Marauder: Synergized Caching and Prefetching for Low-Risk Mobile App Acceleration

Murali Ramanujam
UCLA

Harsha V. Madhyastha
University of Michigan

Ravi Netravali
Princeton University

ABSTRACT

Low interaction response times are crucial to the experience that mobile apps provide for their users. Unfortunately, existing strategies to alleviate the network latencies that hinder app responsiveness fall short in practice. In particular, caching is plagued by challenges in setting expiration times that match when a resource's content changes, while prefetching hinges on accurate predictions of user behavior that have proven elusive. We present Marauder, a system that synergizes caching and prefetching to improve the speedups achieved by each technique while avoiding their inherent limitations. Key to Marauder is our observation that, like web pages, apps handle interactions by downloading and parsing structured text resources that entirely list (i.e., without needing to consult app binaries) the set of other resources to load. Building on this, Marauder introduces two *low-risk* optimizations directly from the app's cache. First, guided by cached text files, Marauder prefetches referenced resources *during* an already-triggered interaction. Second, to improve the efficacy of cached content, Marauder judiciously prefetches about-to-expire resources, extending cache lives for unchanged resources, and downloading updates for lightweight (but crucial) text files. Across a wide range of apps, live networks, interaction traces, and phones, Marauder reduces median and 90th percentile interaction response times by 27.4% and 43.5%, while increasing data usage by only 18%.

CCS CONCEPTS

• **Networks** → **Mobile networks**; *Network measurement*; • **Human-centered computing** → **Mobile phones**.

KEYWORDS

Smartphones, mobile apps, performance, caching, prefetching

ACM Reference Format:
Murali Ramanujam, Harsha V. Madhyastha, and Ravi Netravali. 2021. Marauder: Synergized Caching and Prefetching for Low-Risk Mobile App Acceleration . In *The 19th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '21), June 24-July 2, 2021, Virtual, WI, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3458864.3466866

1 INTRODUCTION

Mobile apps have become the predominant medium through which mobile users access Internet services, accounting for over 80% of user attention time on smartphones [27, 39]. Crucial to the success of mobile apps are low response times and user-perceived latency. Recent reports have shown that users negatively respond to delays on the order of just 100 ms [6], and that they will abandon interactions or even delete apps if response times exceed 2-3 seconds [7, 13, 28]. Thus, inflated response times not only affect user experience, but can also have a significant impact on content provider revenue [15, 47].

Given the importance of mobile app performance, much work has been devoted to improving their responsiveness [8, 9, 11, 21, 24, 31, 37, 61]. Yet, apps continue to underperform in practice, delivering median response times of 2.9 seconds even on state-of-the-art phones and LTE networks (§3.2). In line with prior studies [9, 51, 61], we find that network transfer delays are the primary culprit for high response times.

Today, there exist two primary classes of techniques to alleviate the negative impact that network delays have on app responsiveness: caching and prefetching. In principle, both are highly effective, particularly given the patterns of repetition exhibited in user interactions with apps [29]. However, each has fundamental drawbacks that have limited their use and effectiveness in practice (§3.3).

- Caching [14, 35, 36, 41, 48, 49, 59, 60, 62], based on HTTP headers streamed from servers, enables mobile apps to store local copies of static resources and eliminate subsequent network fetches for those resources until they change. Although caching is widely used (74% of resources are cacheable in our experiments), existing policies are far from optimal, foregoing 52% of potential cache hits compared to policies that perfectly evict resources only when their content has changed. We discover that the issue is not simply a lack of aggressive policies. Instead, owing to the large variance in the rate at which a given resource's content varies over time, developers typically opt for conservative (i.e., low) time-to-live values (TTLs) to fully retain control of content changes and prevent the loading of stale content.
- Prefetching systems [8, 9, 24, 31, 37, 61] aim to predict future user interactions and download the required content ahead of time for storage in the client cache. The drawbacks of prefetching are well-documented [44, 50], and largely stem from the difficulty in predicting precisely what interactions users will make, when, and what resources will be required. Incorrect predictions result in wasted bandwidth and smartphone energy. Indeed, the best prefetching policies supported by the apps in our experiments deliver improvements of 59-82% for app responsiveness, but inflate data usage by 2.4-4.1×; prior prefetching systems similarly report up to 4.2× data overheads [9]. Consequently, only 6% of apps in our corpus employ prefetching by default.

We present **Marauder**, a system that combines caching and prefetching in a way that improves the speedups realized with each technique while sidestepping the aforementioned risks and practical challenges (i.e., low TTLs+hit rates or stale content for caching, wasted bandwidth for prefetching). Our high-level insight underpinning Marauder is that, despite the installation of client-side

MobiSys '21, June 24-July 2, 2021, Virtual, WI, USA

ACM ISBN 978-1-4503-8443-8/21/07.
https://doi.org/10.1145/3458864.3466866

binaries, mobile apps commonly respond to user interactions using a similar model to that in web page loads [40, 42]. In particular, apps discover subsequent requests to make by parsing previously fetched text files. Further, we find that those text files are structured such that it is possible to derive most (94%) of the URLs that they reference purely by analyzing their text contents (rather than analyzing app binaries [9, 61]). Using this insight, Marauder optimizes app responsiveness directly from the app's cache in two ways.

Caching for just-in-time (JIT) prefetching. Rather than prefetching according to (potentially inaccurate) predictions of what a user may do in the future, Marauder leverages the structure in text files to prefetch *during* the current interaction. To do this, Marauder statically analyzes each downloaded text file to extract a list of referenced URLs, taking special care to handle factors like dynamic query strings and relative URLs. Then, when that text file is requested at the start of a future interaction, in addition to serving it from the cache or fetching it over the network (if uncacheable), Marauder also issues asynchronous prefetch requests for the resources it references. Importantly, this approach is low-risk because Marauder only prefetches resources that are referenced by a file that was explicitly requested and is about to be processed. Yet, this prefetching strategy is fruitful since referenced resources cannot be fetched until the corresponding text file is downloaded (if it is not cached) and also parsed.

Prefetching to improve the efficacy of cached resources. Building on our observation that TTLs are fundamentally difficult to set correctly, we find that 47% of resources expire in the cache despite their content not changing. As a result, to maximize the efficacy of already-cached content, Marauder proactively prefetches about-to-expire resources in hopes of extending their TTLs. However, as with traditional prefetching, there is a chance of wasting bandwidth since Marauder is unaware of what resources will be requested in the future. To mitigate this risk, Marauder employs a hybrid approach in which it uses cheap HEAD requests (that do not ship payloads) to extend the TTLs for non-text resources whose content has not changed, and preferentially downloads content updates for text resources using conditional GET requests. This strategy leverages the importance of text files with respect to guiding JIT prefetching and blocking downstream fetches, and is also low-risk as text files account for only 4% of the bytes downloaded in the median interaction.

Marauder is *immediately deployable* today, preserves end-to-end HTTPS security (unlike proxy-based systems [9, 11, 21, 42]), and does not require developers to modify servers, app binaries, or smartphone operating systems. Instead, as with our current prototype, Marauder simply replaces the caching libraries that most apps rely on with a version that embeds the aforementioned optimizations (Figure 1). Further, none of the optimizations in Marauder can break app functionality or change the content displayed to the user; in the end, the app will load all of the up-to-date resources.

We evaluated Marauder using a wide range of 50 popular Android apps, real mobile phones, live networks (WiFi and LTE) and servers, and realistic user interaction traces. Our experiments across these conditions reveal that Marauder reduces median and 90th percentile response times by 27.4% and 43.5% as compared to default caching and prefetching policies, while adding negligible (18%) bandwidth overheads. Further, Marauder provides 2.1× more benefits than recent prefetching systems [61], while imposing 91% lower data overheads. The source code and experimental data for Marauder are available at https://github.com/muralisr/marauder.

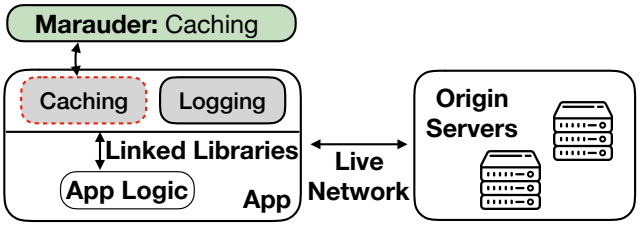


Figure 1: Marauder is immediately deployable in the existing app ecosystem, requiring only a direct swapping of an app's caching library with Marauder's version.

2 METHODOLOGY

We start by describing the experimental setup that we used throughout this paper. Our setup, illustrated in Figure 2, covers a wide range of live mobile apps, real mobile phones and networks, and realistic user interaction traces.

Apps used. We crawled the Google Play Store [18] in July 2020 and collected an Android Application Package (APK) for 75 popular apps. Our crawl considered a variety of categories including news, entertainment, weather, lifestyle, sports, art & design, personalization, and shopping. From this set, in order to operate with our current implementation of Marauder (§5), we focus on the 50 apps that use unobfuscated versions of the popular OkHttp caching library [54].

User interaction traces. User interaction patterns heavily influence mobile app performance and the efficacy of network acceleration techniques like caching and prefetching. To generate realistic user interaction traces for our entire corpus, we use the Humanoid app testing framework [32]. Humanoid employs deep neural networks to learn interaction patterns from actual user traces, and then explore new apps and UIs like a real user would. Humanoid was trained on the 10k Rico app dataset [12]. During exploration, Humanoid guides the AndroidViewClient UI monkey [2] by specifying, at each screen, where the monkey should interact. The output of a Humanoid session is a trace that specifies a series of actions to perform (e.g., "tap(x,y)"), as well as the delay between actions (to account for user think time). We generate 20 such traces for each app in our corpus, where each trace includes a median of 20 clicks and spans 2-3 minutes to match prior reports of user session times [16, 61].

Running experiments. For each app in our corpus, we consider the default APK, as well as a variant that embeds Marauder; we describe how to generate the latter in §5. We load both variants onto two powerful phones, a Google Pixel 4 (Android 10; 2.0 GHz octa-core processor; 6 GB RAM) and a Samsung Galaxy Note 9 (Android Oreo; 2.3 GHz octa-core; 6 GB RAM). Due to space constraints, we present results for the Pixel 4, but note that all reported results and trends were comparable with the Galaxy Note 9.

