Leveraging Program Analysis to Reduce User-Perceived Latency in Mobile Applications

Yixue Zhao, Marcelo Schmitt Laser, Yingjun Lyu, Nenad Medvidovic

University of Southern California
Los Angeles, CA, USA
{yixue.zhao, yingjunl, neno}@usc.edu

Pontifical Catholic University of Rio Grande do Sul
Porto Alegre, RS, Brazil
marcelo.laser@gmail.com

ABSTRACT

Reducing network latency in mobile applications is an effective way of improving the mobile user experience and has tangible economic benefits. This paper presents PALOMA, a novel client-centric technique for reducing the network latency by prefetching HTTP requests in Android apps. Our work leverages string analysis and callback control-flow analysis to automatically instrument apps using PALOMA's rigorous formulation of scenarios that address "what" and "when" to prefetch. PALOMA has been shown to incur significant runtime savings (several hundred milliseconds per prefetchable HTTP request), both when applied on a reusable evaluation benchmark we have developed and on real applications.

1 INTRODUCTION

In mobile computing, user-perceived latency is a critical concern as it directly impacts user experience and often has severe economic consequences. A recent report shows that a majority of mobile users would abandon a transaction or even delete an app if the response time of a transaction exceeds three seconds [6]. Google estimates that an additional 500ms delay per transaction would result in up to 20% loss of traffic, while Amazon estimates that every 100ms delay would cause 1% annual sales loss [42]. A previous study showed that network transfer is often the performance bottleneck, and mobile apps spend 34-85% of their time fetching data from the Internet [32]. A compounding factor is that mobile devices rely on *wireless* networks, which can exhibit high latency, intermittent connectivity, and low bandwidth [21].

Reducing network latency thus becomes a highly effective way of improving the mobile user experience. In the context of mobile communication, we define *latency* as the response time of an HTTP request. In this paper, we propose a novel client-centric technique for minimizing the network latency by prefetching HTTP requests in mobile apps. Prefetching bypasses the performance bottleneck (in this case, network speed) and masks latency by allowing a response to a request to be generated immediately, from a local cache.

ICSE '18, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5638-1/18/05…$15.00
https://doi.org/10.1145/3180155.3180249

Prefetching has been explored in distributed systems previously. Existing approaches can be divided into four categories based on *what* they prefetch and *when* they do so. *(1) Server-based* techniques analyze the requests sent to the server and provide "hints" to the client on what to prefetch [13, 28, 34, 35]. However, most of today's mobile apps depend extensively on heterogeneous third-party servers. Thus, providing server-side "hints" is difficult, not scalable, or even impossible because app developers do not have control over the third-party servers [42]. *(2) Human-based* approaches rely on developers who have to explicitly annotate application segments that are amenable to prefetching [25, 26]. Such approaches are error-prone and pose significant manual burden on developers. *(3) History-based* approaches build predictive models from prior requests to anticipate what request will happen next [14, 22, 28, 36, 43]. Such approaches require significant time to gather historical data. Additionally, building a precise predictive model based on history is more difficult in today's setting because the context of mobile users changes frequently. *(4) Domain-based* approaches narrow down the problem to one specific domain. For example, approaches that focus on the social network domain [40, 44] only prefetch the constant URLs in tweets based on user behavior and resource constraints. These approaches cannot be applied to mobile apps in general.

To address these limitations of current prefetching approaches, we have developed PALOMA (Program Analysis for Latency Optimization of Mobile Apps), a novel technique that is *client-centric, automatic, domain-independent, and requires no historical data.* In this paper, we focus on native Android apps because of Android's dominant market share [31] and its reliance on event-driven interaction, which is the most popular style used in mobile apps today. Our guiding insight is that an app's *code* can provide a lot of useful information regarding *what* HTTP requests may occur and *when.* In addition, a mobile user usually spends multiple seconds deciding what event to trigger next—a period known as "user think time" [26]—providing an opportunity to prefetch HTTP requests in the background. By analyzing an Android program, we are able to identify HTTP requests and certain user-event sequences (e.g., `onScroll` followed by `onClick`). With that information, we can prefetch requests that will happen next during user think time.

User-event transitions are captured as callback control-flow [46] relationships in PALOMA, and we only perform very targeted, short-term prefetching—a single callback ahead. There are several reasons we opted for this strategy. First, short-term prefetching minimizes the cache-staleness problem that is commonly experienced by longer-term prefetching because the newly updated cache

will be used immediately when the user transitions to the next event. Second, the information needed to send the HTTP requests (e.g., a parameter in an HTTP request that depends on user input) is more likely to be known since the prefetching occurs very close in time to the actual request. Third, short-term prefetching takes advantage of user think time *between callbacks*, which has been shown to be sufficient for prefetching HTTP requests [14, 26]. By contrast, prefetching *within the same callback* would not provide a performance gain since the relevant statements would execute within a few milliseconds of one another.

PALOMA comprises four major elements. *(1) String Analysis* identifies the "prefetchable" requests by interpreting each URL string. *(2) Callback Analysis* determines the prefetching points for the "prefetchable" requests by analyzing callback control-flow relationships. *(3) App Instrumentation* modifies the original app based on the information extracted in the previous phases and outputs an optimized app that prefetches the HTTP requests. *(4) Runtime prefetching* involves the optimized app and a local proxy that is in charge of prefetching HTTP requests and managing the responses. PALOMA's first two elements are adaptations and extensions of existing techniques, while the latter two have been newly developed.

PALOMA has been evaluated for accuracy and effectiveness in two different ways. First, we developed a microbenchmark (MBM) that isolates different prefetching conditions that may occur in an Android app. The MBM can be reused for evaluating similar future approaches. Second, we applied PALOMA on 32 real Android apps. Our evaluation shows that PALOMA exhibits perfect accuracy (in terms of precision and recall) and virtually eliminates user-perceived latency, while introducing negligible runtime overhead.

This paper makes the following contributions: (1) PALOMA, a novel client-side, automated, program analysis-based prefetching technique for mobile apps; (2) a rigorous formulation of program analysis-based prefetching scenarios that addresses "what" and "when" to prefetch; (3) a comprehensive, reusable MBM to evaluate prefetching techniques for Android apps; and (4) the implementation of an open-source, extensible framework for program analysis-based prefetching. PALOMA's source code and supporting materials are publicly available [15].

The paper is organized as follows. Section 2 motivates the problem and defines the terms used by PALOMA. Sections 3 and 4 describe PALOMA's approach and implementation. Sections 5 and 6 detail PALOMA's evaluation using a benchmark and real apps. Section 7 presents related work, and Section 8 concludes the paper.

2 BACKGROUND AND MOTIVATION

In this section, we use a concrete example to introduce the fundamental building blocks and execution model of mobile apps, with a particular focus on Android. We then introduce our insights and motivation, followed by the definition of several key terms.

2.1 Mobile App Example

Mobile apps that depend on network generally involve two key concepts: *events* that interact with user inputs and *network requests* that interact with remote servers. We explain these concepts via Listing 1's simplified code fragment of an Android app that responds to user interactions by retrieving weather information.

Events: In mobile apps, user interactions are translated to internal app events. For instance, a screen tap is translated to an `onClick` event. Each event is, in turn, registered to a particular application UI object with a callback function; the callback function is executed when the event is triggered. For instance in Listing 1, the button object `submitBtn` is registered with an `onClick` event (Line 9), and the corresponding callback function `onClick()` (Line 10) will be executed when a user clicks the button. Similarly, the drop-down box object `cityNameSpinner` is registered with an `onItemSelected` event that has an `onItemSelected()` callback function (Lines 5-7).

Network Requests: Within an event callback function, the app often has to communicate with remote servers to retrieve information. The communication is performed through network requests over the HTTP protocol in most non-realtime apps [12]. Each HTTP request is associated with a URL field that specifies the endpoint of the request. For instance in Listing 1, the `onClick` event callback sends three HTTP requests, each with a unique URL (Lines 12-14).

