MOBILESoft: G: Reducing Latency in Mobile Apps via Program Analysis

Yixue Zhao
University of Southern California
yixue.zhao@usc.edu

ABSTRACT

Reducing network latency in mobile applications is an effective way of improving the mobile user experience and has tangible economic benefits. We present 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 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 3s [2]. Google estimates that an additional 500ms delay would result in up to 20% traffic loss, while Amazon estimates that every 100ms delay would cause 1% annual sales loss [14].

Reducing latency thus becomes a highly effective way of improving mobile user experience. In the context of mobile communication, we define *latency* as the response time of an HTTP request. In this work, 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 responses to be generated immediately from a local cache.

The key challenges to efficiently prefetching HTTP requests involve determining (1) which requests to prefetch, (2) what their destination URL values are, and (3) when to prefetch them. To address those challenges, prior prefetching approaches can be divided into four categories. *(1) Server-based* techniques analyze the requests sent to the server and provide "hints" to the client on what to prefetch [8, 9]. However, as most apps today depend extensively on heterogeneous third-party servers, providing server-side "hints" is difficult, not scalable, or even impossible because app developers have no control over the third-party servers [14]. *(2) Human-based* approaches rely on developers to explicitly annotate application segments that are amenable to prefetching [6, 7], which is error-prone and involves significant manual efforts. *(3) History-based* approaches predict future requests based on prior requests [8, 15], which requires significant time to gather historical data. *(4) Domain-based* approaches are limited to only one specific domain thus are not applicable to apps in general, such as only prefetching the constant URLs in tweets [12] in social network domain.

To address these limitations, 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* [18, 19]. Our guiding insight is that an app's *code* can provide useful information on *what* requests may occur and *when*. Additionally, a mobile user usually spends multiple seconds deciding what event to trigger next—a period known as "user think time" [7]—providing an opportunity to prefetch requests in the background. By analyzing an Android program[1], we are able to identify HTTP requests and certain event sequences (e.g., `onScroll` followed by `onClick`) so that we can prefetch requests that will happen next during user think time.

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 app. 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 work 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 repositories are publicly available [1].

2 BACKGROUND AND MOTIVATION

Mobile apps that depend on network generally involve *events* that interact with user inputs and *network requests* that interact with remote servers. We explain these two concepts via Listing 1, which is a code excerpt from an app for retrieving weather information.

```
1  class MainActivity {
2    String favCityId, cityName, cityId;
3    protected void onCreate(){
4      favCityId = "ID123";//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("http://weather?&cityId="+favCityId);
13         URL url2 = new URL("http://weather?cityName="+cityName);
14         URL url3 = new URL("http://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     }});
22   }
23 }
```

Listing 1: Code snippet with callbacks and HTTP requests

Events: In mobile apps, user interactions are translated to events, such as `onClick`. Each event is registered to a particular UI object with a callback function that is executed when the event is triggered. In Listing 1, the button `submitBtn` is registered with an `onClick` event (Line 9), and the corresponding callback function `onClick()` (Lines 10-21) will be executed when a user clicks the button, and similar for the drop-down box `cityNameSpinner` (Lines 5-8).

Network Requests: Within an event callback function, the app often communicates with remote servers to retrieve information

[1] We focus on native Android apps because of its dominant market share and its popular event-driven interaction style.

via network requests over the HTTP protocol. There are two types of URL values associated with HTTP requests, depending on when the value is known: *static* and *dynamic*. In Listing 1, `favCityId` is static because its value is known by static analysis (Lines 4, 12). In contrast, `cityName` is dynamic since its value depends on which item a user selects from `cityNameSpinner` at runtime (Lines 7, 13).

The motivation for PALOMA is that user-perceived latency can be significantly reduced by prefetching certain network requests. For instance, Listing 1 corresponds to a scenario in which a user selects a city name from `cityNameSpinner` (Line 7), then clicks `submitBtn` (Line 9) to get the city's weather information through an HTTP request and wait for that response. In this case, an effective prefetching scheme would submit that request immediately after the user selects a city name, i.e., before the user clicks the button.

2.1 Terminology

We define several terms needed for describing our approach.

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 *dynamic* URL value 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 $\texttt{L7}_{2,1}$ for `url2` because `cityName` is the second substring in `url2` and Line 7 is the first definition of `cityName`.

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.