We randomly select 5 user interaction traces per app, and apply them δ minutes apart. In accordance with prior studies of user-app interactions [29], we consider δ values spanning 10 mins–1 day; if unreported, the default value is $\delta = 60mins$.

During experiments, the apps contact live origin servers using home WiFi or Verizon LTE networks with strong signal strength. Thus, to ensure a fair comparison with regards to app content and network/server delays, we run each trace back-to-back using the default and Marauder versions of the corresponding app. As our focus is primarily on accelerating subsequent (warm cache) interactions, we ignore the first trace for each app as it is entirely used to prime the

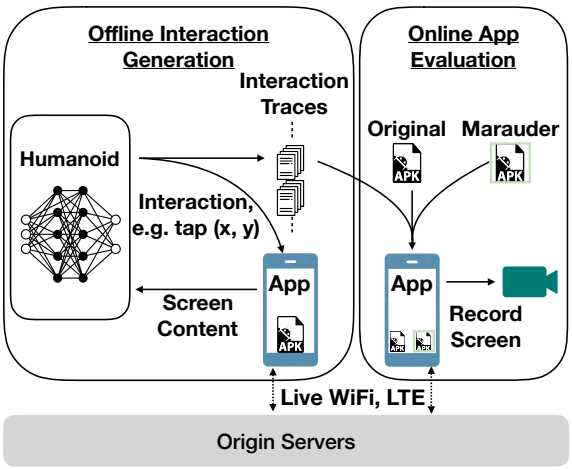


Figure 2: Overview of our evaluation setup. Realistic user interaction traces are generated offline, and are randomly cycled through during online experiments; experiments use real phones, live mobile networks, and live origin servers.

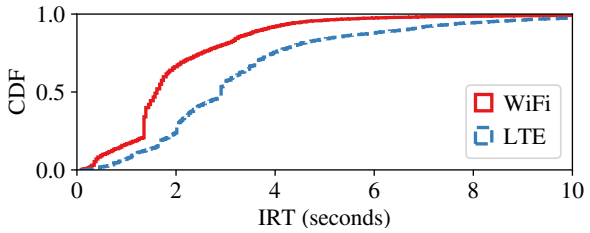


Figure 3: Mobile app interaction delays on WiFi and LTE.

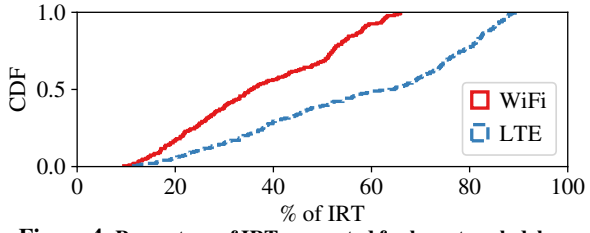


Figure 4: Percentage of IRT accounted for by network delays.

cache. Note that each trace represents a distinct interaction session, and thus a subset of interactions in subsequent traces will also experience cold caches; we preserve such interactions in our results, and provide a breakdown of warm versus cold cache interactions in §6.

Performance metrics. Our primary performance metric is interaction response time (IRT), or the time between when a user performs an on-screen tap to trigger an interaction, and the time when the final screen for that interaction is completely rendered. To measure IRT, we record each phone's screen for each interaction using ffmpeg [3]. We then process the collected video using the scene-cut tool [4] to track when the screen stopped visually changing. We identify and exclude dynamic pixels, or those that continually change by design (e.g., videos), using the same techniques employed by similar web performance metrics such as Speed Index [17].

3 THE STATE OF APP PERFORMANCE

In this section, we first review how mobile apps respond to user interactions today (§3.1), and present results to highlight the negative impact that network delays have on app responsiveness (§3.2). We then describe why classic and well-studied optimizations fail to sufficiently alleviate such network overheads (§3.3).

3.1 Background on App Operation

Each interaction that a user performs with a mobile app (e.g., a screen tap) triggers the firing of event handlers/callbacks (e.g., onTouch()) that are defined by the app binary [5, 51, 52]. Each event handler quickly begins responding to the specific event, either by updating the screen with pre-downloaded information (e.g., expanding a dropdown menu) or by issuing network requests to origin servers. Like the web, requests most often use the HTTP protocol. Upon receiving responses from servers, the app's handlers process them using logic that is either embedded in the downloaded files or in the app binary. Such processing potentially results in subsequent requests and rendering updates to the screen, and this process continues recursively until the final screen is rendered for the interaction.

Apps often use third-party caching libraries (e.g., OkHttp [54], Volley [19]) to mediate issued requests and downloaded responses between the app and servers [37]. Like typical HTTP caches (e.g., in web browsers), previously downloaded responses are used to service subsequent requests for the same resource based on the cacheability set by servers in HTTP headers. In particular, servers embed cache directives in HTTP headers indicating the time-to-live (TTL) for the corresponding resource, which in turn dictates how long that resource can be safely reused [41]. However, unlike traditional caches, app caching libraries store *all* downloaded resources, including those that are marked as uncacheable and those whose TTLs have expired (and are thus unsafe to directly serve). When such a resource is subsequently requested by the app, the caching library issues a conditional `GET` request for that resource to download its content only if it has changed. Thus, this approach attempts to eliminate redundant data transfers (and save bandwidth), but does not reduce fetch latencies.

3.2 Motivation

Mobile app interactions are too slow. Figure 3 plots the distribution of interaction response times (IRTs) for the apps and traces in our corpus. As shown, median and 90th percentile IRT values are 1.6 and 3.7 seconds on a WiFi network, and jump to 2.9 and 6.7 seconds on an LTE network. Thus, interaction response times frequently (and considerably) exceed the 2-3 seconds that users are willing to tolerate [7, 13, 28].

Network delays are key contributors. Mobile app response times are determined by both the network delays incurred during content fetches, as well as the client-side computation delays to parse/render that content and other code in the APK. In order to dissect the high response times from above, we evaluated our corpus of apps and traces in a setting where network delays were set to $\approx$0ms. To do this, for each interaction, we loaded it twice, back-to-back, with the first run evaluating standard performance, and the second run evaluating performance without network delays. To ensure high cache hit rates in the second run despite the intrinsic nondeterminism in certain app requests (e.g., those that embed timestamps), we configured OkHttp's cache hit logic to employ the Mahimahi URL matching heuristic [43] that identifies safe scenarios to serve cached resources despite URL discrepancies. Even with this heuristic, 9% of resources still failed to hit in the cache. To limit network delays for such resources, we relayed their fetches to origin servers using low-latency wired networks.

For each interaction, we compared IRTs with and without network delays. As Figure 4 shows, network delays account for 38% and 64% of IRT for the median interaction on WiFi and LTE, respectively.

Resource Type	% of resources cacheable	% of bytes cacheable
Images	85.3%	99.9%
HTML	44.7%	94.9%
JSON	64.6%	42.5%
CSS	100%	100%
JavaScript	67.6%	96.2%
XML	90.2%	60.7%
Binary	98.2%	93.0%

Table 1: Cacheability properties of different resource types.

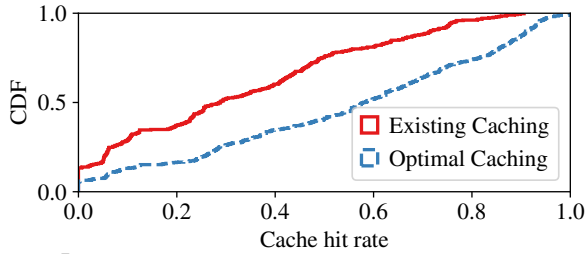


Figure 5: Comparing per-interaction cache hit rates with existing caching strategies and the optimal content-based one.

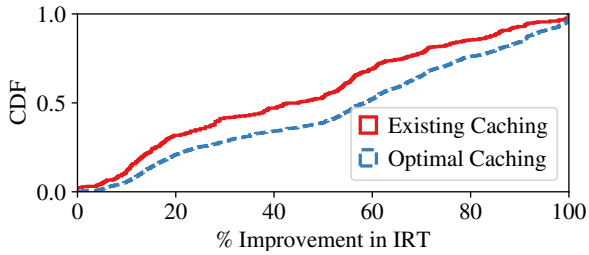


Figure 6: Speedups with existing and optimal caching strategies, as compared to cold cache interactions (i.e., no caching). Results are for the LTE network.

3.3 Limitations of Existing Optimizations

The above results reflect the default operation mode of the apps in our corpus. In other words, any caching or prefetching specified by an app's APK or origin servers is employed. Yet response times remain too high. Here, we explore how much and how effectively these optimizations are applied.

3.3.1 Caching

Caching is widely used and helps. Across our corpus and traces, we observe that 74% of resources are marked as cacheable for a non-zero amount of time by servers. Table 1 further breaks this down by resource type. For instance, 85.3% and 99.9% of image files and bytes are cacheable for some period; note that the discrepancy in files and bytes is entirely due to single-pixel tracking images that are often not cacheable. In our experiments, this translates to an overall cache hit rate of 28% for the median interaction (the "Existing Caching" line in Figure 5), and a 44% median speedup over entirely cold cache interactions (the "Existing Caching" line in Figure 6).

Current caching is suboptimal. To understand how effective existing caching policies are, we compared them with an optimal caching strategy that is based on resource content (rather than HTTP caching headers). More specifically, we analyzed the results from Figure 3 to determine the *ideal* time-to-live (TTL) for each resource, which is defined as the time until the resource's content changed (if it did at all). We then used the same setup applied above to evaluate performance without network delays (§3.2) to replay the interactions with the ideal TTLs enforced. As shown in Figure 5, the

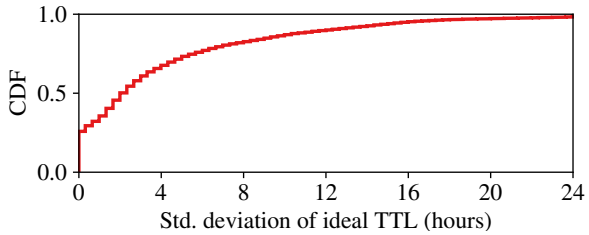


Figure 7: Variation in the ideal TTL (i.e., based on when content actually changes) for each resource fetched in our experiments. Zeros pertain to resources that never changed, changed once, or changed at fixed rates.

optimal caching strategy increases the median cache hit rate by 2.1× compared to existing policies. Figure 6 depicts the impact that these improved cache hit rates have on interaction response times: median IRTs drop by 32% compared to existing policies.

