# Web Caching on Smartphones: Ideal vs. Reality

Feng Qian
University of Michigan

Kee Shen Quah
University of Michigan

Junxian Huang
University of Michigan

Jeffrey Erman
AT&T Labs Research

Alexandre Gerber
AT&T Labs Research

Z. Morley Mao
University of Michigan

Subhabrata Sen
AT&T Labs Research

Oliver Spatscheck
AT&T Labs Research

## ABSTRACT

Web caching in mobile networks is critical due to the unprecedented cellular traffic growth that far exceeds the deployment of cellular infrastructures. Caching on handsets is particularly important as it eliminates all network-related overheads. We perform the first network-wide study of the redundant transfers caused by inefficient web caching on handsets, using a dataset collected from 3 million smartphone users of a large commercial cellular carrier, as well as another five-month-long trace contributed by 20 smartphone users. Our findings suggest that redundant transfers contribute 18% and 20% of the total HTTP traffic volume in the two datasets. Also they are responsible for 17% of the bytes, 7% of the radio energy consumption, 6% of the signaling load, and 9% of the radio resource utilization of *all* cellular data traffic in the second dataset. Most of such redundant transfers are caused by the smartphone web caching implementation that does not fully support or strictly follow the protocol specification, or by developers not fully utilizing the caching support provided by the libraries. This is further confirmed by our caching tests of 10 popular HTTP libraries and mobile browsers. Improving the cache implementation will bring considerable reduction of network traffic volume, cellular resource consumption, handset energy consumption, and user-perceived latency, benefiting both cellular carriers and customers.

## Categories and Subject Descriptors

C.2.2 [**Computer-Communication Networks**]: Network Protocols – Applications; C.4 [**Performance of Systems**]: Performance attributes

## General Terms

Measurement, Performance

## Keywords

HTTP Caching, Redundant Traffic, Cellular Networks, Redundancy Elimination, Smartphone Applications

## 1. INTRODUCTION

Caching plays a vital role in HTTP that dominates the Internet traffic usage [21]. Web caching can effectively decrease network traffic volume, lessen server workload, and reduce the latency perceived by end users.

**Web caching in cellular networks** is even more critical. HTTP traffic generated by mobile browsers and smartphone applications far exceeds any other type of traffic. HTTP accounts for 82% of the overall downstream traffic based on a recent study of a large cellular ISP [20]. From network carriers' perspective, cellular networks operate under severe resource constraints [31] due to the explosive growth of cellular data traffic (a growth of 5000% of its data traffic over 3 years reported by a major U.S. carrier [20]). Therefore even a small reduction of the total traffic volume by 1% leads to savings of tens of millions of dollars for carriers which are expected to spend $40.3 billion on cellular infrastructures in 2011 [4] (for all U.S. carriers).

The benefits of caching are also significant from customers' perspective. *(i)* Fewer network data transfers cut cellular bills since most carriers impose a monthly limit on data plan usage. *(ii)* The user experience improvement brought by caching is more notable in cellular networks whose latency is usually higher than those in Wi-Fi and wired networks. *(iii)* Transferring less data also improves handset battery life, as the power consumed by the cellular radio interface contributes 1/3 to 1/2 of the total device power during the normal workload [32].

**The location of a cache** can be at one or more places on a path from a handset (*i.e.,* a mobile device) to a web server. Although most of prior works focus on caching proxies placed in the network (*e.g.,* hierarchical caching proxies [18], caching proxy placement [21], and caching within 3G networks [20]), we emphasize that in cellular networks, *caching on handsets* is particularly important because it does not incur any network-related overhead. In particular, it eliminates data transmitted over the last-mile, *i.e.,* the radio access network, which is known as the performance and resource bottleneck of a cellular network [31]. In contrast, a network cache does not bring the aforementioned benefits but can cut the wide-area latency and enable sharing across users. In this study we focus on handset caches, which can coexist with network caches.

**The rules of web caching** are defined in HTTP 1.1 protocol (RFC 2616 [23]), although there exist numerous caching proposals [17, 26, 27, 29] that were never widely deployed. As specified in RFC 2616, HTTP employs *expiration* and *revalidation* to ensure cache consistency: the server sets for each cacheable file an expiration time before which the client should safely assume the freshness of the cached file. After the cache entry expires, the client must send a small revalidation message to the server to query the freshness of the cache entry. We detail the procedure in §2.

**Redundant network transfers due to cache implementation.** Let an *ideal* handset cache be a cache that has unlimited size and strictly follows the aforementioned caching rules. Any practically implemented handset cache is not ideal due to one or more reasons below: *(i)* it has limited size, *(ii)* the implemented caching logic does not satisfy RFC 2616, and *(iii)* the cache is not persistent (*i.e.,* it does not survive a process restart or a device reboot). A non-ideal cache potentially incurs *redundant transfers* that do not occur if an ideal cache were used. Also, redundant transfers can be caused by developers not utilizing the caching support even if it is provided by the HTTP library.

In this paper, we present to our knowledge the first network-wide study of the redundant transfers (defined above) by investigating the following important characteristics:

- The prevalence of redundant transfers within today's smartphone traffic, in terms of both the traffic volume contribution and the resource impact;
- The root cause for the redundant transfers;
- The handset caching logic, identified to be the main reason of redundant transfers, of HTTP libraries and browsers on popular smartphone systems.

We summarize our main findings as follows.

- We leveraged a large dataset containing 695M HTTP transaction records collected from 2.9M customers of a commercial cellular network in the U.S. To complement this data set which has a short duration of 24 hours, we collected another five-month trace from 20 smartphone users using smartphones instrumented by us. We measured the prevalence of redundant transfers for both datasets using an accurate caching simulation algorithm, assuming an ideal cache. To our surprise, redundant transfers due to caching implementation issues contribute 18% and 20% of the total HTTP traffic volume, for both datasets, respectively. The fraction slightly decreases to 17% at the scope of *all* traffic for the user study trace. Almost all redundant transfers are caused by the handsets instead of the server. Further, the redundancy ratio varies among applications. Some popular smartphone apps have unacceptably high fractions (93% to 100%) of redundant transfers.
- By strategically changing the simulation algorithm and analyzing the cache access patterns, we found the impact of limited cache size and non-persistent cache on the redundancy to be limited. For example, the fraction of redundant bytes is at least 13% (compared to 18% for an ideal cache) within all HTTP traffic even when our simulation uses a small cache of 4 MB. This implies the problematic caching logic is the main reason for redundant transfers.
- We investigated the resource overhead (handset radio energy, radio resources, and signaling load) incurred by redundant transfers, by leveraging our previously designed Radio Resource Control (RRC) simulation tool for 3G networks [32]. The results indicate that the impact on resource consumption is smaller than their impact on the traffic volume (yet still significant, *i.e.,* 7% for radio energy, 9% for radio resources, and 6% for signaling load, based on our analysis of the user study trace), due to reasons explained in §5.
- We performed comprehensive tests of ten state-of-art HTTP libraries and mobile browsers, many of which were found to not fully support or strictly follow the HTTP 1.1 caching specifications. In particular, among the eight HTTP libraries, four (three for Android and one for iOS) do not support caching at all. Smartphone apps using these libraries thus cannot benefit from caching. By exposing the shortcomings of existing implementations, our work helps encourage library and platform developers to improve the state of the art, and helps application developers choose the right libraries to use for better performance.

Overall, our findings suggest that for web caching, *there exists a huge gap between the protocol specification and the protocol implementation on today's mobile devices*, leading to significant amount of redundant network traffic, most of which could be eliminated if the caching logic of HTTP libraries and browsers fully supports and strictly follows the specification, and developers fully utilize the caching feature provided by the libraries. Fixing this cache implementation issue can bring considerable reduction of network traffic volume, cellular resource consumption, handset energy consumption, and user-perceived latency, benefiting both cellular carriers and customers.

**Paper organization.** §2 provides background of HTTP caching. §3 describes the measurement goal, data, and methodology. Then we measure the traffic volume impact and the resource impact of redundant transfers in §4 and §5, respectively. In §6 we perform caching tests for popular HTTP libraries and mobile browsers. We summarize related work in §7 before discussing our future work and concluding the paper in §8.

## 2. BACKGROUND: CACHING IN HTTP

This section provides background of web caching defined in HTTP 1.1 [23]. As described in §1, our study focuses on caching on handsets instead of within the network. In the remainder of the paper, a *cache* refers to an HTTP cache on a handset unless otherwise specified. We refer to a web object (*e.g.,* an HTML document or an image) carried by an HTTP response as a *file*.

**Caching consistency** (*i.e.,* keeping cached copies fresh) is the key issue. To realize that, the server sets the expiration time for each file by specifying either `Expires` or `Cache-Control:max-age` header directive. Then before a cache entry expires, a handset should safely assume the freshness of the file by serving the request using the cached copy without generating any network traffic. After a cache entry expires, a handset should perform cache *revalidation*, *i.e.,* asking the origin server whether the file has changed, by issuing conditional requests using `If-Modified-Since:<time>` or `If-None-Match:<eTag>` directive (or both). The former directive instructs the server to send a new copy if the file has changed since a specified date, which is usually the last modified time indicated by the `Last-Modified` directive in the previous response. The latter allows the server to determine the freshness using a file "version identifier" called eTag (entity tag), which is a quoted string attached to the file in the previous response. An eTag might be implemented using a version name or a checksum of the file content. In either case, if the file has changed, the server sends a new copy to the handset. Otherwise, a small `304 Not Modified` response is returned to the handset, without a document body, for efficiency.