There are two types of URL values, depending on when the value is known: *static* and *dynamic*. For instance, `favCityId` in Listing 1 is static because its value is obtained statically by reading the application settings (Lines 4, 12). Similarly, `getString("domain")` reads the constant string value defined in an Android resource file [20] (Line 12, 13, 14). In contrast, `cityName` is dynamic since its value depends on which item a user selects from the drop-down box `cityNameSpinner` during runtime (Lines 7, 13). Similarly, `cityId` is also a dynamic URL value (Lines 11, 14).

```
1  class MainActivity {
2    String favCityId, cityName, cityId;
3    protected void onCreate(){
4      favCityId = readFromSetting("favCityId");//static
5      cityNameSpinner.setOnItemSelectedListener(new OnItemSelectedListener(){
6        public void onItemSelected() {
7        cityName = cityNameSpinner.getSelectedItem().toString();//dynamic
8        }});
9      submitBtn.setOnClickListener(new OnClickListener(){
10       public void onClick(){
11         cityId = cityIdInput.getText().toString();//dynamic
12         URL url1 = new URL(getString("domain")+"weather?&cityId="+favCityId);
13         URL url2 = new URL(getString("domain")+"weather?cityName="+cityName);
14         URL url3 = new URL(getString("domain")+"weather?cityId="+cityId);
15         URLConnection conn1 = url1.openConnection();
16         Parse(conn1.getInputStream());
17         URLConnection conn2 = url2.openConnection();
18         Parse(conn2.getInputStream());
19         URLConnection conn3 = url3.openConnection();
20         Parse(conn3.getInputStream());
21         startActivity(DisplayActivity.class);
22     }});
23   }
24 }
```

Listing 1: Code snippet with callbacks and HTTP requests

2.2 Motivation and Challenges

The motivation for PALOMA is that one can significantly reduce the user-perceived latency by prefetching certain network requests. For instance, Listing 1 corresponds to a scenario in which a user selects a city name from the drop-down box `cityNameSpinner` (Line 7), then clicks `submitBtn` (Line 9) to get the city's weather information through an HTTP request. To reduce the time the user will have to wait to receive the information from the remote server, a prefetching scheme would submit that request immediately after the user selects a city name, i.e., before the user clicks the button.

Prefetching HTTP requests is possible for two reasons. First, an HTTP request's destination URL can sometimes be known before the actual request is sent out, such as the static URL `url1` (Line 12) in Listing 1. Second, there is often sufficiently long slack between the

time a request's URL value is known and when the request is sent out, due to other code's execution and the "user think time" [14, 26]. Prefetching in effect "hides" the network latency by overlapping the network requests with the slack period.

The key challenges to efficiently prefetching HTTP requests involve determining (1) which HTTP requests to prefetch, (2) what their destination URL values are, and (3) when to prefetch them. Prior work addressed these challenges by relying on various server hints [13, 28, 34], developer annotations [25, 26], and patterns of historical user behaviors [14, 22, 28, 36, 43]. Our goal is to avoid relying on such external information that may be difficult to obtain, and instead to use only program analysis on the app.

2.3 Terminology

We define several terms needed for describing our approach to program analysis-based prefetching of network requests.

URL Spot is a code statement that creates a URL object for an HTTP request based on a string denoting the endpoint of the request. Example URL Spots are Lines 12, 13, and 14 in Listing 1.

Definition Spot$_{m,n}$ is a code statement where the value of a dynamic URL string is defined, such as Lines 7 and 11 in Listing 1. m denotes the m^{th} substring in the URL string, and n denotes the n^{th} definition of that substring in the code. For example, Line 7 would contain Definition Spot $L7_{3,1}$ for `url2` because `cityName` is the third substring in `url2` and Line 7 is the first definition of `cityName`. A single statement of code may represent multiple Definition Spots, each of which is associated with a dynamic string used in different URLs.

Fetch Spot is a code statement where the HTTP request is sent to the remote server. Example Fetch Spots are Lines 16, 18, and 20.

Callback is a method that is invoked *implicitly* by the Android framework in response to a certain event. Example callbacks from Listing 1 include the `onItemSelected()` (Line 6) and `onClick()` (Line 10) methods. These are referred to as *event handler callbacks* in Android as they respond to user interactions [18]. Android also defines a set of *lifecycle callbacks* that respond to the change of an app's "life status" [17], such as the `onCreate()` method at Line 3.

Call Graph is a directed graph representing the *explicit* invocation relationships between procedures in the app code.

Target Method is a method that contains at least one Fetch Spot. It is named that because identifying methods that contain Fetch Spots is the target of PALOMA's analysis (see Section 3). For example, the `onClick()` method is a Target Method because it contains three Fetch Spots. A Target Method may or may not be a Callback.

Target Callback is a Callback that can reach at least one Target Method in a Call Graph. If a Target Method itself is a Callback, it is also a Target Callback. For example, the `onClick()` Callback defined at Lines 10-22 of Listing 1 is a Target Callback.

Callback Control-Flow Graph (CCFG) represents the *implicit*-invocation flow involving different Callbacks [46]. In a CCFG, nodes represent Callbacks, and each directed edge $f \rightarrow s$ denotes that s is the next Callback invoked after f. Figure 1 illustrates the CCFG extracted from Listing 1 using GATOR, a recently-developed analysis technique [33, 46]. A *wait node* in a CCFG (e.g., `wn1` in Figure 1) indicates that the user's action is required and the event she triggers will determine which one of the subsequent callbacks is invoked.

Trigger Callback is any Callback in the CCFG that is an immediate predecessor of a Target Callback with only a wait node between them. For instance, in Listing 1 the Trigger Callbacks for the Target Callback `onClick()` are `onCreate()` (path 1→2) and `onItemSelected()` (path 5→2). Note that `onClick()` cannot be the Trigger Callback for `DisplayActivity`'s `onCreate()` method (path 3) because there is no wait node between them.

Trigger Point is the program point that triggers the prefetching of one or more HTTP requests.

3 APROACH

This section presents PALOMA, a prefetching-based solution for reducing user-perceived latency in mobile apps that does not require any developer effort or remote server modifications. PALOMA is motivated by the following three challenges: (1) which HTTP requests can be prefetched, (2) what their URL values are, and (3) when to issue prefetching requests. Our guiding insight is that static program analysis can help us address all three challenges. To that end, PALOMA employs an offline-online collaborative strategy shown in Figure 2. The offline component automatically transforms a mobile app into a prefetching-enabled app, while the online component issues prefetching requests through a local proxy.

PALOMA has four major elements. It first performs two static analyses: it (1) identifies HTTP requests suitable for prefetching via string analysis and (2) detects the points for issuing prefetching requests (i.e., Trigger Points) for each identified HTTP request via callback analysis. PALOMA then (3) instruments the app automatically based on the extracted information and produces an optimized, prefetching-enabled app. Finally at runtime, the optimized app will interact with a local proxy deployed on the mobile device. The

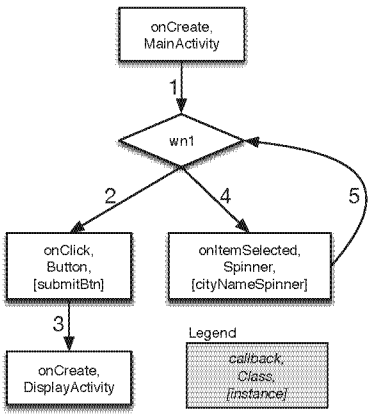

Figure 1: CCFG extracted from Listing 1 by GATOR [33, 46]

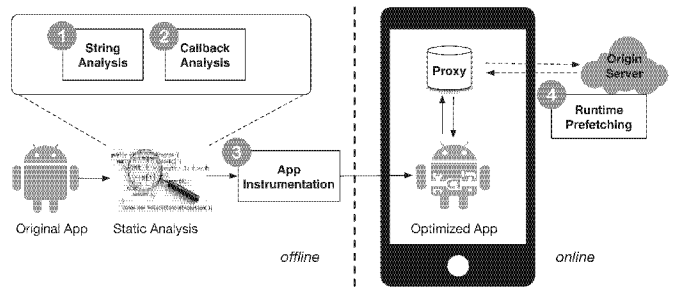

Figure 2: High-level overview of the PALOMA approach

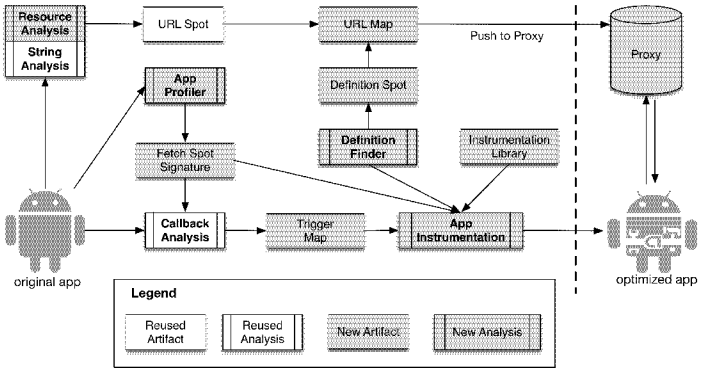

Figure 3: PALOMA's detailed workflow. Different analysis tools employed by PALOMA and artifacts produced by it are depicted, with a distinction drawn between those that are extensions of prior work and newly created ones.

local proxy (4) issues prefetching requests on behalf of the app and caches prefetched resources so that future on-demand requests can be serviced immediately. We detail these four elements next.