The problem. Existing caching practices necessitate that developers explicitly set a TTL for each resource that they serve. Unfortunately, setting such TTLs is difficult as the ideal TTL for each resource varies over time. For example, as shown in Figure 7, the standard deviation of ideal TTLs for the median resource in our experiments was 2 hours. In such cases, developers are faced with a tradeoff. On the one hand, developers can be conservative and select the low end of the ideal TTL spectrum for a resource to ensure that they retain control to quickly disseminate any updates to its content. However, this foregoes many cache hits during periods when the ideal TTL is larger than the minimum one. Alternatively, developers can select a higher value in the ideal TTL spectrum to improve cache hit rates at the risk of having the client use a stale version of the resource. The gap between existing and optimal caching strategies in Figure 5 confirms that (as expected) developers typically opt for conservative TTLs to ensure up-to-date content for all clients.

3.3.2 Prefetching

Prefetching is less common. 6% (i.e., 3) of the apps in our corpus have prefetching enabled by default when the user is connected to WiFi. These apps are all in the news category, and they employ custom prefetching policies to download, at regular intervals (e.g., every 2 hours), the text articles that are referenced by the app's home screen. In addition, the apps give users the ability to manually alter the prefetching policy with respect to the prefetching time interval, the content downloaded (e.g., prefetching images in addition to text files), and the settings under which prefetching should happen (e.g., WiFi only, WiFi+LTE). Further, 1 app has such prefetching as an option that is disabled by default. Our results thus far have considered the default prefetching policies for each app. However, to develop a holistic view of prefetching today, we also reran experiments under all possible prefetching policies that each app supports; the results below only consider the 4 apps with support for at least one prefetching policy.

Prefetching can help. To understand the potential speedups with prefetching, we compared the performance of each app under two settings: the best supported prefetching policies that delivered the largest speedups, and a policy in which prefetching was entirely disabled. As shown in Figure 8, the best prefetching policies improved median IRTs by 59.3-82.4% across the WiFi and LTE networks.

The problem. Prefetching inherently requires accurate predictions of what clients will request in the future. Many prior works [44, 50] have highlighted the difficulty in making such predictions accurately for different users and apps. As a result, as noted above, apps with

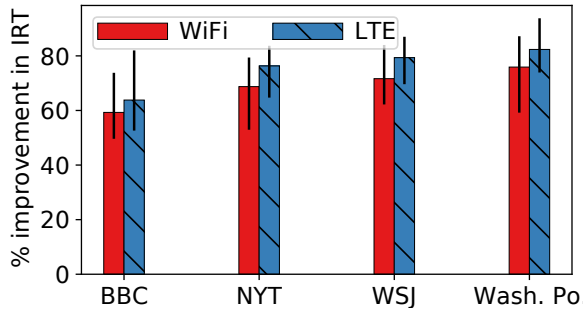


Figure 8: Speedups with the best (i.e., max speedups) prefetching policies supported by each app, as compared to no prefetching at all.

support for prefetching commonly opt for generic policies that fetch large amounts of content, much of which goes unused. Thus, despite the potential speedups, prefetching in practice is highly wasteful in terms of resource usage. For example, the best policies from Figure 8 impose data overheads of 2.4-4.1×. Such wastage can result in high cellular data plan costs, and can also eliminate IRT speedups, especially in bandwidth-constrained settings where explicitly-requested resources must contend with resources that were unnecessarily prefetched. In contrast, the most conservative prefetching policies (excluding no prefetching at all) supported by the apps in our corpus result in median bandwidth wastage of 6.7%; however, these policies yield speedups of only 9-13%.

4 DESIGN OF MARAUDER

Marauder is a mobile app acceleration framework whose goal is to collectively harness the power of caching and prefetching in a manner that sidesteps the risks and limitations of each technique in isolation (§3.3). At a high level, the key idea behind Marauder is to use judicious prefetching to maximize the utility (i.e., cache hits) for already-cached objects, and to then use those cached objects to guide just-in-time (i.e., during the current interaction) prefetching. Crucially, in designing Marauder, our overarching principle is to ensure direct deployability, i.e., operation with existing (legacy) apps, servers, and operating systems.

In this section, we first outline several key observations that we made from the experiments in §3 that both highlight opportunities for improvement with existing caching and prefetching strategies, and guide the operation of Marauder (§4.1). We then describe the end-to-end workflow of Marauder by separately discussing its offline/background operation (§4.2) and online handling of app requests (§4.3).

4.1 Guiding Observations

Three measurement observations from our study of caching and prefetching policies (§3.3) motivate the set of optimizations employed by Marauder. We describe them in turn, along with the corresponding implications for Marauder's design.

Observation 1. In line with the way that apps respond to user interactions today (§3.1), we find that the text files (i.e., JSON, HTML, JavaScript, CSS, and XML) fetched for a given interaction directly embed listings for the majority of non-text resources required for that interaction. To illustrate this, we parsed the text files fetched in our experiments in search of the URLs for non-text files that were fetched; we describe our static parsing methodology in §4.2. We find that the URLs for 94% of the fetched non-text resources could be derived from previously-fetched text files. This is surprising in that, despite the installation of client-side binaries, apps follow a request model similar to that in web page loads [40], whereby subsequent requests are made

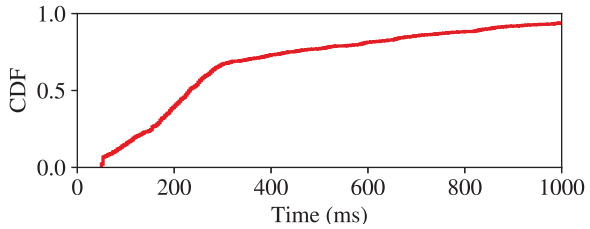


Figure 9: Delay between when the client's cache serves a text file and receives a subsequent request for a referenced resource.

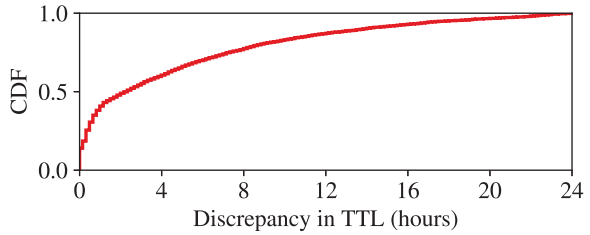


Figure 10: Discrepancy between existing TTLs and ideal TTLs that are based on when a resource's content changes.

by parsing previously fetched text resources. Further, text files are structured to clearly delineate referenced resources, as opposed to seemingly arbitrary data blobs that only the app binary can interpret.

The implication of this observation is that, as text files are being parsed and executed by the app, we can issue prefetch requests for any referenced file directly from the app's cache. In doing so, the early-stage parsing and execution delays of text files would be overlapped with the network fetch delays for referenced non-text files. Importantly, such JIT prefetching is low risk (as compared to the existing strategies discussed in §3.3.2) because it only considers resources that are directly listed in a file that was explicitly requested and is about to be parsed. Yet it would be fruitful for two reasons. As shown in Figure 9, the delay (from the cache's perspective) between when a text file is served to an application and when the first referenced file is requested is non-negligible, with a median value of 230 ms. Further, during this time, the network is entirely idle because, as noted above, the text file dictates the set of other files that the app must fetch.

Observation 2. Building off of the results in §3.3.1 which show that TTLs are often set too conservatively, we observe that the content of cached resources often remains unchanged despite the corresponding TTLs expiring; this occurs for 47% of resources. Figure 10 quantifies this, showing the discrepancy between the time when a resource expires in the cache and the time when its content actually changes. As shown, the TTL for the median resource is set to 1.5 hours below its optimal value.

In order to address this inefficiency, our goal is to prefetch about-to-expire resources in hopes of extending their TTLs when they have not changed (thereby improving their hit rates). However, such prefetching would be risky since, similar to the challenges of traditional prefetching, we are unaware of whether those resources will be requested in the future and the expended bandwidth is justified.

To mitigate this risk, we leverage the fact that text files, which are typically the root resource fetched at the beginning of an interaction, are both high priority (since they block downstream requests and guide the aforementioned JIT prefetching) and lightweight. In particular, the median text file in our experiments only consumes 2.3 KB; for comparison, the median image constitutes 1.3 MB. Further, text resources account for only 4% of the bytes downloaded in the median

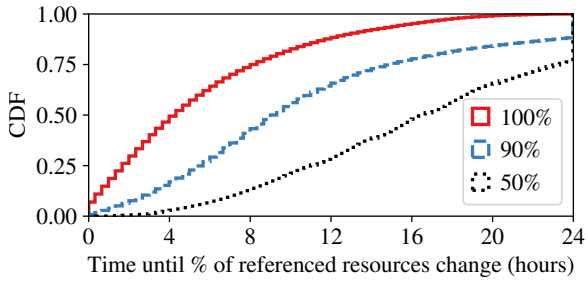


Figure 11: Resources referenced by a text file remain stable for long periods. In each distribution, a data point is the duration during which the listed percentage of referenced resources for a text file are unchanged.

interaction, while images account for 94%. Thus, while attempting to extend TTLs, Marauder preferentially downloads updates only for text files that have changed, and not for other resource types that consume more bytes and are not blocking in the interaction handling process. For a non-text resource, Marauder only checks if its content has changed, but does not exchange any payloads with servers.

Observation 3. Lastly, as depicted in Figure 11, we find that the set of files referenced by a given text file often remains stable for long durations, even as the text content changes. For example, the set of referenced files remains identical for 4 hours for the median text file, despite 50% of text files remaining unchanged for only 20 minutes. Relaxing this condition to having only 90% of referenced files being unchanged increases the median duration to 9 hours.

