with a new set of user experience challenges. Mobile phones are largely used during idle times in between real-world tasks, and users want information instantly during these periods. Yet, the reality is that delays in refreshing popular apps like *Email, Facebook, Twitter, News* and *Weather* that rely on network connectivity to download app content is often of the order of seconds or tens of seconds. The fickle nature of cellular connectivity means that these apps can have highly variable wait times before the content is displayed to the user. Prior work has shown that the top apps available in the market have an average network latency greater than 11 seconds [15]. In short, we face a dire challenge — app loading times are increasing as apps fetch more content from the cloud, and the cellular data networks continue to play catch up with these trends.

There have been several prior efforts at tackling the increase in app usage latency, but ultimately they fall short of addressing a plethora of practical considerations. First, prior work ignores the freshness of loading predicted apps — for example, *Email* may be predicted to be the next application to be used, but should new emails be prefetched now or an hour later? Second, several past efforts propose to modify either the mobile OS or apps in order to speed up launch time [11, 15] — these methods lack easy deployability through app stores or require modifications of 3rd party app source code. Third, prior approaches propose to use contextual data such as location to improve prediction accuracy. However, users are often reluctant to install apps that require contextual information, in particular location context, diminishing the broader appeal of

such methods. Fourth, prediction methods proposed in prior work often have high training requirements (several weeks of data), but users expect system adaptation within days, not weeks.

In response, we have developed *PREPP*[1], a preeminently practical approach to prefetch. By practical, we mean that the two major facets inherent to fast app launch — App Prediction, and Prefetch Execution — have been designed for performance on today's stock mobile devices with users seeing immediate launch time speedups. At the core of our system, we have two models. The first is *App Prediction by Partial Match (APPM)*, a *prediction model* that adaptively learns the probability distribution of the apps to be used next and requires no privacy-sensitive or power-hungry contextual information like location. The second model is a *Time Till Usage (TTU) temporal model* that utilizes the learnt distribution of time spent before app use to estimate appropriate time for prefetch in a bandwidth cost-aware manner – an aspect which is of key importance for prefetching dynamic application content.

The outputs of the prediction and TTU models are combined in *PREPP* to dynamically decide when to prefetch app content, thereby delivering significant fresh content with low energy overhead, while requiring neither app nor OS modifications. We are able to accomplish this by designing *PREPP* to take advantage of app foreground-background execution semantics that are common across mobile OSs while operating within their constraints.

We use a combination of a deployment in the Android app store with 7,630 active users, controlled user study with 22 participants, trace-based analysis and micro-benchmarks of an Android smartphone to show that *PREPP*:

- Reaches over 80% accuracy when predicting a top 5 ranking for the next app to be used. In fact, we perform better than widely used approaches such as MRU, as well as previous technical proposals [11, 15] while requiring neither privacy-sensative location information nor offline training time.
- Delivers content to the user that is on average fresh within 3 minutes, and even up to 1.5 minutes fresh on average for some users.
- Causes negligible additional energy expenditure.

BACKGROUND AND SHORTCOMINGS

In this section, we outline three drawbacks of existing systems that we address in this work.

Slow User Adaptation. Approaches that use prediction-based strategies have suffered from prediction model training cold-start: once a user starts to use the system, it has taken as long as 3-6 weeks of data collection before meaningful predictions can be made [15]. Unfortunately, for mobile systems where time and attention spans are limited, users expect benefits within days, not weeks, particularly since app usage patterns continually change over time. For example, long-term traces of app usage (e.g. LiveLab iPhone usage dataset [10]) show that users download more than one hundred apps each year, and use them only for a few weeks or months. Therefore, prediction systems that require lengthy training periods are not suitable.

Lack of Freshness. Prior work only answers the question of which application will be used next and not *when* it will be used. Predicting when an application will be used is particularly important when optimizing freshness of prefetched application content. For example, *Email* may be predicted to be the next application to be used, but should it be prefetched now or an hour later? Prefetching content frequently would incur bandwidth and energy overhead, and infrequently would result in stale content. Therefore, temporal prediction is an important missing piece for cost-aware prefetch.

Ease of Deployability Mobile OSs have unique constraints that make it difficult to implement a practical prefetch system. Hence existing approaches either propose modifications to mobile OSs [15], or modifications to the app source to prefetch content [8]. However, this hurts their deployability — even with an open source system like Android, manufacturers often lock down the OS bootloader such that unsigned OS changes are not installable. Similarly, a potential approach that aimed to modify only the server push decision algorithm would also lack deployability because push notification infrastructure is under OS vendor control and is not accessible for modification. Finally, approaches that propose to modify only apps are challenged with the necessity for access to and modification of 3rd party app source [8]. These constraints from the OS and Apps are unlikely to change anytime soon. While modifications to the OS or apps represent useful design alternatives, a key question that has remained unanswered is whether it is possible to design an easily deployable, store-compatible prefetching system for mobile devices.

REQUIREMENTS

Our goal is to design *PREPP*, a practical prediction and prefetch system that satisfies the following key requirements:

- ► *PREPP* should require a *small training* overhead, and be able to converge to high accuracy within a small number of days of use, while adapting to changing usage patterns and applications quickly.
- ► *PREPP* should maximize freshness for applications that need to update their content, while minimizing the resource overhead incurred in terms of network access and energy cost.
- ► *PREPP* should work on unmodified off-the-shelf phones and bring speedups to existing apps, thereby making it immediately deployable to mobile app stores, installable on current mobile OSs and usable by all mobile users.

The next three sections address these requirements.

PREPP: SYSTEM DESIGN

The problem that we solve in *PREPP* can be defined as follows: given a sequence of content-based apps that a user has

[1]Predictive Practical Prefetch

used, and the times when the user has used them, can we prefetch content in a timely manner while keeping the overall network prefetch costs low. This high-level problem can be divided into three sub-problems: a) can we accurately predict what app is going to be used next? b) can we predict when that app is going to be used next? and c) can we decide when to prefetch to maximize freshness while keeping network access costs bounded? We describe next the three key system components in *PREPP* that accomplish these goals.