3.1 String Analysis

The goal of string analysis is to identify the URL values of HTTP requests. Prefetching can only happen when the destination URL of an HTTP request is known. The key to string analysis is to differentiate between static and dynamic URL values. A static URL value is the substring in a URL whose concrete value can be determined using conventional static analysis. In contrast, a dynamic URL value is the substring in a URL whose concrete value depends on user input. For this reason, we identify the Definition Spots of dynamic URL values and postpone the actual value discovery until runtime.

As Figure 3 shows, the output of string analysis is a URL Map that will be used by the proxy at runtime (Section 3.4), and the Definition Spot in the URL Map will be used by the App Instrumentation step (Section 3.3). The URL Map relates each URL substring with its concrete value (for static values) or Definition Spots (for dynamic values). In the example of Listing 1, the entry in the URL Map that is associated with `url2` would be

`{url2: ["http://weatherapi/", "weather?&cityName=", L7`$_{3,1}$`]}`

We now explain how the URL Map is created for static and dynamic URL values.

Static value analysis – To interpret the concrete value of each static substring, we must find its use-definition chain and propagate the value along the chain. To do that, we leveraged a recent string analysis framework, Violist [24], that performs control- and data-flow analyses to identify the value of a string variable at any given program point. Violist is unable to handle implicit use-definition relationships that are introduced by the Android app development framework. In particular, in Android, string values can be defined in a resource file that is persisted in the app's internal storage and retrieved during runtime. For instance in Listing 1, all three URLs have a substring `getString("domain")` (Lines 12-14), which is defined in the app's resource file [20]. PALOMA extends Violist to properly identify this case and include the app's resource file that is extracted by decompiling the app in the control- and data-flow analysis. In the end, the concrete value of each static substring in each URL is added to the URL Map.

Dynamic value analysis – Dynamic URL values cannot be determined by static analysis. Instead, PALOMA identifies the locations where a dynamic value is defined, i.e., its Definition Spots. The Definition Spots are later instrumented (see Section 3.3) such that the concrete values can be determined at runtime.

The key challenge in identifying the Definition Spots is that a URL string may be defined in a callback different from the callback where the URL is used. Recall that, due to the event-driven execution model, callbacks are invoked *implicitly* by Android. Therefore, the control flow between callbacks on which the string analysis depends cannot be obtained by analyzing the app code statically. Solving the inter-callback data-flow problem is outside the scope of this paper. This is still an open problem in program analysis, because of the implicit control-flow among callbacks as well as the complex and varied types of events that can trigger callbacks at runtime, such as GUI events (e.g., clicking a button), system events (e.g., screen rotation), and background events (e.g., sensor data changes). Research efforts on understanding callbacks are limited to specific objectives that prevent their use for string analysis in general. Such efforts have included a focus restricted to GUI-related callbacks [46, 47] (which we do use in our callback analysis, detailed in Section 3.2), assumption that callback control-flow can be in any arbitrary order [7], and analysis of the Android framework-level, but not app-level, code to construct callback summaries [11, 30].

To mitigate these shortcomings, we developed a hybrid static/dynamic approach, where the static part conservatively identifies all *potential* Definition Spots, leaving to the runtime the determination of which ones are the *actual* Definition Spots. In particular, we focus on the Definition Spots of class fields because a field is a common way to pass data between callbacks. We identify all potential Definition Spots in two ways. First, if a string variable is a private member of a class, we include all the Definition Spots inside that class, such as constructor methods, setter methods, and definitions in the static block. Second, if a variable is a public member of a class, that variable can be defined outside the class and we conduct a whole-program analysis to find all assignments to the variable that propagate to the URL.

At the end of the analysis, all substring Definition Spots for a URL are added to the URL Map. It is worth noting that although the static analysis is conservative and multiple Definition Spots may be recorded in the URL Map, the true Definition Spot will emerge at runtime because false definitions will either be overwritten by a later true definition (i.e., a classic write-after-write dependency) or will never be encountered if they lie along unreachable paths.

3.2 Callback Analysis

Callback analysis determines where to prefetch different HTTP requests, i.e., the Trigger Points in the app code. There may be multiple possible Trigger Points for a given request, depending on how far in advance the prefetching request is sent before the on-demand request is actually issued. The most aggressive strategy would be to issue an HTTP request immediately after its URL value is discovered. However, this approach may lead to many redundant network transmissions: the URL value may not be used in any on-demand requests (i.e., it may be overwritten) or the callback containing the HTTP request (i.e., the Target Callback) may not be reached at runtime at all. In contrast, the most accurate strategy would be to issue

Algorithm 1: IDENTIFYTRIGGERCALLBACKS

```
Input: CCFG, ECG, App
Output: TriggerMap
1  INSTRUMENTTIMESTAMP(App)
2  NetworkMethodLogs ← PROFILE(App)
3  Signature ← GETFETCHSIGNATURE(NetworkMethodLogs)
4  Requests ← GETREQUESTS(Signature)
5  TriggerMap = ∅
6  foreach req ∈ Requests do
7      tarMethod ← GETTARGETMETHOD(req)
8      TargetCallbacks ← FINDENTRIES(tarMethod, ECG)
9      foreach tarCallback ∈ TargetCallbacks do
10         TriggerCallbacks ←
             GETIMDIATEPREDECESSORS(tarCallback, CCFG)
11         foreach trigCallback ∈ TriggerCallbacks do
12             TriggerMap.ADD(trigCallback, req.url)
13 return TriggerMap
```

the prefetching request right before the on-demand request is sent. However, this strategy would yield no improvement in latency.

Our approach is to strike a balance between the two extremes. Specifically, PALOMA issues prefetching requests at the end of the callback that is the immediate predecessor of the Target Callback. Recall from Section 2 that we refer to the Target Callback's immediate predecessor as Trigger Callback, because it triggers prefetching. This strategy has the dual benefit of (1) taking advantage of the "user think time" between two consecutive callbacks to allow prefetching to take place, while (2) providing high prefetching accuracy as the Trigger Point is reasonably close to the on-demand request.

As Figure 3 shows, PALOMA creates a Trigger Map at the end of callback analysis that is used by app instrumentation (Section 3.3). The Trigger Map maps each Trigger Callback to the URLs that will be prefetched at the end of that callback. In the example of Listing 1, the Trigger Map will contain two entries:

{ [`onCreate`]: [`url1`, `url2`, `url3`] }
{ [`onItemSelected`]: [`url1`, `url2`, `url3`] }

because both `onCreate()` and `onItemSelected()` are Trigger Callbacks that are the immediate predecessors of the Target Callback `onClick()`, which in turn contains `url1`, `url2`, and `url3`.

Algorithm 1 details how PALOMA identifies Trigger Callbacks and constructs the Trigger Map. In addition to the app itself, the algorithm relies on two additional inputs, both obtained with the help of off-the-shelf-tools: the Callback Control-Flow Graph (CCFG) [46] and the Call Graph (CG) [37]. Note that the CCFG we use in our callback analysis is restricted to GUI callbacks that are triggered by user actions (recall Section 2). However, this fits PALOMA's needs given its focus on user-initiated network requests. The CCFG captures the implicit-invocation flow of Callbacks in Android, and thus allows us to find the Trigger Callbacks of a given Target Callback. On the other hand, the CG, which is extracted by Soot [37], captures the control flow between functions, and thus allows us to locate the Callbacks that contain any given method. However, the CG does not include the direct invocations that are initiated from the Android framework. We identified such invocations from Android's documentation and extended the CG with the resulting direct edges. An example is the `execute()`→`doInBackground()` edge from the `AsyncTask` class [16] that is widely used for network operations in Android. We refer to the thus extended CG as ECG.