Target Method is a method that contains at least one Fetch Spot, such as `onClick()` in Listing 1 because it contains three Fetch Spots.

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 10-21 of Listing 1 is a Target Callback.

Callback Control-Flow Graph (CCFG) represents the *implicit*-control flow of callbacks [17]. Nodes represent callbacks, and each directed edge $f \rightarrow s$ denotes that s is the next callback invoked after f. A special *wait node* indicates that user action is required to trigger the event that determines which subsequent callback will be invoked.

Trigger Callback is any callback in CCFG that is an immediate predecessor of a Target Callback with one wait node between them.

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

3 APROACH

PALOMA has four major elements as Figure 1 shows. 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 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 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 now detail these four elements.

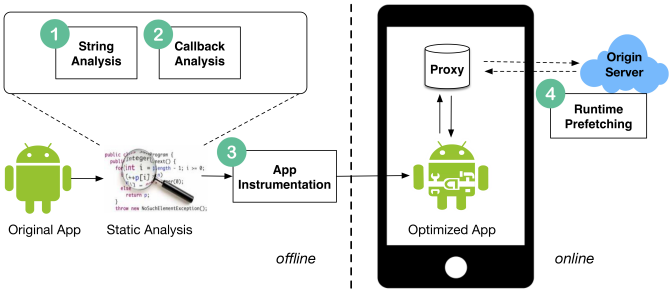

Figure 1: High-level overview of the PALOMA approach

3.1 String Analysis

The goal of string analysis is to identify the URL values of HTTP requests because prefetching can only happen when the destination URL of an HTTP request is known. Thus, we maintain a map to store the URL values that PALOMA interprets. As Figure 2 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 Listing 1, the entry in the URL Map that is associated with `url2` would be `{url2: ["http://weather?&cityName=", L7`$_{2,1}$`]}`

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. We leveraged a recent string analysis framework, Violist [5], that performs static 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 framework, such as the constant strings defined in the resource file. PALOMA extends Violist to analyze the resource file that is extracted by decompiling the app. In the end, the concrete value of each static substring in each URL is added to the URL Map.

Dynamic value analysis – Dynamic values cannot be determined by static analysis. Instead, PALOMA identifies their Definition Spots which are later instrumented (Section 3.3) such that the concrete values can be determined at runtime. To identify Definition Spots, 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 the end, all Definition Spots are added to the URL Map. It is worth noting that although the static analysis is conservative and multiple Definition Spots may be recorded, the true Definition Spot will emerge at runtime because false definitions will either be overwritten by a later true definition or will never be encountered if they lie along unreachable paths.

3.2 Callback Analysis

Callback analysis determines where to prefetch HTTP requests, i.e., the Trigger Points. 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 issued. The most aggressive strategy is to issue a request immediately after its URL value is known, but it may lead to redundant network transmissions: the URL value may not be used in any on-demand requests at runtime. In contrast, the most accurate strategy is to issue the

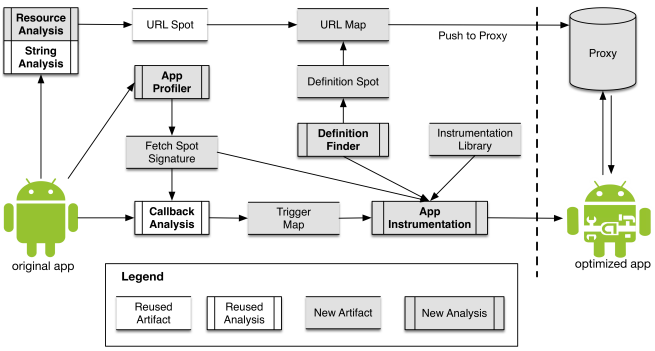

Figure 2: PALOMA's detailed workflow.

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

Our approach is to strike a balance between the two extremes. PALOMA issues prefetching requests at the end of the callback that is the immediate predecessor of the Target Callback, i.e., the end of Trigger Callback. This strategy has the dual benefit of (1) taking advantage of the "user think time" between two consecutive callbacks to prefetch, while (2) providing high prefetching accuracy as the Trigger Point is reasonably close to the on-demand request.