APP PREDICTION ALGORITHM

The problem of app prediction is as follows: given the app usage sequence for an individual, can we predict what app the user is likely to use next? Intuitively, this problem has parallels with text compression, where the preceding character sequence can be used to determine the most likely next character, which in turn can be leveraged to compress the text. (For example, a character following the sequence *natio* in English language is highly likely to be *n*). Similarly, one can view each app as a "character" and the sequence of app usages as a character stream, and apply text compression techniques to our problem. We now look at what text compression algorithm to leverage, and how to adapt it to our needs.

One of the widely used methods in text compression is Prediction by Partial Match (PPM) [6]. At a high level, PPM operates by scanning character sequences, and building up a variable-length Markov-based predictor on the fly. PPM uses the longest preceding character sequence to compute the conditional probability distribution for the following character. To compute this conditional probability distribution, PPM maintains frequencies for characters that have been seen before in all prefix-sequences that have occurred before, up to some maximum order, where order is the length of the prefix. For example, PPM with order 3 maintains frequencies for all prefixes of length 3, 2, 1, and 0 as shown in Table 1. To predict what character is likely to appear next in the stream, PPM computes the conditional probability for each character given the highest order node having a non-zero frequency for the character and scales it using weights, with weights being highest for the longest prefix, since longer matches often provide more precise contexts for character prediction.

Since PPM is designed for text compression rather than app usage patterns, we need to address some of the differences between these two cases. The main difference is that in text prediction, the longest prefix is often the most relevant, whereas in app usage, both long and short prefixes can be highly informative. For example, we found that in some cases, a user tends to strongly favor recently used items whereas in other cases, a user exhibits highly sequential app usage behavior.

We address this issue by allowing the prediction accuracy from the different orders in PPM determine how they are weighted. Thus, we do not make strong assumptions about what length prefixes are likely to be useful, and instead let the empirical accuracies from prior predictions dictate which nodes are favored. With accuracy-driven weights, we compute weighted sum of the outputs of predictions from each of the prefix nodes. In this manner, *APPM*'s vocabulary is personalized to each user, and can learn their preferred patterns over time. Table 1 shows an example scenario where *APPM* weights are given by accuracy obtained using the top-2 predictions for each node.

Order	prefix	character frequency						T_m	PPM w_m	APPM w_m
		n	t	r	f	s	o			
3	sio	2	1					3	1	0.5
2	io	3	1	1				1	$\frac{1}{4}$	0.5
1	o	3	1	2	6			6	$\frac{1}{4.2}$	0.64
0	ϕ	3	1	2	6	4	12	16	$\frac{1}{4.2.7}$	0.38

Table 1. Example of PPM nodes for current context sio in string *"sionofionsionofofioroforofsiotofsio"*. Nodes for prefixes *sio, io, o* and empty sequence ϕ along with character frequencies, sum of frequencies for characters in node that are not present in higher order node (T_m), PPM weights $w_m = \prod_{i=m+1}^{3} \frac{1}{T_i+1}$ and APPM weights given by the prediction accuracy observed for the node are shown.

In summary, APPM is an appealing app prediction algorithm that satisfies our key requirements: (1) it has low training overhead — incremental training requires O(1) operations that involve a simple increment of counts, (2) it continuously adapts to changes in the sequence's character distribution by discounting older history, and (3) it learns what prefixes are likely to be more useful for prediction and places its bets appropriately.

TEMPORAL MODELING

One of the limitations of prior work on app prediction is that it does not consider *freshness*, i.e. how recently an application's content was prefetched prior to an application use. Consider the following naïve prefetch example — every time APPM predicts that Email will be used as the next app, prefetch Email. This has high freshness if Email is used immediately but has poor freshness if the user opens Email an hour later. An alternative might be to prefetch Email every, say 10 minutes, until the user opens Email. But this suffers from high network access overhead if the delay is of the order of an hour or more. Clearly, if we want to optimize cost, we need to first predict *when* an app is likely to be opened next.

The question we are addressing here is: If we have just opened an app, and if we know that the next app to be used is e, what is the probability that e will be used in time interval Δt from the current time (denoted $p(TTU \leq \Delta t | nextapp = e)$, where TTU, time till usage for $nextapp$, is the time that elapses between the current time and when the $nextapp$ is used). Note that we only need to evaluate this probability every time an app is used (we refer to this as an *app-change event*), since no additional information is available between app usage events.

To answer this, we learn the conditional cumulative distribution function (CDF) $F_{TTU|nextapp=e}$ for an app e from the app usage history. Given an app usage history $[app_1, ..., app_n]$ where app_j is the j^{th} app used and launched at time t_j, we can learn the conditional CDF using the distribution of all durations $(t_j - t_{j-1})$ such that $app_j = e$. Figure 1 shows an example of the conditional CDF obtained for a representative user where $nextapp$ being modeled is *Email*. The conditional CDF is computed from a user trace of app usages obtained from LiveLab iPhone usage dataset [10].

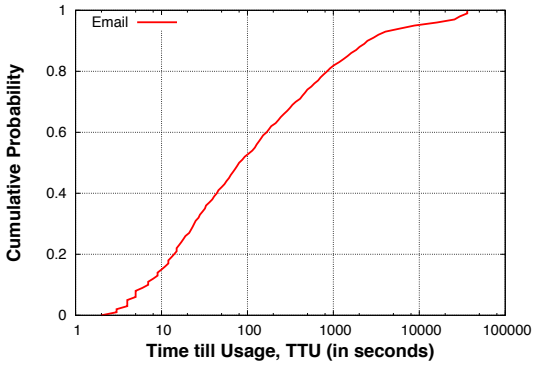


Figure 1. The cumulative probability distribution of *Email* to be launched as the next app as a function of time till usage, obtained for a representative user.