Given these inputs, PALOMA first identifies the signature of a Fetch Spot, i.e., the method that issues HTTP requests, by profiling the app (Lines 1-3 of Algorithm 1). We found that the profiling is needed because the methods that actually issue HTTP requests under different circumstances can vary across apps. For example, the `getInputStream()` method from Java's `URLConnection` library may consume hundreds of milliseconds in one app, but zero in another app where, e.g., the `getResponseCode()` method consumes several hundred milliseconds.[1] Thus, we obtain the signatures by instrumenting timestamps in the app, and select the most time-consuming network operations according to our profiling results. Using the signatures, we then identify all HTTP requests that the app can possibly issue (Line 4). In the example of Listing 1, the signature would be `getInputStream()` and the Requests would be `conn1.getInputStream()`, `conn2.getInputStream()`, and `conn3.getInputStream()`. We iterate through each discovered request and identify the method in which the request is actually issued, i.e., the Target Method (Line 7). Using the control flow information that the ECG provides, we locate all possible Target Callbacks of a Target Method (Line 8). We then iterate through each Target Callback and identify all of its immediate predecessors, i.e., Trigger Callbacks, according to the CCFG (Line 10). Finally, we add each {Trigger Callback, URL} pair to the Trigger Map (Lines 11-12).

3.3 App Instrumentation

PALOMA instruments an app automatically based on the information extracted from the two static analyses, and produces an optimized, prefetching-enabled app as Figure 3 shows. At runtime, the optimized app will interact with a local proxy that is in charge of issuing prefetching requests and managing the prefetched resources (Section 3.4). While PALOMA's app instrumentation is fully automated and it does not require the source code of the app, PALOMA also supports app developers who have the knowledge and the source code of the app to further improve runtime latency reduction via simple prefetching hints. We describe the two instrumentation aspects next.

3.3.1 Automated Instrumentation

PALOMA performs three types of instrumentation automatically. Each type introduces a new API that we implement in an instrumentation library. Listing 2 shows an instrumented version of the app from Listing 1, with the instrumentation code bolded. We will use this example to explain the three instrumentation tasks.

1. Update URL Map – This instrumentation task updates the URL Map as new values of *dynamic* URLs are discovered. Recall that the values of *static* URLs are fully determined and inserted into the URL Map offline. This instrumentation is achieved through a new API, `sendDefinition(var, url, id)`, which indicates that `var` contains the value of the $\texttt{id}^{th}$ substring in the URL named `url`. The resulting annotation is inserted right after each Definition Spot. For instance at Line 8 of Listing 2, PALOMA will update the

[1] In this paper, we focus on `URLConnection`, a built-in Java standard library widely used by Android developers. If the developer is using a different library and/or knows which method(s) to optimize, then PALOMA's profiling step may not be needed.

third substring in `url2` with the runtime value of `cityName`. This ensures that the URL Map will maintain a fresh copy of each URL's value and will be updated as soon as new values are discovered.

2. Trigger Prefetching – This instrumentation task triggers prefetching requests at each Trigger Point. A Trigger Point in PALOMA is at the end of a Trigger Callback. We made this choice for two reasons: on one hand, it makes no discernible difference in terms of performance where we prefetch within the same callback; on the other hand, placing the Trigger Point at the end is more likely to yield known URLs (e.g., when the Definition Spot is also within the Trigger Callback). PALOMA provides this instrumentation via the `triggerPrefetch(url1, ...)` API. The URLs that are to be prefetched are obtained from the Trigger Map constructed in the callback analysis (recall Section 3.2). For instance, PALOMA triggers the proxy to prefetch `url1`, `url2`, and `url3` at the end of `onItemSelected()` (Line 9) and `onCreate()` (Line 26) of Listing 2, which is consistent with the Trigger Map built in Section 3.2.

3. Redirect Requests – This instrumentation task redirects all on-demand HTTP requests to PALOMA's proxy instead of the origin server. This allows on-demand requests to be served from the proxy's cache, without latency-inducing network operations. The cases where the proxy's cache does not contain the response to a request are discussed in Section 3.4. The request redirection is achieved through the `fetchFromProxy(conn)` API, where `conn` indicates the original URL connection, which is passed in case the proxy still needs to make the on-demand request to the origin server. This instrumentation replaces the original methods at each Fetch Spot: calls to the `getInputStream()` method at Lines 16, 18, and 20 of Listing 1 are replaced with calls to the `fetchFromProxy(conn)` method at Lines 19, 21, and 23 in Listing 2.

3.3.2 Developer Hints

Although PALOMA can automatically instrument mobile apps without developer involvement, it also provides opportunities for developers to add hints in order to better guide the prefetching. In particular, PALOMA enables two ways for developers to provide hints: by using its instrumentation APIs and by directly modifying its artifacts. These two approaches are described below.

```
1  class MainActivity {
2     String favCityId, cityName, cityId;
3     protected void onCreate(){
4        favCityId = readFromSetting("favCityId");//static
5        cityNameSpinner.setOnItemSelectedListener(new OnItemSelectedListener(){
6           public void onItemSelected() {
7           cityName = cityNameSpinner.getSelectedItem().toString();//dynamic
8           sendDefinition(cityName, url2, 3);
9           triggerPrefetch(url1, url2, url3);
10          }});
11       submitBtn.setOnClickListener(new OnClickListener(){
12          public void onClick(){
13             cityId = cityIdInput.getText().toString();//dynamic
14             sendDefinition(cityId, url3, 3);
15             URL url1 = new URL(getString("domain")+"weather?&cityId="+favCityId);
16             URL url2 = new URL(getString("domain")+"weather?cityName="+cityName);
17             URL url3 = new URL(getString("domain")+"weather?cityId="+cityId);
18             URLConnection conn1 = url1.openConnection();
19             Parse(fetchFromProxy(conn1));
20             URLConnection conn2 = url2.openConnection();
21             Parse(fetchFromProxy(conn2));
22             URLConnection conn3 = url3.openConnection();
23             Parse(fetchFromProxy(conn3));
24             startActivity(DisplayActivity.class);
25       }});
26       triggerPrefetch(url1, url2, url3);
27    }
28 }
```

Listing 2: Example code of the optimized app

API support – PALOMA's three API functions defined in the instrumentation library—`sendDefinition()`, `triggerPrefetch()`, and `fetchFromProxy()`—can be invoked by the developers explicitly in the app code. For instance, if a developer knows where the true Definition Spots are, she can invoke `sendDefinition()` at those locations. Developers can also invoke `triggerPrefetch()` at any program point. For example, prefetching can happen farther ahead than is done automatically by PALOMA if a developer knows that the responses to a prefetching request and its corresponding on-demand request will be identical.

Artifact modification – Using PALOMA's instrumentation APIs in the manner described above requires modifications to the app source code. An alternative is to directly modify the artifacts generated by PALOMA's static analyses—Trigger Map, Fetch Spot Signature, and Definition Spot (recall Figure 3)—without altering the code. For example, a developer can add an entry in the Trigger Map; as a result, PALOMA's instrumenter will automatically insert a call to `triggerPrefetch()` at the end of the Trigger Callback specified by the developer.

We now introduce two frequently occurring instances where developers are well positioned to provide prefetching hints with very little manual effort. These hints can be provided using either of the above two approaches.

Prefetching at app launch – Launching an app may take several seconds or more because many apps request remote resources, typically toward the end of the launch process. The URLs of the launch-time requests are usually statically known, but the ways in which the URL values can be obtained are highly app-dependent. For instance, apps may retrieve the request URLs from a configuration file or a local database. Supporting those cases in PALOMA's string analysis would mean that PALOMA must understand the semantics of each individual app, which is not a reasonable requirement. However, a practical alternative is for developers to provide prefetching hints because they understand their own apps' behavior. One way developers could implement this is to insert into the URL Map additional static URLs and then call `triggerPrefetch()` at the beginning of `onCreate()`, which for PALOMA's purposes can be treated as the app entry point in most Android applications.

