Remote-Control Caching: Proxy-based URL Rewriting to Decrease Mobile Browsing Bandwidth

Ravi Netravali
MIT CSAIL
ravinet@mit.edu

James Mickens
Harvard University
mickens@g.harvard.edu

ABSTRACT

Mobile browsers suffer from unnecessary cache misses. The same binary object is often named by multiple URLs which correspond to different cache keys. Furthermore, servers frequently mark objects as uncacheable, even though the objects' content is stable over time.

In this paper, we quantify the excess network traffic that mobile devices generate due to inefficient caching logic. We demonstrate that mobile page loads suffer from more redundant transfers than reported by prior studies which focused on desktop page loads. We then propose a new scheme, called Remote-Control Caching (RC2), in which web proxies (owned by mobile carriers or device manufacturers) track the aliasing relationships between the objects that a client has fetched, and the URLs that were used to fetch those objects. Leveraging knowledge of those aliases, a proxy dynamically rewrites the URLs inside of pages, allowing the client's local browser cache to satisfy a larger fraction of requests. Using a concrete implementation of RC2, we show that, for two loads of a page separated by 8 hours, RC2 reduces bandwidth consumption by a median of 52%. As a result, mobile browsers can save a median of 469 KB per warm-cache page load.

CCS CONCEPTS

• **Information systems → Web applications**; • **Networks → Middle boxes / network appliances**; **Mobile networks**;

KEYWORDS

Web proxies, caching, content aliasing

ACM Reference Format:
Ravi Netravali and James Mickens. 2018. Remote-Control Caching: Proxy-based URL Rewriting to Decrease Mobile Browsing Bandwidth. In *HotMobile '18: 19th International Workshop on Mobile Computing Systems & Applications, February 12–13, 2018, Tempe , AZ, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3177102.3177118

1 INTRODUCTION

Users desire web pages that load quickly. However, on mobile devices with limited cellular data plans, users also desire pages that load with *a minimal amount of network traffic*. The average mobile-optimized web page is still 3.1 MB in size [8], which is close to the 3.6 MB size of the average desktop page [7]. HTTP objects are downloaded not just by traditional browsers, but by the many applications that use GUI-stripped browsers [4, 5] to fetch and render user-facing content. HTTP traffic, regardless of its source, consumes data plan bandwidth that is also coveted by video applications and non-HTTP-based programs. So, reducing the transfer bandwidth for mainline web content (i.e., HTML, CSS, JavaScript, and images) is important.

Mainline content is amenable to client-side caching. Objects like CSS files and JavaScript files are often used by multiple pages; furthermore, a given page will often use the same version of an object across multiple page reloads. Unfortunately, traditional caching suffers from low hit rates on mobile devices. The reasons are myriad. For example, web servers often use time-based expiration instead of content-based `ETags` expiration [1], resulting in browsers making unnecessary fetches of unchanged content. Caching rules are also defined using *exact match* semantics for URLs. Exact-match semantics can result in cache misses even if the necessary bytes already reside in the cache (§2).

In this paper, we provide two contributions. First, we perform an empirical study of how traditional caching logic misses opportunities to avoid redundant downloads. For example, we show that 89% of mobile pages have at least one object which is named via multiple URLs that only differ in the query string (§3.5). We also find that 95% of pages contain objects that do not change across reloads, but are marked as uncacheable by servers. Comparing our results to those of prior studies which focused on content aliasing in desktop web pages [9, 10], we find that redundant transfers are a *worse* problem in the mobile setting, causing at least twice as many cache misses as in the desktop setting.

Our second contribution is the design and evaluation of *Remote-Control Caching (RC2)*. RC2 allows mobile carriers or device manufacturers to improve cache hit rates without changing mobile browsers or mobile operating systems. In the current world, mobile browsers often use web proxies to compress content [1]; in RC2, proxies also *actively rewrite the embedded URLs in HTML and JavaScript*. In particular, an RC2 proxy tracks information about the contents of a phone's browser cache. When a phone requests a page, the RC2 proxy loads that page using a headless browser, determining the external objects (e.g., images and CSS files) whose raw bits are cached on the phone, but stored under different URLs than the ones used by the page to load. The RC2 proxy rewrites those URLs so that the page references the associated objects via URLs that will guarantee cache hits. The result is that phones download much less data per page load. For example, on an LTE network and a Nexus 5 phone, across the 500 most popular sites, RC2 reduces bandwidth costs by 52% for the median page that was reloaded

HotMobile '18, February 12–13, 2018, Tempe , AZ, USA

ACM ISBN 978-1-4503-5630-5/18/02…$15.00
https://doi.org/10.1145/3177102.3177118

eight hours after its initial load; this reduction translates into raw bandwidth savings of 469 KB.