This does not have any bearing for text files that are up-to-date in the app cache. However, in order to retain fine-grained flexibility with respect to content dynamism for a given interaction, apps occasionally (17% for the median app) mark text files as uncacheable using the `no-cache` directive in HTTP headers. Such text files are retained in the app's cache but cannot be directly served to the app, and instead mandate a conditional check with the origin server (§3.1). In these cases, subsequent requests are not only blocked by the parsing of the text file (as in observation 1), but also by the network fetch to validate or update that text file. To alleviate such delays, similar to the implication of observation 2, we seek to issue prefetch requests for the files referenced by an uncacheable text file while that text file is being validated/updated and then parsed.

Summary. Taken together, these observations present opportunities for bolstering caching and prefetching speedups in a low-risk manner. We next describe how Marauder (Figure 12) incorporates these principles into the background and online operation of existing app caches.

4.2 Background Operation

Marauder's first background task is to (when possible) **refresh cached resources** to improve overall cache hit rates. For each resource in the cache with a non-zero TTL (i.e., excluding resources marked as no-cache), Marauder adds a timer event that checks, upon expiration, whether the resource's content has not changed and its TTL can be extended. As per observation 2 above, Marauder performs the TTL extension process differently for text and non-text files. Note that, in both cases, Marauder only uses HTTP request formats and cache update mechanisms that are standardized and widely supported by unmodified servers.

For text files, Marauder issues Conditional GET HTTP requests to the corresponding origin servers using the same HTTP headers that were used when those text files were explicitly requested by the app in the past. These requests identify the version (and thus, content)

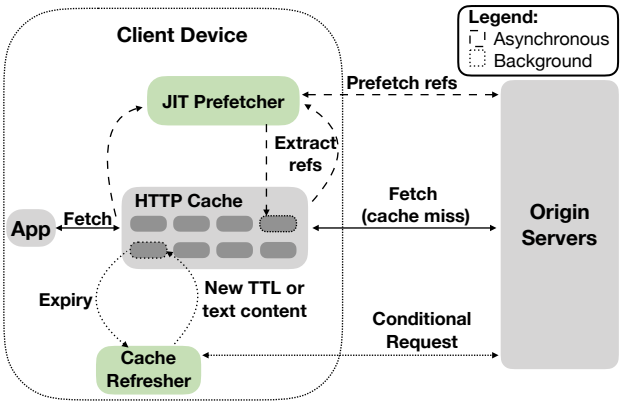


Figure 12: Overview of Marauder. The app's HTTP cache services requests as normal. Marauder adds two optimization components. First, for each downloaded text file, the just-in-time (JIT) Prefetcher extracts a list of referenced URLs; those URLs are asynchronously prefetched (through the cache; omitted for space) as soon as the corresponding text file is re-requested. Second, for each resource that is about to expire, the Cache Refresher conditionally extends its TTL (if its content has not changed) or updates its content (if it is a text file).

of the cached resource by relaying HTTP cache validation headers such as ETag and Last-Modified; ETag headers list unique IDs for the current version of a resource such as a hash value or version number, while Last-Modified headers indicate the creation time of the current resource version [41]. Upon receiving a response, Marauder does one of two things. If the response indicates that the resource has not been modified (i.e., a 304 Not Modified), Marauder updates the TTL for the cached resource to match the duration specified by the current response headers. In contrast, if the response indicates that the resource has been modified (i.e., a 200 OK), then Marauder replaces the cached resource with the new version and its corresponding HTTP headers and TTL. Marauder's handling of non-text files works much in the same way. However, in order to minimize wasted bandwidth from updating resources that may not be requested in the future, Marauder eschews Conditional GETs in place of HTTP HEAD requests that enable origin servers to share information about the current version of a resource, but preclude them from updating a cached resource when its content has changed. Note that updating prior GET requests with HEAD requests is compliant with the HTTP RFC [1]. Unlike GET requests, HEAD requests do not include cache validation headers. Instead, those headers are embedded in the response to enable clients to determine the validity of a cached resource. Thus, upon receiving a response, Marauder compares the values of cache validation headers to the corresponding values for the cached version of the resource. If the values match, Marauder extends the resource's TTL in the cache based on the Cache-Control headers in the HEAD's response. Alternatively, if the HEAD response indicates that the resource's content has changed (i.e., a mismatched cache validation header), Marauder does not update the cached resource and instead treats it as stale moving forward; note that we do not remove the resource from the cache, and instead leave this to the caching library's default eviction policy.

Marauder's other background task is to **facilitate the online prefetching of resources referenced by text files**. To do this, whenever a text file is added to the app cache (even if it is marked as no-cache, as per §3.1), Marauder asynchronously spawns a worker thread to parse the corresponding text file's body in search of referenced

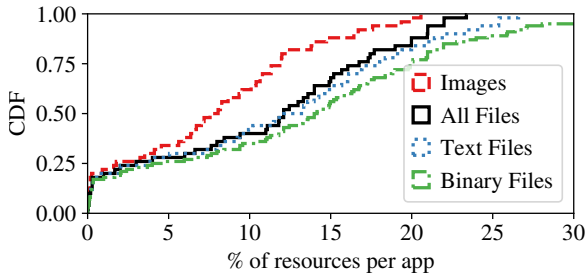


Figure 13: Percentage of resources per app that require types of dynamic query strings other than those used by other resources of the same type and from the same origin servers.

URLs. The worker statically analyzes the text file using a variety of regular expressions and the LinkedIn URL Detector Library [34] to find all strings that resemble a URL. Recall from observation 1 that such static analysis results in high coverage of the URLs that comprise an interaction. Each entry in the output of the regular expressions is either (1) an *absolute* URL that embeds a protocol, hostname, and resource URI, e.g., `https://www.foo.com/bar.jpg`, or (2) a *relative* URL which lists the resource URI and optionally the hostname, e.g., `/bar.jpg`. Relative URLs most often (96% of the time in our corpus) adopt the hostname and protocol of the referencing text file. Thus, Marauder follows this strategy to convert each relative URL into an absolute version.

Now that we have a full list of absolute URLs, the final challenge is to ensure that the URLs are listed in precisely the way that an app would request them during a client interaction. In particular, apps can augment absolute URLs with query parameters listing properties such as the time, date, latitude/longitude coordinates, screen size, and resource size/quality, e.g., `?width=400;time=1608759986`. Missing or incorrect values for any query parameter will result in a cache miss and thus wasted bandwidth during prefetching.

Certain query strings, in particular those that pertain to resource size, are most often statically listed in tandem with the corresponding URL in the referencing text files, and are thus captured by the aforementioned steps. However, other query parameters are inherently dynamic (e.g., timestamp, location), and are thus filled in during a client interaction. To handle such cases, we leverage our finding that, for a given origin, the set of dynamic query parameter types are almost entirely shared across all requests for a given resource type. For example, as listed in Figure 13, only 12% of requests for the median app involve different dynamic query parameters than those used by other requests for the same resource type and origin. Based on this finding, for each absolute URL, Marauder identifies a previously cached resource with the same content type and origin, and appends the corresponding dynamic query parameter types to the URL; note that the dynamic query parameter values are filled in at prefetch time (described in §4.3).

Certain text files clearly delineate sets of resources that should be fetched only after a subsequent interaction (i.e., after the one that triggered the fetch of the text file) is performed. In such cases, subsequent interactions are marked using their handler descriptions, e.g., onScroll. At the extreme, these files list thousands of resources to cover numerous future interactions. For instance, a handful of text files for the news apps in our corpus reference sets of resources that pertain to articles at different positions on the home screen. Since Marauder eschews predicting future interactions and instead performs JIT prefetching for already-triggered interactions, we do not include URLs that pertain to future interactions in our list of referenced resources to prefetch.

4.3 Handling Client Requests

During user interactions, requests made by the app hit the cache as normal. There are three potential scenarios, each of which warrants a different workflow with Marauder.

- For the first request for a text file (i.e., it is not in the cache, even as expired), or any request for a non-text file that misses in the app cache, Marauder immediately issues the request over the network. Upon receiving the response, Marauder sends the content to the app, adds it to the cache, and begins the background tasks from §4.2 to extract referenced resources if it is a text file.
- For text file requests that hit in the app cache (according to the caching library's default criteria for a hit), Marauder responds to the app with the corresponding content, and immediately issues asynchronous prefetch requests for all of the files referenced by that text file (using the list generated offline). Note that prefetch requests first pass through the app cache, ensuring that already cached resources will not be re-downloaded. To issue prefetch requests, Marauder first fills in values for dynamic query parameters. Then, Marauder applies the same set of request headers used in the request for the text resource with two exceptions. First, content negotiation headers such as "Accept-Encoding" are not carried over since the text and referenced files may differ in content type; Marauder defers the setting of values for these headers to the underlying caching library. Second, Marauder adds a "Referer" header listing the text resource; this header provides context for the server with regards to the specific interaction that the client is performing.

  To ensure that an app's explicit request for a resource maps to the version prefetched during the same interaction, Marauder tolerates discrepancies in certain dynamic query parameter values. For example, Marauder tolerates timestamp mismatches of several seconds and different random number values that are used for in-network cache busting query parameters.
- Finally, for text file requests that miss in the app cache but pertain to an entry marked as no-cache, Marauder issues a request for the text file followed by prefetch requests for all of its referenced children. In the event that the text file arrives prior to certain prefetched resources, Marauder is careful to queue subsequent requests that the app explicitly makes for which there is already an outstanding prefetch request. In this way, Marauder avoids duplicate requests and wasted bandwidth; queued requests are serviced as soon as the corresponding prefetched responses arrive.

4.4 Discussion

Preventing storage overheads. Caching libraries employ their own eviction policies, with the most common one being least-recently-used (LRU). With LRU, when space is needed by the cache or the host OS, the caching library deletes the resource whose last request was in the most distant past. However, using the above optimizations, Marauder issues requests for resources either to refresh their cache entries or to prefetch them just before the client requests them. To ensure that such requests do not skew resource access patterns and alter app eviction decisions such that less important resources are preserved, Marauder's requests bypass the caching library's bookkeeping of resource access frequency. In doing so, Marauder does not add any storage overheads to apps or caching libraries, and instead lets evictions happen as they normally do. Further, this ensures that Marauder does not continually refresh resources such that they are never evicted.