**A handset determines a cache entry has expired** if and only if $t_{\text{arrive\_age}} + t_{\text{cache\_age}} \geq t_{\text{fresh\_life}}$. $t_{\text{arrive\_age}}$ is the age of the file when it arrives at the cache. It is computed from the `Date` or `Age` directive of a response, plus an estimated round-trip time compensating for the network delay. $t_{\text{cache\_age}}$, which can be trivially calculated, indicates how long the file has been in the cache. $t_{\text{fresh\_life}}$ is the cached copy's freshness lifetime (similar to the shelf life of food in grocery

**Table 1: Our measurement datasets.**

| Dataset | ISP | UMICH |
|---|---|---|
| Data collection period | May 20 2011 0:00 GMT $\sim$ May 20 2011 23:59 GMT | May 12 2011 $\sim$ Oct 12 2011 |
| Collection point | Commercial cellular core network | Directly on users' handsets |
| Number of users | 2.92 million (estimated) | 20 |
| Dataset size | 271 GB | 119 GB |
| Traffic volume | 24.3 TB | 118 GB |
| Platforms | Multiple (mainly iOS and Android) | Android 2.2 |
| Data format | 695 million records of HTTP transactions | Full packet trace (including payload) of all traffic |

stores) derived from the **`Expires`** or **`Cache-Control:max-age`** directive where the latter one overrides the former one if both exist. $t_{\text{fresh\_life}}$ is usually specified by the server while a handset can also specify **`Cache-Control`** request directives (**`max-age`**, **`max-stale`**, and **`min-fresh`**) to tighten or loosen expiration constraints although they are rarely used in practice.

**Non-storable and must-revalidate files**. A *non-storable* file (marked by **`Cache-Control:no-store`**) forbids a cache from storing (*i.e.,* caching) the file. A *must-revalidate* file (marked by **`Cache-Control:must-revalidate`**, **`Pragma:no-cache`**, or **`Cache-Control:no-cache`**) can be stored in a cache. However, the cache must bypass the freshness calculation mechanism and always revalidate with the origin server before serving it ("**`no-cache`**" is misleading because the file actually can be cached). A handset can also explicitly set the above caching directives in a request, indicating it will not store or will always revalidate the file.

Clearly, well-behaved caching logic requires correct implementation at both the handset and the server side. Our analysis described in §3 helps identify inefficient caching behaviors and which side is responsible for any of them.

## 3. MEASUREMENT GOAL, DATA, AND METHODOLOGY

This section highlights our measurement goal (§3.1), describes the measurement data (§3.2), and then details our analysis approach for redundant data transfers (§3.3).

### 3.1 The Measurement Goal

We define *inefficient caching* as caching behaviors that *(i)* lead to redundant data transfers, whose negative impact on performance, resource consumption, and billing is particularly high for mobile users, and *(ii)* relate to the *implementation* (as opposed to the *semantics*, as described below) of the HTTP caching mechanism. Inefficient caching can be caused by multiple reasons including the following factors covering the most important aspects of cache implementation (we discuss less significant factors in §3.3.2).

- **Problematic caching logic** due to any of the following reasons. *(i)* The handset does not fully support or strictly follow the protocol specification[1]. *(ii)* The server does not properly follow the caching specification. *(iii)* In order to leverage the caching support provided by an HTTP library, a developer still needs to configure it. Many developers may skip that for simplicity, or simply be unaware of it, therefore missing the opportunity of caching even if it is supported.
- **A limited cache size** causing the same content to be fetched twice if the first cached copy is evicted from the cache.
- **A *non-persistent* cache** whose cached data does not survive a process restart or a device reboot (unlike a *persistent* cache that survives both).

[1]Not following the specification can also cause consumption of expired contents. But this was never observed in our tests in §6.

As highlighted in §1, we aim at understanding the impact of redundant transfers caused by the above inefficient caching factors, for commercial cellular networks (§4 and §5), as well as how the caching logic of popular HTTP libraries and mobile browsers deviates from the specification (§6).

**Caching implementation vs. semantics.** All the factors listed above relate to caching *implementation*. Redundant transfers may also be attributed to the misconfiguration of caching parameters, which ideally should be properly set by a server according to the *semantics* of files. For example, for a news website, conservatively marking all news articles and images as non-storable or setting for them very short freshness lifetime values may lead to redundant transfers while bringing negligible benefits given that the article contents rarely change. A thorough study of the file semantics and caching parameter settings is out of the scope of this paper, although we show examples where caching parameter settings are obviously too conservative (§8).

From this point on, unless otherwise specified, *redundant transfers* refer to redundant transfers caused by caching implementation.

### 3.2 The Smartphone Measurement Data

We collected two diverse datasets summarized in Table 1 to measure redundant transfers caused by inefficient caching.

#### *3.2.1 The* ISP *Dataset*

The ISP dataset was collected from a large U.S. based cellular carrier at a national data center on May 20, 2011, on the interface between the GGSN (Gateway GPRS Support Node) and SGSNs (Serving GPRS Support Node) without any sampling. Each record in the dataset corresponds to one HTTP transaction, containing three pieces of information: *(i)* a 64-bit timestamp, *(ii)* summaries of header fields in the request and the response, and *(iii)* the actual amount of data transferred based on the TCP data associated to each HTTP transaction. To preserve subscribers' privacy, the URLs were anonymized using a 128-bit hash function and also all cookie information was removed from the HTTP headers[2].

**Subscriber identification.** Since our cache simulation is performed at a per-user basis (§3.3), we need to identify the subscriber ID for each record. Instead of using MSISDN (the phone number) or IMEI (the device ID) information that was not collected due to privacy concern, we used anonymized session-level information to correlate multiple HTTP transaction records with a single subscriber. One disadvantage of our approach is that one real

[2]In HTTP, caching and cookies are decoupled, and a server is responsible for explicitly disabling caching when appropriate [16].

subscriber may be identified with multiple subscriber IDs (but one subscriber ID never maps to multiple real subscribers). This leads to an underestimation of the amount of redundant data due to increased cold start cache misses [34] (explained in §3.3.2).

### 3.2.2 *The* UMICH *Dataset*

The ISP dataset is representative due to its large user base, however it is limited in terms of the trace duration and recorded content, as only summarized HTTP transaction records were captured. This is complemented by our second dataset called UMICH, collected from 20 smartphone users for five months, allowing us to keep detailed track of each individual user's web cache for a much longer period. These participants consisted of students from 8 departments at University of Michigan[3]. The 20 participants were given Motorola Atrix (11 of them) or Samsung Galaxy S smartphones (9 of them) with unlimited voice, text and data plans of the same cellular carrier from which we obtained the ISP dataset. All smartphones use Android 2.2. The participants were encouraged to take advantage of all the features and services of the phones. We kept collected data and users' identities strictly confidential.

We developed custom data collection software and deployed it on the 20 smartphones. It continuously runs in the background and collects two types of data: *(i)* full packet traces in **`tcpdump`** format including both headers and payload, and *(ii)* the process name responsible for sending or receiving each packet, using the method described in [32] by efficiently correlating the socket, the inode, and the process ID in Android OS in realtime. Both cellular and Wi-Fi traces were collected without any sampling performed. The data collector incurs no more than 15% of CPU overhead although the overhead is much lower when the throughput is low (*e.g.,* less than 200 kbps).

We also built a data uploader that uploads the captured data (stored on the SD card) to our server when the phone is idle. The data collection is paused when the data is being uploaded so the uploading traffic is not recorded. Also the upload is suspended (and the data collection is resumed) by any detected network activity of user applications. The entire data collection and uploading process is transparent to the users, although we do advise the users to keep their phones powered on as often as possible.

## 3.3 Analyzing Redundant Transfers

We explain our data analysis approach. We feed each user's HTTP transactions in the order of their arrival time to a web cache simulator developed by us. The simulator behaves like a cache that strictly follows the HTTP 1.1 caching mechanism specified in RFC 2616 (§2). Redundant transfers can be identified through the simulation process. Our approach differs from previous trace-driven cache simulations [17, 34, 21, 20] in two ways. *(i)* Our simulation is performed at a *per-user* basis to capture redundant transfers for each handset, while previous ones consider aggregated HTTP transfers of all users to compute, for example, the cache hit ratio, for a network cache. *(ii)* Ours is more fine-grained in that it distinguishes various causes of the redundancy (and also non-redundancy).

In the remainder of the paper, a *handset cache* and the *simulated cache* refer to the cache on a real handset and the cache maintained by our simulator, respectively.

**The basic simulation algorithm** is illustrated in Figure 1. It assigns to each HTTP transaction a label indicating its caching status. A file can be **`NOT_STORABLE`** due to its **`Cache-Control:no-store`** directive or causes **`CACHE_ENTRY_NOT_EXIST`** because it has not yet been cached. **`NOT_EXPIRED_DUP`** is an undesired case where a handset issues a request for a file cached in the simulator *before* it expires, resulting in redundant transfers ("**`DUP`**" means duplication).

```
01 foreach HTTP transaction r
02     if (file is not storable) then
           //the file contains "Cache-Control: no-store"
03         assign_label(r, NOT_STORABLE);
04         continue;
05     else if (cache entry not exists) then
           //cache entry not found
06         assign_label(r, CACHE_ENTRY_NOT_EXIST);
07     else if (cache entry not expired) then
           //a request is issued before the file expires
08         assign_label(r, NOT_EXPIRED_DUP);
           //the response is ignored by the simulator because the
           //request should not have been generated
09         continue;
10     else if (file changed) then
           //the file has changed after the cache entry expires
11         assign_label(r, FILE_CHANGED);
12     else if (HTTP 304 used) then
           //the file has not changed after the cache entry expires,
           //and a cache revalidation is properly performed
13         assign_label(r, HTTP_304);
14     else if (revalidation not performed) then
           //the file has not changed after the cache entry expires,
           //but the handset does not perform cache revalidation
15         assign_label(r, EXPIRED_DUP);
16     else
           //the file has not changed after the cache entry expires,
           //but the server does not recognize the cache revalidation
17         assign_label(r, EXPIRED_DUP_SVR);
18     update_cache_entry(r);//update the simulated cache
19 endfor
```

**Figure 1: The basic caching simulation algorithm.**

Then Line 10 to 17 deal with the scenario where a cached file has expired. If the file has changed, then the entire file needs to be transferred again (**`FILE_CHANGED`**). If the file remains unchanged, the ideal way to handle it is that the handset performs cache revalidation and the server sends back an **`HTTP_304`** response. Otherwise, the problem either comes from the handset side, which does not issue a conditional request (**`EXPIRED_DUP`**), or from the server side, which does not recognize a conditional request (**`EXPIRED_DUP_SVR`**). Among the aforementioned labels, **`NOT_EXPIRED_DUP`**, **`EXPIRED_DUP`** and **`EXPIRED_DUP_SVR`** correspond to redundant transfers.

