# CacheKeeper: A System-wide Web Caching Service for Smartphones

**Yifan Zhang**
College of William and Mary
Williamsburg, VA 23185, USA
yzhang@cs.wm.edu

**Chiu Tan**
Temple University
Philadelphia, PA 19122, USA
cctan@temple.edu

**Qun Li**
College of William and Mary
Williamsburg, VA 23185, USA
liqun@cs.wm.edu

## ABSTRACT

Efficient web caching in mobile apps eliminates unnecessary network traffic, reduces web accessing latency, and improves smartphone battery life. However, recent research has indicated that current mobile apps suffer from poor implementations of web caching. In this work, we first conducted a comprehensive survey of over 1000 Android apps to identify how different types of mobile apps perform in web caching. Based on our analysis, we designed CacheKeeper, an OS web caching service transparent to mobile apps for smartphones. CacheKeeper can not only effectively reduce overhead caused by poor web caching of mobile apps, but also utilizes cross-app caching opportunities in smartphones. Furthermore, CacheKeeper is backward compatible, meaning that existing apps can take advantage of CacheKeeper without any modifications. We have implemented a prototype of CacheKeeper in Linux kernel. Evaluation on 10 top ranked Android apps shows that our CacheKeeper prototype can save 42% networks traffic with real user browsing behaviors and increase web accessing speed by 2x under real 3G settings. Experiments also show that our prototype incurs negligible overhead in most aspects on cache misses.

### Author Keywords

Smartphone apps; HTTP caching; Cache system; Redundant network traffic reduction

### ACM Classification Keywords

C.2.2 Computer-Communication Networks: Network Protocols − Applications; C.5.3 Computer System Implementation: Microcomputers − Portable devices

## INTRODUCTION

Web traffic is the dominant type of Internet traffic [7], and with the popularity of smartphones and tablets, an increasing amount of web traffic originates from mobile devices. The mobile web traffic has grown 35% in under a year [5], and now accounts for 20% of the U.S. web traffic [4]. Unlike conventional PCs, where web browser is the main source of web traffic, smartphones have another significant source of web traffic: dedicated mobile apps. The popularity of the ubiquitous smartphone is partly driven by these useful and entertaining mobile apps, all available for little or no cost. Since most mobile apps utilize some form of network connectivity, the network behavior of mobile apps is an important area of research. In this paper, we consider providing a system-wide HTTP caching service for mobile apps to fully exploit the benefits of web caching in smartphones.

An appropriate web caching implementation in mobile apps will benefit both users and network operators. With such an implementation, users can (a) experience a higher quality of service, since the data can be accessed faster locally, (b) lower costs, since users may have to pay a higher fee for downloading more data, and (c) conserve energy by reducing unnecessary data transmissions. Network operators also benefit when mobile apps implement web caching correctly since this reduces the congestion on the network, especially the last mile radio connections.

Despite the importance of web caching, large numbers of mobile apps have *imperfect web caching*, meaning that web caching is either implemented for only certain HTTP resources the apps request, or is not implemented at all. The reason is twofold: lack of library support and negligence from developers. For example, the Android platform provides two official HTTP client classes: `HttpURLConnection` and `Apache HTTP Client` [2]. Before Android 3.2 (API level 13), the `HttpURLConnection` class only provided an interface for caching implementation. Developers have to implement their own client-side caching mechanisms. Heavy programming burden will hold developers from doing so. Later, Android added an official implementation of client-side caching (i.e., the `HttpResponseCache` class) for `HttpURLConnection`. However, it still requires developers to call the library to add caching capability. Since apps without caching or with poor caching will still have the "look-and-feel", some developers will spend less time implementing and testing the caching behavior of their apps.

Our approach is to reduce the burden of mobile app developers by providing a caching-as-a-service layer. We propose CacheKeeper, an OS web caching service transparent to mobile apps for smartphones. CacheKeeper provides the correct web caching implementation with no effort on the part of mobile app developers. Developers do not need to install any additional libraries or incorporate any additional API calls to take advantage of CacheKeeper. Our caching layer is also backward compatible with existing apps. In addition, CacheKeeper is an OS-wide service, meaning that all mobile


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

http://dx.doi.org/10.1145/2493432.2493484

apps that generate web traffic will make use of it. This allows for cross-app caching opportunities, where an app can take advantage of cached data from another app. Finally, to prevent malicious apps from peeking web contents downloaded by other apps, CacheKeeper provides a means allowing apps to specify if the HTTP objects they downloaded can be stored by the caching service or not.

In this paper, we make the following contributions. *First*, to the best of our knowledge, we have conducted the first systematic and extensive measurement study of individual Android apps' caching behaviors. Existing studies all investigated from a perspective of network-wide traces [14,15]. Our measurement considered top ranked apps in each app category in Google Play (i.e., the official Android app store). In total, we have investigated 1300 top ranked Android apps. Our measurement results indicate that over 40% of the apps that generate HTTP traffic have the flaw of imperfect web caching. This number jumps to 58% when considering heuristic expiration. The average redundant HTTP traffic ratio due to imperfect web caching was 0.19 without counting heuristic redundant HTTP traffic. Since our measurement study was from a perspective of individual apps, we were able to make several observations that would not be possible by the existing studies. For example, we observed that categories of apps with high HTTP traffic intended to perform worse in web caching. For example, for the "News & Magazines" category, the redundant HTTP traffic ratio was as high as 0.45. We also observed that a notable number of apps had the problem of *same-click HTTP traffic redundancy*, where only one click on mobile apps generated multiple downloads for the same HTTP object even if the object was used only once by the apps. We incorporated special attention on same-click HTTP traffic redundancy in the design of CacheKeeper.

*Second*, we designed CacheKeeper, an OS web caching service for smartphones. CacheKeeper performs HTTP 1.1 compliant web caching transparently for mobile apps running above. CacheKeeper is user-configurable. Moreover, we provide privacy control in CacheKeeper: apps making privacy sensitive HTTP communications can declare their HTTP transactions (i.e., HTTP request/response pairs) as private so that they will not be cached by the caching service.

*Finally*, we implemented a prototype of CacheKeeper in Linux kernel, and evaluated it with extensive experiments. Our evaluation on 10 top ranked Android apps shows that our CacheKeeper prototype can save 42% HTTP traffic with real user browsing behaviors and reduce web accessing latency by half under real 3G settings. In the case of cache miss, where no HTTP request is served by the caching service, CacheKeeper incurs negligible overhead in most aspects.

## RELATED WORK

**Measurements of Web Usage in Smartphones.** The popularity of smartphones and tablets has driven a growing number of works on studying web usage in smartphones. Based on a dataset containing one-year-long web accessing log from 24 iPhone users, recent work [16] studies users' Internet accessing behaviors on smartphones. The study results show that dedicated mobile apps are used by users to visit the web much more frequently than browsers. This demonstrates the needs to ensure properly working web functions, including web caching, for mobile apps. Work [17] specifically investigates smartphone web traffic related to advertisements based on a large dataset collected in a major European mobile network. The results suggest that ad traffic is a major component of overall mobile web traffic. Work [14] compares smartphone web traffic and laptop web traffic based on a 3-week-long wireless communication trace collected in an enterprise environment. As one of the findings, the authors suggest that web caching in smartphones is not as effective as that in laptops. Similar to [14], Qian *et al.* [15] conduct a comprehensive measurement study on web caching in smartphones. By examining a one-day smartphone web traffic dataset collected from a cellular carrier and a five-month web access trace collected from a small user base, the study reveals that about 20% of the total web traffic examined is redundant because of poor web caching. In this work, we investigate the effectiveness of web caching in smartphones from a different perspective. Instead of analyzing mobile web traffic collected from service provider, we inspect web caching function of 1300 top ranked apps downloaded from the Google Play. This way, we can explicitly get, rather than inferring, information about how different types of mobile apps perform in web caching, which we believe will be helpful for future mobile apps and mobile platforms design.