As Figure 2 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] }

To generate the Trigger Map, PALOMA relies on the CCFG that captures the implicit-invocation flow of callbacks in Android [17], and the Call Graph (CG) that captures the control flow between methods [10]. PALOMA first identifies all HTTP requests that the app can possibly issue based on the signatures at Fetch Spots obtained by profiling the app[2], such as `conn1.getInputStream()`, `conn2.getInputStream()`, and `conn3.getInputStream()` in Listing 1. Then PALOMA iterates through each request and identifies the method in which the request is actually issued, i.e., the Target Method. PALOMA then locates all possible Target Callbacks of each Target Method based on the CG. Thereafter, PALOMA iterates through each Target Callback and identifies all of its immediate predecessors, i.e., Trigger Callbacks, according to the CCFG. Finally, we add each {Trigger Callback, URL} pair to the Trigger Map.

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. 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.

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.

1. Update URL Map – This instrumentation task updates the URL Map as new *dynamic* URL values are discovered. Recall that the *static* values 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 second 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, which is at the end of each Trigger Callback in PALOMA because it makes no discernible performance difference regarding where we prefetch within the same callback and placing the Trigger Point at the end is more likely to yield known URLs. PALOMA provides this instrumentation via `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 25) of Listing 2.

3. Redirect Requests – This instrumentation task redirects all on-demand requests to PALOMA's proxy instead of the origin server via 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.

```
1  class MainActivity {
2     String favCityId, cityName, cityId;
3     protected void onCreate(){
4        favCityId = "ID123";;//static
5        cityNameSpinner.setOnItemSelectedListener(new OnItemSelectedListener(){
6           public void onItemSelected() {
7           cityName = cityNameSpinner.getSelectedItem().toString();//dynamic
8           sendDefinition(cityName, url2, 2);
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("http://weather?&cityId="+favCityId);
16             URL url2 = new URL("http://weather?cityName="+cityName);
17             URL url3 = new URL("http://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       }});
25       triggerPrefetch(url1, url2, url3);
26    }
27 }
```

Listing 2: Example code of the optimized app

Developer Hints: PALOMA automatically instruments apps without developer involvement, it also provides opportunities for developers to add hints to better guide the prefetching in two ways.

1. API support – PALOMA's three APIs can be invoked by the developers explicitly in the code. For instance, if a developer knows where the true Definition Spots are, she can insert `sendDefinition()` only at true locations. Developers can also insert `triggerPrefetch()` at any program point with their domain knowledge of the app.

2. Artifact modification – Developers can also directly modify the artifacts generated by PALOMA's static analyses (recall Figure 2) without altering the code, such as Trigger Map, Definition Spot. For example, a developer can add an entry in the Trigger Map and PALOMA will automatically insert a call to `triggerPrefetch()` at the end of the Trigger Callback specified by the developer.

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.

1. Update URL Map – When the concrete value of the dynamic URL is obtained at runtime, the inserted `sendDefinition(var,url,id)` is executed to send the concrete runtime value 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 `cityNameSpinner` (Line 7), the concrete value of `cityName` will be known, e.g., "LA". Then `cityName` is sent to the proxy (Line 8) and the URL Map entry for `url2` will be updated to `{url2: ["http://weather?&cityName=","LA"]}`.

2. Trigger Prefetching – When `triggerPrefetch(url1,...)` is executed, it triggers the proxy to check each request to see if the whole URL value is known but the response to the request has not

[2]We found that the profiling is needed because the methods that actually issue HTTP requests under different circumstances can vary across apps.

yet been cached. If both conditions are met, a prefetching request will be sent and the response will be cached. In Listing 2, when the app reaches the end of `onCreate` (Line 25), it triggers the proxy to check `url1`, `url2`, `url3`. Only `url1` meets both conditions: the whole URL value is concrete and the response is not cached. The proxy thus prefetches `url1` from the origin server and caches the response. 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), all three urls are checked again and `url2` will be prefetched because its URL is known (its *dynamic* value obtained at Line 8) and has not been previously prefetched.

3. Redirect Requests – When the on-demand request is sent at the Fetch Spot, the replaced `fetchFromProxy(conn)` will be executed, and it will in turn trigger the proxy to return the response immediately from the cache with no network operations involved. If the response is not cached, the proxy issues an on-demand request using the original URL connection `conn` to fetch the response from the server, caches the response, and returns the response to the app. 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, 23). The proxy in turn returns the responses to `url1` and `url2` from the local cache immediately because `url1` and `url2` are prefetched at Lines 25 and 9 respectively, as discussed above. `url3` is not known at any of the Trigger Points, so its response will be fetched from the server as in the original app.