**Is our simulated cache complete?** Let us first assume our simulated cache is persistent with unlimited size (*i.e.,* an ideal cache defined in §1). Consider the Venn diagram shown in Figure 2. Let $U$ be all HTTP transactions carried out by applications. What we observe in the data, $D$, is a subset of $U$ since requests of $U \backslash D$ (the relative complement of $D$ in $U$) are already served by the handset cache so $U \backslash D$ does not appear on the network. However, remember that in order for a file $f$ to be cached, it must be transferred over the network (*i.e.,* $f \in D$) at least once. Therefore our simulated cache will not miss any file that is in a handset cache, if the simulated cache is ideal (we discuss the only exception in §3.3.2). More importantly, as explained in §3.1, our goal is to study the redundant data transfers that always belong to $D$ instead of $U \backslash D$.

[3]This user study has been approved by the University of Michigan IRB-HSBS #HUM00044666.

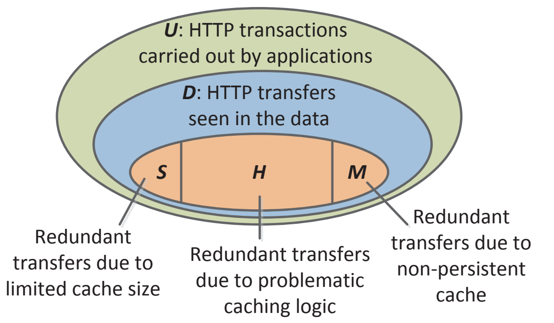
U: HTTP transactions carried out by applications
D: HTTP transfers seen in the data
S
H
M
Redundant transfers due to limited cache size
Redundant transfers due to problematic caching logic
Redundant transfers due to non-persistent cache

**Figure 2: A Venn diagram showing HTTP transactions observed by applications and by our simulator, as well as the redundant transfers.**

On the other hand, a file in our simulated cache may be missing in a handset cache because of its problematic caching logic (the set $H$ in Figure 2), the limited size (the set $S$), or a lack of persistent storage (the set $M$) of the handset cache. Ideally we want to distinguish the three cases. However, the redundant transfers identified by our simulator are in fact $H \cup S \cup M$ where $H$, $S$, and $M$ are indistinguishable, given that the simulated cache is ideal.

### *3.3.1 Algorithm Details*

Figure 1 sketches the basic simulation algorithm. We provide details of the algorithm below.

**The key of a cache entry** (our simulated cache was implemented using a hash map) consists of three parts: *(i)* the host name indicated by the **`Host`** request directive, *(ii)* the file name followed by the **`GET`** command, and *(iii)* the eTag. Part *(i)* and *(ii)* must exist otherwise the HTTP transaction is assigned a special label "**`OTHER`**" (Table 2) while the eTag part is optional. Also the file name includes the entire **`GET`** string containing query strings so **`/a.php?para=1`** and **`/a.php?para=2`** have different cache entries. Empty or error responses (*e.g.*, **`404 Not Found`**) are also counted as **`OTHER`**, which only accounts for 0.5% of all HTTP traffic in both datasets.

**The change of a file** is identified through a different **`Last-Modified`**, **`Content-Length`** or **`Content-MD5`** value for the same cache entry.

**A heuristic freshness lifetime** can be used when neither **`Expires`** nor **`Cache-Control:max-age`** exists in a response, according to RFC 2616. We use a heuristic lifetime of 24 hours and later we show our analysis results are not sensitive to this value (§4.1).

**A partially cached file** is caused either by a byte-range request (using the **`Range`** directive) for a subrange of the origin file, or by a prematurely broken connection. Our simulator supports partial caching by allowing a cache entry to contain one or more subranges of a file. We implemented the following logic according to RFC 2616. Assume one or more subranges of a file have been cached, and an incoming response transfers another subrange $R$. The new subrange $R$ is then combined with the existing range(s) if both the existing and the new range have the same eTag value (the eTag value must exist). Otherwise, all previously cached range(s) are removed before $R$ is put into the cache entry.

If a file is partially cached, then a single transfer of the whole or a part of the file may contain both redundant and non-redundant bytes. To handle such a case, for each of the **`*EXPIRED_DUP*`** labels, we use two new labels, **`*EXPIRED_DUP*_CACHED`**, and **`*EXPIRED_DUP*_UNCACHED`** (not shown in Figure 1), to distinguish the redundant and the non-redundant ranges, respectively.

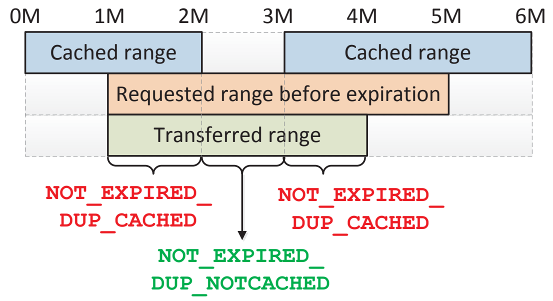

**Figure 3: A partially cached file and a partial cache hit.**

Consider a file of 6 MB shown in Figure 3. Assume two ranges [0, 2M) and [3M, 6M) are already cached by the simulator. The handset makes a byte-range request of [1M, 5M) before the cache entry expires. However the user cancels the transfer in the middle so only [1M, 4M) is actually transferred, as observed in our dataset. In this example, ideally the HTTP library should only request for [2M, 3M), which is not in the cache, using the **`Range`** and the **`If-Range`** directives[4]. We therefore label [2M, 3M) as **`NOT_EXPIRED_DUP_NOTCACHED`**, the non-redundant range, and label [1M, 2M) and [3M, 4M) as **`NOT_EXPIRED_DUP_CACHED`**, the redundant ranges. The two labels substitute for the original **`NOT_EXPIRED_DUP`** label which is not used for partially cached files. We introduce similar labels for **`EXPIRED_DUP_SVR`** and **`EXPIRED_DUP`**. Table 2 summarizes all labels.

### *3.3.2 Limitations*

We discuss five limitations of our simulation approach.

- Inefficient caching and hence redundant transfers exist in both unencrypted HTTP and encrypted HTTPS traffic. However, the ISP trace contains only HTTP records. For the UMICH trace, the simulator cannot parse the HTTPS traffic that was collected by **`tcpdump`** running below the SSL library. HTTPS accounts for only 11.2% of the total traffic volume, compared to 85.4% for HTTP transfers.
- As mentioned before, the simulator cannot precisely distinguish $S$, $H$, and $M$ shown in Figure 2. We qualitatively address such indistinguishability in §4.2 based on robust heuristics.
- A file may already be cached in a handset cache before the data collection started but the simulator does not know that. As an inherent problem (called cold start cache miss [34]) of any trace-driven cache simulation algorithm, it leads to an underestimation of the cache hit ratio (or the redundancy ratio in our case). But both our traces (in particular the UMICH dataset) are sufficiently long so the impact of cold start cache miss is expected to be small.
- Redundant transfers can also be caused by users explicitly reloading a file (*e.g.*, refreshing a web page). In that case, the application may override the default caching behavior by, for example, requesting for a file before its cached copy expires, but the simulator has no way to identify such manually triggered redundant transfers.

[4]The **`Range`** directive is often used together with an **`If-Range:<eTag>`** conditional request. It means "if the file is unchanged, send me the range that I am missing; otherwise, send me the entire new file".

Table 2: Detailed breakdown of caching entry status.

| Label | Cache hit or miss? | Fully or partially cached? | Redundant due to caching? | % HTTP bytes ISP (GC)* | UMICH (GC) | UMICH (PC)* |
|---|---|---|---|---|---|---|
| 1. NOT_STORABLE | - | - | - | 15.4% | 19.7% | 19.7% |
| 2. CACHE_ENTRY_NOT_EXIST | Miss | - | - | 47.5% | 42.0% | 42.3% |
| 3. FILE_CHANGED | Miss | - | - | 1.9% | 0.5% | 0.5% |
| 4. HTTP_304 | Hit | Either | - | 0.1% | 0.0% | 0.0% |
| 5. NOT_EXPIRED_DUP | Hit | Full | Yes | 13.6% | 15.0% | 14.7% |
| 6. NOT_EXPIRED_DUP_CACHED | Hit | }Partial | Yes | 2.3% | 1.3% | 1.3% |
| 7. NOT_EXPIRED_DUP_NOTCACHED | Miss | | - | 16.0% | 17.0% | 17.0% |
| 8. EXPIRED_DUP | Hit | Full | Yes | 1.7% | 4.0% | 4.0% |
| 9. EXPIRED_DUP_CACHED | Hit | }Partial | Yes | 0.1% | 0.0% | 0.0% |
| 10. EXPIRED_DUP_NOTCACHED | Miss | | - | 0.9% | 0.0% | 0.0% |
| 11. EXPIRED_DUP_SVR | Hit | Full | - | 0.0% | 0.0% | 0.0% |
| 12. EXPIRED_DUP_SVR_CACHED | Hit | }Partial | - | 0.0% | 0.0% | 0.0% |
| 13. EXPIRED_DUP_SVR_NOTCACHED | Miss | | - | 0.0% | 0.0% | 0.0% |
| 14. OTHER | - | - | - | 0.5% | 0.5% | 0.5% |

* GC: all processes on a handset share one single global cache; PC: each process has its own cache.

Table 3: Statistics of file cacheability.

| Count by | Dataset | Normally Cacheable | Must-revalidate | Non-storable | Other |
|---|---|---|---|---|---|
| Bytes | ISP | 69.8% | 14.3% | 15.4% | 0.5% |
| | UMICH | 78.2% | 1.6% | 19.7% | 0.5% |
| Files | ISP | 72.4% | 12.4% | 14.9% | 0.4% |
| | UMICH | 65.6% | 6.8% | 25.4% | 2.1% |

- Similarly, our simulator cannot identify redundant transfers caused by users manually clearing the cache after browsing sessions. The amount of such redundant data is expected to be small due to the observed strong temporal locality of accessing the same cache entry (described in §4.2).

## 4. THE TRAFFIC VOLUME IMPACT

In this section, we investigate the *traffic volume* impact of redundant transfers caused by inefficient caching behaviors.

### 4.1 Basic Characterization

We first assume our simulated cache is ideal (*i.e.,* it is persistent with unlimited cache size). Thus all redundant transfers caused by the three factors described in §3.1 can be identified.