**Reducing Web Accessing Latency in Smartphones.** A considerable amount of efforts have been invested in reducing web accessing latency in smartphones. To increase the operation speed of web browsers, work [19] proposes improved web caching on style/layout data. Work by Wang *et al.* [18] also studies the causes of slow web mobile browsers. The authors suggest the root cause is slow content loading. They propose a method of speculative loading to reduce web accessing latency when using smartphone browsers. PocketSearch [11] proposes to put results of certain cloud services like web search in smartphones' local storage to expedite service speed. Similarly, PocketWeb [12] proposes, using machine learning on a per-user basis, to prefetch web pages into smartphone's local storage to reduce web accessing latency. In this work, we take a different approach to reduce web accessing latency for smartphones. We propose to run web caching as a system service, so that we can compensate for the flaw of imperfect web caching in many mobile apps.

## MOTIVATION

Two major observations have led us to believe that it is desirable to provide web caching as a system-wide service for smartphones: *web caching imperfection in mobile apps* and *cross-app caching opportunities*.

### Web Caching Imperfection in Mobile Apps

We have conducted an extensive measurement study of top-ranked Android apps in Google Play to study the web caching behaviors of individual Android apps.

#### *Measurement Setup*

**Apps Selection.** The Google Play organizes apps into around 24 categories (shown in first column of Table 1). We downloaded the top 50 ranked free apps from each category, except

**Table 1. Summary of the app measurement study.**

| Categories on Google Play | I: Setup | | | II: Inter-click. redundancy | | III: Same-click redundancy | | IV: Advertisement | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 1 | 2 | 1 | 2 | 1 | 2 | 3 | 4 |
| | Apps tested | Has HTTP traffic | HTTP traf. per-click | Apps cnt.† | Traf. ratio† | Apps cnt.† | Traf. ratio† | Apps cnt. | Ad only | Ad traf. per-click | Cacheable traf. ratio† |
| books & refs | 50 | 30 | 42.3 KB | 12\|14 | 0.23\|0.26 | 3\|3 | 0.04\|0.04 | 19 | 10 | 10.3 KB | 0.77\|0.87 |
| business | 50 | 23 | 16.0 KB | 4\|15 | 0.06\|0.26 | 1\|1 | 0.01\|0.01 | 14 | 6 | 5.1 KB | 0.92\|0.93 |
| comics | 50 | 39 | 125.1 KB | 12\|23 | 0.19\|0.29 | 3\|6 | 0.03\|0.04 | 31 | 18 | 13.0 KB | 0.68\|0.89 |
| communication | 50 | 17 | 18.0 KB | 3\|10 | 0.02\|0.25 | 2\|2 | 0.01\|0.02 | 9 | 4 | 11.4 KB | 0.61\|0.94 |
| education | 50 | 31 | 130.3 KB | 9\|14 | 0.16\|0.27 | 2\|2 | 0.01\|0.01 | 24 | 15 | 13.7 KB | 0.88\|0.89 |
| entertainment | 50 | 37 | 105.7 KB | 18\|20 | 0.21\|0.25 | 3\|4 | 0.04\|0.05 | 29 | 4 | 32.0 KB | 0.89\|0.92 |
| finance | 50 | 16 | 29.4 KB | 3\|4 | 0.14\|0.19 | 0\|0 | 0\|0 | 14 | 7 | 15.7 KB | 0.87\|0.88 |
| health & fitness | 50 | 35 | 74.4 KB | 8\|12 | 0.11\|0.20 | 2\|3 | 0.04\|0.06 | 33 | 13 | 37.5 KB | 0.86\|0.94 |
| libs & demos | 50 | 29 | 82.7 KB | 10\|12 | 0.16\|0.17 | 4\|4 | 0.04\|0.04 | 24 | 16 | 19.9 KB | 0.81\|0.95 |
| lifestyle | 50 | 30 | 98.8 KB | 8\|13 | 0.10\|0.15 | 1\|1 | 0.01\|0.01 | 24 | 5 | 12.5 KB | 0.70\|0.78 |
| media & video | 50 | 37 | 122.9 KB | 8\|15 | 0.13\|0.18 | 6\|8 | 0.03\|0.04 | 32 | 16 | 38.0 KB | 0.94\|0.97 |
| medical | 50 | 33 | 31.1 KB | 3\|14 | 0.04\|0.15 | 0\|0 | 0\|0 | 30 | 20 | 16.6 KB | 0.85\|0.91 |
| music & audio | 50 | 28 | 98.6 KB | 11\|15 | 0.17\|0.19 | 4\|5 | 0.04\|0.04 | 25 | 6 | 39.0 KB | 0.87\|0.93 |
| news & mgzns | 150 | 129 | 232.2 KB | 92\|106 | 0.45\|0.50 | 41\|45 | 0.12\|0.12 | 94 | 7 | 39.4 KB | 0.85\|0.89 |
| personalization | 50 | 34 | 53.5 KB | 5\|14 | 0.06\|0.16 | 1\|1 | 0.01\|0.01 | 33 | 16 | 27.9 KB | 0.81\|0.88 |
| photography | 50 | 40 | 47.6 KB | 9\|16 | 0.05\|0.10 | 1\|2 | 0.01\|0.01 | 37 | 12 | 23.6 KB | 0.91\|0.93 |
| productivity | 50 | 26 | 27.7 KB | 3\|13 | 0.01\|0.07 | 1\|2 | 0.01\|0.01 | 22 | 10 | 12.5 KB | 0.80\|0.82 |
| shopping | 50 | 34 | 197.7 KB | 27\|28 | 0.44\|0.52 | 11\|13 | 0.11\|0.15 | 20 | 1 | 17.4 KB | 0.94\|0.97 |
| social | 50 | 17 | 112.1 KB | 8\|12 | 0.20\|0.24 | 1\|1 | 0.01\|0.01 | 16 | 3 | 30.7 KB | 0.87\|0.95 |
| sports | 50 | 44 | 227.2 KB | 34\|38 | 0.42\|0.47 | 17\|18 | 0.09\|0.09 | 31 | 3 | 18.1 KB | 0.82\|0.86 |
| tools | 50 | 40 | 25.6 KB | 6\|15 | 0.04\|0.08 | 0\|3 | 0\|0.01 | 36 | 21 | 19.4 KB | 0.78\|0.84 |
| transportation | 50 | 40 | 102.6 KB | 17\|23 | 0.19\|0.26 | 2\|2 | 0.03\|0.03 | 32 | 7 | 37.0 KB | 0.88\|0.92 |
| travel & local | 50 | 29 | 178.5 KB | 17\|22 | 0.31\|0.40 | 4\|5 | 0.02\|0.02 | 23 | 4 | 7.9 KB | 0.61\|0.89 |
| weather | 50 | 45 | 171.2 KB | 27\|32 | 0.26\|0.31 | 11\|12 | 0.03\|0.03 | 43 | 6 | 30.7 KB | 0.76\|0.83 |
| **Overall** | 1300 | 863 | 121.2 KB | 354\|500 | 0.19\|0.26 | 121\|143 | 0.03\|0.04 | 695 | 230 | 24.2 KB | 0.84\|0.90 |

†: Presented in the format of $a|b$, where $a$ and $b$ are the values without and with heuristically redundant traffic counted respectively.

for the "News and Magazines" category. In this category, we selected the top 150 ranked free apps. We paid more attention to the news apps because they all access web contents. In total we selected 1300 top ranked apps. We also examined the selected apps to ensure no app appears in two different categories.