DECISION ENGINE

Our decision engine executes on each app-change event and utilizes the prediction model and the temporal model to decide when to prefetch an app such that it balances the need to improve freshness, while at the same time limits the network bandwidth costs associated with frequently connecting to a server and downloading the data from it. The output of the decision engine is the time Δt we should wait from the app-change event to execute prefetch.

Recall that our prediction model predicts *what* app is going to be used next and the temporal model predicts *when* some app will be used assuming it knows what app will be used next. Our decision fuses the uncertainty in *what* and uncertainty in *conditional when* to get the joint estimate of the probability of some target app $nextapp$ to be used next within some time interval(Δt). Let us denote this probability using p_{fetch}. Now, if we choose to prefetch after time Δt has elapsed, then with probability $1 - p_{fetch}$, the next app will be used after prefetch and will see a freshness benefit of Δt, and with probability p_{fetch}, the next app is opened prior to prefetch and it will see no benefits in freshness. If we select a small value for p_{fetch}, it has two effects: a) more apps are prefetched since even apps that are predicted to be opened next with low probability will qualify for prefetch, and b) Δt will be smaller and hence, we will typically prefetch apps quickly. This means higher bandwidth cost and apps used a while later see less prefetch benefit. On the other hand, a higher value of p_{fetch} means fewer app prefetches and longer Δt and hence, we prefetch less frequently but the fewer target app usages that benefit from it see better freshness due to larger Δt benefit.

Thus, the key question for the decision engine is how to choose p_{fetch} to balance freshness and bandwidth cost. To address this, we set a target bandwidth cost for each app, and learn from history the largest possible value of p_{fetch} that would have been within the target cost while ensuring the least missed opportunities. This learning is not expensive since we only need to make a decision upon each app change event, which restricts our search space. We now present our intuition in a more formal manner.

Algorithm

Our prediction model can be used to compute the probability $p(nextapp = e)$ and our temporal model gives us the probability of $nextapp = e$ being used within time Δt conditioned on e being the next app. This probability is provided using the conditional cumulative distribution function $F_{TTU|nextapp=e}(\Delta t)$. The decision engine fuses these two outputs multiplicatively to obtain a cumulative distribution function ($F_{TTU}(\Delta t)$) that can be used to compute the probability of e being used as the next app within the time interval Δt.

$$F_{TTU}(\Delta t) = p(nextapp = e) \times F_{TTU|nextapp=e}(\Delta t)$$

Once the distribution function F_{TTU} is known, we need to choose a threshold p_{fetch} that balances freshness and cost as described earlier. We define *network bandwidth cost* (C) as the multiplicative factor of the rate of use of the target app. Thus, if a target app is used N times then the number of prefetches can be at most $N \times C$. Given such a cost C and app usage history, we can estimate the appropriate value for p_{fetch} to limit the cost to C. The detailed description of our algorithm can be found in Algorithm 1, but intuitively, if there are N occurrences of $nextapp$ in the user's history, the multiplicative cost is C and we know the probability values attained by the joint distribution function F_{TTU} just before each app-change event in the history, we pick the NC^{th} largest probability value as p_{fetch} since it results in exactly NC prefetches on the historical trace.

Algorithm 1 Compute Time To Prefetch

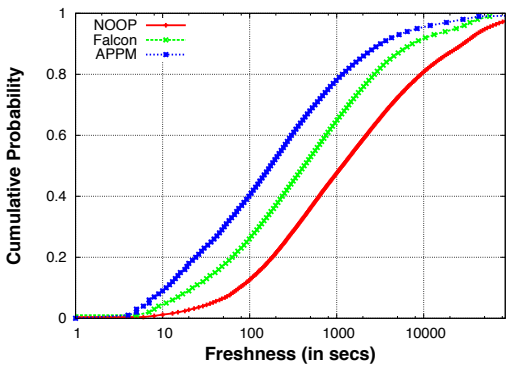
1: **Input:** Network Bandwidth Cost C; TTU distribution function for target app F_{TTU}; TTU probability history $d[1...L]$; Count of target app in user's history N.
2: **Output:** Time to wait for prefetch Δt.
3: Sort d in decreasing order.
4: $p = d[N_e * C]$ i.e. NC^{th} highest TTU probability.
5: $\Delta t = F^{-1}_{TTU}(p)$.
6: **return** Δt

TRACE-DRIVEN EVALUATION

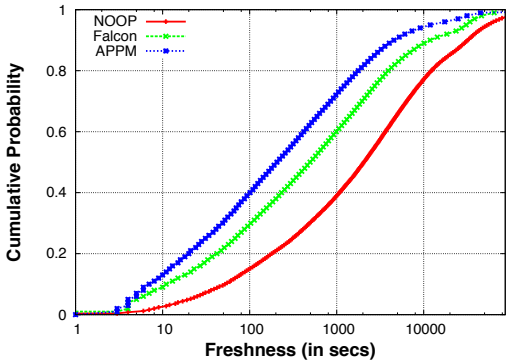
In this section, we evaluate the benefits of our techniques using long-term app usage traces provided by the LiveLab project at Rice University [10]. This dataset consists of traces for thirty four volunteers collected on iPhone 3GS for a period of up to fourteen months. We look at the app-usage traces from this dataset, which provide the start time and the duration of each app usage observed on the phone. In addition, we use location traces to obtain the location where apps are used. Since location traces are obtained at a coarse granularity compared to app usage data, we interpolate location information to find the missing location information for app usage data. In all, we have 576,491 records of app use across 14 months.

