Flywheel: Google's Data Compression Proxy for the Mobile Web

Victor Agababov, Michael Buettner, Victor Chudnovsky, Mark Cogan, Ben Greenstein, Shane McDaniel, and Michael Piatek, *Google, Inc.;* Colin Scott, *University of California, Berkeley;* Matt Welsh and Bolian Yin, *Google, Inc.*

https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/agababov

This paper is included in the Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15).

May 4–6, 2015 • Oakland, CA, USA

ISBN 978-1-931971-218

Flywheel: Google's Data Compression Proxy for the Mobile Web

Victor Agababov[*] Michael Buettner Victor Chudnovsky Mark Cogan Ben Greenstein
Shane McDaniel Michael Piatek Colin Scott[†] Matt Welsh Bolian Yin
Google, Inc. [†]*UC Berkeley*

Abstract

Mobile devices are increasingly the dominant Internet access technology. Nevertheless, high costs, data caps, and throttling are a source of widespread frustration, and a significant barrier to adoption in emerging markets. This paper presents Flywheel, an HTTP proxy service that extends the life of mobile data plans by compressing responses in-flight between origin servers and client browsers. Flywheel is integrated with the Chrome web browser and reduces the size of proxied web pages by 50% for a median user. We report measurement results from millions of users as well as experience gained during three years of operating and evolving the production service at Google.

1 Introduction

This paper describes our experience building and running a mobile web proxy for millions of users supporting billions of requests per day. In the process of developing and deploying this system, we gained a deep understanding of modern mobile web traffic, the challenges of delivering good performance, and a range of policy issues that informed our design.

We focus on mobile devices because they are fast becoming the dominant mode of Internet access. Trends are clear: in many markets around the world, mobile traffic volume already exceeds desktop [23], and double-digit growth rates are typical [32].

Despite these trends, web content is still predominantly designed for desktop browsers, and as such is inefficient for mobile users. This situation is made worse by the high cost of mobile data. In developed markets, data usage caps are a persistent nuisance, requiring users to track and manage consumption to avoid throttling or overage fees. In emerging markets, data access is often priced per-byte at prohibitive cost, consuming up to 25% of a user's total income [18]. In the face of these costs, supporting the continued growth of the mobile Internet and mobile web browsing in particular is our primary motivation.

Although the number of sites that are tuned for mobile devices is growing, there is still a huge opportunity to save users money by compressing web content via a proxy. This paper describes Flywheel, a proxy service integrated into Chrome for Android and iOS that compresses proxied web content by 58% on average (50% median across users). While proxy optimization is an old idea [15, 22, 29, 39, 41] and the optimizations we apply are known, we have gained insights by studying a modern workload for a service deployed at scale. We describe Flywheel from an operational and design perspective, backed by usage data gained from several years of deployment and millions of active users.

Flywheel's data reduction benefits rely on cooperation between the browser and server infrastructure at Google. For example, Chrome has built-in support for the SPDY [11] protocol and the WebP [12] image compression format. Both improve efficiency, yet are rarely used by website operators because they require cumbersome, browser-specific configuration. Rather than waiting for all sites to adopt best practices, Flywheel applies optimizations automatically and universally, transcoding content on-the-fly at Google servers as it is served.

This paper makes two key contributions. First, our experience with Flywheel has given us a deep understanding of the performance issues with proxying the modern mobile web. Although proxy optimization delivers clear-cut benefits for data reduction, its impact on latency is mixed. Measurements of Flywheel's overall performance demonstrate the expected result: compression improves latency. In practice however we find that Flywheel's impact on latency varies significantly depending on the user population and metric of interest. For example, we find that Flywheel decreases load time of large pages but increases load time for small pages.

Our second contribution is a detailed account of the many design tradeoffs and measurement findings that we encountered in the process of developing and deploying Flywheel. While the idea of an optimizing proxy is conceptually simple, our design has evolved continuously in response to deployment experience. For example, we find that middleboxes within mobile carri-

[*] Authors are listed in alphabetical order.

ers are widespread, and often modify HTTP headers in ways that break naïve proxied connections. While use of HTTPS and SPDY would prevent tampering, always encrypting traffic to the proxy is at odds with features such as parental controls enforced by mobile carriers. Perhaps unsurprisingly, addressing these tussles consumes significant engineering effort. We report on the incidence and variety of these tussles, and map them to a clearer picture of mobile web operation with the hope that future system designs will be informed by the practical concerns we have encountered. As far as we know, we are the first to publish a discussion of these tradeoffs.

2 Background

We built Flywheel in response to the practical stumbling blocks of today's mobile web. Ideally, Flywheel would be unnecessary. Mobile data would be cheap, and content providers would be quick to adopt new technologies. Neither is true today.

Mobile Internet usage is large and growing rapidly. The massive growth of mobile Internet traffic has created a tremendous opportunity for automatic optimization. In Asia and Africa, 38% of web page views are performed on mobile devices as of May 2014, a year-over-year increase exceeding 10% [36]. In North America, mobile page loads are 19% of total traffic volume with 8% growth yearly. In February 2014, research firm comScore reported that time spent using the Internet on mobile devices exceeded desktop PCs for the first time in the United States [23]. These trends match our experience at Google. Mobile is increasingly dominant.

Growth in emerging markets is hampered by cost. Emerging markets are growing faster than developed markets. Year-over-year growth in mobile subscriptions is 26% in developing countries compared to 11.5% in developed countries. In Africa, growth exceeds 40% annually [32]. Despite surging popularity, the high cost of mobile access encumbers usage. One survey of 17 countries in sub-Saharan Africa reports that mobile phone spending was 10-26% of individual income in the lower-75% income bracket [18].

Site operators are slow to adopt new technologies. Adapting websites for mobile involves manual and often complex optimizations, and most sites are poorly equipped to make even simple changes. For example, measurements of Flywheel's workload show that 42% of HTML bytes on the web that would benefit from compression are uncompressed, despite GZip being universally supported in modern web browsers [13]. This is in part because GZip is not enabled by default on most web servers, yet only a single-line change to the server configuration is needed. While hosting-providers and CDNs deal with such configuration issues on the behalf of content providers, the pervasive lack of GZip usage indicates that most content providers still do not employ these services.

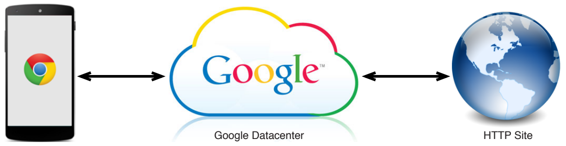

Figure 1: Flywheel sits between devices and origins, automatically optimizing HTTP page loads.

More recent optimizations such as WebP [12] and SPDY [11] have been available for years yet have very low adoption rates. We find that 0.8% of images on the web are encoded in the WebP format, and only 0.9% of sites support SPDY [49].