**Web Traffic Generation.** We installed and used each app on a smartphone running Android 4.0 to see if the app generates web traffic. To achieve automated testing, we developed a tool (using the `ADB getevent/sendevent` utility) that can record and replay user inputs on the touch screen. Prior to running the automated measurement experiment, we first recorded the user inputs when we used an app. To ensure comprehensive app usage, we clicked all the representative buttons/tabs/links when recording the user inputs. During the measurement experiment, we replayed the recorded user inputs to test all the 1300 apps. The experiment has been run twice with a one-week interval between the two executions.

**Web Traffic Recording.** During the measurement experiment, we configured the smartphone to access the Internet via an HTTP debugging proxy [3], through which we could capture all the HTTP traffic the smartphone generated. The captured HTTP traffic was saved into trace files for later processing. Among the 1300 apps, there are 863 apps generating HTTP traffic. Table 1 column `I.2` shows the number of apps with HTTP traffic for each category. To quantify how much HTTP traffic an app generates, we computed the per-click HTTP traffic volume for each app, which is the ratio of an app's total HTTP traffic volume over the app's total number of clicks. Table 1 column `I.3` shows the average per-click HTTP traffic volume for each category.

**Web Caching Imperfection Identification.** When testing an app, we executed the app twice by replaying the user inputs twice with a short interval, and collected traces for the two executions. We chose a short execution interval because we wanted to ensure that the cacheable HTTP objects (defined in RFC 2616 [9]) obtained in the first execution are still fresh when the second execution happens. If the second trace contained the same cacheable HTTP objects as in the first one, and the cacheable objects in the first one were still fresh when the second execution occurred, then the app would be identified to have imperfect web caching, and the corresponding HTTP transaction (i.e., the HTTP request/response pair) in the second trace would be labeled as redundant. For an HTTP response that does not contain expiration time or validators (e.g., ETag, Last-Modified time), if it neither contains the `Cache-Control:no-store` directive, we treat it as *heuristic cacheable* (because in this case, according to RFC2616, HTTP caches can assign a heuristic expiration time to the response).

### *Measurement Findings*

**App HTTP Traffic and Web Caching Imperfection.** Figure 1 (a) plots, for the 24 categories of apps, the relationship between each category's per-click HTTP traffic and the category's percentage of apps with imperfect web caching. We can see that the ratio of apps with imperfect web caching in a category is roughly proportional to the category's average per-click HTTP traffic. We can also learn that almost all the (four out of five) categories whose per-click HTTP traffic is greater than 150 KB have more than half apps with imperfect web caching. This suggests that *imperfect web caching is a common among apps with high HTTP traffic volumes*.

**Inter-click HTTP Traffic Redundancy.** We label a redundant HTTP transaction as *inter-click redundant* if the origi-

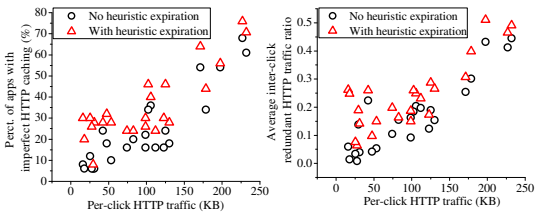



**Figure 1. Correlation between per-click HTTP traffic and (a) the number of apps with imperfect web caching, and (b) the average inter-click redundant HTTP traffic ratio.**

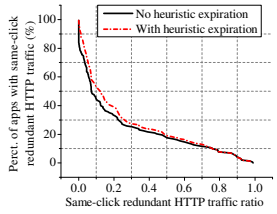



**Figure 2. CCDF of the same-click HTTP redundant traffic ratio.**

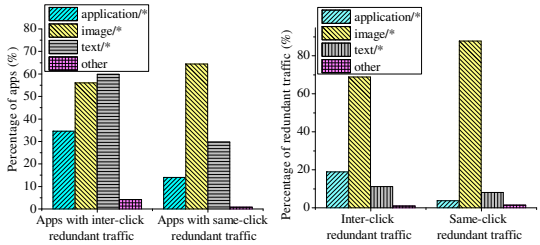



**Figure 3. Content type breakdowns for (a) number of apps and (b) redundant HTTP traffic.**

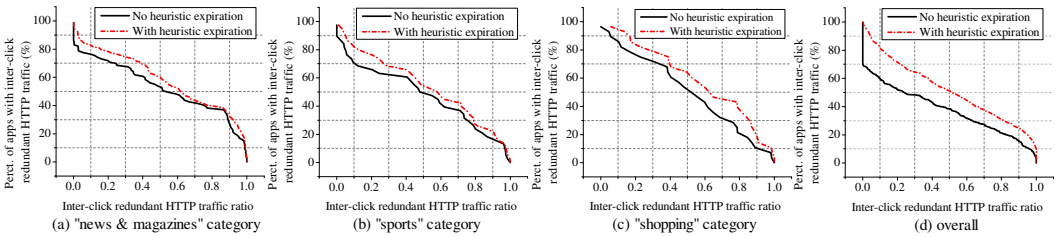



**Figure 4. Distribution of inter-click redundant traffic ratio: (a)-(c) show the CCDF of the redundant ratio for the apps with inter-click redundant traffic in the top 3 categories with the most per-click HTTP traffic; (d) shows the same statistics for all the imperfect apps.**

nal transaction and the redundant transaction occur as results of two different clicks on the same app. Table 1 column `II.1` shows the number of apps with inter-click redundant HTTP traffic for each category. We calculate the inter-click redundant traffic ratio of a category as the ratio of the category's total inter-click redundant traffic over its total HTTP traffic. Table 1 column `II.2` shows this value of each category. The inter-click redundant traffic ratio is 0.19 for all the apps tested. This number increases to 0.26 when counting heuristically redundant traffic. Figure 1 (b) plots, for the 24 categories, the relationship between each category's per-click HTTP traffic and its inter-click redundant traffic ratio. We can observe that those categories with high per-click HTTP traffic have much higher inter-click redundant traffic ratios. For example, the inter-click redundant traffic ratios for the top 3 categories with the most HTTP traffic are 0.45 (News & Magazines), 0.42 (Sports) and 0.44 (Shopping). To further study the distribution of the inter-click redundant traffic ratio among apps, we plot in Figure 4 the CCDFs (Complementary Cumulative Distribution Function) of the inter-click redundant traffic ratio for the previous three categories and for all the apps tested. From the figure we can learn that for the top three categories with the most per-click HTTP traffic, half of the apps with inter-redundant traffic have a redundant ratio greater than 0.5, which suggests *imperfect web caching is not only a common, but also a serious flaw for apps with high HTTP traffic volumes*.