2 TRADITIONAL CACHING

A browser cache is a key/value store. A key is a full HTTP request, which includes the URL of the object to fetch, and the HTTP request headers in the fetch. A cache value represents the binary contents of an object, e.g., the bits in an image.

When a web server returns an object to a browser, the server uses HTTP response headers to describe the cacheability of the object. For example, the header `Cache-Control: max-age=500` indicates that a browser may cache the object for 500 seconds. `Cache-control: no-store` indicates that the browser should never cache the object.

A server can also indicate that an object is *conditionally cacheable*, meaning that the browser's cached version should only be used if a particular condition is true. When a server returns a conditionally cacheable object, the response includes a `Last-Modified: timestamp` header, and/or an `ETag: opaque-string` header. A `Last-Modified` header indicates the creation time of the object; an `ETag` header provides a unique id for the object. An `ETag` id may be strong validator like a hash value, or a weak validator like a version number. When the browser issues a request for a conditionally cached object, the request includes the appropriate `If-Modified-Since: timestamp` or `If-None-Match: opaque-string` headers. The server parses those headers, determines whether the requested object has changed, and returns either a `200: OK` response with a new version of the object, or a `304: Not modified` response with no object data.

The problem with the traditional caching protocol is that *cache keys are defined by HTTP request state instead of raw object content.* A single binary object can be named by an arbitrary number of HTTP requests; each of those requests is a different cache key, even though each request maps to the same object. Consider the URL `http://a.foo.com/image.jpg?abcd`.

- **Modified domain names:** Popular sites use CDNs to distribute load and push content servers closer to users. The hostnames for CDN servers often differ in just a few positions (e.g., `a.foo.com` versus `b.foo.com`). However, those differences mean that if a browser fetches the same object twice, from two different CDN servers, the two HTTP requests will represent different cache keys, even if the responses contain the same bits.
- **Modified resource names:** A resource name specifies an object to fetch (e.g., `image.jpg`). Web servers often add client-specific strings to resource names, such that clients refer to the same underlying object using different resource names.
- **Modified query strings:** Query strings like `?abcd` are typically used to embed user-provided data, e.g., from a form. Query strings in an HTTP request convey important information to a server. However, servers often return the same bytes in response to requests with different query strings.
- **Improper caching headers:** Even if a request exactly matches an earlier one, a server may have marked the response to the earlier request as uncacheable. If this happened, the second request will suffer a cache miss, and fetch bytes that already exist on the client.

Metric	Median	95th Percentile
# of objects	95	369
# of bytes	1.4 MB	6.1 MB

Figure 1: Summary of our 500 page test corpus.

As we empirically demonstrate in Section 3.5, all four of these scenarios are common. Note that `ETag` headers are not a solution—`ETag` headers enable content-based caching for a particular URL, but the scenarios that we described above involve *multiple, distinct URLs* that refer to the same object.

3 PREVENTING ALIAS-INDUCED MISSES

In this section, we first describe an optimal content-based caching scheme; this optimal scheme defines an upper bound on the bandwidth savings that a concrete scheme can provide. We then describe the design of RC2. RC2 is a web proxy that rewrites embedded URLs in HTML, allowing the proxy to effectively take control of a phone's cache management strategy. By tracking the content in client-stored objects, and rewriting URLs to refer to those objects whenever possible, RC2 unlocks 99.1% of the optimal bandwidth savings.

3.1 Methodology

We used a 500 page test corpus that was collected using Mahimahi, a tool for recording and replaying HTTP traffic [18]. Each test page was the landing page for an Alexa Top 500 site [2]; we used the mobile-optimized version of a page when such a version existed. Figure 1 provides a high-level summary of the pages in the corpus.

All experiments used a Nexus 5 phone that ran Android 5.1.1 and Google Chrome v53. For experiments that involved page load times, we loaded each individual page five times using cold browser and DNS caches, and five times using warm browser and DNS caches; we recorded the median load time for each scenario. We defined "load time" as the elapsed time between the `navigationStart` and `loadEventEnd` JavaScript events.