Users should not have to wait for sites to catch up to best practices. Modern browsers such as Chrome are updated as often as every six weeks, providing a constant stream of new opportunities for optimization that are difficult for web developers to track. Moreover, as mobile devices proliferate, the complexity of optimizing sites to conform to the latest platforms (e.g. high-resolution tablets requiring higher image quality) is a daunting task for all but the most committed site owners.

In sum, it is not surprising that most site owners do not take advantage of all browser- and device-specific optimizations. Just as we do not expect programmers to manually unroll loops, we should not expect site owners to remember to apply an ever-expanding set of optimizations to their sites. We need an optimizing compiler for the web—a service that automatically applies optimizations appropriate for a given platform.

3 Design & Implementation

This section describes Flywheel's design and implementation. The high-level design (depicted in Figure 1) is conceptually simple: Flywheel is an optimizing proxy service. Chrome sends HTTP requests to Flywheel servers running in Google datacenters. These proxy servers fetch, optimize, and serve origin responses. An example optimization is transcoding a large, lossless PNG image into a small, lossy WebP. The remainder of this section describes our goals, the flow of requests and responses, and the optimizations applied in transit. We conclude the section with a discussion of fault tolerance.

3.1 Goals & Constraints

Flywheel's primary goal is to reduce mobile data usage for web traffic. To achieve this goal we must address the practical requirements of integrating with the Chrome browser, used by hundreds of millions of people.

Opt-in. Recognizing that many users are sensitive to the privacy issues of proxying web content through Google's servers, we choose to keep the Flywheel proxy off by default. Users must explicitly enable the service.

Proxy HTTP URLs only. Flywheel applies only to HTTP URLs. HTTPS URLs and page loads from incognito tabs[1] do not use the proxy. While it is technically feasible to proxy through Flywheel in these cases, we are deliberately conservative when given an explicit signal that a request is privacy sensitive.

Maintain transparency for users, network operators, and site owners. Flywheel does not depend on mobile carriers to change their networks or site operators to change their content. Once enabled, Flywheel is transparent to users: websites should look and behave exactly as they would without the proxy in use. This requirement means we must be fairly conservative in terms of the types of optimizations we perform—for example, we cannot modify the DOM of a given page without risking adverse interactions with JavaScript on that page. Further, unavailability of Flywheel service should not prevent users from loading pages.

Comply with standards. All of the optimizations performed by Flywheel must be compliant with modern web standards, including the practical reality of middleboxes that may cache or transform the optimized content. Further, we use standard protocols (SPDY and HTTP) for transferring content in order to ensure that Flywheel can be widely deployed without compatibility issues.

While improving web page load times through Flywheel is desirable, it is not always possible, as we describe later. There is a latency cost to proxying web content through third-party proxies, and although compressing responses tends to reduce load times, this benefit does not always outweigh the increased latency of fetching content through Flywheel servers.

3.2 Client Support: Chrome

Flywheel compression is a feature of the Chrome browser on Android and iOS. Users enable Flywheel in Chrome's settings, which shows a graph of compression over time once enabled.

SPDY (HTTP/2). By default, client connections to Flywheel use the SPDY[2] [11] transport protocol, which multiplexes the transfer of HTTP content over a single TLS connection. SPDY is intended to improve performance as well as security by avoiding the overhead of multiple TCP connections and prioritizing data transfer. For example, HTML and JavaScript are often on the critical path for rendering a page and hence have higher transfer priorities.

For users of Flywheel, SPDY support is universal. Each client application maintains a single SPDY connection to the Flywheel proxy over which multiple HTTP requests are multiplexed. Flywheel in turn translates these to ordinary HTTP transactions with origin servers.

Proxy bypass. A practical reality is that Flywheel cannot proxy all pages on the web. Some sites are simply inaccessible to the Flywheel proxy, such as those behind a corporate intranet or private network. Other sites actively block traffic from Flywheel, for example, due to automated DoS prevention that interprets the volume of traffic from Flywheel IP addresses as an attack.

To avoid unavailability, Flywheel is automatically disabled in these circumstances using a mechanism we call *proxy bypass*. Proxy bypass is implemented using a special HTTP control header that informs the browser to disable the proxy and reload the affected content directly from the origin site. Proxy bypasses are configurable, giving us the ability to disable Flywheel for a set time (e.g., to cover an entire page load) or for just a single URL. This signal also provides us with a convenient load shedding mechanism: we can remotely disable Flywheel for specific clients as needed. We describe the uses of proxy bypass in greater detail in §3.3.5.

HTTP fallback. SPDY is desirable for Flywheel proxy connections due to its performance advantages (§4) and insulation from middlebox interference. However, because SPDY traffic is encrypted, its use can interfere with adult content filtering deployed by mobile carriers, schools, etc. Many mobile carriers also perform selective modification of HTTP request headers from clients in their network, for example to support automatic login to a billing portal site. Although web content filtering can be used as a means of censorship, our goal is not to circumvent such filtering with Flywheel; we wish to be "filter neutral."

We therefore provide a mechanism whereby the connection to the Flywheel proxy can fall back to unencrypted HTTP. This is typically triggered using the proxy bypass mechanism described earlier, although the effect is to switch the proxy connection from SPDY to HTTP, rather than disabling Flywheel entirely.

We also provide a mechanism whereby network operators can disable the use of SPDY Flywheel connections for specific clients in their network [3]. While establishing a SPDY connection, the client makes an unencrypted HTTP request to a well-known URL hosted by Google. If the response contains anything other than an expected string, the client assumes that the URL was blocked by an intermediary and disables use of SPDY for the Flywheel connection. This mechanism is straightforward for carriers to use and allows them to achieve their goals. More complicated approaches that we considered would have required significant integration work between carriers and Google.

In some cases, SPDY must be disabled to avoid triggering bugs in sites. One example we encountered is a

[1] Incognito mode is a Chrome feature that discards cookies, history, and other persistent state for sites visited while it is enabled.

[2] As the HTTP/2 protocol based on SPDY moves to the final stages of standardization, we are migrating from SPDY to HTTP/2.