**Same-click HTTP Traffic Redundancy.** We found that a notable amount of apps we tested downloaded the same resource multiple times for the same user click. We call those redundant HTTP transactions occurring for a single click on the app as *same-click redundant* HTTP transactions. Table 1 column III.1 and column III.2 list, for each category, the number of apps with same-click redundant HTTP traffic and the same-click redundant HTTP traffic ratio. Overall, about 10% of the apps have same-click redundant HTTP traffic, and the average same-click redundant traffic ratio is 0.03. However, similar to the case of inter-click HTTP traffic redundancy, these two figures are much higher for those categories with high HTTP traffic volumes. For example, for the top three categories with the most HTTP traffic, more than 20% of the apps have same-click redundant HTTP traffic, and the redundant traffic ratio is around 10%. We plot the CCDF of the same-click redundant ratio in Figure 2, which shows that about 40% of all the apps with same-click redundant HTTP traffic have a redundant ratio greater than 10%.

By carefully examining the web contents that involved same-click redundant HTTP transactions, we confirmed that those redundant downloads for the same click were not because the same resources needed to be displayed at several places on the same web page. We believe the main cause for same-click redundant HTTP transactions is developer error. As an evidence, a well-known online shopping and auction app had a self-redundant traffic ratio of 0.64 for the version we tested, and the problem was fixed in a new version when we retested the app several months later.

**Content Types of Redundant HTTP Traffic.** By extracting the `Content-Type` field from the HTTP response headers, we identified three major types of HTTP resources appeared in the measurement experiment: `application/*`, `image/*` and `text/*`. Figure 3 (a) shows for all the apps with redundant HTTP traffic, the percentage of apps neglecting to cache each type of HTTP resources. In the figure, all the types other than the three major types are labeled as `other`. According to our experience, many of the apps with redundant traffic on image resources only cache large images, but fail on caching small images like thumbnail images for news lists. Meanwhile, almost all the apps with redundant traffic on text resources fail to cache all kinds of text objects such as configuration files and data files. Figure 3 (b) shows the content type breakdown for the redundant HTTP traffic. We can learn that image resources took the most redundant traffic. In

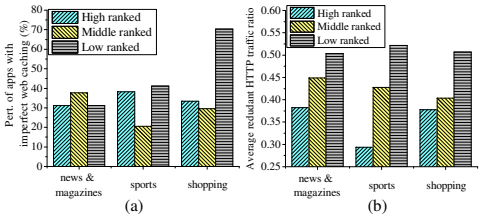



**Figure 5. Web caching imperfection and app rankings.**

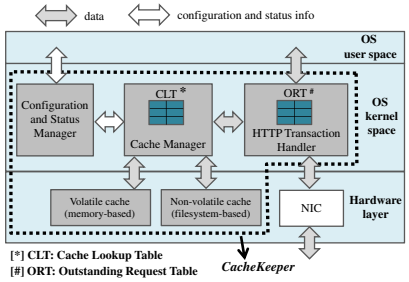



**Figure 6. CacheKeeper architecture.**

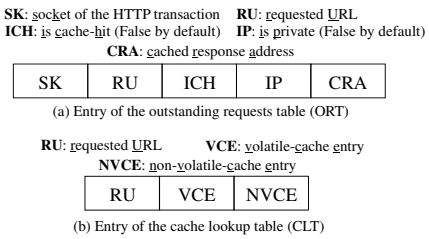



**Figure 7. Entry structures of ORT and CLT.**

the meantime, text resources also account for about 10% of all the redundant HTTP traffic.

**App Ranking and Web Caching Imperfection.** We have investigated whether app rankings have relationship with imperfect web caching. For the top 3 categories with the most per-click HTTP traffic we tested, we divide their apps into three groups (i.e., high, middle and low ranked) according to the app rankings on Google Play. We plot the percentage of apps with imperfect web caching and the average redundant HTTP traffic ratio of each group in Figure 5 (a) and (b) respectively. The shopping category has a much higher percentage of imperfect apps in the low ranked group. Meanwhile, for all the three categories, there is a clear increasing trend for redundant traffic ratio from the high ranked group to the low ranked group. Thus, we can cautiously make the conclusion that apps with lower ranking are more likely to have poor web caching implementation. This is reasonable because high ranked apps are usually developed by experienced and well-known developers, who are more likely to pay attention to details like web caching for their apps.

### Cross-app Caching Opportunities

Same-app web caching reduces web accessing latency and saves bandwidth for an app when it accesses the same cacheable content more than once. Meanwhile, cross-app web caching can also achieve the same benefit for different apps accessing the same web content. We have identified two types of cross-app caching opportunities for mobile apps.

**Opportunities by User Behaviors**. The first type of opportunity comes when a user uses different apps to access the same web content. For example, many top-ranking news reader apps on Google Play (such as *Flipboard*, *Pulse* and *Yahoo!*) allow users to view the news they are browsing on phone's web browser. This is a useful feature because usually a web browser provides more full-fledged web content rendering support. With this feature, users may access the same piece of news several times with both the news reader app and a web browser. Another example is that when a user wants to do online shopping with his smartphone, he may first use a web browser to search for the product and compare prices and reviews. After seeing that an online retailer, Amazon.com for example, provides the product for the lowest price, the user opens Amazon's dedicated shopping app to complete the transaction.

**Opportunities by Shared Libraries**. The second type of opportunity comes when two different apps use the same shared library that regularly accesses web contents. Mobile advertising network SDKs are the most notable kind of shared library. The way that a developer puts ads in his app is to call functions from an ad library provided by the mobile ad network. The app will download (or the ad network will push) advertisements to the smartphone running the app dynamically when the app is being used. Mobile ads are common in free mobile apps. For example, among the 1300 apps in our measurement experiment, 695 apps generate ad HTTP traffic (Table 1 column `IV.1`); HTTP traffic of 230 apps are all ad traffic (Table 1 column `IV.2`); and the per-click ad traffic is 24.2 KB (Table 1 column `IV.3`), which accounts for 20% of the per-click HTTP traffic. In the mean time, most of the ad traffic is cacheable: as shown in Table 1 column `IV.4`, the overall cacheable ad traffic ratio is 0.84 (or 0.9 if considering heuristic expiration). Considering that the mobile ads market is dominated by just a few ad networks [17] and that the ads to be shown are usually determined based on the user information such as user's location [10], it's likely that different apps running on the same phone will display the same set of ads over time. According to our experience, even two different ads from the same ad network usually share common cacheable objects like configuration scripts and data files.

## SYSTEM DESIGN

### Design Goals and Challenges

**Design Goals.** We design CacheKeeper (CK for short) with the following goals in mind.

1. CK should be able to perform standard-compliant (RFC 2616 [9]) web caching for all the entities (e.g., apps) making HTTP requests in the device. This is the fundamental goal of designing CK.
2. CK should be transparent to all the entities that it serves. In other words, entities making HTTP requests should be able to perform normally without any modifications. This is to ensure backward-compatibility for existing apps.
3. Since CK is essentially a shared client-side cache, the design of CK should provide means to protect apps' cache privacy.
4. While cache hits will bring benefits, CK should also incur low overhead on cache misses to ensure good usability.
5. CK should provide interfaces allowing users to configure the web caching services (e.g., cache size, cache location and heuristic expiration time) and to obtain service status.