3.2 Optimal Content-based Caching

We used Mahimahi to record three versions of each page: an initial version, a version that was recorded a few seconds later, and a version that was recorded eight hours later. The first and second versions represented the pages that a browser would see if it loaded a page and then immediately reloaded it. The first and third versions represented what a browser would see if it loaded a page, and then waited eight hours to reload the page.

For each of the three versions, we examined the caching headers for each object in the version, determining which HTTP requests would incur network fetches in the immediate-reload scenario and the delayed-reload scenario. We also examined the *contents* of the objects in the three versions of each page. For each object, we calculated a SHA1 hash. Using those hashes, we determined when an omniscient content-based cache would incur network fetches in the immediate-reload and delayed-reload scenarios. In particular, let $bits(obj_i)$ correspond to the binary content in obj_i, and let $bits(req, t)$ correspond to the bits associated with the HTTP request req when fetched at time t. For a page reload at $t_{reload} > t_{initial\_load}$, we declared a particular request to hit

in the cache if $bits(req, t_{reload}) = bits(obj_i)$ for any obj_i that was fetched at $t_{initial\_load}$. This calculation ignored all of the traditional caching logic; for example, even if an object's response headers at $t_{initial\_load}$ marked the object as uncacheable, the object might still provide a content-based cache hit for some request at t_{reload}. This setup evaluates a perfect content-based cache with infinite storage space and a priori knowledge of the mappings between URLs and raw objects.

3.3 RC2 Caching

An RC2 proxy is a remote dependency resolution (RDR) proxy. Before explaining how RC2 rewrites HTML, we first provide a brief overview of RDR proxying.

RDR Fundamentals: When a mobile browser issues an HTTP fetch for a page's top-level HTML, an RDR proxy loads the entire page locally, using a headless browser that runs on the proxy.[1] The proxy loads the page fully, parsing the top-level HTML and fetching all of the external objects that are referenced by that HTML. After returning the fetched HTML, most RDR proxies will proactively push the fetched external objects to the client, allowing the client to load the objects locally when the associated HTTP requests are generated during the local HTML parse. This approach, used by Amazon Silk [3], Opera Mini [19], and PARCEL [23], reduces page load times, but requires a modified client browser to handle the proxy's object-push logic. In contrast, RC2 is designed to work with unmodified mobile browsers. So, after an RC2 proxy returns HTML to the client, the proxy buffers the fetched external objects, draining the buffer as the phone issues HTTP requests for those objects.

RC2: In addition to an object buffer, an RC2 proxy maintains a per-client data structure that persists across individual page loads; this data structure represents the proxy's belief about which objects are stored in the client's browser cache. The data structure is a table which maps the hash value of a client-resident object to 1) the URL that the client used to fetch that object, and 2) the cache expiration date, as indicated by the HTTP response headers for the associated fetch.

A client's proxy-side table is initially empty. At some point, the client issues an HTTP fetch for a page's top-level HTML. The proxy fetches and loads the HTML in a headless browser, triggering additional proxy-side HTTP fetches for the external objects referenced by the top-level HTML. For each HTTP fetch via URL U_i, the proxy calculates the hash value H of the retrieved object. If the client has no table entry for H, the proxy adds the table entry $t[H] = <U_i, exp_{U_i}>$, where exp_{U_i} is the cache expiration date for the object, as indicated by the HTTP response headers. If $t[H]$ is not empty, then there is an aliasing relationship between $<U_i, exp_{U_i}>$ and some preexisting table value $<U_{alreadyCached}, exp_{U_{alreadyCached}}>$. Assuming that $exp_{U_{alreadyCached}}$ is still in the future, the proxy rewrites the HTML's reference to U_i, changing the reference to $U_{alreadyCached}$; this change prevents an aliasing-based cache miss on the mobile browser. If $exp_{U_{alreadyCached}}$ has already passed, then the proxy updates $t[H]$ with $<U_i, exp_{U_i}>$, and does *not* rewrite the reference to U_i in the HTML.

An RC2 proxy uses DOM shimming [13] to rewrite the URLs for HTML tags that are dynamically created via DOM methods like `document.appendChild()`. An RC2 proxy also rewrites URLs for JavaScript-initiated `XMLHttpRequests` that issue during a page load. The proxy detects aliasing relationships for `XMLHttpRequests` during the proxy-side page load, creating a map between the baseline URLs and the rewritten ones. The proxy includes this map in a small JavaScript library which the proxy injects into the page's HTML. This library also uses DOM shimming [13] to interpose on the mobile browser's creation of `XMLHttpRequest` objects. When the mobile browser creates `XMLHttpRequests`, the shimmed objects consult the URL map to determine which objects to fetch.