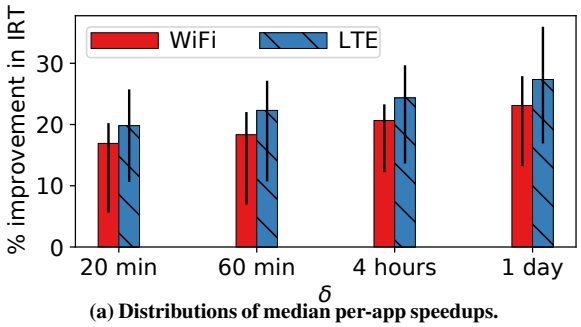
Preserving application behavior. Marauder only operates on (i.e., refreshes in the cache or JIT prefetches) resources fetched using HTTP GET or HEAD request types. These request types are defined to be "safe methods," in that the corresponding response generation logic should be idempotent and is not intended to trigger state changes on the server [1]; in this way, such methods are meant to enable tasks like web crawling, cache optimization, and prefetching, without risk of causing harm to the intended application behavior. As a result, Marauder's optimizations of downloading and serving cached or prefetched versions of resources should not affect the functionality for apps that respect these guidelines for servicing HTTP requests.

During operation, Marauder always respects the TTLs set by application developers via HTTP caching headers, and does not require any modifications to those TTLs by developers. More specifically, cache management for each resource that is prefetched or updated in the cache reflect the cache expiration headers during the latest request for that resource, and resources are only served to clients until their TTLs (and accordingly, their cache entries) expire. Thus, Marauder may not always serve the latest version of a resource to a client (e.g., if the content for a resource is updated on the server-side prior to its expiration in the client-side cache), but all of the resources served by Marauder have been marked by app developers as acceptable to use (via the TTLs that they have set). This behavior mirrors that of other HTTP caches [41].

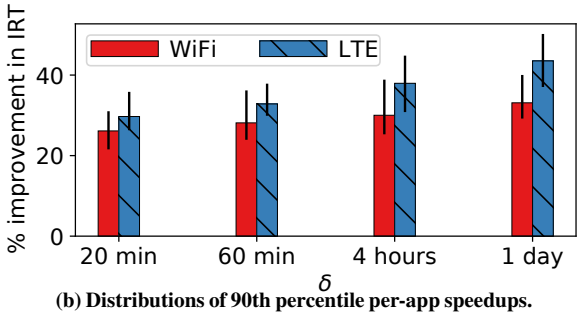
Generalizability and future outlook for Marauder. Marauder's optimizations extensively leverage the text-based communication between client-side apps and servers, i.e., by preferentially updating about-to-expire text resources in the cache, and using those resources to determine other required resources to prefetch for an ongoing interaction. The diverse apps used in our motivating measurements and evaluation (§2) highlight the prevalence of this behavior in the current app ecosystem. Moreover, we expect apps to continue employing this operational paradigm moving forward for several reasons: (1) many apps have converged on this behavior following over a decade of optimizations and alterations to their ecosystem since their inception [25], and (2) text-based communication simplifies cross-platform development, e.g., page-embedded JavaScript running in web browsers could interpret the same text files to determine resources to fetch during a web page load. However, for apps that do not use text files in this manner or if apps abort this practice in the future, we note that Marauder could still employ JIT prefetching in a less transparent way, e.g., by having developers explicitly specify referenced URLs in HTTP headers.

5 IMPLEMENTATION

To implement Marauder, we forked version 3.12.0 of the OkHttp caching library [54] and added ≈ 2500 lines of code to support the low-risk caching and prefetching optimizations from §4; we verified that identical changes can be made to later versions of OkHttp (e.g., v3.14.x and v4.8.x), but we used v3.12.0 as it was the most commonly used by the apps in our corpus. Marauder stores references to cached resources in need of refreshing in a PriorityQueue with constant access time, and performs refreshing (i.e., TTL extensions and text file updates) using Java ExecutorServices. For JIT prefetching, referenced resources are discovered with the help of the LinkedIn URL detection library [34]. Importantly, to avoid blocking the handling of explicit user requests, cache refreshing occurs in the background and JIT prefetching is performed asynchronously, both on separate worker threads. Further, to ensure that network and CPU resources are not flooded in the face of complex apps with hundreds of resources



(a) Distributions of median per-app speedups.



(b) Distributions of 90th percentile per-app speedups.

Figure 14: IRT improvements over default app caching and prefetching policies. Bars list median or 90th percentile speedups for the median app, and error bars span the 25-75th percentiles. δ is time between user sessions (traces) with an app.

per interaction, Marauder caps the number of outstanding requests that it makes (across cache refreshing and JIT prefetching) to 32.

To modify apps, we start with a JAR file housing Marauder's caching library and then disassemble that file using apktool [55] in order to extract the source code for the compiled Marauder classes. We then use apktool to disassemble our target app and check to see if it uses an unobfuscated version of the OkHttp library that does not contain custom modifications. If so, we replace the app's OkHttp library with Marauder's, and recompile the app into a new APK.

6 EVALUATION

Using the methodology and testbed described in §2, we evaluated Marauder across a wide range of popular apps, live mobile networks, real phones, and realistic user interaction traces. Our key findings are:

- Marauder reduces median and 90th percentile per-app interaction response times (IRTs) for the median apps in each case by 27.4% and 43.5%, compared to the default caching and prefetching policies embedded in our apps (§6.1).
- Marauder increases overall data usage for the median app by only 18%, and can also achieve 59% of its total benefits by only using its cache refreshing optimizations that minimally inflate data usage by 3% (§6.2).
- Marauder delivers 2.1× larger speedups than the recent Paloma prefetching system [61], while imposing 91% lower data overheads. Further, compared to app-supported prefetching policies that result in comparable (within 3%) data overheads, Marauder delivers 3.3× larger speedups.

6.1 App speedups

Figure 14 illustrates Marauder's ability to lower interaction response times (IRTs) compared to the default caching and prefetching policies employed by the apps in our corpus. For example, on LTE networks,

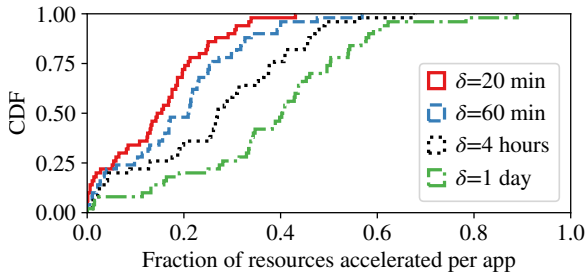


Figure 15: Larger durations between user sessions (δ) result in more opportunities for Marauder to accelerate resource fetches.

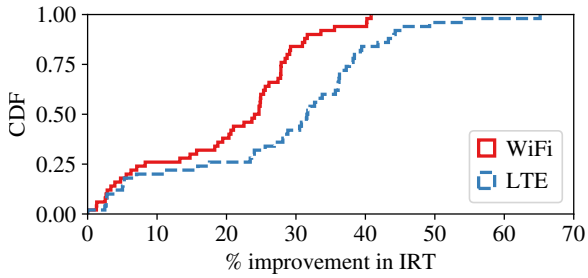


Figure 16: Distribution of median per-app IRT speedups for repeat interactions as compared to default caching+prefetching policies. Results consider a δ of 4 hours.

Marauder accelerates the median interaction for the median app by 19.8-27.4% (or 0.58-0.81 seconds); 90th percentile IRT improvements were 29.7-43.5% (or 0.87-1.27 seconds) for the median app. Marauder's median and 90th percentile benefits drop to 16.9-23.1% and 26.1-33.2% on WiFi due to the lower network latencies to servers.

Figure 14 also reveals that Marauder's improvements are more pronounced when the time between user sessions grows (δ from §2). For example, on LTE, median IRT improvements for the median app grow from 19.8% to 24.4% as δ increases from 20 minutes to 4 hours. Digging deeper into this, the reason is that larger δ values provide more opportunities for Marauder to speed up the fetches of required resources. In other words, larger δ values imply that fewer resources will hit in the client's cache under existing caching policies (due to expirations). Each cache miss provides Marauder with an acceleration opportunity, either via refreshing the cached entry (by extending its TTL or updating its text content), or by prefetching it during the subsequent interaction if its non-text content has changed. Figure 15 depicts the impact of this relationship, showing that larger δ values result in more resource fetches being accelerated by Marauder.

Recall from §2 that our setup considers realistic user traces with different sets of interactions in each. As a result, even after excluding the first trace in a given experiment, some interactions are not repeats of prior ones (i.e., they are cold cache interactions). Neither Marauder nor existing caching policies provide any speedups for cold cache interactions. To hone in on Marauder's target interactions, we analyzed the results in Figure 14 and excluded any interactions whose initial request was never seen by the cache. As shown in Figure 16, Marauder's median benefits grow to 24.4-31.7% when focusing solely on repeat interactions; for context, these benefits are 20.7-24.4% when all interactions are considered (as per Figure 14).

6.2 Analyzing Marauder

Ablation study. In order to understand the importance and contributions of each of Marauder's optimization techniques, we performed an ablation study in which we selectively disabled each one. As shown in Figure 17, background cache refreshing to improve hit rates for already-cached resources is the largest contributor to Marauder's benefits, delivering 13.3-16.1% speedup at the median across the considered δ values. In contrast, just-in-time prefetching provides median speedups of 3.4-8.6%. Figure 17 also shows that, as δ values grow, even though both optimizations remain important, the relative importance gap between the two techniques shrinks. The reason is that, especially for apps with significant amounts of dynamic content, the contents of more resources change, and the cache refresher's ability to extend TTLs for non-text resources becomes more limited.