Technique	Section
HTTP caching	§3.3.4
Multiplexed fetching	§3.3.1
Image transcoding	§3.3.2
GZip compression	§3.3.2
Minification	§3.3.2
Lightweight 404s	§3.3.2
TCP preconnect	§3.3.3
Subresource prefetching	§3.3.3
Header integrity check	§3.3.5
Anomaly detection	§3.3.5

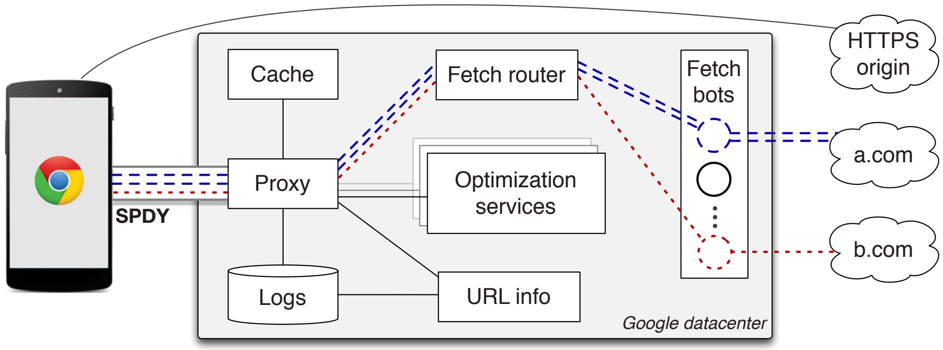

Figure 2: The Flywheel server architecture within a datacenter. Lines indicate bidirectional communication via RPC or HTTP. Each logical component is comprised of replicated, load-balanced tasks for scalability and fault tolerance. The majority of Flywheel code is written in Go, a fact we mention only to dispel any remaining notion that Go is not a robust, production-ready language and runtime environment.

site that uses JavaScript to inspect the response headers of an AJAX request. To improve compression, SPDY sends all headers as lowercase strings, as permitted by the HTTP standard. However, the site's JavaScript code uses a case-sensitive comparison of header names in its logic, leading to an unexpected execution path that ultimately breaks page rendering if SPDY is enabled.

Safe Browsing. Safe Browsing is a feature of Chrome that displays a warning message if a user is about to visit a known phishing or malware site [2]. Because of the overhead of synchronizing Safe Browsing data structures, this feature was disabled for mobile clients and, as of February 2015, is only now being gradually enabled. The main obstacle to its deployment is the tradeoff between bandwidth consumption, power usage, coverage, and timeliness of updates. Ideally, all clients would learn of new bad URLs instantly, but synchronization overhead can be significant, requiring care in managing the tradeoffs. At the Flywheel server, however, many of these tradeoffs do not apply. Flywheel checks all incoming requests against malware and phishing lists, providing an additional layer of protection that is always up-to-date. For requests that match bad URLs, the server signals the client to display a warning.[3]

3.3 Server

The Flywheel server runs in multiple Google datacenters spread across the world. Client connections are directed to a nearby datacenter using DNS-based load balancing; e.g., a client resolving the Flywheel server hostname in Europe is likely to be directed to a datacenter in Europe.

The remainder of this section describes the data reduction and performance optimizations applied at the Flywheel server. Figure 2 provides an overview of our techniques and architecture.

3.3.1 Multiplexed Fetching

The proxy coordinates all aspects of handling a request. The first step is to match incoming requests against URL patterns that should induce a Safe Browsing warning or a proxy bypass. For requests that match, a control response is immediately sent to the client. Otherwise, the request is forwarded via RPC to a separate fetch service that retrieves the resource from the origin.

The fetch service is distinct from the proxy for two reasons. First, a distinct fetch service simplifies management. Many teams at Google need to fetch external websites, and a shared service avoids duplicating subtle logic like rate-limiting and handling untrusted external input. The second benefit to a separate fetch service is improved performance. As shown in Figure 2, the fetch service uses two-level request routing. Request RPCs are load balanced among a pool of fetch routers, which send requests for the same destination to the same fetching bot. Bots are responsible for the actual HTTP transaction with the remote site. Request affinity facilitates TCP connection reuse—requests from multiple users destined for the same origin can be multiplexed over a pool of hot connections rather than having to perform a TCP handshake for each request. Similarly, the fetch service maintains a shared DNS cache, reducing the chance that DNS resolution will delay a request.

3.3.2 Data Reduction

After receiving the HTTP response headers from the fetch service, the proxy makes a decision about how to compress the response based on its content type.[4] These optimizations are straightforward and we describe them briefly. A theme of our experience is that data reduction is the easy part. The bulk of our exposition and engineering effort is dedicated to robustness and performance.

[3] Of course, client checks are still beneficial since Flywheel is not enabled for all users and does not proxy HTTPS and incognito requests.

[4] The proxy respects Cache-Control: No-Transform headers used by origins to inhibit optimization.

Some optimizations are performed by the proxy itself; others are performed by separate pools of processes coordinated via RPC (see Figure 2). Separating optimization from the serving path allows us to load balance and scale services with different workloads independently. For example, an image conversion consumes orders of magnitude more resources than a cache hit, so it makes sense to consolidate image transcoding in a separate service.

Distinct optimization services also improve isolation. For example, because of the subtle bugs often encountered when decoding arbitrary images from the web, we wrap image conversions in a syscall sandbox to guard against vulnerabilities in image decoding libraries. If any optimization fails or a bug causes a crash, the proxy recovers by serving the unmodified response.

Image transcoding. Flywheel transcodes image responses to the WebP format, which provides roughly 30% better compression than JPEG for equivalent visual quality [12]. To further save bytes, we also adjust the WebP quality level according to the device screen size and resolution; e.g. tablets use a higher quality setting than phones. Animated GIF images are transcoded to the animated WebP format. Very rarely, the transcoded WebP image is larger than the original, in which case we serve the original image instead.

Minification. For JavaScript and CSS markup, Flywheel minifies responses by removing unnecessary whitespace and comments. For example, the JavaScript fragment

```
// Issue a warning if the browser
// doesn't support geolocation.
if (!navigator.geolocation) {
  window.alert(
    "Geolocation is not supported.");
}
```

is rewritten (without line breaks) as:

```
if(!navigator.geolocation){window.alert
("Geolocation is not supported.");}
```

GZip. Flywheel compresses all text responses using GZip [26]. This includes CSS, HTML, JavaScript, plain text replies, and HTML. Unlike image optimization, which requires buffering and transcoding the complete response, GZipped responses are streamed to clients. Streaming improves latency, as the browser can begin issuing subresource requests before the HTML download completes.

Lightweight error pages. Many requests from clients result in a 404 error response from the origin, for example, due to a broken link. However, in many cases the 404 error page is not shown to the user. For example, Chrome automatically requests a small preview image called a favicon when navigating to a new site, which often results in a 404. Analyzing Flywheel's workload shows that 88% of page loads result in a 404 error being returned for the favicon request. These error pages can be quite large, averaging 3.2KB—a fair number of "invisible" bytes for each page load lacking a favicon. Flywheel returns a small (68 byte) response body for favicon and apple-touch-icon requests that return a 404 error since the error page is not typically seen by the user.

3.3.3 Preconnect and Prefetch

Preconnect and prefetch are performance optimizations that reduce round trips between the client and origin server. The key observation is that while streaming a response back to the client, the proxy can often predict additional requests that the client will soon make. For example, image, JavaScript and CSS links embedded in HTML will likely be requested after the HTML is delivered. Flywheel parses HTML and CSS responses as they are served in order to discover subresource requests.

Another source of likely subresource requests comes from the URL info service (see Figure 2), which is a database populated by an analysis pipeline that periodically inspects Flywheel traffic logs to determine subresource associations, e.g. requests for `a.com/js` are followed by requests for `b.com/`. This offline analysis complements online parsing of HTML and CSS since it allows Flywheel to learn associations for resources requested by JavaScript executed at the client. We eschew server-side execution of JavaScript because of the comparatively high resource requirements and operational complexity of sandboxing untrusted JavaScript.