**Challenges.** The design of a client-side system-wide caching service such as CK is different from implementing a cache in an individual app.

In particular, when compared with app-based client cache, there are two challenges: The *first* challenge is the ability to handle a large volume of concurrent HTTP transactions while incurring low overhead. This is different from caching in individual apps where HTTP requests are issued less frequently and usually in a sequential manner. The *second* challenge is that, unlike individual apps where web caching is part of the operations handled by HTTP libraries, CK is not in the network operations flow of the apps it serves. Thus, it is challenging to design and implement CK without making any modifications to the apps. For example, since fetching content from web cache is fast, it is designed as a synchronous operation in individual apps (i.e., the program execution blocks until the fetching operation finishes). However, fetching content from web cache cannot be designed as a simple synchronous operation in CK. This is because apps use asynchronous requests (i.e., request-then-poll) to retrieve content from web servers. Acting as a transparent middle layer between apps and web servers, CK cannot serve asynchronous requests from apps by using simple synchronous web cache retrieving. Otherwise it will be extremely inefficient and unscalable.

#### CacheKeeper Architecture

The architecture of CK is shown in Figure 6. CK is designed as an OS kernel space component providing web caching service to apps running in the user space. We choose to place CK in OS kernel space for three reasons. *First*, this approach has clear performance advantage over user space based approaches (e.g., user-level HTTP proxy). *Second*, this allows the web caching service to be *portable* across different devices running the same type of OS kernel. *Third*, by placing it in kernel space and not changing the interfaces connecting user and kernel spaces, we could easily achieve *backward-compatibility*.

CK contains the following components: the HTTP transaction handler, the cache manager, the configuration and status manger and the physical caches. Next, we give a description of each component, followed by the description of how the components cooperate in CK operations.

**HTTP Transaction Handler.** The HTTP transaction handler handles HTTP requests issued from apps and HTTP responses retrieved from network connections. The transaction handler consults the cache manager for cached responses, and passes incoming responses to the cache manager for caching processing.

The transaction handler uses a key data structure, *outstanding requests table* (ORT), to handle the asynchronous web content requests from apps mentioned previously. Each ORT entry corresponds to an HTTP request waiting to be served. Figure 7 (a) shows the ORT entry structure. Since CacheKeeper needs to process a large amount of concurrent HTTP transactions, we use socket address plus the requested URL to identify individual HTTP transactions. The `SK` field and the `RU` field record the socket address and the requested URL of the corresponding HTTP transaction respectively. The `ICH` field records if CK has a fresh cached response for the HTTP request. The `IP` field records whether the app issuing the request has declared that the HTTP transaction is private and thus should not be cached by CK. The default value of both `ICH` and `IP` is False. If the request can be served by CK, the field `CRA` holds the address of the buffer that is prepared by the cache manager and stores the cached response.

**Cache Manager.** The cache manager performs the following tasks. It accepts and processes queries for cached response from the transaction handler. It accepts newly coming responses from the transaction handler, caches them in a proper physical cache, and performs cache replacement if necessary. It accepts and processes configuration or status query requests from the configuration and status manager.

To help manage the cache entries, the cache manager maintains a key data structured named *cache lookup table* (CLT). Each CLT entry, with the entry structure shown in Figure 7 (b), corresponds to the cached HTTP transaction (i.e., a cached HTTP request/response pair) of a certain URL. The field `RU` records the URL of the cached transaction. The fields `VCE` and `NVCE` store the addresses of the volatile cache entry (i.e., memory-based) and the non-volatile cache entry (i.e., filesystem based) of the HTTP transaction respectively.

**Configuration and Status Manager.** The configuration and status manager provides interfaces to user space programs to configure CK and to query the running status of CK for debugging purposes.

**Physical Caches.** CK supports two types of caching media: *volatile cache* residing in device's memory and *non-volatile cache* residing in device's filesystem. The volatile cache is for efficient cache lookup, and the non-volatile cache is to ensure persistent cache content after reboots.

#### CacheKeeper in Operation

**On Cache Hits/Misses/Validations**. The transaction handler handles every HTTP request from apps and HTTP response from the network. Upon receiving an HTTP request, the transaction handler creates a new ORT entry, and consults the cache manager to see if CK has a freshed cached response for the request. The cache manager looks up the CLT by comparing the URL provided by the transaction handler and the `RU` field in the CLT entries, and sends the result back to the transaction handler, which in turn updates the `ICH` field in the ORT entry based on the result. *If there is a cache hit*, the cache manager retrieves the cached response from either the memory based cache or the filesystem based cache based on the `VCE` and the `NVCE` fields in the CLT entry, and notifies the transaction handler about the address of the buffer storing the cached response. The transaction handler then records this address in the `CRA` field of the ORT entry. Till now, the transaction handler has the complete ORT entry, through which the handler knows how to serve the later polls from the app (recall that apps use request-then-poll to retrieve contents from web servers). *If there is a cache miss or the cache response is expired*, the HTTP request is sent out as normal. The transaction handler passes the HTTP response to the cache manager for storing if the response is cacheable. *If the cached response needs to be validated* before it can be served to the apps, the transaction handler uses the validator (e.g., ETag, Last-Modified time) provided in the cached response to issue a conditional request to the web server. Based on the result of

the conditional request, the following operations are similar to the cache hit or miss situation described previously.

**Dealing with Same-click Redundant Requests.** During the measurement experiment, we observed that a notable amount of apps generated same-click redundant HTTP traffic. CK will naturally eliminate same-click redundant traffic if it has cached the previous response for a redundant HTTP request. However, a deeper investigation into the same-click redundant HTTP transactions we obtained shows that HTTP requests of about 20% of the same-click redundant transactions were issued before the full responses of the first HTTP transactions were received, in which case CK would send out those redundant requests. To solve this problem, the transaction handler postpones sending out an HTTP request for a short period of time if the requested URL is found in an ORT entry. Based on our experience, we set the length of this period to 200 ms in our prototype implementation.

**Declaring Private HTTP Transactions.** As one of the design goals, CK should provide means to protect apps' cache privacy. In our design, we allow apps to decide if they want their HTTP traffic to be cached by CK. Specifically, we provide an interface for apps to declare privacy for each HTTP transaction they generate. If an HTTP transaction is declared as private, it will not be cached by CK. We will present the implementation of the interface in the following section.

## SYSTEM IMPLEMENTATION

We have implemented a prototype of CK as a loadable Linux kernel module (kernel version: 3.0.15).

**Location in Linux Kernel.** Since HTTP communication usually takes place over TCP connections [9], the CK module intercepts TCP data flow at a location between socket and the TCP protocol implementation (Figure 8 (a)). We make this choice for the following three reasons. *First*, running CK under the system call interface can guarantee its backward-compatibility, since the interfaces between user space and kernel space remain untouched. *Second*, implementing the caching service above the TCP layer allows us to use socket information to distinguish different HTTP transactions. *Third*, running CK at a hight level in kernel's network data flow avoids the needs of considering packet fragmentation, which reduces implementation complexity.