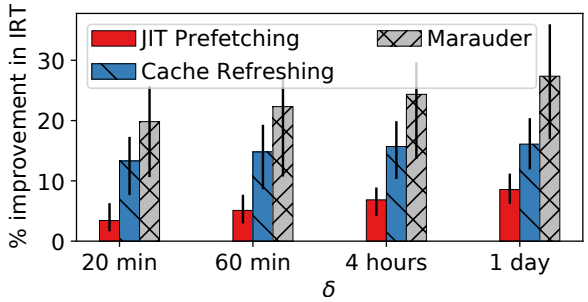


Figure 17: Breaking down Marauder's benefits into its constituent optimizations. Results are for LTE, and show distributions of median per-app IRT speedups for different δ values.

App	Cache Refreshing	JIT Prefetching	Both	Neither
Fox News	59.6%	12.8%	19.1%	8.5%
Uniqlo	25.9%	11.1%	44.4%	18.5%
Guardian	26%	13%	52.1%	8.7%
UEFA	18.2%	7%	11%	63.7%
Weather	22.6%	0%	0%	77.4%

Table 2: The source of Marauder's speedups varies across apps, and across the interactions within a given app. Values list the fraction of interactions for each app that benefit from each of Marauder's optimizations: cache refreshing and JIT prefetching. 'Both' pertains to interactions sped up by both of Marauder's optimizations, while 'Neither' includes those interactions that Marauder does not provide any speedups for.

Perhaps most importantly, these results also highlight the synergy between Marauder's optimization techniques: Marauder consistently delivers benefits that exceed the sum of those from its two optimizations in isolation. As an example, consider the speedups for a δ of 1 day. Median speedups with Marauder are 27.4%, while only cache refreshing or JIT prefetching yields improvements of 16.1% and 8.6%, respectively. The reason is that cache refreshing keeps text content up-to-date, which in turn ensures that Marauder JIT prefetches the appropriate resources required for an interaction. In other words, as text files grow out of date, the set of children that they reference also becomes increasingly stale with respect to the interaction.

Case study of interactions. The effectiveness of Marauder's optimizations is influenced by the loading patterns in the specific interactions being targeted. To better understand these relationships, we analyzed the interactions for the apps in our corpus, and identified three predominant patterns that affect the magnitude of speedups delivered by Marauder. Table 2 presents results for representative apps; note that a given interaction can exhibit multiple of these patterns simultaneously, e.g., the 'Both' column of Table 2.

First, Marauder's cache refreshing optimization is particularly helpful for interactions whose resources remain largely unchanged

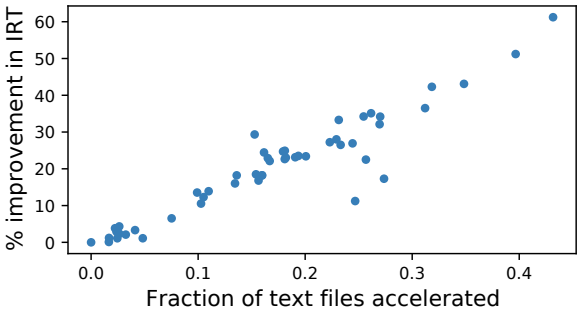


Figure 18: Marauder's ability to accelerate text file downloads is an accurate indicator of overall speedups. Each point represents one app, and results consider the LTE network.

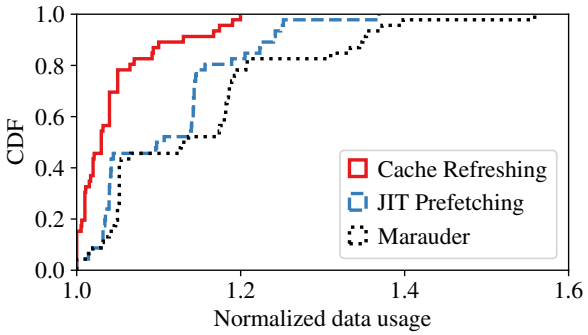


Figure 19: Distribution of normalized per-app bandwidth overheads with Marauder (and its individual optimizations) relative to default app operation. Results consider a δ of 1 hour.

across instantiations, and are assigned TTLs that are too conservative with respect to the rate at which the corresponding content actually changes. 74% of all interactions in our experiments (spread across all 50 apps in our corpus) exhibited this behavior, and involved synchronous downloads of resources that expired in the cache but whose content had not changed. This behavior is particularly common for item pages on e-commerce apps and for recent articles on news apps. Developers conservatively set resource TTLs in these cases so that they can update ratings/availability data for items and update articles as events unfold. For instance, as shown in Table 2, 70.3% of the interactions for Uniqlo are accelerated by Marauder's cache refreshing technique, of which 81.4% are item pages. Similarly, 78.7% of Fox News' interactions are sped up by Marauder's cache refreshing, of which 73.4% of are recent news articles.

Second, Marauder's JIT prefetching optimization provides benefits primarily for interactions that load lists or grids of content such as image galleries, catalogs, or topic pages. Such pages are typically indexed by text files that reference the set of resources to be loaded, and those resources can be JIT prefetched if they cannot be serviced by the app cache, i.e., if their content has changed since their last load, or if they have been newly added to the referencing text file. Across our corpus, 53% of interactions exhibited this pattern, including 31.9% of Fox News interactions and 65.1% of Guardian interactions.

Third, certain interactions load content that either (1) never changes, e.g., old news articles, sizing charts for a given e-commerce item, or statistics of completed sporting events, or (2) continually change, e.g., stock tickers or live sports scores. Marauder does not provide any speedups for such interactions as their resources are most often tagged with long TTLs (reflecting ossification) or no-cache headers (reflecting impending changes). We observe this pattern in 43 apps in our corpus, representing 8% of all interactions. Most notably, in the UEFA Scores app, 63.7% of interactions were unoptimized by Marauder (see Table 2), of which 73.3% involved rapidly-changing content and 26.7% involved purely static content.

The importance of text files. Key to both of Marauder's optimizations is the observation that text files are the highest priority resource to load in an interaction for two reasons: they are typically fetched in a blocking manner at the start of an interaction, and they list the remaining resources needed to handle the interaction (thereby guiding JIT prefetching). Unsurprisingly, we find that Marauder's ability to accelerate the loading of text files is a good indicator of Marauder's overall speedups for an app. Figure 18 illustrates this, showing that Marauder's overall improvements are larger for apps that have a high fraction of text file downloads accelerated by Marauder.

Bandwidth overheads. Figure 19 shows the amount of extra bandwidth that Marauder consumes to realize its speedups, as compared to default app operation. As shown, Marauder only consumes 1.18× more data for the median app in a given experiment, i.e., data is summed across all interaction traces used for the app in an experiment; this is far lower than that of existing prefetching systems and app-supported prefetching approaches (§6.3). Breaking this down further, there are two ways in which Marauder can increase an app's bandwidth usage: data overheads in refreshing cached resources, and JIT prefetching of resources that the app will not require. As expected, the former contributes only 16.7% of the overhead, with the majority of extra bytes coming from unnecessary prefetching. The reasons for unnecessary prefetching are that certain text files embed references to resources that are either (1) required by subsequent interactions which may or may not be triggered by the user, or (2) a superset of resources required for the current interaction from which the app binary chooses. While Marauder eliminates unnecessary prefetches for the former case (§4.2), it cannot for files that list resources in a seemingly random order and rely on app logic to select the subset to fetch. Importantly, because cache refreshing provides the bulk of speedups with Marauder, data-conscious users can opt to disable JIT prefetching to realize substantial IRT speedups with minimal bandwidth overheads.

6.3 Comparison with State-of-the-Art

We compared Marauder with three acceleration approaches: the recent Paloma [61] prefetching system, as well as the most aggressive and conservative prefetching policies supported by the apps in our corpus. We describe the comparisons in turn.

Paloma [61]. The open-source Paloma prefetching system performs static program analysis (both control and data flow) on app source code to identify (1) a set of static and dynamically-constructed URL strings to prefetch (these can be symbolic, with values determined at runtime), and (2) trigger points in the code at which to prefetch each URL. To determine trigger points, Paloma extracts a reachability graph across an app's callback handlers [58]; a user interaction is typically handled by a single callback handler. With this graph, Paloma denotes the trigger point for a URL that is fetched in callback Y as the end of the callback that immediately precedes Y in the graph. In doing so, Paloma only performs short-term prefetching, i.e., one callback/interaction ahead of the current one being handled.

Paloma failed to run on 33 apps in our corpus, so we report comparison results for the 17 that succeeded. As shown in Figure 20, Marauder provides larger IRT speedups than Paloma does. For example, with

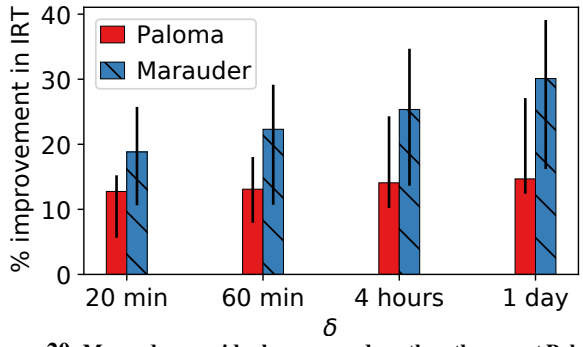


Figure 20: Marauder provides larger speedups than the recent Paloma prefetching system [61]. Bars list median speedups for the median app, and error bars span the 25-75th percentiles. Results are for LTE.

Approach	IRT improvement	Bandwidth overhead
Marauder	41.2% (49.6%)	1.13×
Most aggressive app-supported policies	78.1% (88.4%)	4.05×
Most conservative app-supported policies	12.3% (20.3%)	1.09×

Table 3: Marauder delivers far larger speedups compared to conservative app prefetching policies, while delivering comparable data overheads. Results use LTE and a δ of 4 hour, and list medians of per-app median (90th percentile) IRT improvements.

a δ of 4 hours, median speedups with Marauder are 11.3% larger than those with Paloma. There are three main reasons for this discrepancy.

First, for each interaction, Paloma often requires at least two rounds of prefetching to fetch the required resources. The reason is that Paloma must first prefetch the interaction's root text file before it can determine the resources that it references. In other words, Paloma's static analysis can deduce that a resource will be requested, but the precise URL can often only be determined by downloading and parsing the referencing text file. In contrast, Marauder keeps text files up-to-date in the background (and incurs little overhead in doing so), enabling the prefetching of an interaction's resources in a single round.