Given subresource associations, Flywheel either prefetches the entire object or opens a TCP preconnect to the origin. Which of these is used is determined by a policy intended to balance performance against server overhead. Server overhead comes from wasted preconnects or prefetches that are not used by a subsequent client request. These can arise in case of a client cache hit, a CSS resource for a non-matching media query, or an origin response that is uncacheable. Avoiding these cases would require Flywheel to maintain complete information about the state of the client's cache and cookies. Because of privacy concerns, however, Flywheel does not track or maintain state for individual users, so we have no basis for storing cache entries and cookies.

Flywheel balances the latency benefits of prefetch and preconnect against overhead by issuing a bounded number of prefetches per request for only the CSS, JavaScript, and image references in HTML and only image references in CSS. Preconnects are similarly limited. We track cache utilization and fraction of warm TCP connections to tune these thresholds, a topic we revisit in §4.

3.3.4 HTTP Caching

Flywheel acts as a customized HTTP proxy cache.

Customized entry lookup. Flywheel may store multiple optimized responses for a single URL, for example a transcoded WebP image with two different quality levels—one for phones and another for tablets. Similarly, only some versions of Chrome support WebP animation, so we also need to distinguish cache entries on the basis of supported features. Dispatching logic is shared by both the cache and optimization path in the proxy, so that a cache hit for a particular request corresponds to the appropriate optimized result. The client information is included in each request, e.g., the User-Agent header identifies the Chrome version and device type.

Private external responses. When serving responses over HTTP rather than SPDY, Flywheel must prevent downstream caches from storing optimized results since those caches will not share our custom logic.[5] Downstream proxy caching can break pages, e.g., by serving a transcoded WebP image to a client that does not support the format. To prevent this, we mark all responses transformed by Flywheel as Cache-Control: private, which indicates that the response should not be cached by any downstream proxy but may be cached by the client.

3.3.5 Anomaly Detection

Flywheel employs several mechanisms for improving robustness and availability.

Proxy bypass. Transient unavailability (e.g. network connection errors, software bugs, high server load) may occur along the path between the client, proxy, and origin. In these cases, Flywheel uses the proxy bypass mechanism described earlier to hide such failures from users. Recall that proxy bypass disables the proxy for a short time and causes the affected resources to be loaded directly from the origin site. Proxy bypass is triggered either when an explicit control message is received from the proxy, or when the client detects abnormal conditions. These include:

- *HTTP request loop:* A loop suggests a misconfigured origin or proxy bug. If the loop continues without Flywheel enabled, client-side detection is triggered.
- *Unreachable origin:* DNS or TCP failures at the proxy suggest network-level unavailability, an attempt to access an intranet site, or Google IP ranges being blocked by the origin.
- *Server overload:* The proxy sheds load if needed by issuing bypasses that disable Flywheel at a particular client for several minutes (§4.4).
- *Blacklisted site or resource:* Sites that are known to have correctness problems when fetched via Flywheel are always bypassed, e.g. carrier portals that depend on IP addresses to identify subscribers.
- *Missing control header:* Middleboxes may strip HTTP control headers used by Flywheel if SPDY is disabled. If the proxy observes that such headers are missing from the client request, it sends a bypass to avoid corner cases wherein non-compliant HTTP caches may break page loads.
- *Unproxyable request:* We bypass requests for loopback, .local, or non-fully qualified domains.

Fetch failures. Some fetch errors can be recovered at the server without bypassing, e.g. DNS resolution or TCP connection failures. Simply retrying a fetch often works, recovering what would otherwise have been a bypass.

While retrying a failed fetch often succeeds, it can increase tail latency in the case that an origin is persistently flaky or truly unavailable. To detect these cases, we use an anomaly detection pipeline to automatically detect flaky URLs; i.e., those that have high fetch failure rates. There are thousands of such URLs, making manual blacklisting impractical.

The analysis pipeline runs periodically, inspects traffic logs, and determines URLs that have high fetch failure rates. These URLs are stored in the URL info service, which Flywheel consults upon receiving each request. Flaky URLs are bypassed immediately without waiting for multiple failed retries. To avoid blacklisting a URL forever, Flywheel allows a small fraction of matching requests to proceed to the origin to test if the URL has become available. The analysis pipeline removes an entry from the blacklist provided the failure rate for the URL has dropped sufficiently.

Tamper detection. As described in Section 3.2, Chrome will occasionally fall back to using an HTTP connection to the Flywheel proxy. The need for unencrypted transport is not uncommon; 12% of page loads through Flywheel use HTTP.

Unencrypted transport means that both the Flywheel client and server need to be robust to modifications by third-party middleboxes. For example, we have found that middleboxes may strip our control headers on either the request or response path. Or, they may ignore directives in the Cache-Control header and serve cached Flywheel responses to other users. They may also optimize and cache responses independently of Flywheel. Our experience echoes other studies: transparent middleboxes are common [45, 56, 57].

To provide robustness to middleboxes, Flywheel is defensive at both the client and server, bypassing the proxy upon observing behavior indicating middlebox tampering. Examples include TLS certification validation failures (typical of captive portals) or missing

[5] Responses delivered via SPDY cannot be cached by intermediate proxy caches because of TLS encryption.

Flywheel headers (typical of transparent caches). In practice, checking for these cases has been sufficient to avoid bugs. An overly conservative policy, however, risks eroding data savings, since we need not disable Flywheel in all circumstances. For example, while non-standard middlebox caching risks breaking pages (e.g. serving Flywheel responses to non-Flywheel users), HTTP-compliant caching is mostly benign (e.g. at worst serving images optimized for tablets to non-tablets). Similarly, Flywheel should be disabled in the presence of a captive portal, but only until the user completes the sign-in process. We continue to refine our bypass policies, and this refinement will continue as the behavior of middleboxes evolves.

4 Evaluation

Our evaluation of Flywheel is grounded in measurements and analysis of our production workload comprising millions of users and billions of queries per day. We focus on data reduction, performance, and fault tolerance.

Data in this section is drawn from two sources: (1) Flywheel server traffic logs, which provide fine-grained records of each request, and (2) Chrome *user metrics* reports, which are aggregated distributions of metrics from Chrome users who opt to anonymously share such data with Google.[6]

4.1 Workload

Since Flywheel is an optional feature of Chrome, only a fraction of users have it enabled. Adoption tends to be higher (14-19%, versus 9% worldwide) in emerging countries such as Brazil where mobile data is costly. Although we do not know whether Flywheel has changed user behavior as we hoped in emerging countries, higher adoption rates indicate a perceived benefit.