Prefetching for ListView – The `ListView` class [19] is commonly used in Android apps to display the information of a list of items. The app "jumps" to another page to display further information based on the item a user selects in the list. The URL fetched for the page to which the app "jumps" is typically only known after the user selects the item in the list. Ordinarily, this would prevent prefetching. However, Android apps tend to exhibit two helpful trends. First, the list view usually displays similar types of information. Second, the further information obtained by selecting an item is related to the information displayed in the list itself. Based on these observations, we identified and are exploiting in PALOMA similar patterns in the URLs for the list and the subsequent page. Consider a wallpaper app for illustration: The URL that is fetched to render an item in the list view may be "image1Url_small.jpg", while the URL that is fetched after the user selects `image1` may be "image1Url_large.jpg". Based on this pattern, we have explored manually adding Definition Spots of the URLs that are fetched in

Algorithm 2: TriggerPrefetch

```
   Input: Requests
1  foreach req ∈ Requests do
2      if IsKnown(req.url) and ¬IsCached(req) then
3          SetWaitFlag(req)
4          response ← req.FetchRemoteResponse()
5          cache.Put(req, response)
6          UnWait(req)
```

the list view and sending modified values to the proxy, such as replacing "small" with "large" in the wallpaper example.

3.4 Runtime Prefetching

PALOMA's first three phases are performed offline. By contrast, this phase captures the interplay between the optimized apps and PALOMA's proxy to prefetch the HTTP requests at runtime. The instrumented methods in an optimized app trigger the proxy to perform corresponding functions. We now use the example from Listing 2 to show how the three instrumented functions from Section 3.3.1 interact with the proxy.

1. Update URL Map – When the concrete value of the *dynamic* URL is obtained at runtime, the inserted instrumentation method `sendDefinition(var, url, id)` is executed and the concrete runtime value is sent to the proxy. In response, the proxy updates the corresponding URL value in the URL Map. For instance in Listing 2, when a user selects a city name from the `cityNameSpinner` (Line 7), the concrete value of `cityName` will be known, e.g., "Gothenburg". Then `cityName` is sent to the proxy (Line 8) and the URL Map entry for `url2` will be updated to `{url2: ["http://weatherapi/", "weather?&cityName=","Gothenburg"]}`.

2. Trigger Prefetching – When the inserted instrumentation method `triggerPrefetch(url1,...)` is executed, it triggers the proxy to perform TriggerPrefetch as shown in Algorithm 2. For each request that is sent to the proxy by `triggerPrefetch (url1,...)`, the proxy checks if the whole URL of the request is known but the response to the request has not yet been cached (Line 2). If both conditions are met, a "wait" flag is set in the cache for that request (Line 3). This ensures that duplicated requests will not be issued in the case when the on-demand request is made by the user before the response to the prefetching request has been returned from the origin server. In the example of Listing 2, when the app reaches the end of `onCreate` (Line 26), it triggers the proxy to perform `TriggerPrefetch(url1,url2,url3)`. Only `url1` meets both conditions at Line 2 of Algorithm 2: the URL value is concrete (it is, in fact, a *static* value) and the response is not in the cache. The proxy thus sets the "wait" flag for `url1` in the cache, prefetches `url1` from the origin server, stores the response in the cache, and finally sends an "unwait" signal to the on-demand request that is waiting for the prefetched request (Line 3-6). Thereafter, when the user selects a city name from the dropdown box, `onItemSelected` (Line 6 of Listing 2) will be triggered. At the end of `onItemSelected` (Line 9), `TriggerPrefetch(url1,url2,url3)` is invoked again and `url2` will be prefetched because its URL is known (its *dynamic* value obtained at Line 8) and has not been previously prefetched. In contrast, the value of `url1` is known at this point but `url1` was already prefetched at Line 26, so the proxy will not prefetch `url1`.

Algorithm 3: ReplacedFetch

```
   Input: req ∈ Requests
   Output: response ∈ Responses
1  if IsCached(req) then
2      if GetWaitFlag(req) is TRUE then
3          Wait(req)
4      return cache.GetResponse(req)
5  else
6      response ← req.FetchRemoteResponse()
7      cache.Put(req, response)
8      return response
```

3. Redirect Requests – When the on-demand request is sent at the Fetch Spot, the replaced function `fetchFromProxy(conn)` will be executed, and it will in turn trigger the proxy to perform ReplacedFetch as shown in Algorithm 3. If the request has a corresponding response in the cache, the proxy will first check the "wait" flag for the request. If the flag is set, the proxy will wait for the signal of the prefetching request (Line 3) and will return the response of the prefetching request when it is back from the origin server (Line 4). If the "wait" flag has not been set, the response is already in the cache and the proxy returns the response immediately with no network operations involved (Line 4). Otherwise, if the cache does not contain the response to the request, the proxy issues an on-demand request using the original URL connection `conn` to fetch the response from the origin server, stores the response in the cache, and returns the response to the app (Line 6-8). For instance in Listing 2, if a user clicks `submitBtn`, `fetchFromProxy(conn)` will be executed to send on-demand requests for `url1`, `url2`, `url3` to the proxy (Lines 19, 21, and 23 of Listing 2). The proxy in turn returns the responses to `url1` and `url2` from the local cache immediately because `url1` and `url2` are prefetched at Lines 26 and 9 respectively, as discussed above. `url3` is not known at any of the Trigger Points, so the response to `url3` will be fetched from the origin server on demand as in the original app. Note that if a user did not select a city name from the dropdown box before clicking `submitBtn`, `onItemSelected` will not be triggered, meaning that Lines 8 and 9 of Listing 2 will not be executed. In this case, only the response for `url1` will be returned from the cache (prefetched at Line 26) while the on-demand requests for `url2` and `url3` will be routed to the origin server.

4 IMPLEMENTATION

PALOMA has been implemented by reusing and extending several off-the-shelf tools, and integrating them with newly implemented functionality. PALOMA's string analysis extends the string analysis framework Violist [24]. The callback analysis is implemented on top of the program analysis toolkit GATOR [33], and by extending GATOR's CCFG analysis [46]. PALOMA's instrumentation component is a stand-alone Java program that uses Soot [37] to instrument an app. The proxy is built on top of the Xposed framework [5] that provides mechanisms to "hook" method calls. The proxy intercepts the methods that are defined in PALOMA's instrumentation library and replaces their bodies with corresponding methods implemented in the proxy. The total amount of newly added code to extend existing tools, implement the new functionality, and integrate them together in PALOMA is 3,000 Java SLOC.

5 MICROBENCHMARK EVALUATION

In this section, we describe the design of a microbenchmark (MBM) containing a set of test cases, which we used to evaluate PALOMA's *accuracy* and *effectiveness*.

MBM thoroughly covers the space of prefetching options, wherein each test case contains a single HTTP request and differs in whether and how that request is prefetched. The MBM is restricted to individual HTTP requests because the requests are issued and processed independently of one another. This means that PALOMA will process multiple HTTP requests simply as a sequence of individual requests; any concurrency in their processing that may be imposed by the network library and/or the OS is outside PALOMA's purview. In practice, the look-up time for multiple requests varies slightly from one execution of a given app to the next. However, as shown in Section 5.3, the look-up time required by PALOMA would not be noticeable to a user even with a large number of requests. As we will show in Section 6, the number of HTTP requests in real apps is typically bounded. Moreover, PALOMA only maintains a small cache that is emptied every time a user quits the app.

In the rest of this section, we will first lay out the goals underlying the MBM's design (Section 5.1), and then present the MBM (Section 5.2). Our evaluation results show that PALOMA achieves perfect accuracy when applied on the MBM, and leads to significant latency reduction with negligible runtime overhead (Section 5.3).

5.1 Microbenchmark Design Goals

The MBM is designed to evaluate two fundamental aspects of PALOMA: *accuracy* and *effectiveness*.

PALOMA's **accuracy** pertains to the relationship between *prefetchable* and actually *prefetched* requests. Prefetchable requests are requests whose URL values are known before the Trigger Point and thus can be prefetched. Quantitatively, we capture accuracy via the dual measures of *precision* and *recall*. Precision indicates how many of the requests that PALOMA tries to prefetch at a given Trigger Point were actually prefetchable. On the other hand, recall indicates how many requests are actually prefetched by PALOMA out of all the prefetchable requests at a given Trigger Point.

PALOMA's **effectiveness** is also captured by two measures: the runtime overhead introduced by PALOMA and the latency reduction achieved by it. Our objective is to minimize the runtime overhead while maximizing the reduction in user-perceived latency.

5.2 Microbenchmark Design

The MBM is built around a key concept—*prefetchable*—a request whose whole URL is known before a given Trigger Point. We refer to the case where the request is prefetchable and the response is used by the app as a *hit*. Alternatively, a request may be prefetchable but the response is not used because the definition of the URL is changed after the Trigger Point. We call this a *non-hit*. The MBM aims to cover all possible cases of *prefetchable* and *non-prefetchable* requests, including *hit* and *non-hit*.