An RC2 proxy must avoid situations in which URL rewriting causes a mobile browser to load a page with inconsistent content, i.e., a set of objects which would never be seen in an unmodified load of the page. To avoid these problems, an RC2 proxy uses two mechanisms.

- First, an RC2 proxy always marks top-level HTML as uncacheable, ensuring that the proxy controls the objects fetched for every load of a page. This caching strategy deviates little from the status quo, since top-level HTML is often dynamically-generated and therefore unsuitable for caching. We examined the cache headers for all of the HTML objects in our 500 page test corpus, and found that only 6.7% of objects were naturally marked as cacheable.
- An RC2 proxy also does not rewrite U_i to $U_{alreadyCached}$ if $exp_{U_{alreadyCached}}$ is less than 30 seconds in the future. This policy ensures that the associated object in the client cache will not expire half-way through the page load, resulting in a live fetch for that object which might lead to inconsistent page content.

RC2 must also be wary of users who unilaterally delete some or all of the mobile browser's cache entries. If this happens, then the mobile browser may issue an HTTP fetch for an object which the RC2 proxy (incorrectly) believed would be satisfied by the client's cache. When the proxy receives such an unexpected HTTP request, the proxy forces the request to fail by terminating the TCP connection. The termination induces a client-side `onerror` JavaScript event; the proxy rewrites each page's HTML to include a custom `onerror` handler that, when fired, forces the entire page to reload via a call to the browser's `Location.reload(true)` JavaScript method. Note that passing `true` to the method forces the reload to bypass the browser cache and load the page directly from the origin server (via the proxy). When the proxy detects such a reload, the proxy discards the entirety of the client's table, under the assumption that the proxy's view of the client-side cache is now totally desynchronized and must be rebuilt from scratch.

As a mobile phone is used, it may switch between different cellular towers, or between cellular service and WiFi service. If the RC2 proxy is maintained by phone vendors, then a phone can redirect its web traffic to the proxy regardless of whether the phone's last-mile link is cellular or WiFi; this network-agnostic approach

[1] A headless browser is one that has no GUI, but is otherwise equivalent to a regular browser. PhantomJS [6] is the most popular headless browser, and is the one used by our RC2 prototype.

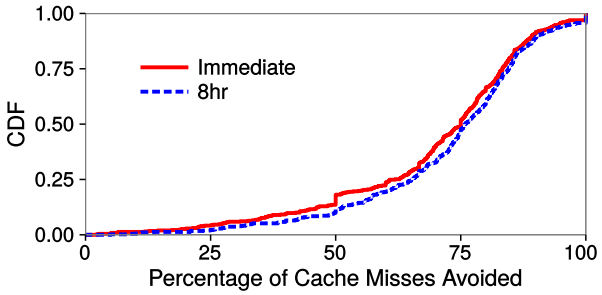

Figure 2: The fraction of cache misses that RC2 avoids, relative to the total number of misses incurred by traditional caching. Results span the 500 page test corpus.

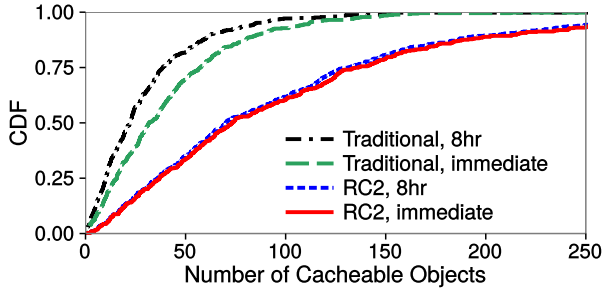

Figure 3: The number of cached objects during an immediate page reload, and an 8-hour-delayed reload.