Prefetch Effectiveness

We evaluate the effectiveness of our prefetch algorithm and compare it with Falcon [15] which is the only other known scheme for app prefetch. Figure 2 shows the results for two heavily used apps that can leverage prefetch, *Email* and *Facebook*. Since Falcon cannot control the bandwidth costs, for a fair comparison, we evaluate our APPM+TTU model based



(a) Freshness in *Facebook* app



(b) Freshness in *Email* app

Figure 2. Freshness observed using i) No prefetching (NOOP) ii) Previously proposed scheme (Falcon), and iii) our APPM prefetch algorithm. While the previous scheme improves median *Email* and *Facebook* freshness to 7 and 8 minutes, respectively, APPM further improves freshness to 3 minutes at the same bandwidth cost.

prefetch scheme using the same bandwidth cost as observed in Falcon. The median freshness (i.e. the time period between the latest network data fetch and the time of actual app use) with the Falcon scheme is near 8 minutes for both the apps. Our scheme improves freshness by reducing median freshness to 3 minutes. Note that for the 'No prefetching' case, we computed freshness as the time elapsed since last use of the app assuming that the app is refreshed by a user upon each use.

In addition, we evaluate the effectiveness of temporal modeling in our scheme by comparing it against a APPM-based scheme which prefetches on app-change events whenever the probability of the target app predicted by APPM algorithm is above a certain threshold. The threshold value in this scheme is determined by the bandwidth cost. Table 2 shows the benefits of using TTU model in 3 quartiles of observed freshness in *Facebook* app for various costs. We see that the benefits are significant, particularly at low bandwidth costs. The median freshness improves by 25% for 2× and 4× cost, demonstrating the benefit of incorporating TTU into the algorithm.

Utility of location context

As described earlier, several prior efforts at predicting application usage have pointed out that location context is very useful [11, 13, 15]. Yet, obtaining location is often undesirable due to user privacy concerns as well as energy overhead. Therefore, we investigate the actual utility of location data and whether we can forgo it. Interestingly, we find that app prefixes largely subsume the need for explicit location.

Cost	Algorithm	Freshness(in secs)		
		25-%ile	50-%ile	75-%ile
2x	*APPM* w/o TTU model	83	383	1966
	APPM w/ TTU model	68	286	1334
4x	*APPM* w/o TTU model	41	190	928
	APPM w/ TTU model	37	155	691
8x	*APPM* w/o TTU model	26	111	506
	APPM w/ TTU model	26	104	452

Table 2. APPM w/ TTU model delivers better freshness, especially for low bandwidth budgets. Three quartiles of observed freshness values for *Facebook* app are shown at varying bandwidth budgets of 2x, 4x and 8x (Evaluated on 34 users from LiveLab dataset).

Context Used	Prediction Accuracy
None	80.85%
Location	81.10%
Time of Day	81.23%
Location, Time of Day	81.35%

Table 3. Aggregate prediction accuracy over all users obtained using APPM with and without various contexts. Additional contexts like Time of Day and Location provide only a marginal improvement in accuracy.

To understand whether location contexts improve prediction accuracy, we design a multi-context version of *APPM* which includes a different predictor for each location context (e.g. home, workplace, etc), and one for each time-segment of the day (e.g. morning, afternoon, etc). Given several context-specific predictors, we compute a weighted combination of these models to predict the next application. Table 3 shows the performance of a single predictor that does not use any context, as well as predictors that use additional contexts. The benefits of additional context is surprisingly small, a puzzling observation given observations made in prior work. A closer look at app usage patterns in different location contexts provides an interesting explanation. As it turns out, "contextual" information is partially captured by the app sequences that are used only in a specific context e.g. Angry Birds at home. We further validated this by looking at the distribution of location contexts for each prefix used by *APPM* for a representative user who had 14 semantic locations. We observed an 80^{th}-percentile entropy of 2.0 bits indicating that a large number of app prefixes have location information encoded in them. If the prefixes had no correlation with location, then the entropy would be close to 3.8 bits corresponding to the 14 semantic locations.

Prediction Evaluation

Tables 4(a) and 4(b) present a comparison of prefix-only APPM against previously proposed prediction techniques including: a) Falcon, which uses location and time contexts [15], b) 2-NB: a Naive-Bayes prediction model that uses location and time-of-day as features [13], c) 3-NB: a Naive-Bayes prediction model that uses location, time-of-day and previous-app-used as features [4], and d) a strawman *Most Frequently Used* (MFU) algorithm.

(a) Predicting all apps in the user traces

Algorithm	Prediction Accuracy
MFU	48.81±1.08 %
2-NB	74.87±1.60 %
3-NB	78.81±1.34 %
APPM	80.85±1.23 %

(b) Predicting follower apps in the user traces

Algorithm	Prediction Accuracy
Falcon	70.16±1.56 %
APPM	74.37±1.41 %

Table 4. Comparison of APPM with previous app prediction algorithms when making Top-5 predictions. APPM performs better than schemes using location and time context.

Table 4(a) compares prefix-only *APPM* against all schemes except Falcon, and Table 4(b) compares against Falcon. We separated these results because Falcon only predicts occurrences of apps that follow the first app used upon unlocking the screen. The results show that the prefix-only version of *APPM* performs just as well as algorithms that use additional contexts including location.

PRACTICAL PREFETCH CONSIDERATIONS

Mobile operating systems such as Android, iOS and Windows Phone support a set of runtime semantics distinct from their desktop OS counterparts. A central challenge in the design of a practical prefetching system is managing the limitations posed by these mobile operating systems. While the remainder of our discussion centers around Android, the design decisions are in fact standard to modern mobile OSs, and apply to iOS and Windows Phone as well.

Addressing Android constraints: Android places several constraints on the scheduling and resource usage of foreground and background threads and processes. The constraints that directly impact the design of prefetch mechanisms are two-fold. First, main threads are not scheduled when the device enters the standby state (which occurs either due to an idle activity timeout or an explicit user button toggle). This means that main threads can only run when the device is active. Second, apps are restricted from performing networking activities on the main thread as it can make an app unresponsive if there is slow or no network connectivity. Thus, the standard model is that apps fetch network data by spawning background threads from the main thread.