4 MICROBENCHMARK EVALUATION

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 a request is prefetchable: (1) k—the number of dynamic values; (2) d_i—the number of Definition Spots for the i^{th} dynamic value; (3) the location of each Definition Spot relative to the Trigger Point. The case where there is no dynamic values, i.e., the whole URL is static, is considered separately. A request is

- **prefetchable:** iff *every* dynamic value has a DefSpot before Trigger Point
- **hit:** iff *all* dynamic value DefSpots are before Trigger Point
- **non-hit:** iff *some* dynamic value DefSpots are after Trigger Point
- **non-prefetchable:** iff *all* DefSpots for a dynamic value are after Trigger Point

Without loss of generality, MBM covers all 25 cases where $k \leq 2$ and $d_i \leq 2$ 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. The simplest case is when the entire URL is known statically (case 0). The remaining 24 cases are diagrammatically encoded in Figure 3. Of particular interest are the six *hit* cases—0, 1, 3, 6, 10, and 16—that should allow PALOMA to prefetch the corresponding requests and significantly reduce the user-perceived latency.

We implemented the MBM as a set of Android apps along with the remote server to test each case on the 4G network. 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).

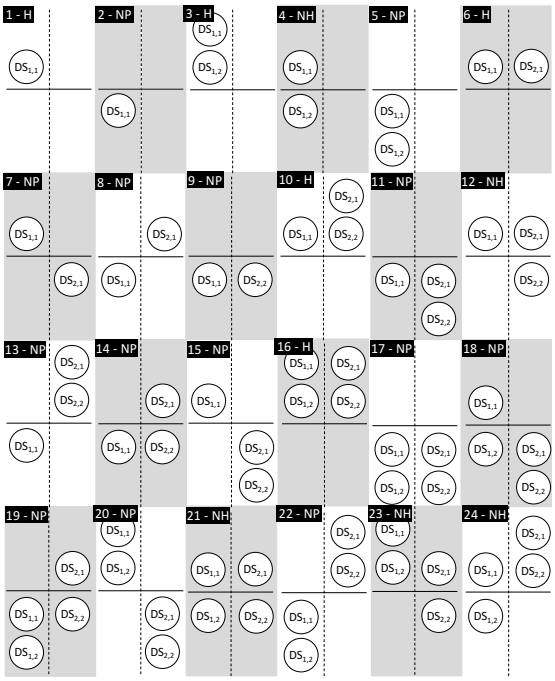

Figure 3: The 24 test cases covering all configurations involving dynamic values. The horizontal divider denotes the Trigger Point, while the vertical divider delimits the two dynamic values. The circles labeled with "$DS_{i,j}$" are the locations of the Definition Spots with respect to the Trigger Point. "H" denotes a *hit*, "NH" denotes a *non-hit*, and "NP" denotes a *non-prefetchable* request.

Table 1 shows the results of each test case corresponding to Figure 3, as well as case 0. Each execution value is the average of multiple executions of the corresponding MBM apps. The highlighted test cases are the *hit* cases where the reduction is massive (≥99%).

5 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. 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.

We asked multiple Android users to actively use the 32 subject apps for two minutes each, and recorded the resulting usage traces. 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 the methods at the Fetch Spots between the original and optimized apps.

Table 2 depicts the averages, outliers (min and max), 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% because the app fetches a large number of ads at runtime whose URLs are non-deterministic, and only a single static URL is prefetched. There are four additional list-view apps whose hit rate is below 20% because they fetch large numbers of requests in the list at the same time. In PALOMA, we set the threshold for the maximum number of requests

Table 1: Results of PALOMA's MBM evaluation. "SD", "TP", and "FFP" denote the runtimes of the three PALOMA instrumentation methods. "Orig" is the response time of the request in 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%

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%

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 is 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.

6 RELATED WORK

Prefetching HTTP requests has been applied successfully in the browser domain [7, 11] but cannot be applied to mobile apps. The bottleneck for page load times is resource loading [13] because one initial HTTP request will require a large number of subresources which can only be discovered after the main source is fetched and parsed. However, in mobile apps, the HTTP requests are light-weight [4]: one request only fetches a single resource. Therefore, our work focuses on prefetching the future requests that a user may trigger next rather than the subresources within a single request.