**File cacheability.** Table 3 breaks down all HTTP bytes (files) transferred over the network into four categories: normally cacheable (*i.e.,* following the standard expiration and freshness calculation mechanism), must-revalidate (§2), non-storable, and other HTTP transfers (§3.3.1). For both datasets, most bytes (70% to 78%) and most files (66% to 72%) are normally cacheable, indicating the potential benefits of caching if handled properly by a handset.

**A detailed breakdown of caching entry status** is shown in Table 2, which lists the 14 labels described in §3.3. We show two simulation scenarios for the UMICH dataset: *(i)* all processes on a handset share one single global cache, and *(ii)* each process has its own cache. They correspond to "GC" and "PC" in Table 2, respectively. The per-process cache simulation is feasible because the UMICH trace contains packet-process correspondence for each packet. We summarize our findings as follows.

- **NOT_EXPIRED_DUP** contributes most bytes (77% for ISP and 74% for UMICH) among the four labels (5, 6, 8, 9) incurring redundant transfers. In other words, redundant transfers are usually caused by a handset issuing unnecessary requests *before* received files expire. For the ISP (UMICH) trace, 14% (31%) of all HTTP transactions (not shown), corresponding to 14% (15%) of all HTTP bytes (Row 5 in Table 2), are unnecessary because if handsets properly cache previous responses, no request needs to be sent out over the network and the responses can be served from local caches. On the other hand, the contribution of **EXPIRED_DUP** is much less. For the ISP (UMICH) trace, for 4.5% (10.2%) of all HTTP transactions (not shown), conditional requests to check the freshness of the cached data need to be sent, so that their responses, corresponding to 1.7% (4.0%) of all HTTP bytes (Row 8 in Table 2) will end up being served from local caches (if handsets properly cache previous responses) because they do not change.
- When **HTTP_304** is not used, it is almost always attributed to the handset instead of the server which properly handles cache revalidation, as indicated by the negligible contribution of **EXPIRED_DUP_SVR***.
- Partially cached files incur limited redundant transfers in that the traffic volume contribution of ***_DUP_CACHED** is small. In contrast, ***_DUP_NOTCACHED** (illustrated in Figure 3) account for considerable amount of non-redundant bytes. We found most (95%)[5] of ***_DUP_NOTCACHED** bytes originate from servers using byte-range responses for streaming large multimedia files. Note that a recent measurement study [22] showed that 98% of multimedia streaming traffic for a commercial cellular network is delivered over HTTP.
- The results of UMICH (GC) and UMICH (PC) are almost identical, indicating small overlap among files requested by different applications.
- Cellular and Wi-Fi traffic exhibit similar breakdown (not shown in Table 2), indicating the caching strategies on both the server and the handset side are independent of the network interface.

[5]Identified by their **User-Agent** strings. See §4.3 for details.

**Table 4: The overall traffic volume impact of redundant transfers when different heuristic freshness lifetime values are used.**

| Dataset | % of redundant bytes of all HTTP traffic [% of redundant bytes of all traffic (HTTP and non-HTTP)] under different values of heuristic freshness lifetime | | | | |
|---|---|---|---|---|---|
| | 1 hour | 6 hours | 1 day* | 1 week | 1 month |
| ISP (GC) | 17.74% [16%]** | 17.74% [16%] | 17.74% [16%] | 17.74% [16%] | 17.74% [16%] |
| UMICH (GC) | 20.24% [17.29%] | 20.25% [17.30%] | 20.28% [17.33%] | 20.32% [17.36%] | 20.34% [17.38%] |
| UMICH (PC) | 19.94% [17.03%] | 19.95% [17.04%] | 19.98% [17.07%] | 20.02% [17.10%] | 20.04% [17.12%] |

* 1 day is the heuristic freshness lifetime used for other results in the paper.
** Assume the fraction of HTTP traffic is 90%, based on a recent large-scale measurement study for a commercial cellular data network [36].

**The overall traffic volume impact** is summarized in Table 4. We highlight key observations below.

- The first number in each grid of Table 4 is the fraction of redundant bytes within all HTTP traffic. Recall the redundant bytes come from label 5, 6, 8, 9 in Table 2, and they correspond to $H \cup S \cup M$ in Figure 2. By eliminating redundant transfers, the reduction of HTTP traffic is as high as 17.7% and 20.3% for ISP and UMICH, respectively.
- Even at the scope of *all* traffic (HTTP and non-HTTP), the redundancy ratio is also significant (17.3% for UMICH) as indicated by the second number. Note that this is an underestimation because we did not consider the redundancy of HTTPS traffic accounting for 11.2% of the total traffic volume of UMICH.

  We do not have the number for the ISP dataset that only contains HTTP records. As reported by a recent measurement study [36], HTTP accounts for at least 90%[6] of the total traffic volume of an aggregated one-week dataset involving 600K cellular subscribers collected in August 2010. If we assume that fraction is representative and apply it to our ISP dataset, then its overall redundancy ratio at the scope of all traffic is at least 16%.
- Recall that our simulator introduced a heuristic freshness lifetime when a response contains no expiration information. Table 4 shows this parameter has negligible impact on the amount of redundant data.
- Although the duration of the ISP trace is much shorter, its redundancy ratio is only marginally smaller than that of UMICH, implying the usage duration has limited impact on the redundancy ratio. This is partly explained by the strong temporal locality of accessing the same cache entry (Figure 5 in §4.2).
- For the UMICH dataset, the difference between redundancy ratios of per-process caches (PC) and a single global cache (GC) is as small as 0.3%.

## 4.2 The Impact of the Cache Size

Now we discard the assumption of unlimited cache size and consider a *finite* cache for the simulator. This helps quantify the impact

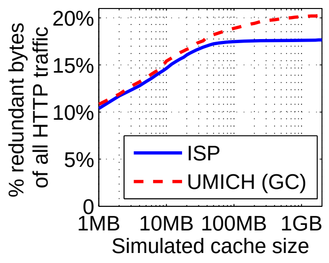

**Figure 4: Relationship between the simulated cache size and the fraction of detected HTTP redundant bytes.**

of limited cache size on redundant transfers. We implemented LRU (Least Recently Used) algorithm for our simulator since all HTTP libraries and browsers we tested in §6 use LRU as the replacement algorithm, if they support caching. LRU discards the least recently accessed files first when the cache is full. With a finite cache size, the simulated cache may not capture all redundant transfers in the trace. We refer to those captured ones as *detected* redundant transfers.

Consider Figure 2 again. Let us decrease the simulated cache size. Then we see fewer detected redundant transfers (each of $|S|$, $|H|$, and $|M|$ decreases) since more previously detected redundant transfers are classified as `CACHE_ENTRY_NOT_EXIST` due to cache misses as the simulated cache becomes smaller. In particular, when the simulated cache size is smaller than the handset cache size, redundant transfers due to limited size of the handset cache will be eliminated (*i.e.,* $|S|$ becomes 0) because if a cache entry is evicted from the handset cache, it must have been evicted from the simulated cache (assuming both use LRU). Therefore, if the detected redundant bytes decrease to $x\%$ when the simulated cache size goes below the handset cache size, then the traffic volume impact of $H \cup M$ is at least $x\%$. Note this is a very loose lower bound in that $|H \cup M|$ also decreases as the simulated cache becomes smaller.

Our measurement results are shown in Figure 4 where we vary the simulated cache size from 1 MB to 2 GB for both datasets (note that the X-axis is in log scale). Figure 4 shows that even when the cache has a very small size of, for example, 4 MB[7], the detected redundant bytes is still as high as 12.8% and 13.2% (compared to 17.7% and 20.3% when the simulated cache has unlimited size),

[6]See Figure 1(a) of the paper [36]. The following categories use HTTP: **`web_browsing`**, **`smartphone_apps`**, **`market`**, and **`streaming`**.

[7]In comparison, our caching tests in Table 9 (§6.2) show that the cache sizes for the Android 2.2 browser and the Safari browser of iOS 4.3.4 / iPhone 4 are 8 MB and 100 MB, respectively.

**Table 5: Measuring redundant transfers for top applications in both datasets.**

| The UMICH dataset (HTTP Bytes: 101 GB) | | | The ISP dataset (HTTP Bytes: 24.3 TB) | | |
|---|---|---|---|---|---|
| Android process* | % HTTP bytes | % redundant HTTP bytes | **`User-Agent`** regex (device/OS name)* | % HTTP bytes | % redundant HTTP bytes |
| Streaming Service 1 | 29.8% | 8.8% | Streaming Service 1 (A)** | 37.8% | 20.1% |
| Streaming Service 2 | 12.4% | 0.5% | Internet Radio (A) | 11.6% | 1.9% |
| Web Browser 1 | 11.5% | 20.4% | Web Browser (A) | 11.3% | 14.6% |
| Entertainment | 6.6% | 12.0% | Media player (A) | 8.9% | 3.1% |
| News and Weather | 6.3% | 55.3% | Map (A) | 3.1% | 0.0% |
| Lifestyle | 3.9% | 99.4% | HTTP Library (B) | 2.4% | 86.6% |
| Music and Audio 1 | 3.4% | 0.0% | Web Browser (C) | 2.1% | 1.6% |
| Music and Audio 2 | 2.9% | 0.1% | Weather (A) | 2.0% | 93.0% |
| Web Browser 2 | 1.8% | 8.6% | Social Networking (A) | 1.6% | 7.3% |
| Social Network Manager | 1.7% | 99.3% | Streaming Service 2 (D) | 1.5% | 19.1% |
| Media and Video | 1.7% | 0.8% | Web Browser (B) | 1.4% | 13.5% |
| Web Browser 3 | 1.4% | 18.3% | Ad library (B) | 1.0% | 100.0% |
| (Total or average) | 83.4% | 18.3% | (Total or average) | 84.7% | 18.2% |

* The process names, **`User-Agent`** regular expressions, and device/OS names have been anonymized.
** For the ISP dataset, A, B, C, D refer to four different device/OS names after anonymization.