We examined the behavior of several apps that could be potential prefetch candidates and found that all of them rely on the main thread to be active to fetch content or load state. For example, *Facebook* and *Email* rely on network fetch and initiate background threads for network calls from the main thread when the app is opened and appears on the screen. Thus, it was clear that we could not initiate prefetch during times that the device was turned off, leaving us with the times that the device was actually unlocked and in use.

A close look at applications that require content refresh reveals that they often auto-refresh upon opening the app and/or upon re-starting the app. This means that once auto-refresh is triggered by the main thread, the asynchronous network fetch task is created and the app can be pushed to the background. Thus, we only needed to bring the app to the foreground for the small amount of time that it needed to initiate auto-refresh. An empirical study of the minimum amount of time needed for an app to be brought to the foreground showed that it ranges from 200ms for apps like *Email* to up to 1 second for *Facebook*. Note that even if auto-sync is turned off for an application, it is possible to kill and re-start an app, at which time it automatically updates its content.

Minimizing disruptiveness to user: Next, we ask what is the most suitable time for prefetch such that it is minimally disruptive to user experience. During periods when the device is actively used, three opportunities present themselves: a) when the screen is unlocked and before an application is started, b) transition times between usage of applications in the same session, and c) between an app being closed and the phone being turned off. While all three are viable, we find that the most convenient option is prefetch upon screen unlock since it allows us to prefetch immediately prior to a user opening an app. Our approach uses k-step prediction, where *APPM* predicts which apps to prefetch among the next k applications that were going to be used. Previous work has analyzed typical session lengths and found that these are typically short [15], therefore a two-step prediction is sufficient for deciding which apps to prefetch upon screen unlock.

Parallel prefetch to minimize energy overhead: The energy consumption in executing prefetch is another consideration that impacts our design. Prior work has shown that data transfer using cellular incurs a tail-energy overhead as the phone's radio remains in active mode for another 10-20 seconds after data transfer is complete [7]. Therefore, batching transfer can provide energy savings in the range 62-75% for 3G networks. We leverage this insight to reduce prefetch cost by executing a number of prefetches in parallel. In *PREPP*, two-step look ahead allows us to merge all the prefetches that are scheduled to execute within 30 seconds of each other and execute them in parallel.

IMPLEMENTATION

To implement the APPM algorithm on Android, we ported the PPMII implementation from D. Shkarin et al. [12] in C++, and modified it to implement our ranking scheme for predictions. Since Android apps are written in Java, we implemented a Java wrapper around this code using Java Native Interface so that other components implemented in Java can interact seamlessly with the predictor.

In addition to prefetching content automatically, we also integrate a dynamic home screen widget that leverages predictions from *APPM* to allow users to quickly find apps among the many that they have installed. The idea of adaptive shortcut menus for smartphones has existed for some time [3, 5, 11, 14]. Our *PREPP* widget builds on prior work, with the change that we update the widget not only when the device is unlocked, but also when the user returns to his homescreen after closing an app. This provides the user with a higher degree of adaptiveness.

Figure 3. *PREPP's* adaptive shortcut menu placed on the homescreen as a widget.

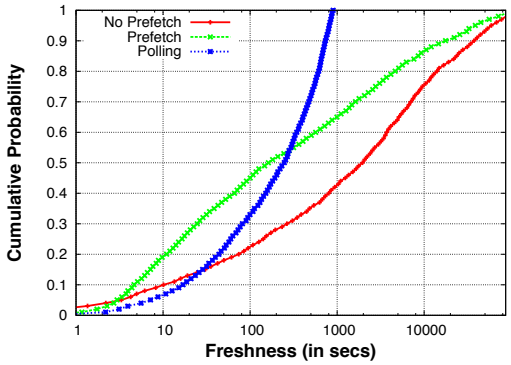


Figure 4. Prefetch achieves better content freshness than 15-minutes polling at a fraction of cost (number of polls were 3.58× number of prefetches). The median freshness improves from 1955s (no prefetch) to 247s with polling and to 162s with prefetch.

Figure 3 shows a screenshot of our *PREPP* adaptive shortcut menu Android prototype. Shortcut icons for the top five predicted apps are shown, and the user can click on any of them to launch it directly. In addition, a timer beneath each shortcut indicating time since prefetch informs users about content freshness.

User Control: Since a user may have sensitive apps like banking applications or apps that use privacy-sensitive information like location upon start, we do not prefetch any app by default. Instead, we provide users with an interface to select the apps they want to prefetch. This interface provides users with a greater control regarding which apps get prefetched.

EXPERIMENTAL EVALUATION

In this section, we present an evaluation of our implementation through two user studies. We first describe our controlled user study where we evaluate benefits of prefetch in detail. Next, we describe our experiences deploying our prediction algorithm part of *PREPP* to Google Play Store and study its impact on real users in the wild. Finally, we conclude with benchmarks characterizing system overhead for our implementation.

Controlled User Study

In order to perform a detailed evaluation of the prefetch benefits from *PREPP*, we conducted a user study[2] consisting of 22 users. Of these 22 users, 6 users used *PREPP* for at least two weeks, segmented as follows: (1) one week of control data with no prefetch, and (2) one week of data with prefetch active, and the multiplicative bandwidth cost C set to 2. The remaining 16 users had a control and test period of 3 days and 4 days respectively. Recall that *PREPP* performs prefetch by bringing an app to the foreground for a brief period upon unlocking the phone screen. To make users unaware of whether it was the control or experimental period, apps were brought to the foreground during the control period without actually performing any content fetch activity. Note that for the controlled user study, we disabled the adaptive shortcut menu shown in Figure 3.

[2]Study conducted with appropriate IRB approval

Prefetch Benefits

The primary metric for evaluating the benefit of prefetch during app usage is the increase in freshness of content. Content freshness reflects how recently content was retrieved at time of use, and is measured as the time between prefetch and app usage.