Prefetching in mobile apps is still in its infancy. One research thread has attempted to answer "how much" to prefetch under different contexts [3], while assuming that "what" to prefetch is handled by the apps already. Another thread focuses on fast prelaunching by predicting what app the user will use next [16]. 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 [8, 9], human-based [6, 7], history-based [8, 15], and domain-based [12]—have limitations which we directly target in PALOMA.

7 CONCLUSION AND FUTURE WORK

We presented PALOMA, a novel program analysis-based technique that reduces the user-perceived latency in mobile apps by prefetching HTTP requests. PALOMA defines formally the conditions under which the requests are prefetchable. This provides guidelines for developers to make their apps more amenable to prefetching, and lay the foundations for further program analysis-based prefetching techniques. PALOMA shows prefetching all possible next requests is effective in practice and this provides motivation and confidence for future research to optimize program analysis-based prefetching techniques, such as studying user behavior patterns to prune unlikely requests and adapting prefetching based on runtime QoS conditions. Finally, PALOMA's MBM forms a foundation for standardized empirical evaluation and comparison of future efforts. Currently, we are conducting a large-scale empirical study of real apps on request properties (e.g., *prefetchability, cachability*) and comparing different prefetching strategies to guide future research in this area.

REFERENCES

[1] PALOMA website: https://softarch.usc.edu/PALOMA.

[2] AppDynamics. The app attention span, 2014.

[3] B. D. Higgins, J. Flinn, T. J. Giuli, B. Noble, C. Peplin, and D. Watson. Informed mobile prefetching. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pages 155–168. ACM, 2012.

[4] D. Li, Y. Lyu, J. Gui, and W. G. Halfond. Automated energy optimization of http requests for mobile applications. In *Proceedings of the 38th International Conference on Software Engineering*, pages 249–260. ACM, 2016.

[5] D. Li, Y. Lyu, M. Wan, and W. G. Halfond. String analysis for java and android applications. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 661–672. ACM, 2015.

[6] Y. Li. 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*, pages 689–698. ACM, 2014.

[7] J. W. Mickens, J. Elson, J. Howell, and J. Lorch. Crom: Faster web browsing using speculative execution. In *NSDI*, volume 10, pages 9–9, 2010.

[8] V. N. Padmanabhan and J. C. Mogul. Using predictive prefetching to improve world wide web latency. *ACM SIGCOMM Computer Communication Review*, 26(3):22–36, 1996.

[9] S. Rosen, B. Han, S. Hao, Z. M. Mao, and F. Qian. 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, pages 459–468, Republic and Canton of Geneva, Switzerland, 2017. International World Wide Web Conferences Steering Committee.

[10] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot: A java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, pages 214–224. IBM Corp., 2010.

[11] X. S. Wang, A. Krishnamurthy, and D. Wetherall. Speeding up web page loads with shandian. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 109–122, 2016.

[12] Y. Wang, X. Liu, D. Chu, and Y. Liu. 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*, pages 67–76. ACM, 2015.

[13] Z. Wang, F. X. Lin, L. Zhong, and M. Chishtie. Why are web browsers slow on smartphones? In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*, pages 91–96. ACM, 2011.

[14] Z. Wang, F. X. Lin, L. Zhong, and M. Chishtie. How far can client-only solutions go for mobile browser speed? In *Proceedings of the 21st international conference on World Wide Web*, pages 31–40. ACM, 2012.

[15] R. W. White, F. Diaz, and Q. Guo. Search result prefetching on desktop and mobile. *ACM Transactions on Information Systems (TOIS)*, 35(3):23, 2017.

[16] T. Yan, D. Chu, D. Ganesan, A. Kansal, and J. Liu. Fast app launching for mobile devices using predictive user context. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pages 113–126. ACM, 2012.

[17] S. Yang, D. Yan, H. Wu, Y. Wang, and A. Rountev. 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, 2015.

[18] Y. Zhao. Toward client-centric approaches for latency minimization in mobile applications. In *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 203–204. IEEE, 2017.

[19] Y. Zhao, M. Schmitt Laser, Y. Lyu, and N. Medvidovic. Leveraging program analysis to reduce user-perceived latency in mobile applications. available at: https://softarch.usc.edu/~yixue/mypapers/ICSE2018_PALOMA.pdf. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, 2018.