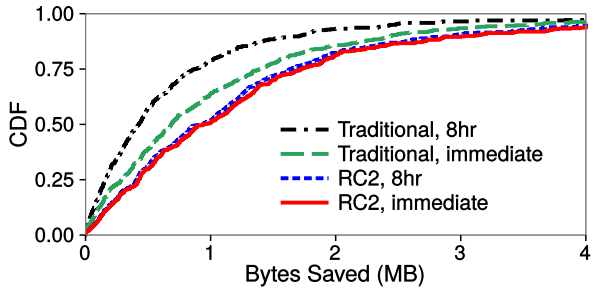

Figure 4: The number of bytes saved during an immediate page reload, and an 8-hour-delayed reload.

is used by Google's compression proxy for Android phones [1]. If the RC2 proxy is instead deployed by a cellular provider, then the proxy will lose visibility into the phone's cache updates when the phone switches to a WiFi network. These updates may cause the phone and the proxy to become desynchronized, forcing the proxy to discard the user's RC2 table when the phone reassociates with the cellular network.

3.4 Bandwidth Savings

We built a prototype RC2 system by modifying the Cumulus RDR proxy [18]. Figure 2 demonstrates the performance of our prototype. For both immediate and delayed page reloads, RC2 eliminates a median of *75% of the cache misses* that would be suffered by a traditional, URL-indexed cache. These savings are roughly *three times* the savings observed in a 2002 study of content aliasing on the web [10, 14], and roughly *two times* the savings observed in a 2011 study of content aliasing on the web [9]. Both studies focused on traffic involving desktop browsers, so additional research is needed to determine whether our findings generalize to non-mobile pages.

	Raw savings	% savings
Optimal	948 (4420) KB	69.9% (72.5%)
RC2	911 (4284) KB	67.5% (70.3%)
Traditional	682 (3543) KB	50.4% (58.1%)

(a) Immediate reloads.

	Raw savings	% savings
Optimal	909 (4191) KB	67.1% (68.7%)
RC2	901 (4093) KB	66.5% (67.3%)
Traditional	432 (2439) KB	31.9% (40.0%)

(b) Delayed reloads (8 hours).

Figure 5: Bandwidth saved at the median (95th percentile): optimal content-based caching (§3.2), RC2 (§3.3), and traditional caching.

Whereas Figure 2 shows the fractional reduction in cache misses, Figures 3 and 4 provides raw numbers for the cacheable objects found per page reload, and the bandwidth saved per page load due to cache hits. For example, during an immediate reload, traditional caching logic found a median of 32 cacheable objects per page; as a result, the reload needed 50.4% less bandwidth than the initial load. RC2 identified a median of 73 cacheable objects, allowing the reload to use 67.5% less bandwidth than the initial load. The additional cache hits represented a raw bandwidth savings of 229 KB over traditional caching.

In a delayed-reload scenario, both traditional caching and RC2 had lower hit rates. However, RC2 retained its performance advantage. With an eight hour separation between the initial visit and the reload, RC2 saved a median of 1.1% less bandwidth than in the immediate reload scenario; traditional caching saved 36.7% less.

As Figure 5 demonstrates, an RC2 proxy achieves 99.1% of the maximum possible savings in the delayed reload case. In the immediate reload case, RC2 achieves 96.1% of the optimal savings. Traditional caching only enables 47.5% of the optimal savings for delayed reloads, and only 71.9% for immediate reloads.

3.5 Analysis of Bandwidth Savings

In Section 2, we described four reasons why traditional caching logic often leads to unnecessary cache misses. Figure 6 uses that taxonomy to explain why RC2-based caching outperformed traditional caching. Each of the four reasons was impactful, but surprisingly, misspecified caching headers ("Req match, resp uncached") were the most popular reason for cache misses with traditional caching logic. In the median page, 33.1% of newly cacheable objects had been fetched during the initial page load, but marked as uncacheable by the server, such that, during a reload, requests for those objects would generate cache misses, even though the bits in the objects had already been fetched.

Modified domain names were the second most popular reason for cache misses, causing 29.9% of the misses for the median page. Modified query strings caused 20.8% of the misses for the median

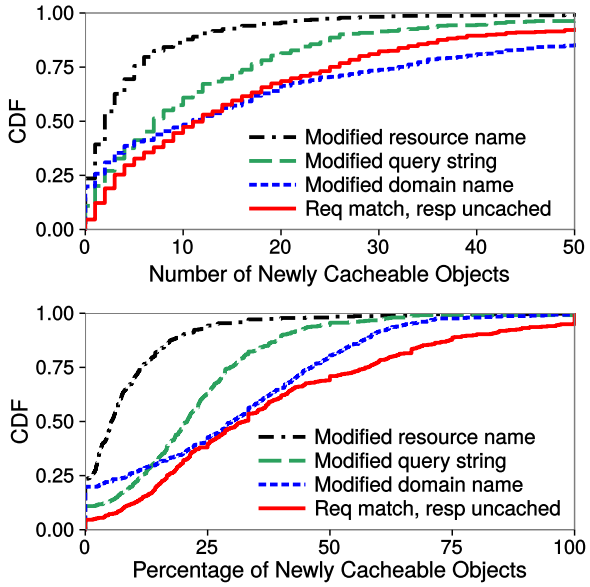

Figure 6: The reasons why traditional caching generated misses for objects that would hit in an RC2-managed cache. These results came from the immediate reload experiments.