Figure 4 shows distributions of freshness observed for all users during the control period with no prefetching and the test period with active prefetching. Also, we show the distribution that can be achieved via 15-minutes polling for the test period as polling is the most popular way of refreshing app content on phones. We see substantial freshness benefits across the board, with an order of magnitude improvements across all three quartiles. The median freshness improves from 32.6 minutes to 4.1 minutes with polling and to 2.7 minutes with prefetch. Most importantly, the number of polls during test period are 3.58× the number of prefetches. Thus, APPM achieves better freshness than polling at a much smaller cost. Also, the median freshness with prefetch for 16 users with 3-days control period and 6 users with 7-days control period are 3.6 minutes and 2.06 minutes respectively.

Additionally, we measure prefetch effectiveness using *precision* i.e. the fraction of prefetches that were followed by a respective app usage and *recall* i.e. the fraction of app usages that were preceded by a respective prefetch. The average precision and recall values observed for all the users are 22.12% and 46.87% respectively. In contrast, 15-minutes polling has 3.82% precision and 28.97% recall.

Prediction Accuracy and Adaptivity

We now show that *APPM* requires very little training and adapts rapidly to changing user behavior. Figure 5 shows the time-series trend for a representative user — the bar graph below shows the number of unique apps seen by *APPM* over time, and the upper bar graph shows *APPM*'s accuracy on those days based on top-5 predictions.

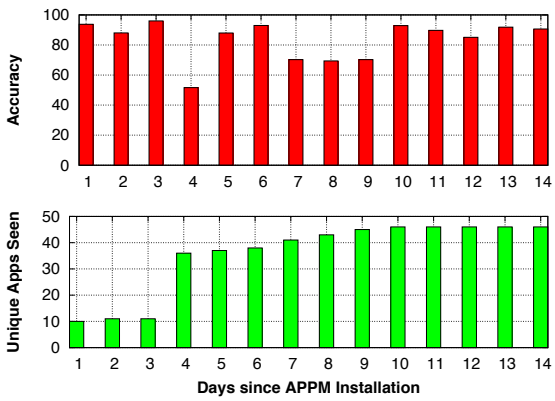


Figure 5. APPM yields high accuracy and adapts quickly to usage of new apps.

The results suggest two interesting conclusions. First, we see that accuracy with top-5 predictions is extremely high (95%) even on the first day, and our average prediction accuracy is 81.89±3.9%, both of which are very high. Second, we see that *APPM* adapts well to usage dynamics — for example, we see that on day 4, the user invokes a large number of new apps that are previously unseen by APPM, As a result, prediction accuracy drops for the day, but rapidly improves again on day 5. We see similar fluctuations between day 7 to day 10 where new apps are used. These results validate the adaptive capability of *APPM*

User Study in the Wild

Next, we study *PREPP* in the wild, and specifically focus on the prediction performance. We packaged the adaptive shortcut menu widget with APPM prediction and other state of the art prediction algorithms and released them as a standalone download — called *AppKicker*[3] — on the Google Play Store.

Methodology

We compared APPM with two other prediction algorithms: i) Most Recently Used (MRU), which is used by all major mobile platforms to show MRU app shortcuts; and ii) Sequential (SEQ), which uses only the previously used app to predict the next app, as suggested by previous work [4, 11].

We used a within-subject A/B/C design to test and compare the different prediction strategies; whenever a new prediction is requested, we randomly choose one of APPM, MRU or SEQ for prediction. We tracked how users interacted with the icon menu with each prediction algorithm. However, our evaluation is limited in that we cannot know what other icons users have on their home screens. For example, we might recommend apps that the user has already pinned to his home screen, or we might not catch app launches that the user initiates outside of our launcher widget.

Results

At the current time, the widget has been installed more than 43,600 times and has more than 7,630 active users.[4] Note

[3]http://goo.gl/wZO0C

[4]According to Android Developer Console

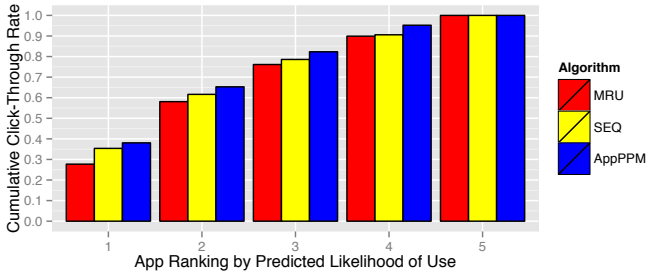


Figure 6. APPM yields better click-through rates for more highly ranked apps.

that both *PREPP* installation and participation in the research study are voluntary. As soon as a user starts *PREPP*, we present a research study disclaimer and offer opt-in consent. Following the Two Buttons Approach [9], users can decline from contributing data but still use the application.

For a fair comparison of the different prediction algorithms, we report results for the top 100 users as ranked by click volume. These users had a median of 112.5 clicks (min 62, max 1,807). This resulted in 16,991 clicks in total.

Figure 6 shows the click-through rate as a function of the prediction algorithm's ranking. APPM yields a click-through rate of 38.1% on its top ranked app, whereas MRU and SEQ produce only 27.7% and 35.4% respectively. Similarly, APPM outperforms other prediction algorithms at other ranked positions as well. APPM's more accurate predictions and better app rankings result in more opportunities for effective prefetch.

Binary Size	0.96MB
Memory	6.5MB
Time for prediction	$<250\mu s$
Time for prefetch decision	<5ms

Table 5. System overhead

MICROBENCHMARKS

System Overhead: We measure the system overhead of our implementation on the Samsung Galaxy Nexus Phone. Table 5 shows the overhead summary of *PREPP*. Using APPM for prediction and prefetch decision requires less than $250\mu s$ and $5ms$ respectively. This overhead is sufficiently low that a prefetch upon unlock is not an issue. The memory requirement of 6.5MB is modest and remains stable during execution. This suffices for running all required background services, keeping statistics for prediction in memory, and keeping uplink state to collect user trace.