There are three factors that affect whether an HTTP request is prefetchable: (1) the number of dynamic values in a URL; (2) the number of Definition Spots for each dynamic values; and (3) the location of each Definition Spot relative to the Trigger Point. We now formally define the properties of *prefetchable* and *hit* considering the three factors. The formal definitions will let us succinctly describe test cases later.

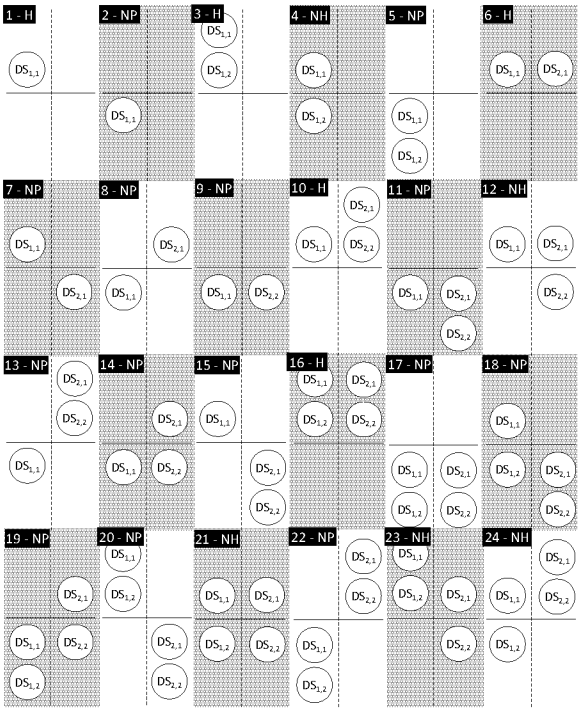

Figure 4: The 24 test cases covering all configurations involving dynamic values. The horizontal divider denotes the Trigger Point, while the vertical divider delimits the two dynamic values. The circles labeled with "$DS_{i,j}$" are the locations of the Definition Spots with respect to the Trigger Point. "H" denotes a *hit*, "NH" denotes a *non-hit*, and "NP" denotes a *non-prefetchable* request.

Formal Definition. Let M be the set of Definition Spots before the Trigger Point and N the set of Definition Spots after the Trigger Point, which is within the Target Callback (recall Sections 2 and 3.2). Let us assume that a URL has $k \geq 1$ dynamic values. (The case where $k = 0$, i.e., the whole URL is static, is considered separately.) Furthermore, let us assume that the dynamic values are the first k values in the URL.[2] The i^{th} dynamic value ($1 \leq i \leq k$) has $d_i \geq 1$ Definition Spots in the whole program. A request is

- *prefetchable* iff $\forall i \; \exists (j \in [1..d_i]) \mid \text{DefSpot}_{i,j} \in M$
  (every dynamic value has a DefSpot before Trigger Point)
- *hit* iff *prefetchable* $\wedge \; \forall (j \in [2..d_i]) \mid \text{DefSpot}_{i,j} \in M$
  (all dynamic value DefSpots are before Trigger Point)
- *non-hit* iff *prefetchable* $\wedge \; \exists (j \in [2..d_i]) \mid \text{DefSpot}_{i,j} \in N$
  (some dynamic value DefSpots are after Trigger Point)
- *non-prefetchable* iff $\forall (j \in [1..d_i]) \; \exists i \mid \text{DefSpot}_{i,j} \in N$
  (all DefSpots for a dynamic value are after Trigger Point)

Without loss of generality, MBM covers all cases where $k \leq 2$ and $d_i \leq 2$. We do not consider cases where $k > 2$ or $d_i > 2$ because we only need two dynamic values to cover the *non-prefetchable* case—where some dynamic values are unknown at the Trigger Point—and two Definition Spots to cover the *non-hit* case—where some dynamic values are redefined after the Trigger Point.

[2]This assumption is used only to simplify our formalization. The order of the values in a URL has no impact on whether the URL is prefetchable and can thus be arbitrary.

There are a total of 25 possible cases involving configurations with $k \leq 2$ and $d_i \leq 2$. The simplest case is when the entire URL is known statically; we refer to it as case 0. The remaining 24 cases are diagrammatically encoded in Figure 4: the two dynamic URL values are depicted with circles and delimited with the vertical line; the location of the Trigger Point is denoted with the horizontal line; and the placement of the circles marks the locations of the dynamic values' Definition Spots ("$DS_{i,j}$" in the figure) with respect to the Trigger Point. These 24 cases can be grouped as follows:

- single dynamic value – cases 1-5;
- two dynamic values, one Definition Spot each – cases 6-9;
- two dynamic values, one with a single Definition Spot, the other with two – cases 10-15; and
- two dynamic values, two definition spots each – cases 16-24.

Each case is labeled with its *prefetchable/hit* property ("H" for *hit*, "NH" for *non-hit*, and "NP" for *non-prefetchable*). Of particular interest are the six cases—0, 1, 3, 6, 10, and 16—that represent the *hit*s that should allow PALOMA to prefetch the corresponding requests and hence significantly reduce the user-perceived latency.

5.3 Results

We implemented the MBM as a set of Android apps along with the remote server to test each of the 25 cases. The server is built with Node.js and deployed on the Heroku cloud platform [3]. The apps interact with the server to request information from a dataset in MongoDB [4]. The evaluation was performed on the 4G network. The testing device was Google Nexus 5X running Android 6.0. Overall, our evaluation showed that PALOMA achieves 100% precision and recall without exception, introduces negligible overhead, and can reduce the latency to nearly zero under appropriate conditions (the *hit* cases discussed above).

Table 1 shows the results of each test case corresponding to Figure 4, as well as case 0 in which the entire URL value is known statically. Each execution value is the average of multiple executions of the corresponding MBM apps. The highlighted test cases are the *hit* cases that should lead to a significant latency reduction. The columns "SD", "TP", and "FFP" show the average times spent in the corresponding PALOMA instrumentation methods in the optimized apps—`sendDefinition()`, `triggerPrefetch()`, and `fetchFromProxy()`, respectively (recall Section 3.3). The "Orig" column shows the execution time of the method invoked at the Fetch Spot in the original app, such as `getInputStream()`.

The final column in the table, labeled "Red/OH" shows the percentage reduction in execution time when PALOMA is applied on each MBM app. The reduction is massive in each of the six *hit* cases (≥99%). It was interesting to observe that applying PALOMA also resulted in reduced average execution times in 11 of the 19 *non-hit* and *non-prefetchable* cases. The more expected scenario occurred in the remaining eight of the *non-hit* and *non-prefetchable* cases: applying PALOMA introduced an execution overhead (shown as negative values in the table). The largest runtime overhead introduced by PALOMA was 149ms in case 11, where the original response time was 2,668ms. This value was due to a couple of outliers in computing the average execution time, and it may be attributable to factors in our evaluation environment other than PALOMA, such as network speed; the remaining measurements were significantly lower. However, even this value is actually not prohibitively expensive: recall that PALOMA is intended to be applied in cases in which a user already typically spends multiple seconds deciding what event to trigger next [26].

Table 1: Results of PALOMA's evaluation using MBM apps covering the 25 cases discussed in Section 5.2. "SD", "TP", and "FFP" denote the runtimes of the three PALOMA instrumentation methods. "Orig" is the time required to run the original app. "Red/OH" represents the reduction/overhead in execution time when applying PALOMA.