Object Type	% newly cacheable objects	% newly cacheable bytes
JavaScript	29.8%	26.6
CSS	5.0%	3.3
Images	46.3%	63.4
Fonts	1.2%	2.1
JSON	13.8%	3.9
Other	3.9%	0.7

Figure 7: The types of newly cacheable objects that RC2 found. Recall that RC2 marks HTML as uncacheable (§3.3).

page. Modified resource names were the least common reason for misses, causing only 5.4% of misses for the median page.

Every page in our corpus suffered from at least one of the four types of unnecessary cache misses. Additionally, 89% of pages contained at least one object that was named by multiple URLs which only differed in their query strings. 95% of pages contained at least one object that was marked uncacheable by a server, but whose bits did not change across reloads. These statistics remained stable for both immediate reloads and delayed reloads.

Figure 7 lists the MIME types for the newly cacheable objects that RC2 caching discovered. Images were the most common type, both in terms of object count (46.3%) and bytes saved (63.4%). JavaScript files provided the second-largest amount of bytes saved (26.6%).

3.6 Page Load Time

Prior studies of mobile caching found that page load times (PLTs) are only moderately affected by cache hits—warm cache PLTs decrease by only 13% relative to cold cache PLTs [24]. The reason is that a traditional cache stores few objects that reside on the critical path for a page load [17]. Since RC2 improves cache hit rates, RC2 should increase the likelihood that critical path objects will hit in the cache. Thus, RC2 should lower PLTs as well as bandwidth consumption.

Scenario	Median PLT	95th-percentile PLT
Traditional	5.23 seconds	14.66 seconds
RC2	5.27 seconds	14.54 seconds

(a) Cold caches.

Scenario	Median PLT	95th-percentile PLT
Traditional	4.31 seconds	9.95 seconds
RC2	3.62 seconds	7.69 seconds

(b) Warm caches, immediate reload.

Scenario	Median PLT	95th-percentile PLT
Traditional	4.68 seconds	10.83 seconds
RC2	3.70 seconds	5.62 seconds

(c) Warm caches, 8-hour-delayed reload.

Figure 8: Page load times for traditional caching and RC2.

To empirically measure the PLT benefits, we USB-tethered a Nexus 5 phone to a Linux desktop machine. The desktop machine ran an RC2 proxy which treated Mahimahi servers [18] as the origin servers for all web content. The phone/proxy connection was an emulated Verizon LTE link which used packet delivery schedules driven by an empirical trace [26]. The proxy/Mahimahi link used an emulated RTT of 5 ms, and an emulated bandwidth of 25 Mbps. We also evaluated traditional caching by having the phone directly contact the Mahimahi servers over an emulated LTE link.

In all warm cache experiments, we warmed the cache (and RC2 proxy) by simulating a user with a 100-day old browser cache. At the beginning of simulated day 1, the browser cache was empty. During each simulated day, the simulated user loaded 10 pages drawn from a Zipfian distribution over the Alexa Top 500 pages. For each page PLT to examine, we then ensured that the cache had objects from either a recent or an 8-hour-old version of the page. Using this methodology, we could measure the expected size of the per-client data structures kept by a proxy—the final size of such a structure on the 100th simulated day was roughly 636 KB.

Figure 8 shows the PLT results. In the cold cache case, median PLTs are slightly higher (0.76%) with RC2 because of proxy overheads for manipulating per-client data structures and loading pages inside of a headless browser. However, RC2 provides significant PLT reductions in warm cache scenarios; compared to baseline PLTs that used cold caches, RC2 reduces warm cache PLTs by 30.8% for immediate reloads, and 29.3% for 8-hour-delayed reloads.