Data Overhead: In this evaluation, we define two categories of apps that can use prefetching and evaluate the data overhead. We use data overhead observed for each app to set the app-specific multiplicative bandwidth cost C to be used by the decision engine.

Persistent data apps

In this category of apps, when refreshed, the apps fetch all the new-arrived data since it was last updated and save the

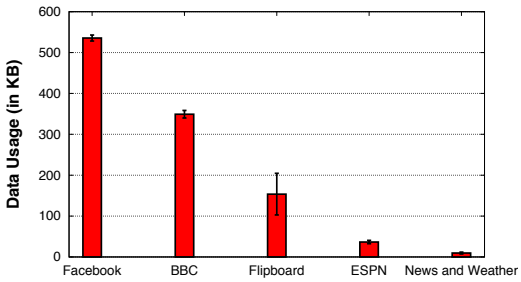


Figure 7. Data usage per refresh for popular apps.

data locally for later retrieval. Example of such apps include Email, Dropbox and Evernote. The data usage in these apps is proportional to the size of payload being fetched. These apps can fetch their payload in one or more prefetches but the actual cost of payload fetch remains independent of the number of prefetches. However, each prefetch incurs a constant overhead that is needed for the synchronization protocol and makes multiple prefetches expensive to execute. We can compute this constant overhead by observing the total data usage as we increase the payload size linearly. By fitting straight lines to these observations, we can solve for the constant. Using this, we estimate constant overhead to be 8447 bytes for the Email app and 2659 bytes for Gmail app.

Live feed apps

This category of apps receives the latest feed from the server. In contrast to Email app, these apps do not fetch all the new feeds since the last update but only the most recent ones. Example of these apps include Facebook, News and Weather, BBC news, etc. Depending on the nature of feed, the size of the feed can be near constant (e.g. live game scores) or it can vary significantly (e.g. Facebook where the number of images and the amount of text in a feed vary). While most of these apps will discard the previous feed upon refresh and fetch the feed again, there are a few apps like Flipboard that fetch only the change in feed since the last update. Overall, the data overhead in these apps is equal to the data used in receiving the new feed. Figure 7 shows average data usage for popular apps observed at various times of day.

Energy Consumption in 3G: We now compare energy consumption for refreshing app content in i) prefetch scheme, and ii) manual refresh. We note that a user refreshes app by opening each app in a sequence. In contrast, our prefetch scheme identifies all the apps that need to be refreshed as soon as the phone is unlocked and starts fetching content for all the identified apps in parallel. Consequentially, the power consumption characteristics are different for prefetch and manual refresh. Now, we show that our prefetch scheme using parallel fetches reduces the energy overhead per app-prefetch, and can save energy up to 47% in comparison to manual refresh if the app predictions are accurate.

Figures 8(a)-8(c) show power consumption characteristics of network data-fetch for 3 distinct apps on a 3G network with different levels of data usage. Also, Figure 8(d) shows power characteristics when data is fetched for all these 3 apps in parallel. We measured this using Monsoon Power monitor on Galaxy Nexus phones running on AT&T's 3G network. In these figures, the first phase with high power consumption is the data transfer phase, followed by a period called the *tail* up to 17s in length when there is no data transfer but the power consumed remains above 1000mW. In the end, we see a baseline power of 783mW required to keep the screen and cpu running on the phone. It is easy to see that parallel fetch has an advantage over sequential fetch as it incurs an energy overhead of 1 tail instead of 3 tails in the worst case for a sequential fetch. Apart from energy consumed in the tail, the energy consumed during the data transfer phase in parallel fetch is smaller than the total energy consumed in all the data transfer phases of sequential fetch of all the apps. Table 6 shows the average energy consumed in parallel and sequential fetch obtained over several runs. In this computation, we subtract the baseline power to accurately estimate the energy consumption for network fetch. We see that the parallel fetch uses 47.14% less energy than sequential fetch. Thus, if apps are predicted accurately, our prefetch scheme can potentially save energy. But if the predictions are inaccurate, energy in prefetch can go waste. In the worst case, the energy consumption of $1875\mu Ah$ for prefetching 3 apps in parallel accounts for 0.13% of the $1400mAh$ battery available on the Galaxy Nexus phone. This overhead is further reduced if upon unlocking the phone, a user starts some app that connects to the network to fetch data or an advertisement or if some widget initiates a network connection. This is because a prefetch will occur in parallel with a user-initiated or a widget-initiated fetch.

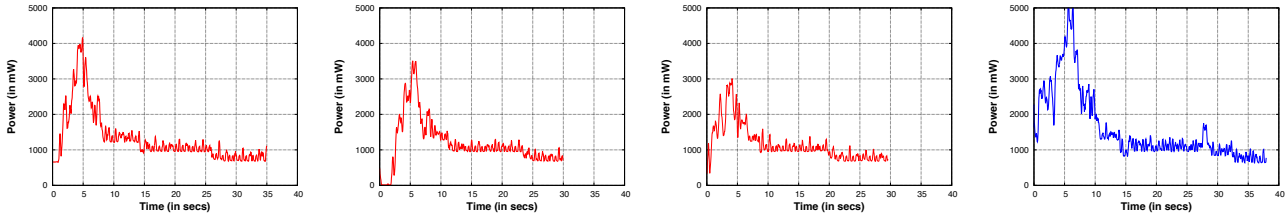
	Energy Consumption(in μAh)	
	Data Transfer Phase	**Total**
Sequential	2320.04	3547.25
Parallel	1407.31	1875.00

Table 6. Energy consumed in parallel fetch vs sequential fetch of content for 3 distinct apps. Parallel fetch saves 39.34% and 47.14% energy in data transfer phase and total respectively.