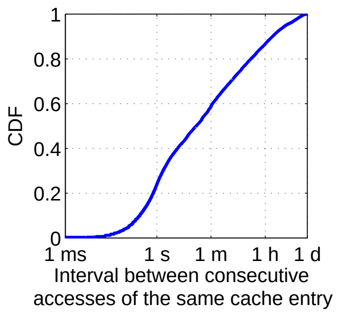

**Figure 5: Distribution of intervals between consecutive accesses of the same simulated cache entry (for the ISP trace).**

which are the aforementioned (loose) lower bounds of $|H \cup M|$, for ISP and UMICH, respectively. We therefore conclude that the problematic caching logic (instead of the limited cache size) takes the major responsibility for redundant transfers.

Figure 4 also suggests how to set the cache size, which can be small enough to entirely fit into today's smartphone memory with very limited additional cache misses incurred. For example, reducing the simulated cache size from infinity to 50 MB causes additional cache misses for only 2.0% and 0.4% of HTTP bytes (they are the loose upper bounds of the reduction of $|S|$) for UMICH and ISP, respectively, as indicated by the decrease of the detected redundant bytes shown in Figure 4.

**Persistent vs. non-persistent cache.** As described in §3.1, a non-persistent cache does not survive a process restart or a device reboot while a persistent cache does survive both. Based on our caching tests of 6 HTTP libraries and mobile browsers that support caching (§6.2), only one library for iOS (**`NSURLRequest`**) uses a non-persistent cache (Table 9). All Android libraries as well as both iPhone and Android browsers use persistent caches.

Our simulation assumes a persistent cache, which is consistent with the UMICH trace involving only Android handsets. For the ISP trace involving iOS devices, redundant transfers can also be caused by the non-persistent cache, which cannot be simulated since we do not know when a user restarts a process or reboots a handset. However, we expect the fraction of redundant transfers caused by the non-persistent cache is small due to two reasons. *(i)* Restarting a process happens infrequently in iOS [11]. On iPhone and iPad, pressing the "home" button simply puts an application to background. *(ii)* More quantitatively, Figure 5 plots the CDF of the intervals between consecutive accesses of the same simulated cache entry for the ISP trace. It is generated during the cache simulation. As shown in Figure 5, 59% of the intervals are less than 1 minute and 87% are less than 1 hour. We expect such strong temporal locality of cache entry access makes a non-persistent cache comparable to a persistent cache in terms of efficiency.

## 4.3 Diversity Among Applications

This subsection investigates the caching efficiency of individual smartphone applications.

**Identifying smartphone applications** is trivial for the UMICH trace, which contains the process name for each packet and hence for each HTTP transaction. 741 unique processes were observed from the UMICH dataset.

For the ISP dataset whose application identification is less straightforward, we used the **`User-Agent`** field in HTTP requests to distinguish different applications. First, from the 48,214 unique **`User-Agent`** strings appeared in the dataset, we picked the top 500 strings with the highest HTTP traffic coverage, yielding an overall traffic coverage ratio of 95.5%. We found many **`User-Agent`** strings belong to the same application. They are only slightly different in OS versions, hardware specifications, and languages, *etc.* For example, Apple iTunes has the following **`User-Agent`** format: **`iTunes-`***Device***`/`***iOS version* **`(`***device version***`;`** *memory size***`)`**, such as **`iTunes-iPhone/4.3.3(4;16GB)`** or **`iTunes-iPhone/4.2.1 (2;8GB)`**. To avoid duplicated applications, we generated 95 regular expressions, each corresponding to an app for a specific device and/or OS, that cover all 500 **`User-Agent`** strings. All regular expressions follow a simple pattern of *app_name*⋆*device/OS_name*⋆, such as **`iTunes*iPhone*`** and **`Pandora*iOS*`**, by ignoring other less significant fields such as the OS version number.

**Redundant transfers for top applications** are measured in Table 5 (assuming an ideal cache for simulation). For the UMICH (ISP) trace, we show the top 12 process names (**`User-Agent`** regular expressions), their contributions of HTTP traffic, and their fractions of redundant bytes (their names have been anonymized). Due to the heavy-tail distribution of the smartphone application

**Table 6: A summary of our findings regarding to the traffic volume impact of redundant transfers.**

| Question | Our finding based on the two datasets | § |
|---|---|---|
| File cacheability | Most bytes (70% to 78%) and most files (66% to 72%) are cacheable. | 4.1 |
| Traffic volume impact of redundant transfers | They account for 18% to 20% of HTTP traffic, and about 17% of the overall traffic for the UMICH trace. | 4.1 |
| Main reason for redundant transfers | Problematic caching logic of the handsets (instead of the sever). | 4.2 |
| Impact of cache size on redundant transfers | Limited. The detected redundant bytes are at least 13% even for a simulated cache of as small as 4 MB. | 4.2 |
| Suggested handset cache size | Reducing the cache size from infinity to 50 (100) MB causes cache misses for at most 2.0% (1.4%) of HTTP bytes. | 4.2 |
| Difference between persistent cache and non-persistent cache | Very small. Both have similar caching efficiency due to strong temporal locality of accessing the same cache entry. | 4.2 |
| Caching efficiency of individual mobile apps | Some popular apps have unacceptably high fractions (93% to 100%) of redundant transfers. | 4.3 |

usage [36], these top apps are responsible for more than 83% of all HTTP traffic. We found that while some apps incur small fractions of redundant data, some have unacceptably high redundancy ratios.

**To validate the cache simulation results**, we further studied the four apps with high redundancy ratios greater than 90% as shown in Table 5. We used them locally by exploring their common application usage scenarios, whose packet traces were simultaneously collected by **`tcpdump`** running on our handsets. By analyzing the traces, we found that all four apps use HTTP as the application-layer protocol but none of them performs caching. For example, for the "Weather (A)" app, HTTP responses of the weather information contain the **`Expires`** and the **`Cache-Control:max-age`** directives both specifying a freshness lifetime of 5 minutes. But when we checked the weather for the same location again (we verified from the trace that the URLs were identical), the handset always issued a non-conditional request regardless of the freshness of the downloaded file.

**Inefficiency caused by HTTP POST.** Table 5 indicates that the "Map (A)" app incurs negligible redundant transfers, because almost all its bytes are not cacheable (not shown). Specifically, we found that instead of employing HTTP GET, the application heavily uses HTTP POST requests that point to a single static file name by including the parameters in the body of a POST request, making it impossible for HTTP to cache the responses. A more caching-friendly approach is to attach query strings to the URLs to make them cacheable.

We summarize important findings regarding to the traffic volume impact of redundant transfers in Table 6.

## 5. THE RESOURCE IMPACT

§4 reveals the traffic volume impact of redundant transfers due to inefficient caching. In cellular networks, resources such as handset battery life, radio resources, and signaling load could also become critical bottlenecks. We now focus on understanding the resource impact of redundant transfers.

### 5.1 Cellular Resource Management Policy

To efficiently utilize the limited radio resources, cellular networks employ a radio resource management policy distinguishing them from wired and Wi-Fi networks. In particular, there exists a radio resource control (RRC) state machine [31] that determines the radio resource usage based on application traffic patterns, affecting the handset energy consumption and the user experience.

**The RRC state machine** of the 3G UMTS cellular carrier used by the 20 participants generating the UMICH trace is depicted in Figure 6, which was inferred by our prior work [31]. The

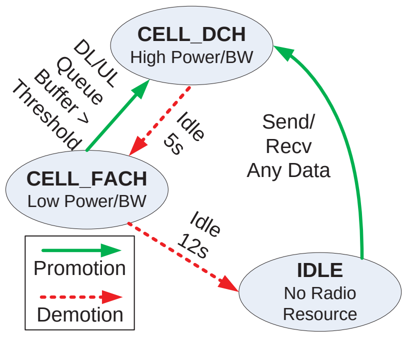

**Figure 6: The RRC state machine of the 3G UMTS cellular carrier used by the participants generating the UMICH trace.**

UMTS RRC state machine consists of three power states: **`IDLE`** (no connection), **`CELL_FACH`** (low-power, low bandwidth state), and **`CELL_DCH`** (high-power, high-bandwidth state). Similar RRC state machines exist in other types of cellular radio access networks such as GPRS/EDGE [12], 3G EvDO [19], and 4G LTE [25].

**A state promotion** (going from low to high power state) is triggered by a relatively high data transmission rate. It incurs a long delay up to several seconds during which tens of control messages are exchanged between a handset and the radio access network for resource allocation. A large number of state promotions increase the signaling overhead and worsen the user experience [31].

**A state demotion** (going from high to low power state) takes negligible time. However it is controlled by an inactivity timer that may cause significant waste of resources, because when waiting for timeout, a handset still occupies the transmission channel and the WCDMA codes, and its radio power consumption is kept at the corresponding level of the state[8]. If the inactivity timer, which is reset by any uplink or downlink packet, finally expires, we say the RRC state is demoted after a *tail*, which is the idle time period matching the inactivity timer value before a state demotion. The negative resource impact of tails in cellular networks was investigated in prior work [13, 31].

[8]In typical UMTS networks, each handset is allocated dedicated high-speed channel on **`CELL_DCH`**. For HSDPA [24], a UMTS extension with higher downlink speed, a high-speed channel may be shared by a limited number of handsets. Occupying it during the tail time can potentially prevent other handsets from using it.

## 5.2 Computing the Resource Impact

**Three metrics** are used to quantify the resource impact. We leveraged the RRC state machine simulator (configured with the state transition model shown in Figure 6) and the handset power model used by the ARO tool [32] for computing the metrics below.

- ***E***: the handset radio energy consumption. It is the energy consumed by the cellular radio interface whose radio power contributes 1/3 to 1/2 of the overall handset power during the normal workload [32]. $E$ is calculated by associating each RRC state or state promotion an average power value measured from a particular phone [31, 32]. The exact radio power values may also depend on other factors such as the signal strength, but our qualitative conclusions in terms of the energy impact should hold.
- ***S***: the signaling overhead quantified by the total state promotion delay.
- ***D***: the radio resources. It is estimated by the total **`CELL_DCH`** (the high-speed dedicated channel) occupation time (including the tail time).

We only studied the UMICH trace, because accurate RRC state reconstruction, a prerequisite for computing the above metrics, requires for each incoming and outgoing packet its precise timing and size, which is not available in the ISP trace. Also, we only considered the 3G traffic within the UMICH trace since the resource management policy of Wi-Fi is much more efficient due to its short-range nature[9].