4 OPEN CHALLENGES AND FUTURE WORK

Web traffic is increasingly shifting from HTTP to HTTPS [15]. The transition is beneficial for security, but detrimental to proxy-based solutions for reducing load times or decreasing bandwidth costs. Compression proxies like Flywheel [1], and RDR proxies like Amazon Silk [3], require access to cleartext HTTP traffic; however, to gain access to that traffic, proxies would have to spoof HTTPS origin servers and break TLS's end-to-end security guarantees. Flywheel and Amazon Silk choose not to break those guarantees, thereby eschewing acceleration for HTTPS traffic. In contrast, Opera Mini does perform man-in-the-middle TLS mediation, such that end-to-end integrity is broken, but both HTTP and HTTPS pages can be

accelerated. The results in Section 3 used Opera Mini-style proxying; RC2's performance would obviously be worse if only HTTP traffic could be proxied. For example, in delayed reload scenarios, RC2 provides 99.1% of the optimal benefits if all traffic can be proxied, but only 81.2% of those benefits if HTTPS traffic must be ignored. RC2 still outperforms traditional caching (which only provides 47.5% of the optimal benefits), but proxy-based web acceleration in general is threatened by the increasing ubiquity of HTTPS. Researchers have begun to investigate secure mechanisms for TLS introspection by middleboxes [16, 22]; further work is needed to apply such techniques to web acceleration proxies.

An RC2 proxy stores a per-client data structure that represents the proxy's view of the client-side browser cache. In our unoptimized RC2 prototype, the data structure for a client with a 100-day-old cache is roughly 636 KB in size. The data structure contains hash values (which are random-looking and thus incompressible), as well as URLs and dates (which are more promising candidates for compression). To maximize proxy scalability, future work should investigate concrete approaches for shrinking per-client data structures.

Using HTML rewriting, an RC2 proxy takes control of a phone's browser cache. We believe that rewriting can enable other proxy-managed optimizations. For example, proxies receive HTTP requests from a large set of phones. By observing cross-request correlations (e.g., "requests for page X are typically followed by requests for page Y"), a proxy can inject `<link>` prefetch tags [25] for Y into X, so that a mobile browser will proactively fetch Y before the local user requests Y.

5 RELATED WORK

Ma et al. examined mobile caching behavior [11], but ignored HTTPS content. Their survey also did not consider cross-site aliasing relationships (as considered in Section 3.6). Additionally, their survey considered a cache hit to be a false hit if, at the time of the cache hit, the server-side version of an object differed from the client-side version retrieved from the cache. This definition is problematic, since sites can define cross-object consistency semantics which Ma et al.'s methodology would flag as leading to false cache hits.

Prior systems have explored client/server protocols for implementing content-based caching. For example, in CZIP [20], a client fetches a page using two HTTP-level RTTs: one to fetch the list of hashes for objects in the page, and another to fetch the raw data for objects that are not client-resident. In Silo [12], a web server inlines all of the external content referenced by an HTML file, and then splits the inlined HTML into chunks. The server returns the chunks to the client, who stores the chunks in DOM storage; in a subsequent request for the HTML, the client indicates which chunks are locally resident, so that the server only has to return the new ones (plus a list of old chunks that are no longer contained in the page's inlined HTML). These prior systems for content-based caching are either incompatible with standard web browsers [20, 21], or cannot track aliasing relationships for the same object across different sites [12]. In contrast, RC2 works on commodity browsers, and can track aliasing relationships across all sites that a client visits.

6 CONCLUSION

Bandwidth is precious on mobile phones. In this paper, we quantify the amount of bandwidth that is needlessly consumed by HTTP fetches that miss in a traditional browser cache, but would have hit in a content-based one. We then propose a new content-based caching scheme, called RC2. RC2 leverages a proxy that rewrites HTML to avoid unnecessary cache misses. Experiments with 500 popular sites demonstrate that RC2 eliminates 99.1% of unnecessary cache misses, while requiring no changes to mobile browsers.

REFERENCES

[1] Victor Agababov, Michael Buettner, Victor Chudnovsky, Mark Cogan, Ben Greenstein, Shane McDaniel, Michael Piatek, Colin Scott, Matt Welsh, and Bolian Yin. 2015. Flywheel: Google's Data Compression Proxy for the Mobile Web. In *Proceedings of NSDI*.

[2] Alexa. 2018. Alexa Top 500 Global Sites. http://www.alexa.com/topsites. (2018).

[3] Amazon. 2018. Silk Browser. http://amazonsilk.wordpress.com/. (2018).

[4] Apple. 2018. https://developer.apple.com/documentation/webkit/wkwebview. (2018).