**The HTTP transaction handler.** The transaction handler inspects every intercepted TCP message, and processes those related to HTTP. Since a long HTTP response may be divided by web server into several shorter HTTP messages, the transaction handler also needs to reassemble partial HTTP response messages into a complete one. Our implementation supports reassembling for both messages with explicit `Content-Length` header field and messages using chunked transfer encoding [9]. The outstanding request table (ORT) is implemented as an array with 128 entries. According to our experience, 128 ORT entries are enough because the amount of concurrent HTTP requests waiting to be served is not a larger than 100 in all of our tests.

**The Cache Manager.** The cache manager executes HTTP caching logic according to the RFC 2616 specification. Our current implementation supports caching with explicit expiration time, caching with validation and caching heuristic expiration. To achieve efficient cache lookup, the cache lookup table (CLT) is implemented as a dynamic hash table indexed by the `RU` field. The initial number of CLT entries is 1024. When the CLT is 80% full, it is expanded by adding 256 empty entries. To achieve hash table indexing and also save memory space for the CLT, we place a hashed URL value, instead of the actual URL (which may be of hundreds of bytes), in the `RU` field of a CLT entry (the same implementation applies to ORT entries). To achieve consistent web caching between reboots, we write the CLT to a file before system reboots or unloading the CK module, and read the CLT into memory right after the CK module is loaded.

**Cache Replacement.** If adding a new HTTP transaction to a cache (volatile or non-volatile) will cause the cache's size exceeds the configured value, the cache manager deletes a cache entry from the cache. Our current implementation adopts the simplest replace policy: deleting the oldest cache entry. In the future we plan to implement different types of cache replacement algorithms, and evaluate how the replacement algorithm can affect the performance of CK.

**Private Transaction Declaration Interface.** To declare an HTTP transaction as private, an app adds a comment string "CK-Private" to the `User-Agent` header field of the request. The transaction handler marks the `IP` field of the corresponding ORT entry as "True" if the comment string is found, and will never cache the response in the shared cache. Since web servers will ignore comments in HTTP headers, this approach does not affect the app's normal function. Note that this method is used by apps to choose whether its HTTP responses can be put in a shared cache. This is different from the `Cache-Control:no-store` directive in RFC 2616, which is used by the web server to declare that a response should not be stored by any cache.

**User Configuration Interface.** We provide an interface, by utilizing the `/proc` filesystem, for users to configure CK and to obtain CK status. Figure 8 (b) shows the screenshot of a CK configuration interface. The configuration options include turning on/off CK, setting caching location, setting sizes of caches, enabling/disabling heuristic caching and setting heuristic expiration time.

## ADDITIONAL DISCUSSION

**Caching HTTPS Traffic.** HTTPS traffic can be cached by web clients. However, the result of our Android app caching measurement study did not contain statistics on HTTPS traffic. This is because contents of HTTPS transactions were encrypted, and could not be parsed by our analysis program. However, among the 1300 selected apps, only 10% of them generate only HTTPS traffic. The current design of CK does not support caching HTTPS traffic. One way to enable HTTPS caching is to generate a CK certificate accepted by both apps and web servers. This way, CK can decrypt and analyze through HTTPS traffic, perform caching and encrypt traffic back. We leave supporting HTTPS caching in CacheKeeper to our future work.

**Privacy Considerations**. Sharing web cache among applications brings privacy concerns. For example, sensitive objects

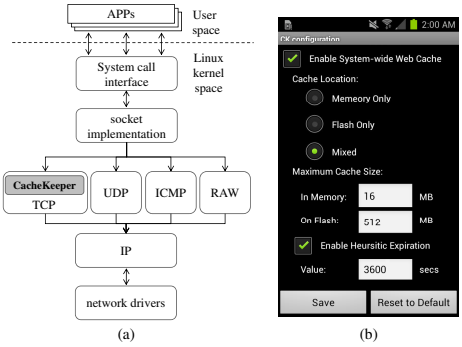



**Figure 8. CacheKeeper implementation: (a) location in Linux kernel; (b) the user configuration interface.**

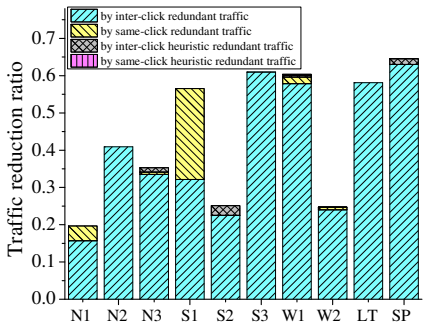



**Figure 9. Source breakdown of HTTP traffic reduction ratio for the 10 tested apps.**

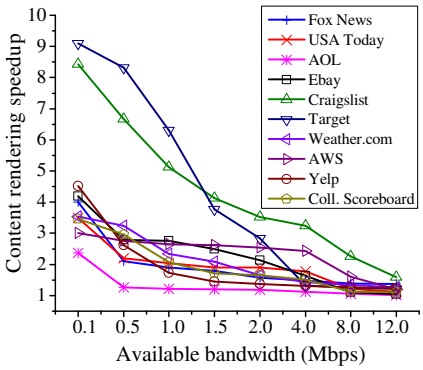



**Figure 10. Web content rendering speedup of the 10 tested apps under different transmission bandwidths.**

of one app may be accessed by other apps. A more sophisticated case is that a malicious app could use the time difference of downloading certain HTTP objects to determine if the user has viewed certain web contents, which is similar to the timing attacks of website accesses [8]. The simplest and most effective solution is to disable shared caching for sensitive HTTP objects. To this end, CK allows an app to declare if an HTTP transaction is private to the app and should not be stored by the shared cache. However, this solution requires app modifications, and thus is not applicable to legacy applications. To better solve this problem, we are considering other solutions including randomizing cache access times [6, 8], and fingerprinting app web access patterns to detect malicious cross-app cache accessing.

**The Userspace Loopback Alternative**. An alternative way to achieve system-wide web caching is to install a web proxy supporting caching at the localhost address. All the HTTP traffic will be looped back through the web proxy, which will handle the HTTP caching for the phone. While this approach might work in theory, we argue that CK still has several notable advantages over it. First, web proxy is not specifically designed for running in a phone. Instead, it is usually running in a standalone computer and has a design goal of efficiently serving HTTP requests from a large amount of computers. Therefore, the loopback approach is not able to take care of issues specifically related to app semantics (e.g., performance and privacy issues) like CK does. Second, the userspace loopback approach apparently adds more time to completing an HTTP transaction. On the contrary, CK is much more lightweight as it operates in the kernel. Third, although our prototype CK currently only supports basic web caching, it is possible to add more functions, such as compression and JavaScript rewriting, to CK, and allow it to be functionally comparable to standalone web proxies.

## SYSTEM EVALUATION

We evaluated our CK implementation in a Samsung Galaxy S2 smartphone running Android 4.0.3.