**To quantify the resource impact** of redundant transfers, we *take the difference* of the computed metrics for the original trace and the modified trace with all redundant transfers removed. Specifically, the radio energy impact is computed as $\Delta E = (E_0 - E_R)/E_0$ where $E_0$ and $E_R$ correspond to the radio energy consumed by the original and the redundant-transfer-free trace, respectively. $\Delta E$ is positive as removing redundant transfers reduces energy consumption. The radio resource impact $\Delta D$ and the signaling impact $\Delta S$ are computed in similar ways. Note this general method [32] can be used to compute the resource impact of any transfers.

**Two factors may lead to underestimation** of the resource impact. *(i)* We conservatively consider all HTTPS traffic non-redundant. *(ii)* In the above approach, by removing redundant transfers (or any transfers of our interest), we assume that any of the remaining traffic is *unaffected* in terms of its schedule and occurrence. This may not be true in reality: if some redundant transfer is eliminated, the subsequent transfer may happen sooner, or some other traffic may not occur at all (*e.g.,* a DNS lookup). Ignoring such dependency leads to an underestimation of the resource impact because in reality the resultant redundant-transfer-free trace has shorter duration and/or less traffic than our simulated one. Although it is difficult to handle all such cases, we do address a common case where a DNS lookup would not occur if its corresponding HTTP transfer were eliminated. Specifically, when removing a redundant transfer $X$, we also try to eliminate a DNS lookup $D$ right before $X$, using a time window $\delta$ tolerating the handset processing delay. We empirically choose $\delta$=300 ms but varying it from 100 to 500 ms has negligible impact on the results.

## 5.3 Measurement Results

Table 7 measures the resource impact of redundant transfers in two scenarios: *(i)* consider only 3G HTTP traffic and exclude 3G non-HTTP traffic, and *(ii)* consider all 3G traffic in the UMICH trace. The "$\Delta E$ (HTC)" and "$\Delta E$ (Nexus)" columns refer to the radio energy impact using power parameters of an HTC TyTn II smartphone and a Google Nexus One smartphone [32], respectively. The presented results also assume an ideal cache. We found that similar to our findings in §4.2, as long as the simulated cache size is reasonably large (*e.g.,* greater than 10 MB), its impact on the measured resource impact of redundant transfers is small (results not shown).

**Table 7: Resource impact of redundant transfers (the UMICH trace), under the scopes of HTTP traffic and all traffic.**

| | $\Delta S$ | $\Delta E$ (HTC) | $\Delta E$ (Nexus) | $\Delta D$ |
|---|---|---|---|---|
| HTTP only | 26.9% | 26.1% | 25.9% | 27.1% |
| All traffic | 6.1% | 7.0% | 6.7% | 9.0% |

As shown in Table 7, by considering non-HTTP traffic, the resource impact of redundant transfers decreases sharply from more than 25% to less than 10%, although non-HTTP traffic accounts for only 13% of the total 3G traffic volume. This is attributed to two reasons explained below.

First, the resource impact of non-HTTP traffic can be significant although their traffic volume contribution is small. Due to the tail effect explained in §5.1, intermittently transmitting very small amount of data may utilize much more resources than transmitting large amount of data in one burst [32]. One representative example of such an inefficient traffic pattern identified in the UMICH trace is Android push notification (identified by TCP port 5228) and XMPP (Extensible Messaging and Presence Protocol, a popular instant messaging protocol using TCP port 5222 [1]) traffic. They account for 1% of the total 3G traffic volume while their resource impact in terms of $E$ (HTC) is 18%[10]. On the other hand, for all HTTP traffic dominating the overall 3G traffic volume (87%), the resource impact in terms of $E$ (HTC) is only 20%. Note that the traffic patterns of push notifications (and in general, delay-tolerant transfers) can be optimized to be more resource efficient [13, 30], resulting in higher resource impact of redundant transfers.

The second reason is *resource sharing*. Unlike the traffic volume measured in §4, resources in cellular networks can be shared by multiple HTTP sessions, or be shared by HTTP and non-HTTP transfers. Recall that in §5.1, the "radio-on" period of a transfer consists of a state promotion, its data transmission period and the following tail. If the radio-on periods of two transfers are fully or partially overlapped, then $D$ and hence $E$ are shared during the overlapped period. The signaling load $S$ is also shared in that only one state promotion is triggered by the two transfers. Resource sharing significantly reduces the resource impact of redundant transfers, many of which do not incur additional resource overhead because their channel occupation periods overlap with those of other transfers. For $S$, $E$ (HTC), $E$ (Nexus), and $D$, the fractions of resource reduction due to resource sharing among redundant transfers and other transfers are 70%, 61%, 63%, and 50%. respectively[11].

[9]Wi-Fi has very short tail time and negligible promotion delay[13].

[10]It is computed using the method described in §5.2 by taking the difference of the radio energy for the entire trace and the modified trace where push notification and XMPP transfers are removed.

[11]It is computed as $1 - (U(T) - U(T_2))/U(T_1)$ where $U(\cdot)$ computes the resource consumption for a certain trace. Trace $T$ is the original trace of all traffic. $T_1$ consists of only redundant transfers. $T_2$ corresponds to $T$ with $T_1$ removed.

**Table 8: Our tested HTTP libraries and smartphone browsers.**

| Name | HTTP library or browser | Platform | Handset |
|---|---|---|---|
| UC | `java.net.URLConnection` | Android 2.3 | Samsung Galaxy S |
| HUC | `java.net.HttpURLConnection` | Android 2.3 | Samsung Galaxy S |
| HC | `org.apache.http.client.HttpClient` | Android 2.3 | Samsung Galaxy S |
| WV | `android.webkit.WebView` | Android 2.3 | Samsung Galaxy S |
| HRC | `android.net.http.HttpResponseCache` | Android 4.0.2 | Samsung Galaxy Nexus |
| T20 | `Three20` (Version 1.0.6.2) | iOS 4.3.4 | iPhone 4 |
| NSUR | `NSURLRequest` | iOS 5.0.1 | iPhone 4S |
| ASIHR | `ASIHTTPRequest` (Version 1.8.1) | iOS 4.3.4 | iPhone 4 |
| AB | The Android web browser | Android 2.3 | Samsung Galaxy S |
| SB | The Safari web browser on iPhone | iOS 4.3.4 and 5.0.1 | iPhone 4 and 4S |

## 6. FINDING THE ROOT CAUSE

We learn from §4.2 that the main reason for redundant transfers is the problematic caching logic. We verify this by performing comprehensive caching tests for state-of-art HTTP libraries and browsers of Android and iOS.

Previously, professional developers also spent efforts investigating HTTP cache implementation issues leading to poor performance, focusing on mobile browsers [5, 7, 8, 6, 9]. Our tests go beyond them in two aspects. *(i)* Our tests are much more complete, covering all important aspects of caching implementation. To our knowledge, only three (Test 7, 10, 11) out of the thirteen tests described in §6.1 were performed before. *(ii)* Prior efforts only investigated browsers but we further examined HTTP libraries that are heavily used by today's smartphone applications.

### 6.1 Test Methodology

We examined eight HTTP libraries listed in Table 8. To the best of our knowledge, they cover all publicly available HTTP libraries for Android and iOS. To test them, we wrote small applications using these libraries as HTTP clients. We also investigated the default browsers on Android and iPhone, using strategically generated HTML pages embedding multiple web objects to perform tests involving multiple files (for Test 11 and 12).

We performed all tests on real handset devices: Samsung Galaxy S with Android 2.3, Samsung Galaxy Nexus with Android 4.0.2, iPhone 4 with iOS 4.3.4, and iPhone 4S with iOS 5.0.1. Each handset has non-volatile storage of at least 10 GB. In each test, a client only requested files, whose caching directives were properly configured, from our controlled HTTP server running Apache 2.2. We ran `tcpdump` on the server to monitor incoming HTTP requests to tell whether a request we made was served by the handset cache or by the server.

We took the following measures to further eliminate external factors that may affect the accuracy. *(i)* Before launching each test, the handset cache (if existed) was always cleared either manually (for browsers) or by calling the corresponding APIs (for libraries). *(ii)* We verified that the caching behaviors of the server in all tests were correct by analyzing the traces collected at the server[12]. The clocks of both the server and the handset were synchronized as well. *(iii)* We also ran `tcpdump` on the handset and compared HTTP requests and responses observed at the handset and at the server. We found both to be identical in all tests, implying that the cellular middleboxes did not change any caching directives. This is further confirmed by the fact that using cellular and Wi-Fi yielded the same testing results.

[12]The server was also tested by `http://redbot.org`, an online tool for checking HTTP caching implementation of web servers.

We designed 13 black-box tests to understand how caching was implemented in the state-of-art HTTP libraries and mobile browsers. Test 1 to Test 7 verify whether key features (*e.g.,* revalidation) were supported. Failure to support any of them may lead to redundant transfers. Test 8 to Test 13 determine important attributes of a cache (*e.g.,* its size) whose implications on redundant transfers cannot be overlooked either. Note the testing methodology is generally applicable to HTTP libraries and browsers on any platform. We detail the tests below.

**Test 1** (Basic caching)**.** The handset requests for a small cacheable file $f$. The server transfers $f$ with a proper `Expires` directive. Then the client requests for $f$ again before it expires. If the basic caching is supported, the second request should not incur any network traffic.

**Test 2** (Revalidation)**.** It is similar to Test 1 except that the client requests for $f$ after it expires. For the second request, the client should issue a conditional request with an `If-Modified-Since` directive. Then the server will respond with an HTTP 304 indicating the file has not changed.

**Test 3** (Non-caching directives)**.** It tests whether the following directives are correctly followed: `Cache-Control:no-store`, `Pragma:no-cache`, and `Cache-Control:no-cache`. Their caching logic is described in §2.

**Test 4** (Expiration directives)**.** It is similar to Test 1 except that the server uses other expiration directives: *(i)* `Cache-control:max-age`, *(ii)* `Expires` and `max-age`, and *(iii)* `max-age` and `age`. In *(ii)*, `max-age` should override `Expires`. In *(iii)*, the file should always expire if `age` is greater than `max-age`.