Case	SD (ms)	TP (ms)	FFP (ms)	Orig (ms)	Red/OH
0	N/A	2	**1**	1318	99.78%
1	0	5	**0**	15495	99.97%
2	0	1	2212	2659	16.81%
3	1	4	**1**	781	99.24%
4	2	5	611	562	-9.96%
5	0	2	2588	2697	3.97%
6	1	4	**2**	661	98.95%
7	1	4	2237	2399	6.54%
8	1	9	585	568	4.75%
9	2	2	611	584	-5.31%
10	1	5	**0**	592	98.99%
11	2	2	2813	2668	-5.58%
12	2	6	546	610	8.16%
13	2	3	2478	2753	10.87%
14	3	3	549	698	20.49%
15	5	1	631	570	-11.75%
16	1	11	**0**	8989	99.87%
17	0	3	418	555	31.83%
18	2	6	617	596	-4.87%
19	4	6	657	603	-10.61%
20	1	3	620	731	17.15%
21	2	10	611	585	-6.50%
22	2	7	737	967	29.62%
23	2	9	608	607	-1.98%
24	1	10	611	715	14.95%

6 THIRD-PARTY APP EVALUATION

We also evaluated PALOMA on third-party Android apps to observe its behavior in a real-world setting. We used the same execution setup as in the case of the MBM. We selected 32 apps from the Google Play store [1]. We made sure that the selected apps span a range of application categories—Beauty, Books & Reference, Education, Entertainment, Finance, Food & Drink, House & Home, Maps & Navigation, Tools, Weather, News & Magazines, and Lifestyle—and vary in sizes—between 312KB and 17.8MB. The only other constraints in selecting the apps were that they were executable, relied on the Internet, and could be processed by Soot.[3]

We asked two Android users to actively use the 32 subject apps for two minutes each, and recorded the resulting usage traces.[4] We then re-ran the same traces on the apps multiple times, to account for variations caused by the runtime environment. Then we instrumented the apps using PALOMA and repeated the same steps the same number of times. Each session started with app (re)installation and exposed all app options to users. As in the case of the MBM, we measured and compared the response times of

[3]Soot is occasionally unable to process an Android app for reasons that we were unable to determine. This issue was also noted by others previously.

[4]While the average app session length varies by user and app type (e.g., [2]), two minutes was sufficiently long to observe representative behavior and, if necessary, to extrapolate our data to longer sessions.

Table 2: Results of PALOMA's evaluation across the 32 third-party apps.

	Min.	Max.	Avg.	Std. Dev.
Runtime Requests	1	64	13.28	14.41
Hit Rate	7.7%	100%	47.76%	28.81%
Latency Reduction	87.41%	99.97%	98.82%	2.3%

the methods at the Fetch Spots between the original and optimized apps.

Unlike in the case of the MBM, we do not have the ground-truth data for the third-party apps. Specifically, the knowable URLs at the Trigger Points would have to be determined manually, which is prohibitively time-consuming and error prone. In fact, this would boil down to manually performing inter-callback data-flow analysis (recall Section 3.1). For this reason, we measured only two aspects of applying PALOMA on the third-party apps: the *hit rate* (i.e., the percentage of requests that have been *hit* out of all triggered requests) and the resulting *latency reduction*. Table 2 depicts the averages, outliers (min and max values), as well as the standard deviations obtained across all of the runs of the 32 apps.

Overall, the results show that PALOMA achieves a significant latency reduction with a reasonable hit rate. There are several interesting outlier cases. The minimum hit-rate is only 7.7%. The reason is that the app in question fetches a large number of ads at runtime whose URLs are non-deterministic, and only a single static URL is prefetched outside those. There are four additional apps whose hit rate is below 20% because those apps are list-view apps, such as a wallpaper app (recall Section 3.3), and they fetch large numbers of requests at the same time. In PALOMA, we set the threshold for the maximum number of requests to prefetch at once to be 5. This parameter can be increased, but that may impact device energy consumption, cellular data usage, etc. This is a trade-off that will require further study.

Similarly to the MBM evaluation, PALOMA achieves a reduction in latency of nearly 99% on average for "hit" cases. Given the average execution time for processing a single request across the 32 unoptimized apps of slightly over 800ms, prefetching the average of 13.28 requests at runtime would reduce the total app execution time by nearly 11s, or 9% of a two-minute session. Note that the lowest latency reduction was 87.41%. This was caused by on-demand requests that happen before the prefetching request is returned (recall the discussion in Section 3.4). In those cases, the response time depends on the remaining wait time for the prefetching request's return. However, there were only 5 such "wait" requests among 425 total requests in the 32 apps. This strongly suggests that PALOMA's choice for the placement of Trigger Points is effective in practice.

7 RELATED WORK

Prefetching of HTTP requests has been applied successfully in the browser domain [26, 27, 38, 39]. Unfortunately, approaches targeting page load times cannot be applied to mobile apps. The bottleneck for page load times is resource loading [41], because one initial HTTP request will require a large number of subresources (e.g., images), which can only be discovered after the main source is fetched and parsed. Thus, existing research efforts have focused on issues such as prefetching subresources [34, 42], developer support for speculative execution [26], and restructuring the page load process [39]. In mobile apps, the HTTP requests are always lightweight [23]: one request only fetches a single resource that does not require any further subresource fetching. Therefore, our work focuses on prefetching the future requests that a user may trigger rather than the subresources within a single request.

Researchers have recently begun exploring prefetching in the mobile app domain. One research thread has attempted to answer "how much" to prefetch under different contexts (e.g., network conditions) [10, 21, 48], while assuming that "what" to prefetch is handled by the apps already. Another thread of work focuses on fast prelaunching by trying to predict what app the user will use next [8, 29, 45]. By contrast, our work aims to provide an automated solution to determine "what" and "when" to prefetch for a given app in a general case. As discussed previously, other comparable solutions—server-based [13, 28, 34], human-based [25, 26], history-based [14, 22, 28, 36, 43], and domain-based [9, 40, 44]—have limitations which we directly target in PALOMA.

To the best of our knowledge, PALOMA is the first technique to apply program analysis to prefetching HTTP requests in mobile apps in order to reduce user-perceived latency. Bouquet [23] has applied program analysis techniques to bundle HTTP requests in order to reduce energy consumption in mobile apps. Bouquet detects *Sequential HTTP Requests Sessions* (SHRS), in which the generation of the first request implies that the following requests will also be made, and then bundles the requests together to save energy. This can be considered a form of prefetching. However, this work does not address inter-callback analysis and the SHRS are always in the same callback. Therefore, the "prefetching" only happens a few statements ahead (within milliseconds most of the time) and has no tangible effect on app execution time.

8 CONCLUSION AND FUTURE WORK

We have presented PALOMA, a novel program analysis-based technique that reduces the user-perceived latency in mobile apps by prefetching certain HTTP requests. While PALOMA cannot be applied to all HTTP requests an app makes at runtime, it provides significant performance savings in practice. Several of PALOMA's current facets make it well suited for future work in this area, both by us and by others. For instance, PALOMA defines formally the conditions under which the requests are prefetchable. This can lead to guidelines that developers could apply to make their apps more amenable to prefetching, and lay the foundations for further program analysis-based prefetching techniques. We have also identified several shortcomings to PALOMA whose remedy must include improvements to string analysis and callback analysis techniques. Another interesting direction is to improve the precision and reduce the waste associated with prefetching by incorporating certain dynamic information (e.g., user behavior patterns, runtime QoS conditions). Finally, PALOMA's microbenchmark (MBM) forms a foundation for standardized empirical evaluation and comparison of future efforts in this area.

9 ACKNOWLEDGMENT

We would like to thank William G.J. Halfond, Atanas Rountev, Yuhao Zhu, and their research groups. This work is supported by the U.S. National Science Foundation under grants no. CCF-1618231 and CCF-1717963, U.S. Office of Naval Research under grant no. N00014-17-1-2896, and by Huawei Technologies Co., Ltd.