### Case Evaluation: App Performance Gains

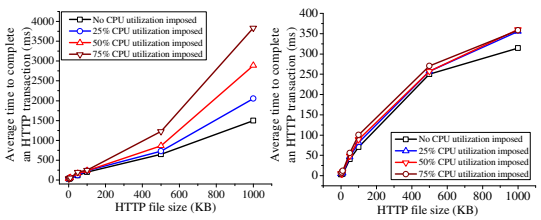
We selected 10 top ranked apps with imperfect web caching from Google Play (listed in the left part of Table 2). All these apps were ranked top 20 in their categories. Before performing the measurement experiments, we first used the 10 apps, each for three minutes, on the Samsung Galaxy S2 smartphone, and recorded the user inputs when using the app. We instrumented CK such that it can record different statistics of HTTP traffic, including the amount of total HTTP traffic, the amount of HTTP traffic served by the caching service and the amount of traffic with different cacheability.

**Table 2. HTTP traffic ratios of the 10 tested apps.**

| Category | App Name | Served by CK | New & cacheable | Non-cacheable |
|---|---|---|---|---|
| News | Fox News (N1) | 0.1967 | 0.4154 | 0.3879 |
| | USA Today (N2) | 0.4091 | 0.2075 | 0.3834 |
| | AOL (N3) | 0.3528 | 0.1710 | 0.4762 |
| Shopping | Ebay (S1) | 0.5654 | 0.1218 | 0.3127 |
| | Craigslist (S2) | 0.2512 | 0.3823 | 0.3665 |
| | Target (S3) | 0.6098 | 0.0734 | 0.3168 |
| Weather | Weather.com (W1) | 0.6035 | 0.0716 | 0.3249 |
| | AWS (W2) | 0.2472 | 0.1363 | 0.6165 |
| Local&Travel | Yelp (LT) | 0.5810 | 0.0164 | 0.4026 |
| Sports | Coll. Scoreboard (SP) | 0.6454 | 0.2161 | 0.1384 |
| Overall | | 0.4205 | 0.2388 | 0.3407 |

**HTTP Traffic Reduction.** In this experiment, we aimed to investigate how the 10 top ranked apps can benefit from CK in terms of HTTP traffic reduction. We ran the 10 apps on the smartphone by replaying the recorded inputs from real user every 30 minutes in one-day period. This is to simulate a user accessing an app every 30 minutes (note that the actual benefits achieved by CK depend on how often the web contents accessed by the user are updated, which is further determined by how often the user uses the app and how often the app updates its web contents).

Table 2 presents the ratios of traffic obtained in the experiment. The third column is the ratio of HTTP traffic served by CK, which is also the traffic reduction ratio. The fourth and the fifth column of Table 2 are the ratio of those first-time appeared cacheable traffic and the ratio of non-cacheable traffic respectively. The sum of the values in these three columns is 1. In the experiment, the overall HTTP traffic reduction ratio is 0.42. Among the 10 apps, 5 of them enjoyed a traffic reduction of over 50%. The traffic reduction ratio of an app is determined by two factors: how well web caching is im-




(a) Serving HTTP transactions from network downloading




(b) Serving HTTP transactions from local cache

**Figure 11. Transaction times under different system loads.**

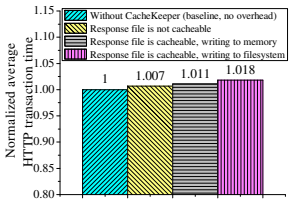
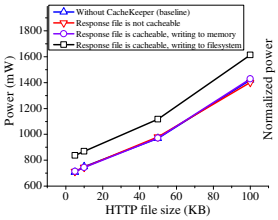
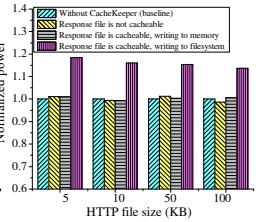



**Figure 12. Processing time O/H.**




(a) Absolute power values for downloading files with differet sizes




(b) Normalized power values for downloading files with differet sizes

**Figure 13. Power consumption O/H.**

plemented in the app and how often the app updates its web contents. Specifically, the worse web caching performance an app has, the higher traffic reduction ratio it can obtain from using CK. For example, the two weather apps had a similar rate regarding content update. The weather app 1 had a higher HTTP traffic reduction ratio than the weather app 2. This is because the weather app 1 has a worse web caching performance. Meanwhile, for two apps with similar web caching performances, the app with less frequent content updates enjoys more traffic reduction. For example, the three shopping apps perform similarly in web caching: all of them do not cache image resources. In the experiment, the shopping app 1 and 3 had higher traffic reduction ratios than the shopping app 2. This is because the shopping app 2 has a much higher content update rate, as it is the official app of Craigslist, which is a popular classified advertisement website where many new listings are posted by users every hour.

Figure 9 shows the source breakdowns of the HTTP traffic reduction ratio. We can see that inter-click redundant HTTP traffic was the only major contributor to the overall traffic reduced for 9 apps. For the shopping app 1, same-click redundant traffic was another main source of traffic reduced. This suggests that it is worthwhile to pay special attention to same-click redundant traffic in CK's design.

**Web Content Rendering Speedup.** We evaluated how the 10 top ranked apps can expedite web content rendering under different connection conditions by using CK. In this experiment, the smartphone was connected to the Internet via our HTTP proxy [3], which could throttle download and upload bandwidths according to user configuration. We set the transmission bandwidth at the proxy according to a recent study on 3G/4G wireless speed [1]. This study suggests that the average 3G download speeds of the four major U.S. wireless service providers range from 0.59 Mbps to 3.84 Mbps with an average value of 2 Mbps. The average 4G download speeds range from 2.81 Mpbs to 9.12 Mpbs with an average value of 6.2 Mbps. Accordingly, we chose 8 values for the download bandwidth: 0.1, 0.5, 1, 1.5, 2, 4, 8, 12, all in the unit of Mbps, and set the upload bandwidth to 1 Mbps. We ran the 10 apps, with CK enabled in the smartphone, by replaying the recorded user inputs under the 8 different download bandwidths for 20 rounds. In each round, we also ran the 10 apps with CK disabled. We recorded the web content rendering time for each user click, which was the time interval between the first HTTP request and the last HTTP response of all the HTTP transactions of a click. The rendering speedup was calculated as the ratio between the rendering times with and without CK running respectively.

Figure 10 shows the average content rendering speedup of the 20 rounds testing for the 10 apps. From the figure we can see that the content rendering speedup increases as the connection condition becomes worse for all the 10 apps. The shopping app 2 and 3 are more sensitive to bandwidth changes. This is because the HTTP resources requested by these two apps are mainly large images. The average speedup of the 10 apps under the average 3G download bandwidth of the four major U.S. wireless providers (2 Mbps, reported in [1]) is 2.0. The average speedup under the average 4G download bandwidth (6.2 Mbps, reported in [1]) is around 1.5.

## Controlled Evaluation

**Effects of High System Load.** Mobile devices usually have constrained computational resources, and thus, mobile app performances are more sensitive to system load changes than their counterparts in PCs. Here, we wanted to investigate how CK can help mobile apps to improve their resilience to high system load. We developed a mobile app that can repeatedly download specified files from our own HTTP server with a configured interval. We used this app to download files with different sizes (1, 5, 10, 50, 100, 500 and 1000, in KB) from the server. For each file size, we repeated the download for 50 times with a 100 ms interval, and calculated the average time needed as the HTTP transaction duration for the file size. During the downloads, we imposed different background workloads on CPU so that we can see how transaction durations responded to system load changes.