**Test 5** (URL with query string)**.** It is similar to Test 1 except that the URL contains a query string (*e.g.,* `query.php?p=123`).

**Test 6** (Partial caching)**.** The handset performs a byte-range request $[p_1, q_1]$ for a small cacheable file $f$. The server transfers the corresponding range of $f$ with a proper `Expires` directive and an eTag. Before the response expires, the client performs another byte-range request $[p_2, q_2]$ for $f$. If $p_2 \geq p_1$ and $q_2 \leq q_1$, then we should not observe the second request over the network.

**Test 7** (Redirection)**.** We test whether a handset caches two types of responses: `301 Moved Permanently` and `302 Found`. They are common ways of performing a permanent and a temporary redirection, respectively. A 301 response is always cacheable unless indicated otherwise (*e.g.,* by `Cache-Control:no-store`). A 302 response is only cacheable if indicated by a `Cache-Control` or `Expires` header field. The testing procedure is similar to Test 1 except that the response of the server is HTTP 301 or 302, redirecting the request to another small file.

**Test 8** (Shared or non-shared cache)**.** We run two applications $A$ and $B$ using the same library to be tested on the same phone. $A$ requests for a small cacheable file $f$. Then $B$ requests for $f$ again before $f$ expires. If the cache is shared by both applications, the second request should not incur any network traffic. Otherwise we will see two requests over the network. We did not find any publicly available API that allows one to read or modify cache entries of the default Android or iPhone browser, whose caches are thus assumed to be non-shared.

**Test 9** (Persistent or non-persistent cache)**.** We perform a test similar to Test 1 except that we reboot the phone after receiving the first response. A persistent cache must survive a device reboot (and therefore a process restart).

**Test 10** (Cache entry size limit)**.** A large file may not be cached by a given cache implementation with a cache entry size limit. We perform binary search for this limit by fetching cacheable files of varying sizes from the server. For a file of size $s_i$, we perform Test 1 after clearing the cache to see whether $s_i$ is above the cache entry size limit. Previous measurement [8] reported that HTML pages and external web objects (*e.g.,* JavaScript and CSS) may have different cache entry size limits, which are measured separately.

**Test 11** (Total cache size)**.** We perform binary search for the total cache size. To test whether it exceeds a particular value $z$, we clear the cache and download $n$ cacheable files $f_1, ..., f_n$ each with a size of $z/n$ (smaller than the cache entry size limit inferred by Test 10). We then request for the $n$ files again. $z$ exceeds the total cache size if and only if any file is transferred over the network in the second pass.

**Test 12** (Replacement policy)**.** We test for five cache replacement algorithms known to be commonly implemented [35]: *(i)* LRU (Least Recently Used), *(ii)* LFU (Least Frequently Used), *(iii)* evicting the oldest cache entry, *(iv)* evicting the cache entry with the nearest expiration time, and *(v)* evicting the cache entry of the largest size.

To test whether the replacement policy is LRU, we first fill up the cache using $n$ cacheable files $f_1, ..., f_n$ such that $\sum_{i=1}^{n} f_i < z$ where $z$ is the total cache size inferred by Test 11. Next, we randomly generate an $n$-permutation $p_1, p_2, ..., p_n$, and then request for $f_{p_1}, ..., f_{p_n}$ again. Subsequently the handset downloads a new file $f_{n+1}$ such that $\sum_{i=1}^{n+1} f_i > z$, thus triggering a cache entry eviction. If $f_{p_1}$ is evicted (based on Test 1), then we know LRU is the replacement policy since $f_{p_1}$ is the least recently accessed file. Other replacement algorithms are tested in similar ways.

**Test 13** (Heuristic freshness lifetime)**.** The HTTP server is configured in a way that it puts neither **`Cache-Control:max-age`** nor **`Expires`** in a response. Then we test whether a small file can be cached by performing Test 1 in which the two requests are sent back to back. If it can, we do binary search for the heuristic freshness lifetime (§3.3.1) by varying the interval between the two requests.

## 6.2 Test Results

Table 9 summarizes the results (refer to Table 8 for acronyms of the libraries and browsers). Each feature in Test 1 to Test 7 can be fully supported (indicated by a "●" symbol), not supported at all ("○") or partially supported ("◑" with the reason explained). For each of the attribute tests (Test 8 to Test 13), a "✕" symbol means the test was not performed since the corresponding API does not support HTTP caching. We highlight key findings as follows.

- To our surprise, among the eight HTTP libraries, four (three for Android and one for iOS) do not support caching at all. Smartphone apps using these libraries thus cannot gain any benefit from caching.
- For libraries and browsers that do support caching, they may not strictly follow RFC 2616, as detailed by footnotes $d, h, j, k$ in Table 9. To our knowledge, only observations $h$ and $j$ were reported by previous measurements [9, 6] (for browsers only). All such cases of non-compliance potentially incur redundant transfers.
- The Android browser uses a small cache of 8 MB. As described in §4.2, increasing the cache size brings non-trivial reduction of cache misses.
- No library or browser supports partial caching, although its impact on redundant transfers is limited (§4.1).
- We found that for all libraries that support caching, in order to leverage the caching support, a developer still needs to configure the library. However, developers can easily skip that for simplicity, or they can simply be unaware of it, therefore missing the opportunity of caching and incurring redundant transfers.

By exposing the shortcomings of existing implementations, our work helps encourage library and platform developers to improve the state of the art, and helps application developers choose the right libraries for better performance.

## 7. RELATED WORK

We describe related work in four categories.

**Extensive research on web caching** has been done since the World Wide Web was in its nascent state. We summarize the important topics. *(i)* Web server workload modeling and characterization, focusing on the implication on caching [14, 16]. *(ii)* Efficient cache replacement algorithms [17, 35]. *(iii)* Efficient cache validation and invalidation techniques for strong consistency [26, 27]. Note a *validation* is initiated by a client which verifies the validity of its cached files (as used in HTTP), while an *invalidation* is performed by the origin server which notifies clients which of its cached files have been modified. *(iv)* Cooperative proxy caching [18, 34] where individual proxies share their cached files with each other's clients. *(v)* Caching-friendly content representation such as delta encoding [29].

**Caching in mobile networks.** Recent study [20] explored the potential benefits of HTTP caching in 3G cellular networks by analyzing traffic traces collected from a large 3G cellular carrier. They found that the cache hit ratio is 33% when caching at the Internet gateway. Another study [22] investigated the potential for caching video content in cellular networks, indicating that 24% of the bytes for progressive video download can be served from a network cache located at the Internet gateway. By comparison, our study investigates caching efficiencies from the perspective of individual handsets.

Another recent study [33] examined three client-only solutions: caching, prefetching, and speculative loading, using web usage data collected from 24 iPhone users over one year. The authors focus on improving the smartphone browsing speed instead of saving the bandwidth. They found that 40% of resource requests can be served by a local browser cache of 6 MB, implying the necessity of HTTP caching. However, its effectiveness of reducing the latency is found to be not as good as that of reducing the traffic volume, mainly because revalidation cannot hide network RTT, which is an important factor affecting mobile browser performance.

**HTTP cache implementation on browsers.** Professional developers spent efforts investigating HTTP cache implementation

**Table 9: Testing results for smartphone HTTP libraries and browsers. ●: fully supported ○: not supported ◑: partially supported ×: not applicable. Refer to Table 8 for acronyms of the libraries and browsers.**

| Test Name | UC | HUC | HC | WV | HRC | T20 | NSUR | ASIHR | AB | SB |
|---|---|---|---|---|---|---|---|---|---|---|
| 1. Basic caching | ○ | ○[b] | ○[a] | ● | ● | ○ | ◑[j] | ● | ◑[j] | ◑[j] |
| 2. Revalidation | ○ | ○ | ○ | ● | ● | ○ | ● | ● | ● | ● |
| 3. Non-caching directives | ○ | ○ | ○ | ● | ● | ○ | ● | ◑[d] | ● | ● |
| 4. Expiration directives | ○ | ○ | ○ | ● | ● | ○ | ● | ● | ● | ● |
| 5. URL with query string | ○ | ○ | ○ | ● | ● | ○ | ● | ● | ● | ● |
| 6. Partial caching | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| 7. Redirection | ○ | ○ | ○ | ○ | ○ | ○ | ◑[h] | ○ | ◑[h] | ◑[h] |
| 8. Shared or non-shared cache | × | × | × | Non-shared | Non-shared | × | Shared | Shared by default[e] | Non-shared | Non-shared |
| 9. Persistent or non-persistent cache | × | × | × | Persistent | Persistent | × | Hybrid[l] | Persistent | Persistent | Persistent |
| 10. Cache entry size limit | × | × | × | 2 MB | No limit | × | NP: 50 KB P: 2 MB[l] | No limit / 512 KB[i] | 2 MB / 4 MB[i] | 250 KB[g] / 4 MB[i] |
| 11. Total cache size | × | × | × | Storage Capacity | Configurable[c] | × | NP: 1 MB P: 40 MB[f] | Storage Capacity | 8 MB | 100 MB |
| 12. Replacement policy | × | × | × | LRU | LRU | × | LRU | LRU | LRU | LRU |
| 13. Heuristic fresh lifetime | × | × | × | 17.5 hrs | 30 mins | × | 48 hrs | 0[k] | 48 hrs | 48 hrs |

[a] Including subclasses **`AbstractHttpClient`**, **`AndroidHttpClient`**, and **`DefaultHttpClient`**. None supports basic caching.
[b] The class provides caching interfaces through the abstract classes **`ResponseCache`**, **`CacheRequest`**, and **`CacheResponse`**. But developers need to implement them by themselves.
[c] The cache size must be specified by developers.
[d] The class does not cache responses with **`Pragma:no-cache`** or **`Cache-Control:no-cache`**.
[e] Developers can make it non-shared by specifying a private cache storage path.
[f] The default sizes are 1 MB for the non-persistent cache and 40 MB for the persistent cache. But they are also configurable by developers.
[g] Safari on iOS 5 has a larger cache entry size limit of 2 MB for an HTML page.
[h] They do not cache a cacheable HTTP 302 response.
[i] The first and the second numbers are the cache entry size limits for an HTML page and an external web object (*e.g.,* JavaScript), respectively.
[j] When loading the same URL back-to-back, the second load is treated as a reload without using a cached copy or issuing a conditional request.
[k] Revalidation is always performed when neither **`Cache-Control:max-age`** nor **`Expires`** exists in a response.
[l] Both a persistent and a non-persistent cache are used. Given a file whose size is $s$, it is stored in the non-persistent cache if $s <$50KB, or stored in the persistent cache if 50KB$\leq s <$2 MB, or not stored if $s \geq$ 2MB.

issues leading to poor performance, focusing on mobile browsers. The following measurement studies were reported on various technical blogs. Early measurement [5] in 2008 shows that the iPhone 3G browser has a non-persistent cache with an entry size limit of 25 KB (for HTML files) and a total size of 500 KB, implying potential performance issue for large web pages. The experiments were revisited in 2010 [7], and larger cache sizes of iOS 4 on iPhone 4 and Android 2.1 were observed. Similar tests of caching sizes were performed in [11]. Blog entry [8] further pinpoints that for iPhone and Android browsers, the cache entry size limit differs depending on the file type (we considered this in our tests). Blog entry [6] revealed an implementation bug of Safari on iOS 4 shown in footnote $j$ in Table 9. We confirmed this and found this problem also exists in the Android 2.3 browser and the **`NSURLRequest`** library. Blog entry [9] discovered that most desktop and mobile browsers do not cache HTTP redirections properly. Our caching tests cover all aforementioned aspects, and are much more complete as described at the beginning of §6.