Access network. Segmented by access network, we find that 78% of page loads are transferred via WiFi, 11% via 3G, 9% via 4G/LTE, and 1% via 2G. Unsurprisingly, the majority of browsing using Flywheel is via WiFi, since the proxy is enabled regardless of the network type the device is using. While the browser could disable Flywheel on WiFi networks, this would eliminate other benefits of Flywheel such as safe browsing. WiFi is prevalent in terms of traffic volume for several reasons: first, it tends to be faster, so users on WiFi generate more page views in less time. A second reason is that tablets are more likely to use WiFi than cellular data.

Traffic mix. Recall that Flywheel does not receive all traffic from the client; HTTPS and incognito page loads are not proxied. For the 28 day period from August 11 through September 8, 2014, we see that 37% of the total bytes downloaded by users (after optimization) with Flywheel enabled are received from the proxy. In comparison, 50% of total received bytes are over HTTPS,[7] and the remainder are incognito requests, bypassed URLs and protocols other than HTTP/HTTPS (e.g. FTP). A notable consequence of this traffic mix is that because our servers only observe a fraction of web traffic, the data reduction observed at Flywheel servers translates into a smaller overall reduction observed at clients.

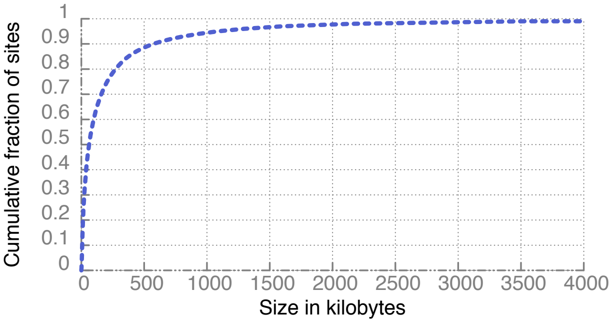

Figure 3: The cumulative distribution of web page size (summation of object sizes) in bytes.

Page footprints. Figure 3 shows the distribution of web page sizes observed in our workload, calculated as the sum of the origin response body bytes for all resources on the page. This distribution is dominated by a small number of larger sites. The median value of 63 KB is dwarfed by the 95th percentile exceeding 1 MB. The tendency for total data *volume* to be dominated by a small number of page loads but the typical page load *time* to be dominated by a large number of very small pages has implications for the latency impact of proxy optimization, a topic we discuss in §4.3.

Video. Flywheel does not currently compress video content, for two reasons. First, most mobile video content is downloaded by native apps rather than the browser. Second, video content embedded in web pages is loaded not by Chrome but by a separate Android process in most cases; hence, video does not pass through the Chrome network stack and cannot be proxied by Flywheel. However, preliminary work on video transcoding using the WebM format suggests that we can expect to achieve 40% data reduction without changing the frame rate or resolution.

4.2 Data reduction

Overall. Excluding request and response headers, Flywheel reduces the size of proxied content by 58% on average. Reduction is computed as the difference between

[6] Chrome users can see a complete list by navigating to about:histograms.

[7] Just 33% of total bytes were received over HTTPS 9 months prior—aggregated between March 11th and April 8th—representing a noteworthy 17 percentage point increase in HTTPS adoption over 9 months.

Type	% of Bytes	Savings	Share of Benefit
Images	74.12%	66.40%	85%
HTML	9.64%	38.43%	6%
JavaScript	9.10%	41.09%	6%
CSS	1.81%	52.10%	2%
Plain text	0.64%	20.49%	.2%
Fonts	0.37%	9.33%	.1%
Other	4.32%	7.76%	1%

Table 1: Resource types and data reduction.

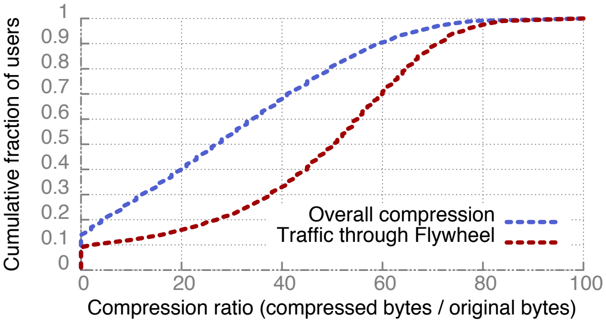

Figure 4: Distribution of overall data reduction across users. The overall reduction is lower than that through Flywheel because we do not proxy HTTPS or incognito traffic.

total incoming and total outgoing bytes at the proxy (excluding bytes served out of cache) divided by total incoming bytes. Table 1 segments traffic and data reduction by content type for a day-long period in August 2014. The 'Other' category includes all other content types, missing content types, and non-200 responses. Note that 'Other' does not include large file transfers: because these are typically binary files that compress poorly, Flywheel sends a proxy bypass upon receiving a request for a large response.

The majority of data reduction benefit comes from transcoding images to WebP, which reduces the size of images by 66% on average. Much of the remaining data reduction benefits come from GZipping uncompressed responses, with larger benefits for CSS and JavaScript due to syntactic minification.

Overall reduction. The data reduction at the server is an upper bound for reduction observed by clients. Recall that HTTPS and incognito page loads, as well as bypassed traffic, are not compressed by Flywheel. Figure 4 quantifies the difference. Across users, the median data reduction for all traffic is 27%, compared to 50% for traffic proxied through Flywheel.

WebP quality. Because images dominate our workload, the aggressiveness of our WebP encoding has a significant influence on our overall data reduction. Our goal is to achieve as much reduction as possible without affecting the perceived quality of the image. To tune the quality, we used the structural similarity index metric (SSIM) [53] to compare the visual similarity of 1500 original vs. compressed images (drawn from a set of 100 popular curated URLs) for different WebP encoder quality values. We picked initial quality values by choosing the knee of the SSIM vs. quality curve.

Of course, the ideal visual quality metric is actual user perception, and there is no substitute for experience. Our experimentation with internal Google users before launch lead us to transcode images at quality 50 for phones and 70 for tablets, which roughly corresponds to an SSIM threshold of ~0.85 and ~0.9, respectively. Prior to tuning, we had received several complaints from internal testers regarding visual artifacts; we have no known reports of users complaining about the current settings.

Lightweight error responses. Flywheel sends a small (68 byte) response body for 404 errors returned for favicon and apple-touch-icon requests. Despite the fact that 404 responses for these images constitute only 0.07% of requests, the full error pages would account for 2% of the total data consumed by Flywheel users. The average 404 page for an apple-touch-icon is 3.3KB, and two such requests are made for every page load. Our lightweight 404 responses eliminate nearly all of this overhead.

Redundancy elimination. Our workload provides a scaffold for evaluating potential data reduction optimizations. Sometimes, this yields a negative result. We conclude our evaluation of data reduction with two such examples.

Many websites do a poor job of setting caching parameters, e.g. using time-based expiration rather than content-based ETags. Others mark resources that do not change for months as uncacheable. The result is that clients unnecessarily download resources that would otherwise be in their cache.

Flywheel could improve data reduction by fixing these configuration errors. For example, we could add a content-based ETag to all responses lacking one, and verify that the origin content is unchanged upon each revalidation request sent by the client.