We first performed the experiment without running CK. In this case, every HTTP transaction was served from network downloading. Figure 11 (a) plots the relationship between file sizes and transaction durations. We can see that transaction duration of a file increases much faster as file size increases if the background system load is high. This is because to transmit a large file, HTTP servers usually divide it into small chunks and transmit them separately. For example, our HTTP server segmented a large file into 8 KB chunks for separated transfers. Since frequent network transfers consume high CPU resource, downloading a large file needs more time under higher system load. We then performed the same experiment with CK running in the phone. In this case, except for the first download, which was served by network downloading, all the other 49 downloads were served by CK. Figure 11 (b) shows the experiment result. We can learn that when HTTP transactions were served by CK, the transaction durations were not only shorter than when served from network downloading (one magnitude less), but also more resilient to system load changes (i.e., the transaction duration for the

same file size increases little as system load increases), which is helpful to offer good user experiences under high system loads. This suggests that CK is desirable in mobile devices with constrained resources.

**Processing Time Overhead.** We evaluated processing time overhead caused by CK in the case of cache miss. There are two cases for processing time overhead on cache miss. First, if the HTTP response to the cache missed request is not cacheable, processing overhead by CK comes from searching the CLT for a matched cached response. Second, if the HTTP response is cacheable, additional processing time overhead comes from caching the response. We performed the experiment by downloading a 100 KB file from our HTTP server for 50 times with a 100 ms interval. We first ran the experiment with CK disabled, and recorded the average transaction duration as the based line value. Then we ran the experiment with CK enabled while enforcing the two cases of cache misses respectively, and recorded the corresponding average transaction duration. For the first case of cache miss, we configured HTTP responses as "`no-stored`". To enforce the second case of cache miss, we used different file names for each downloading. We also configured CK so that we could compare the difference of caching responses to memory and caching responses to filesystem files.

Figure 12 shows the normalized transaction durations under different scenarios. When responses to cache missed requests were non-cacheable, the processing time overhead was less than 1%. When responses are cacheable, the case of writing responses to memory had an processing time overhead of 1.1%. and the case of caching responses to files had an overhead of 1.8%. This result suggests that our implementation of CK incurs a small processing overhead on cache misses.

**Energy Overhead.** In this experiment, we evaluated the energy overhead of CK in the case of cache miss. Similar to processing time overhead, energy overhead on cache miss also has two cases: the case that responses are not cacheable and the case that responses are cacheable. When responses are non-cacheable, the energy overhead is looking up the CLT for a matched cached response. When responses are cacheable, additional energy overhead comes from writing responses to memory and/or files. We performed the experiment by downloading files with different sizes (5, 10, 50, 100, in KB) from our HTTP server. For each file size, we repeated the download for 50 times with a 100 ms interval. We measured phone power consumption using the Monsoon power monitor [13]. To obtain the baseline power consumption, we first ran the experiment with CK disabled. Then we ran the experiment with CK enabled while enforcing the two cases of cache miss using the same methods as in the processing time overhead experiment.

Figure 13 (a) plots the absolute power values, and Figure 13 (b) shows the normalized power values for different file sizes. From the result we can learn that CK incurs negligible energy overhead when responses to cache-missed HTTP requests are non-cacheable and when cacheable response are only written to memory. When cacheable responses are written to files, about 15% power overhead is incurred.

## CONCLUSION

In this paper, we propose and design CacheKeeper, an OS web caching service for smartphones. To motivate the work, we have performed a comprehensive measurement study on web caching functionality of 1300 top ranked Android apps. The measurement results suggest that imperfect web caching is a common and serious flaw for Android apps generating web traffic. We have implemented CacheKeeper in Linux kernel, and performed extensive evaluations on Android smartphone. Our evaluation indicates that CacheKeeper can effectively remedy the flaw of imperfect web caching for mobile apps with small overhead.

## ACKNOWLEDGMENTS

We would like to thank the reviewers for their valuable and helpful comments. This project was supported in part by US National Science Foundation grants CNS-1117412 and CAREER Award CNS-0747108.

## REFERENCES

1. 3G and 4G Wireless Speed Showdown: Which Networks Are Fastest? http://www.pcworld.com/article/253808/3g_and_4g_wireless_speed_showdown_which_networks_are_fastest_.html.
2. Androids HTTP Clients. http://android-developers.blogspot.com/2011/09/androids-http-clients.html.
3. Charles Web Debugging Proxy. http://www.charlesproxy.com.
4. Mobile Devices Now Make Up About 20 Percent of U.S. Web Traffic. http://allthingsd.com/20120525/mobile-devices-now-make-up-about-20-percent-of-u-s-web-traffic.
5. Study: Mobile Web Traffic Up 35% in Under a Year; PC Web Usage Peaks Early Morning. http://insights.chitika.com/2012/study-mobile-web-traffic-up-35-in-under-a-year-pc-web-usage-peaks-early-morning.
6. Bortz, A., and Boneh, D. Exposing private information by timing web applications. In *WWW* (2007).
7. Erman, J., Gerber, A., Hajiaghayi, M. T., Pei, D., Sen, S., and Spatscheck, O. To Cache or Not to Cache: The 3G Case. *IEEE Internet Computing 15*, 2 (2011).
8. Felten, E. W., and Schneider, M. A. Timing attacks on Web privacy. In *ACM CCS* (2000).
9. Fielding, et al. RFC 2616 - Hypertext Transfer Protocol - HTTP/1.1.
10. Guha, S., Jain, M., and Padmanabhan, V. Koi: A Location-Privacy Platform for Smartphone Apps. In *NSDI* (2012).
11. Koukoumidis, E., Lymberopoulos, D., Strauss, K., Liu, J., and Burger, D. Pocket cloudlets. In *ASPLOS* (2011).
12. Lymberopoulos, D., Riva, O., Strauss, K., Mittal, A., and Ntoulas, A. PocketWeb: instant web browsing for mobile devices. In *ASPLOS* (2012).
13. Monsoon Solutions Inc. Power Monitor. http://www.msoon.com/LabEquipment/PowerMonitor.
14. Papapanagiotou, I., Nahum, E. M., and Pappas, V. Smartphones vs. laptops: comparing web browsing behavior and the implications for caching. In *SIGMETRICS* (2012).
15. Qian, F., Quah, K. S., Huang, J., Erman, J., Gerber, A., Mao, Z. M., Sen, S., and Spatscheck, O. Web caching on smartphones: ideal vs. reality. In *MobiSys* (2012).
16. Tossell, C., Kortum, P. T., Rahmati, A., Shepard, C., and Zhong, L. Characterizing web use on smartphones. In *CHI* (2012).
17. Vallina-Rodriguez, N., Shah, J., Finamore, A., Grunenberger, Y., Papagiannaki, K., Haddadi, H., and Crowcroft, J. Breaking for commercials: characterizing mobile advertising. In *IMC* (2012).
18. Wang, Z., Lin, F. X., Zhong, L., and Chishtie, M. How far can client-only solutions go for mobile browser speed? In *WWW* (2012).
19. Zhang, K., Wang, L., Pan, A., and Zhu, B. B. Smart caching for web browsers. In *WWW* (2010).