**Data compression.** Besides caching, another orthogonal approach for redundancy elimination is data compression, which can be performed at each single object (*e.g.,* gzip [10]), across multiple objects (*e.g.,* shared dictionary compression over HTTP [15]), or for packet streams (*e.g.,* MODP [28]). Compression can be jointly applied with caching to further save the network bandwidth.

## 8. DISCUSSION AND CONCLUSION

Web caching in mobile networks is critical due to the unprecedented cellular traffic growth that far exceeds the deployment of cellular infrastructures. Caching on handsets is particularly important as it eliminates all network-related overheads. We have performed the first network-wide study of the redundant transfers caused by inefficient web caching on handsets. We found that redundant transfers account for 17% and 20% of the HTTP traffic, for the two large datasets, respectively. Further analysis on the UMICH trace suggests that redundant transfers are responsible for 17% of the bytes, 6% of the signaling load, 7% of the radio energy consumption, and 9% of the radio resource utilization of *all* cellular data traffic. Most of the redundancy can be eliminated by making the caching logic fully support and strictly follow the protocol specification, and making developers fully utilize the caching support provided by the HTTP libraries.

We plan to pursue three potential directions for further optimizing web caching for mobile networks.

**The offline application cache** is a new feature in HTML5, the latest HTML standard [3]. It differs from HTTP caching in two ways. *(i)* Caching information of all objects embedded in an HTML page is specified in a small *cache manifest* file associated with the HTML page. *(ii)* There is no explicit expiration, which is instead indicated by a new version of the manifest file that is always downloaded whenever the HTML page is fetched over the network. Although the usage of HTML5 caching is very unpopular in our datasets (only one app in the UMICH trace used it), analysts envision it will eventually be widely used as almost all smartphones are expected to support HTML5 by 2013 [2]. We expect strategically employing this coarse-grained caching mechanism with the traditional per-file-based HTTP caching can achieve more reduction of revalidation traffic causing non-trivial resource consumption despite their small sizes [33].

**Optimizing caching parameter settings** based on the file semantics is not addressed in this paper, as described in §3.1. However we do observe from our datasets examples where caching parameter settings are obviously too conservative. For example, in the UMICH trace, for the built-in weather app of Motorola Atrix, 95% of its bytes are marked by server as non-storable. We are conducting a more in-depth investigation on optimizing caching parameter configurations.

**Previous caching proposals** such as delta encoding [29] and piggyback cache validation [26] may provide additional benefits in cellular networks. For example, batching multiple validation requests into a single message [26] potentially reduces the resource overhead as otherwise each individual validation request may incur a separate tail. We plan to revisit both studies in our future work.

## 9. ACKNOWLEDGEMENTS

We thank Mark Gordon and Zhiyun Qian for their contribution to the data collection software for collecting the UMICH dataset, as well as the 20 students at University of Michigan who participated in our user study. We appreciate the valuable comments from Thomas Andrews, Yudong Gao, Mark Gordon, Sanae Rosen, Zhiyun Qian, and Qiang Xu. We would also like to thank the Mobisys reviewers and especially Matt Welsh for shepherding the paper. This work is partly funded by NSF grants CNS-1059372, CNS-1050157, CNS-1039657 and Navy grant N00014-09-1-0705.

## 10. REFERENCES

[1] Extensible Messaging and Presence Protocol. `http://xmpp.org/xmpp-protocols/`.

[2] HTML5-enabled phones to hit 1 billion in sales in 2013. `http://news.cnet.com/8301-1023_3-57339156-93/html5-enabled-phones-to-hit-1-billion-in-sales-in-2013/`.

[3] HTML5 (W3C working draft). `http://www.w3.org/TR/html5/`.

[4] Invest in Cell Phone Infrastructure for Growth in 2010. `http://pennysleuth.com/invest-in-cell-phone-infrastructure-for-growth-in-2010/`.

[5] iPhone Cacheability - Making it Stick. `http://www.yuiblog.com/blog/2008/02/06/iphone-cacheability/`.

[6] (lack of) Caching for iPhone Home Screen Apps. `http://www.stevesouders.com/blog/2011/06/28/lack-of-caching-for-iphone-home-screen-apps/`.

[7] Mobile Browser Cache Limits: Android, iOS, and webOS. `http://www.yuiblog.com/blog/2010/06/28/mobile-browser-cache-limits/`.

[8] Mobile cache file sizes. `http://www.stevesouders.com/blog/2010/07/12/mobile-cache-file-sizes/`.

[9] Redirect caching deep dive. `http://www.stevesouders.com/blog/2010/07/23/redirect-caching-deep-dive/`.

[10] The gzip home page. `http://www.gzip.org/`.

[11] Understanding Mobile Cache Sizes. `http://www.blaze.io/mobile/understanding-mobile-cache-sizes/`.

[12] GERAN RRC State Mchine. 3GPP GAHW-000027, 2000.

[13] N. Balasubramanian, A. Balasubramanian, and A. Venkataramani. Energy Consumption in Mobile Phones: A Measurement Study and Implications for Network Applications. In *IMC*, 2009.

[14] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web Caching and Zipf-like Distributions: Evidence and Implications. In *INFOCOM*, 1999.

[15] J. Butler, W.-H. Lee, B. McQuade, and K. Mixter. A Proposal for Shared Dictionary Compression over HTTP. `http://groups.google.com/group/SDCH`.

[16] R. Caceres, F. Douglis, A. Feldmann, G. Glass, and M. Rabinovich. Web proxy caching: the devil is in the details. *SIGMETRICS Perf. Eval. Rev.*, 26(3), 1998.

[17] P. Cao and S. Irani. Cost-aware WWW proxy caching algorithms. In *USITS*, 1997.

[18] A. Chankhunthod, P. Danzig, C. Neerdaels, M. Schwartz, and K. Worrell. A hierarchical internet object cache. In *USENIX ATC*, 1996.

[19] M. Chatterjee and S. K. Das. Optimal MAC State Switching for CDMA2000 Networks. In *INFOCOM*, 2002.

[20] J. Erman, A. Gerber, M. Hajiaghayi, D. Pei, S. Sen, and O. Spatscheck. To Cache or not to Cache: The 3G case. *IEEE Internet Computing*, 2011.

[21] J. Erman, A. Gerber, M. Hajiaghayi, D. Pei, and O. Spatscheck. Network-Aware Forward Caching. In *WWW*, 2009.

[22] J. Erman, A. Gerber, K. Ramakrishnan, S. Sen, and O. Spatscheck. Over The Top Video: the Gorilla in Cellular Networks. In *IMC*, 2011.

[23] R. Fielding, J. Gettys, J. Mogul, H. F. L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol - HTTP/1.1 . RFC 2616, 1999.

[24] H. Holma and A. Toskala. HSDPA/HSUPA for UMTS: High Speed Radio Access for Mobile Communications. John Wiley and Sons, Inc., 2006.

[25] J. Huang, F. Qian, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck. A Close Examination of Performance and Power Characteristics of 4G LTE Networks. In *Mobisys*, 2012.

[26] B. Krishnamurthy and C. Wills. Study of Piggyback Cache Validation for Proxy Caches in the World Wide Web. In *USITS*, 1997.

[27] C. Liu and P. Cao. Maintaining Strong Cache Consistency in the World-Wide Web. In *ICDCS*, 1997.

[28] C. Lumezanu, K. Guo, N. Spring, and B. Bhattacharjee. The Effect of Packet Loss on Redundancy Elimination in Cellular Wireless Networks. In *IMC*, 2010.

[29] J. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy. Potential benefits of delta encoding and data compression for HTTP. In *SIGCOMM*, 1997.

[30] F. Qian, Z. Wang, Y. Gao, J. Huang, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck. Periodic Transfers in Mobile Applications: Network-wide Origin, Impact, and Optimization. In *WWW*, 2012.

[31] F. Qian, Z. Wang, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck. Characterizing Radio Resource Allocation for 3G Networks. In *IMC*, 2010.

[32] F. Qian, Z. Wang, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck. Profiling Resource Usage for Mobile Applications: a Cross-layer Approach. In *Mobisys*, 2011.

[33] Z. Wang, F. X. Lin, L. Zhong, and M. Chishtie. How Far Can Client-Only Solutions Go for Mobile Browser Speed? In *WWW*, 2012.

[34] A. Wolman, G. Voelker, N. Sharma, N. Cardwell, A. Karlin, and H. Levy. On the scale and performance of cooperative Web proxy caching. In *SOSP*, 1999.

[35] K.-Y. Wong. Web Cache Replacement Policies: a Pragmatic Approach. *IEEE Network*, January/February, 2006.

[36] Q. Xu, J. Erman, A. Gerber, Z. M. Mao, J. Pang, and S. Venkataraman. Identifying Diverse Usage Behaviors of Smartphone Apps. In *IMC*, 2011.