We evaluate the potential benefits of redundancy elimination using trace replay: we measure the possible increase in data reduction from eliminating all redundant transfers across all client sessions, where a session is defined as the duration between browser restart events.[8] We define a redundant transfer as two response bodies with exactly matching contents (we discuss partial matches next).

The result of redundancy elimination is a modest improvement in data reduction. Overall, 11.5% of the bytes served are redundant after data reduction. Restricting

[8] This typically extends beyond a single foreground session, since Android does not prune tasks except under memory pressure.

consideration to only JavaScript, CSS, and images reduces the benefit to 7.8%. Given this data, we concluded that this opportunity was not worth prioritizing. The assumption of complete redundancy elimination is optimistic, and impact of data reduction by the server would be reduced by the limited fraction of traffic handled by Flywheel at the client. We may return to this technique, but not before pursuing simpler and more fruitful optimizations such as video compression.

Delta Encoding. Most compression techniques focus on reducing data usage for a single response object. If an object is requested multiple times however, the origin may have only made small modifications to the object between the first and subsequent request.

Chrome supports a delta encoding technique known as Shared Dictionary Compression over HTTP (SDCH) designed to leverage such cross-payload redundancy [17] by only sending the deltas between modified objects rather than the whole object. We modified Flywheel to support server-side SDCH functionality (e.g. dictionary construction) on behalf of origin servers. We then duplicated a fraction of user traffic, applied SDCH to the duplicated traffic, and measured what the data reduction would have been if we had served the SDCH responses to users. We found that SDCH only improved data reduction for HTML and plain text objects from ∼35% to ∼41%, equivalent to less than 1% improvement in overall data savings. We therefore opted to not deploy SDCH.

4.3 Performance

We next examine Flywheel's impact on latency. Compared to data reduction, evaluating performance is significantly more complex. The results are mixed: Flywheel improves some performance metrics and degrades others. These results reflect a tradeoff between compression, performance, and operating environment. Table 2 summarizes the performance data underlying these results, which we describe in detail below.

Methodology. Our evaluation answers two main questions: (1) Does Flywheel improve latency compared to loading pages directly? And, (2) how effective are the server-side mechanisms used to improve latency?

We use Chrome user metrics reports (described earlier) to gather client-reported data on page load time (PLT), time to first paint, time to first byte (TTFB), and so on. These anonymous reports are aggregated and can be sliced by a variety of properties, e.g. the client version, device type, country, and network connection. We use server-side logs to measure the effectiveness of performance optimizations such as multiplexed fetching, preconnect, and prefetch. Clients are unaware as to whether or not these optimizations are applied, so their use is not reflected in user reports. Instead, we evaluate these using server traffic logs.

For both client-side and server-side measurements, all comparisons are made relative to a *holdback experiment*, a random sampling of 1% of users for which the proxy is silently disabled, despite the feature being turned on by the user. A holdback group is essential for eliminating sampling bias. For example, comparing the latency observed by users with Flywheel on and off suggests a significant increase in page load time due to Flywheel. However, the holdback experiment shows that the typical page load time of a user who enables Flywheel is higher than the overall population of Chrome users. In retrospect, this is unsurprising—users are more likely to enable Flywheel if they are bandwidth-limited, and Flywheel adoption rates are highest in countries with comparatively high page load times.

Flywheel reduces page load time when pages are large. For most users and most page loads, Flywheel increases page load time. This is reflected in Table 2; cf. rows for 'Holdback' and 'Flywheel'. For the majority of page loads, the increase is modest: the median value increases by 6%. The benefits of compression appear in the tail of the distribution, with the PLT reduction at the 95th percentile being 5%. We attribute this to our production workload: data reduction improves latency when pages are large, and as shown in Figure 3, the distribution of page load sizes is heavily skewed.

Flywheel's performance benefit arises from a combination of individual mechanisms. For example, we find that SPDY provides a median 4% reduction in page load time relative to proxying via HTTPS. Data reduction improves latency, but only for the minority of large pages; e.g. disabling all data reduction optimizations increases median page load time through the proxy by just 2%, but the 95th percentile PLT increases by 7%. On the whole, the contribution of individual mechanisms varies significantly based on characteristics of clients and sites.

Flywheel increases time to first paint. Page load time is an upper bound on latency. But, long before a page is loaded fully, it may display useful content and become interactive. Moreover, displaying a partially rendered page increases perceived responsiveness even if the overall load time is unchanged. To capture a lower bound on page load performance, we consider time to first paint; i.e., the time after loading begins when the browser has enough information to begin painting pixels.

As shown in Table 2, Flywheel increases median time to first paint by 10%. This increase is modest, and drops off in the tail of the distribution. A probable cause of this increase is a corresponding inflation of time to first byte; i.e., the delay between sending the first request in a page load and receiving the first byte of the response. Flywheel increases median TTFB by 40%. Unlike time to first paint, we observe inflated TTFB at all quantiles

Flywheel configuration	*Median*		*70th*		*80th*		*90th*		*95th*		*99th*	
						Page load time quantiles (milliseconds)						
Holdback	2075		3682		5377		9222		14606		39380	
Flywheel	2207	6.4%	3776	2.6%	5374	-0.1%	8951	-2.9%	13889	-4.9%	36130	-8.3%
Holdback (Beta)	2123		3650		5151		8550		13192		32650	
Images only (Beta)	2047	-3.6%	3447	-5.6%	4944	-4.0%	8214	-3.9%	12476	-5.4%	31650	-3.1%
						Japan, page load time quantiles (milliseconds)						
Holdback	1355		2289		3133		4939		7211		14926	
Flywheel	1674	23.5%	2715	18.6%	3647	16.4%	5502	11.4%	7797	8.1%	15927	6.7%
						Time to first byte quantiles per pageload (milliseconds)						
Holdback	185		360		547		999		1688		5064	
Flywheel	259	40.0%	485	34.7%	687	25.6%	1164	16.5%	1903	12.7%	5808	14.7%
						Time to first paint quantiles per pageload (milliseconds)						
Holdback	803		1429		2084		3493		5650		19233	
Flywheel	888	10.6%	1547	8.3%	2194	5.3%	3581	2.5%	5723	1.3%	20374	5.9%

Table 2: Page load time for various Flywheel configurations. Flywheel improves page load time only when pages are large and users are close to a Google data center. All percentages are given relative to the baseline holdback measurement. 'Holdback' refers to a random sampling of 1% of users with Flywheel enabled for whom we disable Flywheel for the browsing session. 'Images only' refers to an experimental configuration wherein only images are proxied through Flywheel. This experiment applies only to Android Chrome Beta users.

Flywheel's latency improvement is not universal. We attribute TTFB inflation primarily to the geographic distribution of Flywheel users. Many users are further from a Google data center than typical web servers, resulting in longer round trips. The extent of TTFB inflation and its relationship to overall latency depends on many factors: latency to Google, from Google to the site, from the user to the site, and the overall benefits of data reduction. Our overall performance data shows that more often than not, the balance of these tradeoffs increases latency.