RELATED WORK

PREPP relates to two distinct lines of prior work:

App Prediction: In the wake of increasing availability of apps in phone's app stores, a number of algorithms have been proposed for utilizing contextual information in app usage prediction. Verkasalo et al. [13] found that location and time of day are the most useful contexts in app usage prediction. The study by Böhmer et al. [4] showed that in addition to location and time contexts, the last app used by a user is a useful predictor for the next app. Yan et al. [15] used the first app opened upon phone unlocking to predict the following apps. Most recently, Shin et al. [11] analyzed a variety of contextual information like accelerometer, WiFi, SMS, Bluetooth, GPS, last used app, etc. to build app-specific naive-bayes predictors by selecting the best set of contextual features for each app. In contrast to all these works, our APPM algorithm does not use any of power-hungry or privacy-sensitive contexts but utilizes only the history of sequence of app usages in prediction, and does not require a long training period unlike previous efforts.



(a) Google+ (b) Play Magazines (c) News & Weather (d) All apps in parallel

Figure 8. Power consumption characteristics for fetching network data shown for 3 apps: *Google+*, *Play Magazines* and *News & Weather*. Google+ has highest data usage(>300KB) whereas News has the least data usage(<10KB). Figure 8(d) gives characteristics when all these 3 apps are fetched in parallel.

Mobile Web Content Prefetch: Prefetching in mobile phones has recently received attention from the research community [2, 8, 15]. Higgins et al. [8] argue for an OS-supported API for prefetch where third party apps are expected to provide hints to execute prefetch. Such a model for prefetch requires modification of third party apps to learn the temporal model. In contrast, *PREPP* does not require any modification of the OS or third party apps. Yet, *PREPP* is complementary to this work in that *APPM* can provide hints to a potential OS-supported API on behalf of legacy third party apps. FALCON [15] considers prefetch under a fixed energy budget, but requires extensive modification of the OS and some apps, requires long training periods, and does not offer as accurate prediction as *PREPP*. Balasubramania et al. [2] proposed the TailEnder protocol that minimizes energy usage while prefetching and meeting user-specified delay tolerance. However, they do not take usage behavior into account, performing prefetch even when a user is unlikely to use apps. In the non-mobile case, browsers have utilized content prefetching and caching for reducing page load times [1], but were not targeted for interoperatibility with mobile apps nor OSs.

CONCLUSION

Mobile apps now serve a dazzling array of functions, but are becoming increasingly complex and reliant on network connectivity to provide up-to-date content and rich interaction. For mobile users where seconds of sluggish loading can dissuade many users, *PREPP* mitigates long network content retrieval times by accurately predicting which apps will be used, and prefetching their app content to improve the user experience. Moreover, *PREPP's* careful design, cognizant of the constraints posed by commodity OSs, has allowed us to implement, evaluate and deploy a *PREPP* prototype on commodity Android devices. Through a combination of a controlled user study and a study "in the wild" of the prediction algorithm, we show that we can accurately predict app usage, and deliver system-level speedups even without modification to current OSs and apps.

REFERENCES

1. Aggarwal, C., Wolf, J. L., and Yu, P. S. Caching on the world wide web. *IEEE Trans. on Knowl. and Data Eng. 11*, 1 (Jan. 1999), 94–107.
2. Balasubramanian, N., Balasubramanian, A., and Venkataramani, A. Energy consumption in mobile phones: a measurement study and implications for network applications. In *Proc. IMC'09*, ACM (2009), 280–293.
3. Böhmer, M., and Bauer, G. Exploiting the icon arrangement on mobile devices as information source for context-awareness. In *Proc. MobileHCI* (2010).
4. Böhmer, M., Hecht, B., Schöning, J., Krüger, A., and Bauer, G. Falling asleep with angry birds, facebook and kindle - a large scale study on mobile application usage. In *Proc. of MobileHCI* (2011).
5. Bridle, R., and McCreath, E. Inducing shortcuts on a mobile phone interface. In *Proc. IUI* (2006).
6. Cleary, J. G., Ian, and Witten, I. H. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications 32* (1984), 396–402.
7. Deng, S., and Balakrishnan, H. Traffic-aware techniques to reduce 3g/lte wireless energy consumption. In *Proc. of CoNEXT 2012*, ACM (2012), 181–192.
8. Higgins, B. D., Flinn, J., Giuli, T. J., Noble, B., Peplin, C., and Watson, D. Informed mobile prefetching. In *Proc. of MobiSys* (2012), 155–168.
9. Pielot, M., Henze, N., and Boll, S. Experiments in app stores - how to ask users for their consent? In *CHI '11 Workshop on Ethics, Logs and Videotape: Ethics in Large Scale Trials & User Generated Content* (2011).
10. Shepard, C., Rahmati, A., Tossell, C., Zhong, L., and Kortum, P. Livelab: measuring wireless networks and smartphone users in the field. *SIGMETRICS Perform. Eval. Rev. 38*, 3 (Jan. 2011), 15–20.
11. Shin, C., Hong, J.-H., and Dey, A. K. Understanding and prediction of mobile application usage for smart phones. In *Proc. of UbiComp'12*, ACM (2012), 173–182.
12. Shkarin, D. Ppm: One step to practicality. In *Proc. of the Data Compression Conference*, DCC '02 (2002), 202–.
13. Verkasalo, H. Contextual patterns in mobile service usage. *Personal Ubiquitous Comput. 13*, 5 (June 2009), 331–342.
14. Vetek, A., Flanagan, J., Colley, A., and Keränen, T. SmartActions: Context-Aware Mobile Phone Shortcuts. In *Proc. INTERACT 2009* (2009).
15. Yan, T., Chu, D., Ganesan, D., Kansal, A., and Liu, J. Fast app launching for mobile devices using predictive user context. In *Proc. of MobiSys*, MobiSys '12, ACM (New York, NY, USA, 2012), 113–126.