[5] Google. 2018. Android Developer Reference: WebView. https://developer.android.com/reference/android/webkit/WebView.html. (2018).

[6] Ariya Hidayat. 2018. PhantomJS. http://phantomjs.org/. (2018).

[7] HTTP Archive. 2018. Desktop Trends. http://httparchive.org/interesting.php#bytesperpage. (2018).

[8] HTTP Archive. 2018. Mobile Trends. http://mobile.httparchive.org/interesting.php#bytesperpage. (2018).

[9] Sunghwan Ihm and Vivek Pai. 2011. Towards Understanding Modern Web Traffic. In *Proceedings of IMC*.

[10] Terence Kelly and Jeffrey Mogul. 2002. Aliasing on the World Wide Web: Prevalence and Performance Implications. In *Proceedings of WWW*.

[11] Yun Ma, Xuanzhe Liu, Shuhui Zhang, Ruirui Xiang, Yunxin Liu, and Tao Xie. 2015. Measurement and Analysis of Mobile Web Cache Performance. In *Proceedings of WWW*.

[12] James Mickens. 2010. Silo: Exploiting JavaScript and DOM Storage for Faster Page Loads. In *Proceedings of USENIX WebApps*.

[13] James Mickens, Jeremy Elson, and Jon Howell. 2010. Mugshot: Deterministic Capture and Replay for Javascript Applications. In *Proceedings of NSDI*.

[14] Jeffrey Mogul, Yee Man Chan, and Terence Kelly. 2004. Design, Implementation, and Evaluation of Duplicate Transfer Detection in HTTP. In *Proceedings of NSDI*.

[15] David Naylor, Alessandro Finamore, Ilias Leontiadis, Yan Grunenberger, Marco Mellia, Maurizio Munafò, Konstantina Papagiannaki, and Peter Steenkiste. 2014. The Cost of the "S" in HTTPS. In *Proceedings of the CoNEXT*.

[16] David Naylor, Kyle Schomp, Matteo Varvello, Ilias Leontiadis, Jeremy Blackburn, Diego R. López, Konstantina Papagiannaki, Pablo Rodriguez Rodriguez, and Peter Steenkiste. 2015. Multi-Context TLS (mcTLS): Enabling Secure In-Network Functionality in TLS. In *Proceedings of SIGCOMM*.

[17] Ravi Netravali, Ameesh Goyal, James Mickens, and Hari Balakrishnan. 2016. Polaris: Faster Page Loads Using Fine-grained Dependency Tracking. In *Proceedings of NSDI*.

[18] Ravi Netravali, Anirudh Sivaraman, Somak Das, Ameesh Goyal, Keith Winstein, James Mickens, and Hari Balakrishnan. 2015. Mahimahi: Accurate Record-and-Replay for HTTP. In *Proceedings of USENIX ATC*.

[19] Opera. 2018. Opera Mini. http://www.opera.com/mobile/mini. (2018).

[20] KyoungSoo Park, Sunghwan Ihm, Mic Bowman, and Vivek S. Pai. 2007. Supporting Practical Content-addressable Caching with CZIP Compression. In *Proceedings of USENIX ATC*.

[21] Sean Rhea, Kevin Liang, and Eric Brewer. 2003. Value-Based Web Caching. In *Proceedings of WWW*.

[22] Justine Sherry, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. 2015. Blind-Box: Deep Packet Inspection over Encrypted Traffic. In *Proceedings of SIGCOMM*.

[23] Ashiwan Sivakumar, Shankaranarayanan Puzhavakath Narayanan, Vijay Gopalakrishnan, Seungjoon Lee, Sanjay Rao, and Subhabrata Sen. 2014. PARCEL: Proxy Assisted BRowsing in Cellular Networks for Energy and Latency Reduction. In *Proceedings of CoNEXT*.

[24] Jamshed Vesuna, Colin Scott, Michael Buettner, Michael Piatek, Arvind Krishnamurthy, and Scott Shenker. 2016. Caching Doesn't Improve Mobile Web Performance (Much). In *Proceedings of USENIX ATC*.

[25] W3C. 2017. Resource Hints. https://w3c.github.io/resource-hints/. (May 4, 2017).

[26] Keith Winstein, Anirudh Sivaraman, and Hari Balakrishnan. 2013. Stochastic Forecasts Achieve High Throughput and Low Delay over Cellular Networks. In *Proceedings of NSDI*.