Usage in Japan provides a concrete example. Flywheel increases median page load time by 23.5% relative to holdback users in Japan, with smaller but still significant increases in the tail. How does this relate to the tradeoffs described above? First, page loads in Japan tend to be fast—34% lower than the overall holdback median. The TTFB inflation is similarly high, but the faster page load time means that the overhead of indirection through Google is proportionally larger. Typical network capacity is also higher in Japan, reducing the benefits of data reduction as round trips represent a larger fraction of overall page load time. While Japan is an extreme case, the overall theme remains: Flywheel's performance benefits are not universal and depend on the interaction of many factors.

At the server, we can further refine the breakdown of TTFB inflation. For Flywheel page loads in the United States over WiFi, for example, median TTFB inflation is 90 milliseconds, of which 60 ms is RTT to Google, 20 ms is RTT from Google to the origin site, and 10 ms is internal routing within Google's network and processing overhead. The precise breakdown of overheads varies by client population, but the dominant factor is typically overhead to reach the nearest Google data center.

Proxying only images improves latency for small pages at the expense of large pages. Given widespread TTFB inflation, the tradeoff between latency and compression is straightforward: Flywheel improves performance when the latency benefit of data reduction outweighs the latency cost of indirect fetching through Google. Trading off data reduction for performance thus requires some notion of *selective proxying*; i.e., sending only some resource loads through Flywheel.

Recall that the majority of Flywheel's data reduction comes from transcoding images to WebP. 74% of bytes passing through Flywheel are images, and 85% of our overall data reduction benefit over a typical day comes from image transcoding (Table 1). This data suggests that proxying requests for images only is likely to eliminate most of Flywheel's latency overhead while retaining most of its data reduction benefit. Implementing this on the client is straightforward: based on surrounding markup, Chrome typically has an expectation of content type; e.g., requests originating from an `<img>` HTML tag are likely to return an image.

The 'Images only' rows in Table 2 show the results of applying this policy as an experiment for Chrome beta users. The results match our intuition. Median page load time is reduced relative to proxying all content. Data reduction is diminished only slightly. On the flip side, the reductions in page load time for the majority of small page loads come at the expense of larger pages. The 99th percentile reduction in PLT from proxying only images is 3% compared with 8% when proxying all content.

Preconnect and prefetch provide modest benefits. We find that although preconnect and prefetch have non-negligible effects on first order metrics (connection reuse and cache hit ratio), they only provide modest 1-2% reductions in median page load time overall. Like other performance metrics, the benefits vary depending on how the data is sliced. The benefits tend to be greater for users with relatively high TTFB inflation. For example, prefetch and preconnect each provide a 2% reduction in PLT for page loads from Japan.

The benefits of preconnect and prefetch are limited by the natural tendency for connection reuse and object caching in our workload. Even without TCP preconnect, for example, 73% of requests from Flywheel are issued over an existing TCP connection. Enabling preconnect increases this fraction to 80%. Similarly, subresource prefetching increases Flywheel's HTTP cache hit rate by 10 percentage points, from 22% to 32%.

Because of their overhead and limited benefits, we have not deployed preconnect or prefetch beyond small experiments because of concerns about the overhead they would impose on origin sites. For example, redundant prefetches increase the number of fetches to origins by 18%. Redundant fetches are caused by two main factors: (1) the prefetched response is already cached at the client, so it will not be requested; or (2) the prefetched response is not cacheable, so we cannot safely use it to satisfy a subsequent client request. We continue to refine our logic for issuing speculative connections and prefetches in an attempt to reduce overhead.

4.4 Fault Tolerance

Our goal is for all Flywheel failures to be transparent to users. If the client cannot contact the Flywheel proxy, or if the proxy cannot fetch and optimize a given URL, our proxy bypass mechanism recovers by transparently requesting resources from the origin server directly (§3.2). Below, we report on the prevalence of bypassed requests, their causes, and mechanisms for improving the precision of proxy bypass.

Bypass causes. A request may be bypassed before it is sent to Flywheel, by Flywheel before it is sent to the origin, or by Flywheel after observing the origin response. We consider each of these cases in turn.

Client-side bypasses are rare, but typically occur due to failure to connect to the proxy. Server-side, 0.89% of requests received by Flywheel result in a bypass being sent to the client. The largest fraction of bypasses (38%) are caused by origin response codes, e.g. 429 indicating a rejected request. Another large fraction are due to fetch errors (28%), e.g. when Flywheel cannot establish a connection to the origin. This could be because the site is down, blocking traffic from the proxy, or because the site is on an intranet or local network not reachable by the proxy. We bypass audio and video files (19% of bypasses) as Flywheel does not currently transcode these response types, as well as large file downloads (0.3%) where we are not able to achieve sufficient data reduction to merit the processing overhead. Requests automatically flagged as problematic by our anomaly detection pipeline constitute 8% of bypasses. We also issue bypasses for blacklisted URLs (5%) for sites with compatibility issues, carrier dependencies, or legal constraints. The remaining 0.25% of bypasses are caused by load shedding; i.e., if an individual Flywheel server becomes unusually slow and accumulates too many in-flight requests, it sends a bypass response.

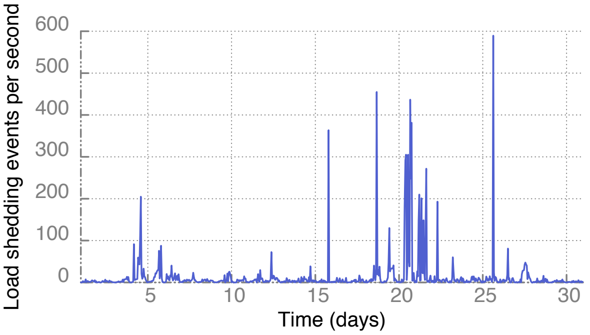

Figure 5: A month-long trace of load shedding events. Transient unavailability is common and is typically resolved without manual intervention.

Load shedding acts as a form of back pressure that provides recovery for many causes of unavailability: congestion, origin slowness, attack traffic, configuration errors, and so on. These events occur frequently, and a stop-gap is essential for smooth operation. A simple load shedding policy of bounding in-flight requests has worked well for us so far. Figure 5 shows a trace of load shedding events collected over one month. Events are typically bursty and short-lived. Crucially, automatic load shedding means that most transient production issues do not require manual intervention for recovery.

Mitigating fetch errors. More than half of bypassed requests are due to fetch errors. Two mechanisms reduce the impact of these errors. First, Flywheel retries failed fetches. Nearly a third of failed fetches succeed after a single retry, and roughly half succeed within 5 retries.

Retrying fetches can impose significant delay if the site is still unreachable after multiple tries. The median latency for fetch errors is ~1 second, with the 90th percentile exceeding 200 seconds. Most fetches that fail after 5 retries are due to DNS lookup or fetch timeouts.