Second, Paloma does not improve hit rates for already-cached resources, which when possible, are the most effective way of eliminating network overheads. Indeed, our results from §6.2 show that background cache refreshing are a major source of Marauder's overall wins. The importance of background prefetching becomes more pronounced as δ values grow (and more cache evictions occur). For instance, as δ grows from 20 minutes to 1 day, the additional benefits that Marauder provides over Paloma rise from 6.1% to 15.4%.

Third, although Paloma does not explicitly rely on error-prone prediction of user behavior, it still results in significant bandwidth wastage, consuming 3.4× more bandwidth than the default operation of the median app. The source of this data wastage is Paloma's policy of prefetching the requests that could be made by all callbacks that are immediately reachable from the current one—not all of those callbacks will be triggered, and requests for untriggered callbacks result in wasted bytes. In contrast, Marauder only prefetches during an already-triggered user interaction, and incurs 96% lower bandwidth overheads (1.21× for the median app in this set).

App-supported prefetching policies. We also compared Marauder with the most aggressive and most conservative prefetching policies that each app supports. The most aggressive policy is the one that downloads as much content as possible, as often as possible, under as many conditions as possible (e.g., on WiFi and LTE); in our setting, these policies delivered the largest IRT speedups. In contrast, the most conservative policy does prefetch content, but downloads as little as possible outside of explicitly-requested content. As in §3.3.2, we exclude apps that do not support any prefetching policies (default or optional) from this comparison.

Table 3 summarizes our results. Unsurprisingly, the most aggressive policies deliver the largest speedups and data overheads (4.05×) since they prefetch all content reachable by a user at a given time. However, Marauder operates at a desirable point amongst the policies that deliver practical data overheads: Marauder increases data overheads by only 4% compared to the most conservative policies, while delivering 3.3× the speedups.

7 RELATED WORK

Prefetching for apps. Multiple systems prefetch resources in anticipation of user requests, each using a different way of determining what to prefetch, when, and where. We describe them in turn, and note that none improve hit rates for already-cached resources, a key source of Marauder's speedups.

One class of systems employ program analysis techniques on app source code to determine both when and what to prefetch [9, 31, 37, 61]. For example, Paloma [62] (described in more detail in §6.3) uses static analysis techniques and a local proxy to prefetch URLs one callback early. Similarly, APPx [9] uses static analysis to identify inter-request dependencies, and then (at a remote proxy) prefetches the dependent resources for each explicit request using online learning to fill in dynamic parameters. In contrast to these systems, our key observation is that downloaded text files have an inherent structure that highlights referenced resources that will be required for a given interaction. As a result, Marauder can make accurate and low-risk prefetching decisions directly from the generic resource caching layer; indeed, Marauder's bandwidth wastage is far lower than that of Paloma (§6.3) and that reported by APPx [9], e.g., median and worst-case data wastage is 1.18× and 1.56× for Marauder, but is reportedly 1.74× and 4.17× for APPx.

Other prefetching systems rely on apps to specify the set of objects to prefetch, and instead focus on shielding developers from determining when it is fruitful to do so [8, 24]. For example, the IMP library [24] dynamically adapts prefetching policies according to tracked hit rates for past prefetches and current network conditions. EBC [8] schedules prefetch requests upon screen unlock by balancing the probability that a given app will be used with the amount of traffic it hopes to prefetch. In contrast, Marauder is immediately deployable with existing apps and does not require developer effort. Further, Marauder does not rely on any predictions about user behavior, and instead prefetches only *after* an interaction is triggered.

Finally, Looxy [23] uses a proxy to passively monitor app requests and cluster them into groups that are typically requested together. Upon receiving a request, Looxy's proxy prefetches all other resources in the same group. Though lightweight, Looxy suffers from several drawbacks that Marauder avoids. First, Looxy is ill-suited for apps with highly dynamic content that result in continually changing clusters and wasted bandwidth. Second, Looxy only prefetches exact URLs seen in the past, and does not have support for dynamic content embedded in those URLs (e.g., query strings), limiting hit rates.

Caching optimizations. Numerous studies have observed inefficiencies in the way that HTTP caches (both for web browsers

and apps) operate [14, 35, 36, 41, 48, 49, 59, 60, 62]. These studies have shown that apps fall short of realizing the significant opportunities for caching [62] for a variety of reasons including lack of adherence to the HTTP caching specification [48, 60], improperly set TTLs [35, 36, 49], and lack of support for aliased URLs that share the same content [26, 30, 35, 36, 41]. Our analysis of existing caching policies (§3.3.1) mirrors these findings, and presents a fundamental tension in resolving the poor performance, i.e., ideal TTLs vary for a given resource. In addition, and unlike these studies, Marauder uses cache refreshing to transparently improve hit rates for already-cached resources, while ensuring low overheads.

In addition to the aforementioned studies, some systems also try to improve cache utility by caching resources at finer granularites (e.g., parts of resources) [38, 46, 56] or employing cross-app resource caching [60]. Marauder is complementary to both: Marauder's cache refreshing requests could benefit from finer-grained caching, and Marauder is agnostic to the apps that run atop it and can thus be used in a multi-app cache as well.

Additional app optimizations. Falcon[57] and PREPP [45] uses location, time, and device sensors to predict what apps a user will soon use, and preload those apps to mask startup delays. Marauder is complementary to these techniques, and instead focuses on optimizing app responsiveness *during* user interactions. Other systems [10, 11, 21, 22, 33] such as Tango [21] and Maui [11] offload computations (and the ensuing network fetches) to well-provisioned proxy servers in a way that balances potential speedups with mobile device energy and data consumption. Though effective, such systems pose significant scalability challenges to maintain proxy servers that can support large numbers of mobile clients [53]. Worse, by relying on proxy servers, these systems violate the end-to-end security guarantees promised by HTTPS which now dominates web/app transfers [20, 42]. In contrast, Marauder operates directly in an app's resource cache and thus preserves HTTPS security. Finally, EdgeReduce [44] and Procrastinator [50] reduce data usage at the cost of responsiveness by delaying resource downloads to determine whether they will be required. Marauder operates at a different point in the design space, aiming to reduce response times while minimizing the data overheads in doing so; recent surveys suggest that app providers typically opt for the latter goal [9].

8 CONCLUSION

This paper presents Marauder, a mobile app acceleration system that carefully packages caching and prefetching techniques in a way that sidesteps their associated risks, i.e., poorly set TTLs or stale content with caching, and wasted bandwidth with prefetching. The primary observation guiding Marauder's operation is that apps handle interactions much like the web, whereby downloaded text resources are structured to entirely list the set of remaining URLs to fetch. Leveraging this, Marauder introduces two *low-risk* optimizations from the app's cache. First, guided by cached text files, Marauder prefetches referenced resources during an already-triggered interaction. Second, to improve the efficacy of cached content, Marauder judiciously prefetches about-to-expire resources, extending cache lives for non-text resources, and downloading updates for lightweight (but crucial) text files. Overall, Marauder reduces median and 90th percentile interaction response times by 27.4% and 43.5%, while increasing data usage by only 18%.

Acknowledgements: We thank Anirudh Sivaraman and Omid Abari for their valuable feedback on earlier drafts of the paper, as well as Aishwarya Sivaraman, Yixue Zhao, Marcelo Laser, and Byungkwon Choi for useful discussions on the Paloma and APPx prefetching baselines. We also thank our anonymous shepherd and the MobiSys reviewers for their constructive comments. This work was supported in part by NSF grants CNS-1943621, CNS-2006437, and CSR-2105773.

REFERENCES

[1] 2014. Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. https://tools.ietf.org/html/rfc7231.
[2] 2020. AndroidViewClient. https://github.com/dtmilano/AndroidViewClient.
[3] 2020. FFmpeg. https://ffmpeg.org/.
[4] 2020. Scenecut Extractor. https://github.com/slhck/scenecut-extractor.
[5] Android Developers. 2019. Input events overview. https://developer.android.com/guide/topics/ui/ui-events.
[6] Android Developers. 2020. Keeping your app responsive. https://developer.android.com/training/articles/perf-anr#Reinforcing.
[7] AppDynamics. 2014. AppDynamics Releases App Attention Span Study Which Shows Nearly 90 Percent Surveyed Stopped. https://bit.ly/39pMAXM.
[8] Paul Baumann and Silvia Santini. 2017. Every byte counts: Selective prefetching for mobile applications. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 1, 2 (2017), 1–29.
[9] Byungkwon Choi, Jeongmin Kim, Daeyang Cho, Seongmin Kim, and Dongsu Han. 2018. Appx: an automated app acceleration framework for low latency mobile app. In *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies*. 27–40.
[10] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. 2011. CloneCloud: Elastic Execution between Mobile Device and Cloud. In *Proceedings of the Sixth Conference on Computer Systems* (Salzburg, Austria) *(EuroSys '11)*. Association for Computing Machinery, 301–314.
[11] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. 2010. MAUI: making smartphones last longer with code offload. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*. 49–62.
[12] Biplab Deka, Zifeng Huang, Chad Franzen, Joshua Hibschman, Daniel Afergan, Yang Li, Jeffrey Nichols, and Ranjitha Kumar. 2017. Rico: A Mobile App Dataset for Building Data-Driven Design Applications. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology* (Québec City, QC, Canada) *(UIST '17)*. Association for Computing Machinery, 845–854.
[13] Dimensional Research. 2015. Failing to Meet Mobile App User Expectations: A Mobile App User Survey. https://techbeacon.com/sites/default/files/gated_asset/mobile-app-user-survey-failing-meet-user-expectations.pdf.
[14] Kaushik Dutta and Debra Vandermeer. 2017. Caching to reduce mobile app energy consumption. *ACM Transactions on the Web (TWEB)* 12, 1 (2017), 1–30.
[15] Tammy Everts and Tim Kadlec. 2019. WPO stats. https://wpostats.com/.
[16] Hossein Falaki, Ratul Mahajan, Srikanth Kandula, Dimitrios Lymberopoulos, Ramesh Govindan, and Deborah Estrin. 2010. Diversity in Smartphone Usage. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services* (San Francisco, California, USA) *(MobiSys '10)*. Association for Computing Machinery, 179–194.
[17] Google. 2012. Speed Index - WebPagetest Documentation. https://sites.google.com/a/webpagetest.org/docs/using-webpagetest/metrics/speed-index.
[18] Google. 2020. Google Play Store. https://play.google.com/store.
[19] Google. 2020. Volley. https://developer.android.com/training/volley/index.html.
[20] Google. 2021. Transparency Report: HTTPS encryption on the web. https://transparencyreport.google.com/https/overview?hl=en.
[21] Mark S Gordon, David Ke Hong, Peter M Chen, Jason Flinn, Scott Mahlke, and Zhuoqing Morley Mao. 2015. Accelerating mobile applications through flip-flop replication. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*. 137–150.
[22] Mark S. Gordon, D. Anoushe Jamshidi, Scott Mahlke, Z. Morley Mao, and Xu Chen. 2012. COMET: Code Offload by Migrating Execution Transparently. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Hollywood, CA, USA) *(OSDI'12)*. USENIX Association, 93–106.
[23] Yao Guo, Mengxin Liu, and Xiangqun Chen. 2017. Looxy: Web Access Optimization for Mobile Applications with a Local Proxy. In *2017 IEEE 85th Vehicular Technology Conference (VTC Spring)*. IEEE, 1–5.
[24] 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*. 155–168.
[25] M. Hort, M. Kechagia, F. Sarro, and M. Harman. 2021. A Survey of Performance Optimization for Mobile Applications. *IEEE Transactions on Software Engineering* (2021).
[26] Sunghwan Ihm and Vivek S. Pai. 2011. Towards Understanding Modern Web Traffic. In *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference* (Berlin, Germany) *(IMC '11)*. Association for Computing Machinery, 295–312.