REFERENCES

[1] [n. d.]. Google Play App Store. http://play.google.com/store/apps. ([n. d.]).
[2] 2015. Average mobile app session length as of 4th quarter 2015. (2015). https://www.statista.com/statistics/202485/average-ipad-app-session-length-by-app-categories/
[3] 2017. Heroku. (2017). https://www.heroku.com/
[4] 2017. MongoDB. (2017). https://docs.mongodb.com/getting-started/shell/import-data/
[5] 2017. Xposed Framework. (2017). http://repo.xposed.info/
[6] AppDynamics. 2014. The app attention span. (2014).
[7] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *ACM SIGPLAN Notices* 49, 6 (2014), 259–269.
[8] Ricardo Baeza-Yates, Di Jiang, Fabrizio Silvestri, and Beverly Harrison. 2015. Predicting The Next App That You Are Going To Use. In *Proceedings of the Eighth ACM International Conference on Web Search and Data Mining (WSDM '15)*. ACM, New York, NY, USA, 285–294. https://doi.org/10.1145/2684822.2685302
[9] Leilani Battle, Remco Chang, and Michael Stonebraker. 2016. Dynamic Prefetching of Data Tiles for Interactive Visualization. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. ACM, New York, NY, USA, 1363–1375. https://doi.org/10.1145/2882903.2882919
[10] Paul Baumann and Silvia Santini. 2017. Every Byte Counts: Selective Prefetching for Mobile Applications. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 1, 2, Article 6 (June 2017), 29 pages. https://doi.org/10.1145/3090052
[11] Yinzhi Cao, Yanick Fratantonio, Antonio Bianchi, Manuel Egele, Christopher Kruegel, Giovanni Vigna, and Yan Chen. 2015. EdgeMiner: Automatically Detecting Implicit Control Flow Transitions through the Android Framework.. In *NDSS*.
[12] Shuaifu Dai, Alok Tongaonkar, Xiaoyin Wang, Antonio Nucci, and Dawn Song. 2013. Networkprofiler: Towards automatic fingerprinting of android apps. In *INFOCOM, 2013 Proceedings IEEE*. IEEE, 809–817.
[13] B De La Ossa, JA Gil, Julio Sahuquillo, and Ana Pont. 2007. Improving web prefetching by making predictions at prefetch. In *Next Generation Internet Networks, 3rd EuroNGI Conference on*. IEEE, 21–27.
[14] Li Fan, Pei Cao, Wei Lin, and Quinn Jacobson. 1999. Web prefetching between low-bandwidth clients and proxies: potential and performance. In *ACM SIGMETRICS Performance Evaluation Review*, Vol. 27. ACM, 178–187.
[15] SoftArch Research Group. 2018. PALOMA Project Website. (2018). https://softarch.usc.edu/PALOMA
[16] Android Developers Guide. 2017. Android AsyncTask. (2017). https://developer.android.com/reference/android/os/AsyncTask.html
[17] Android Developers API Guides. 2017. The Activity Lifecycle. (2017). https://developer.android.com/guide/components/activities/activity-lifecycle.html
[18] Android Developers API Guides. 2017. Android Input Events. (2017). https://developer.android.com/guide/topics/ui/ui-events.html
[19] Android Developers API Guides. 2017. Android ListView. (2017). https://developer.android.com/guide/topics/ui/layout/listview.html
[20] Android Developers API Guides. 2017. String Resources. (2017). https://developer.android.com/guide/topics/resources/string-resource.html
[21] Brett D Higgins, Jason Flinn, Thomas J Giuli, Brian Noble, Christopher Peplin, and David Watson. 2012. Informed mobile prefetching. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*. ACM, 155–168.
[22] Vassilis Kostakos, Denzil Ferreira, Jorge Goncalves, and Simo Hosio. 2016. Modelling Smartphone Usage: A Markov State Transition Model. In *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp '16)*. ACM, New York, NY, USA, 486–497. https://doi.org/10.1145/2971648.2971669
[23] Ding Li, Yingjun Lyu, Jiaping Gui, and William GJ Halfond. 2016. Automated energy optimization of HTTP requests for mobile applications. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, 249–260.
[24] Ding Li, Yingjun Lyu, Mian Wan, and William GJ Halfond. 2015. String analysis for Java and Android applications. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 661–672.
[25] Yang Li. 2014. Reflection: enabling event prediction as an on-device service for mobile interaction. In *Proceedings of the 27th annual ACM symposium on User interface software and technology*. ACM, 689–698.
[26] James W Mickens, Jeremy Elson, Jon Howell, and Jay Lorch. 2010. Crom: Faster Web Browsing Using Speculative Execution.. In *NSDI*, Vol. 10. 9–9.
[27] Ravi Netravali, Ameesh Goyal, James Mickens, and Hari Balakrishnan. 2016. Polaris: faster page loads using fine-grained dependency tracking. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*.
[28] Venkata N Padmanabhan and Jeffrey C Mogul. 1996. Using predictive prefetching to improve world wide web latency. *ACM SIGCOMM Computer Communication Review* 26, 3 (1996), 22–36.
[29] Abhinav Parate, Matthias Böhmer, David Chu, Deepak Ganesan, and Benjamin M. Marlin. 2013. Practical Prediction and Prefetch for Faster Access to Applications on Mobile Phones. In *Proceedings of the 2013 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp '13)*. ACM, New York, NY, USA, 275–284. https://doi.org/10.1145/2493432.2493490
[30] Danilo Dominguez Perez and Wei Le. 2017. Generating Predicate Callback Summaries for the Android Framework. In *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft '17)*. IEEE Press, Piscataway, NJ, USA, 68–78. https://doi.org/10.1109/MOBILESoft.2017.28
[31] QUARTZ. 2016. Android just hit a record 88% market share of all smartphones. (2016).
[32] Lenin Ravindranath, Jitendra Padhye, Sharad Agarwal, Ratul Mahajan, Ian Obermiller, and Shahin Shayandeh. 2012. AppInsight: mobile app performance monitoring in the wild. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. 107–120.
[33] PRESTO research group. 2017. GATOR: Program Analysis Toolkit For Android. (2017). http://web.cse.ohio-state.edu/presto/software/gator/
[34] Sanae Rosen, Bo Han, Shuai Hao, Z. Morley Mao, and Feng Qian. 2017. Push or Request: An Investigation of HTTP/2 Server Push for Improving Mobile Performance. In *Proceedings of the 26th International Conference on World Wide Web (WWW '17)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, Switzerland, 459–468. https://doi.org/10.1145/3038912.3052574
[35] Vaspol Ruamviboonsuk, Ravi Netravali, Muhammed Uluyol, and Harsha V Madhyastha. 2017. VROOM: Accelerating the Mobile Web with Server-Aided Dependency Resolution. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 390–403.
[36] N. Swaminathan and S. V. Raghavan. 2000. Intelligent prefetch in WWW using client behavior characterization. In *Proceedings 8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (Cat. No.PR00728)*. 13–19. https://doi.org/10.1109/MASCOT.2000.876424
[37] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java Bytecode Optimization Framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON '99)*. IBM Press, 13–. http://dl.acm.org/citation.cfm?id=781995.782008
[38] Haoyu Wang, Junjun Kong, Yao Guo, and Xiangqun Chen. 2013. Mobile web browser optimizations in the cloud era: A survey. In *Service Oriented System Engineering (SOSE), 2013 IEEE 7th International Symposium on*. IEEE, 527–536.
[39] Xiao Sophia Wang, Arvind Krishnamurthy, and David Wetherall. 2016. Speeding up web page loads with Shandian. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. 109–122.
[40] Yichuan Wang, Xin Liu, David Chu, and Yunxin Liu. 2015. EarlyBird: Mobile Prefetching of Social Network Feeds via Content Preference Mining and Usage Pattern Analysis. In *Proceedings of the 16th ACM International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc '15)*. ACM, New York, NY, USA, 67–76. https://doi.org/10.1145/2746285.2746312
[41] Zhen Wang, Felix Xiaozhu Lin, Lin Zhong, and Mansoor Chishtie. 2011. Why are web browsers slow on smartphones?. In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*. ACM, 91–96.
[42] Zhen Wang, Felix Xiaozhu Lin, Lin Zhong, and Mansoor Chishtie. 2012. How far can client-only solutions go for mobile browser speed?. In *Proceedings of the 21st international conference on World Wide Web*. ACM, 31–40.
[43] Ryen W. White, Fernando Diaz, and Qi Guo. 2017. Search Result Prefetching on Desktop and Mobile. *ACM Trans. Inf. Syst.* 35, 3, Article 23 (May 2017), 34 pages. https://doi.org/10.1145/3015466
[44] C. Wu, X. Chen, Y. Zhou, N. Li, X. Fu, and Y. Zhang. 2016. Spice: Socially-driven learning-based mobile media prefetching. In *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*. 1–9. https://doi.org/10.1109/INFOCOM.2016.7524568
[45] Tingxin Yan, David Chu, Deepak Ganesan, Aman Kansal, and Jie Liu. 2012. Fast App Launching for Mobile Devices Using Predictive User Context. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services (MobiSys '12)*. ACM, New York, NY, USA, 113–126. https://doi.org/10.1145/2307636.2307648
[46] Shengqian Yang, Dacong Yan, Haowei Wu, Yan Wang, and Atanas Rountev. 2015. Static control-flow analysis of user-driven callbacks in Android applications. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 89–99.
[47] Shengqian Yang, Hailong Zhang, Haowei Wu, Yan Wang, Dacong Yan, and Atanas Rountev. 2015. Static window transition graphs for android (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 658–668.
[48] Y. Yang and G. Cao. 2018. Prefetch-Based Energy Optimization on Smartphones. *IEEE Transactions on Wireless Communications* 17, 1 (Jan 2018), 693–706. https://doi.org/10.1109/TWC.2017.2769646