[27] Immobi. 2020. Mobile App Vs Website Statistics: How App Usage Compares To Mobile Web Visits In The United States. https://bit.ly/3sgObI9.

[28] Tushar Jain. 2017. 7 pre-launch mobile app performance metrics to measure. https://kaysharbor.com/blog/mobile/7-pre-launch-mobile-app-performance-metrics.

[29] Simon L. Jones, Denzil Ferreira, Simo Hosio, Jorge Goncalves, and Vassilis Kostakos. 2015. Revisitation Analysis of Smartphone App Use. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing* (Osaka, Japan) *(UbiComp '15)*. Association for Computing Machinery, 1197–1208.

[30] Terence Kelly and Jeffrey Mogul. 2002. Aliasing on the World Wide Web: Prevalence and Performance Implications. In *Proceedings of the 11th International Conference on World Wide Web* (Honolulu, Hawaii, USA) *(WWW '02)*. Association for Computing Machinery, New York, NY, USA, 281–292.

[31] Ding Li, Yingjun Lyu, Jiaping Gui, and William GJ Halfond. 2016. Automated energy optimization of http requests for mobile applications. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 249–260.

[32] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2019. Humanoid: A Deep Learning-Based Approach to Automated Black-Box Android App Testing. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering* (San Diego, California) *(ASE '19)*. IEEE Press, 1070–1073.

[33] Chit-Kwan Lin and H. T. Kung. 2014. Mobile App Acceleration via Fine-Grain Offloading to the Cloud. In *Proceedings of the 6th USENIX Conference on Hot Topics in Cloud Computing* (Philadelphia, PA) *(HotCloud'14)*. USENIX Association, 8.

[34] Linkedin. 2020. Url Detector. https://github.com/linkedin/URL-Detector.

[35] Xuanzhe Liu, Yun Ma, Yunxin Liu, Tao Xie, and Gang Huang. 2015. Demystifying the imperfect client-side cache performance of mobile web browsing. *IEEE Transactions on Mobile Computing* 15, 9 (2015), 2206–2220.

[36] Yun Ma, Xuanzhe Liu, Shuhui Zhang, Ruirui Xiang, Yunxin Liu, and Tao Xie. 2015. Measurement and analysis of mobile web cache performance. In *Proceedings of the 24th International Conference on World Wide Web*. 691–701.

[37] Ivano Malavolta, Francesco Nocera, Patricia Lago, and Marina Mongiello. 2019. Navigation-aware and personalized prefetching of network requests in Android apps. In *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. IEEE, 17–20.

[38] James Mickens. 2010. Silo: Exploiting JavaScript and DOM Storage for Faster Page Loads. In *Proceedings of the 2010 USENIX Conference on Web Application Development* (Boston, MA) *(WebApps'10)*. USENIX Association, 9.

[39] MindSea. 2020. 28 Mobile App Statistics To Know In 2020. https://mindsea.com/app-stats/.

[40] Ravi Netravali, Ameesh Goyal, James Mickens, and Hari Balakrishnan. 2016. Polaris: Faster Page Loads Using Fine-grained Dependency Tracking. In *Proceedings of the 13th USENIX Conference on Networked Systems Design and Implementation* (Santa Clara, CA) *(NSDI)*. USENIX Association, Berkeley, CA, USA.

[41] Ravi Netravali and James Mickens. 2018. Remote-Control Caching: Proxy-based URL Rewriting to Decrease Mobile Browsing Bandwidth. In *Proceedings of the 19th International Workshop on Mobile Computing Systems & Applications* (Tempe, Arizona, USA) *(HotMobile '18)*. ACM, 63–68.

[42] Ravi Netravali, Anirudh Sivaraman, James Mickens, and Hari Balakrishnan. 2019. WatchTower: Fast, Secure Mobile Page Loads Using Remote Dependency Resolution. In *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services* (Seoul, Republic of Korea) *(MobiSys '19)*. ACM, 430–443.

[43] R. Netravali, A. Sivaraman, K. Winstein, S. Das, A. Goyal, J. Mickens, and H. Balakrishnan. 2015. Mahimahi: Accurate Record-and-Replay for HTTP *(Proceedings of ATC '15)*. USENIX.

[44] Andreas Pamboris and Peter Pietzuch. 2015. Edge Reduce: Eliminating Mobile Network Traffic Using Application-Specific Edge Proxies. In *2015 2nd ACM International Conference on Mobile Software Engineering and Systems*. IEEE, 72–82.

[45] 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*. 275–284.

[46] KyoungSoo Park, Sunghwan Ihm, Mic Bowman, and Vivek S. Pai. 2007. Supporting Practical Content-Addressable Caching with CZIP Compression. In *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference* (Santa Clara, CA) *(ATC'07)*. USENIX Association, Article 14, 14 pages.

[47] Liliana Pinho. 2018. Mobile app performance: 1-second delay costs $0,08 per user. https://bit.ly/2Xwy5ff.

[48] Feng Qian, Kee Shen Quah, Junxian Huang, Jeffrey Erman, Alexandre Gerber, Zhuoqing Mao, Subhabrata Sen, and Oliver Spatscheck. 2012. Web caching on smartphones: ideal vs. reality. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*. 127–140.

[49] Feng Qian, Subhabrata Sen, and Oliver Spatscheck. 2014. Characterizing resource usage for mobile web browsing. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*. 218–231.

[50] Lenin Ravindranath, Sharad Agarwal, Jitendra Padhye, and Chris Riederer. 2014. Procrastinator: pacing mobile apps' usage of the network. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*. 232–244.

[51] Lenin Ravindranath, Jitendra Padhye, Sharad Agarwal, Ratul Mahajan, Ian Obermiller, and Shahin Shayandeh. 2012. AppInsight: Mobile App Performance Monitoring in the Wild. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Hollywood, CA, USA) *(OSDI'12)*. USENIX Association, 107–120.

[52] Lenin S. Ravindranath, Jitendra Padhye, Ratul Mahajan, and Hari Balakrishnan. 2013. Timecard: Controlling User-Perceived Delays in Server-Based Mobile Applications. In *The 24th ACM Symposium on Operating Systems Principles*. Nemacolin Woodlands Resort, PA.

[53] Ahiwan Sivakumar, Chuan Jiang, Seong Nam, P.N. Shankaranarayanan, Vijay Gopalakrishnan, Sanjay Rao, Subhabrata Sen, Mithuna Thottethodi, and T.N. Vijaykumar. 2017. Scalable Whittled Proxy Execution for Low-Latency Web over Cellular Networks. In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking* (Snowbird, Utah) *(Mobicom)*. ACM.

[54] Square. 2019. OkHttp. https://square.github.io/okhttp/.

[55] Connor Tumbleson and Ryszard Wisniewski. 2020. Apktool – A tool for reverse engineering Android apk files. https://ibotpeaches.github.io/Apktool/.

[56] Xiao Sophia Wang, Arvind Krishnamurthy, and David Wetherall. 2014. How much can we micro-cache web pages?. In *Proceedings of the 2014 Conference on Internet Measurement Conference*. 249–256.

[57] 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*. 113–126.

[58] 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* (Florence, Italy) *(ICSE '15)*. IEEE Press, 89–99.

[59] Ming Zeng, Tzu-Heng Lin, Min Chen, Huan Yan, Jiaxin Huang, Jing Wu, and Yong Li. 2018. Temporal-spatial mobile application usage understanding and popularity prediction for edge caching. *IEEE Wireless Communications* 25, 3 (2018), 36–42.

[60] Yifan Zhang, Chiu Tan, and Li Qun. 2013. CacheKeeper: a system-wide web caching service for smartphones. In *Proceedings of the 2013 ACM international joint conference on Pervasive and ubiquitous computing*. 265–274.

[61] Yixue Zhao, Marcelo Schmitt Laser, Yingjun Lyu, and Nenad Medvidovic. 2018. Leveraging program analysis to reduce user-perceived latency in mobile applications. In *Proceedings of the 40th International Conference on Software Engineering*. 176–186.

[62] Yixue Zhao, Paul Wat, Marcelo Schmitt Laser, and Nenad Medvidović. 2018. Empirically assessing opportunities for prefetching and caching in mobile apps. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 554–564.