To deal with this, the second mitigation mechanism involves analyzing the server traffic logs periodically to identify URLs that exhibit a high failure rate. If the majority of fetches to a given URL fail, we flag that URL as flaky and push a blacklisting rule into the URL info service to send bypasses for requests to that URL.

Results show that this technique eliminates ~1/3 of all fetch errors. The pipeline achieves a low false positive rate; for 90% of the URLs classified as flaky, at least 70% of the fetches would have failed if they had not been bypassed preemptively. Moreover, a third of bypassed requests would have resulted in a timeout exceeding one second if not bypassed preemptively. Because flaky URLs constitute a small fraction of overall traffic, correcting these errors has limited impact on aggregate page load time. However, exceptionally slow page loads tend to be particularly unpleasant for users, leading us to focus on reducing their impact.

Tamper detection. Our tamper detection mechanisms track cases of middleboxes modifying our HTTP traffic. While we do observe cases of benign tampering, we find that obstructive tampering is rare. For example, over the period of a week, 45 mobile carriers modified our content length header at least once, and 115 carriers appended an extra Via header (indicating the presence of an additional proxy). However, these cases do not significantly hinder user experience; we have only dealt with obstructive tampering on a handful of occasions.

5 Related Work

Optimizing proxy services have received significant attention in the literature, and this work informs our design. This paper differs in two main ways. First, our environment is unique; we focus on the co-design of a modern mobile browser, operating system, and proxy infrastructure. The second difference is scale; we report operational and performance results from millions of users spread across the globe. As far as we know we are the first to report on the incidence and variety of issues encountered by a proxy of this kind.

Web proxies. In the late 90's researchers investigated proxies for improving HTTP performance [22, 39, 41] and accommodating mobile devices [19,29–31,40]. We revisit these ideas in the context of modern optimizations, client platforms, and workloads.

User studies. Others studied the effects of data pricing [21,46], performance [43], and page layout [58] on user behavior. This work reinforces our motivation.

Performance optimizations. At the proxy, we employ known proxy optimizations such as prefetch [16, 35, 41]. By virtue of building on Google infrastructure, we also benefit from transport-level performance optimizations [28, 44]. We do not apply more aggressive optimizations such as 'whole-page' content rewriting [20,34,38] or client offload [48,52], since in our experience even simple changes like CSS import flattening [5] can break some web pages, and our goal is full compatibility with existing pages.

Other work has focused on evaluating existing performance optimizations [16, 24, 27, 47, 50, 51]. Our measurements are derived from a large scale production environment with real user traffic.

Data reduction optimizations. Data reduction techniques beyond those we employ include WiFi offload [14] and differential caching [17,33,37,42,54,59]. Since differential caching only applies to text resources its effect on overall data reduction are limited compared to optimization of images and video, as we quantified in our evaluation of SDCH [17].

Alternate designs. VPN-based compression [4,6,7] offers an alternative to HTTP interposition. The main advantage of VPN-interposition is ubiquity: all traffic can be optimized without modifying applications. But, interposition at the level of raw packets does not lend itself to transport optimizations like SPDY or application-specific mechanisms like proxy bypass, which depends on the client reissuing requests. Flywheel instead integrates with Chrome, which allows us to retain the protocol information required for flexible failure recovery.

Transparent web proxies, which are deployed by many carriers today [25,55–57], present another design option. The main benefit of in-network optimization is that it requires no client modifications whatsoever. But, as with VPNs, interposing at the network level limits options for optimization and failure recovery, and transparent proxies are applicable only within a particular network.

The proxy service with the closest design to ours is Opera Turbo [9]. Although Opera has not published the details of their optimizations or operation, we performed a point comparison of Flywheel and Turbo's data reduction gains, and found that Flywheel provides comparable data reduction.

Other mobile browsers [1, 8, 10] feature more aggressive optimization based on server-side transcoding of entire pages; e.g. Opera Mini rewrites pages into a proprietary format optimized for mobile called OBML [8], and offloads some JavaScript execution to servers rather than clients. While whole-page transcoding can significantly improve data reduction, pages that rely heavily on JavaScript or modern web platform features are often broken by the translation; e.g. touch events are unsupported by Opera Mini [8]. Maintaining an alternative execution environment to support whole-page transcoding is not feasible for Flywheel given our design goal of remaining fully compatible with the modern mobile web.

6 Summary

We have presented Flywheel, a data reduction proxy service that provides an average 58% byte size reduction of HTTP content for millions of users worldwide. Flywheel has been in production use for several years, providing experience regarding the complexity and tradeoffs of operating a data reduction proxy at Internet scale. We find that data reduction is the easy part. The practical realities of operating with geodiverse users, transient failures, and unpredictable middleboxes consume most of our effort, and we report these tradeoffs in the hope of informing future designs.

References

[1] Amazon Silk. http://s3.amazonaws.com/awsdocs/AmazonSilk/latest/silk-dg.pdf.

[2] Chrome SafeBrowse. http://www.google.com/transparencyreport/safebrowsing/.

[3] Data Compresion Proxy Canary URL. https://support.google.com/chrome/answer/3517349?hl=en.

[4] Microsoft Data Sense. http://www.windowsphone.com/en-us/how-to/wp8/connectivity/use-data-sense-to-manage-data-usage.

[5] mod pagespeed. https://developers.google.com/speed/pagespeed/module.

[6] Onavo. http://www.onavo.com/.

[7] Opera Max. http://www.operasoftware.com/products/opera-max.

[8] Opera Mini and Javascript. https://dev.opera.com/articles/opera-mini-and-javascript/.

[9] Opera Turbo. http://www.opera.com/turbo.

[10] Skyfire - A Cloud Based Mobile Optimization Browser. http://www.skyfire.com/operator-solutions/whitepapers.

[11] SPDY Whitepaper. http://www.chromium.org/spdy/spdy-whitepaper.

[12] WebP: A New Image Format For The Web. http://developers.google.com/speed/webp/.

[13] Which Browsers Handle ‘Content-Encoding: gzip‘? http://webmasters.stackexchange.com/questions/22217/.

[14] A. Balasubramanian, R. Mahajan, and A. Venkataramani. Augmenting Mobile 3G with WiFi. MobiSys ’10.

[15] H. Bharadvaj, A. Joshi, and S. Auephanwiriyakul. An Active Transcoding Proxy to Support Mobile Web Access. Reliable Distributed Systems ’98.

[16] C. Bouras, A. Konidaris, and D. Kostoulas. Predictive Prefetching on the Web and its Potential Impact in the Wide Area. WWW ’04.

[17] J. Butler, W.-H. Lee, B. McQuade, and K. Mixter. A Proposal for Shared Dictionary Compression over HTTP. http://lists.w3.org/Archives/Public/ietf-http-wg/2008JulSep/att-0441/Shared_Dictionary_Compression_over_HTTP.pdf.

[18] A. Chabossou, C. Stork, M. Stork, and P. Zahonogo. Mobile Telephony Access and Usage in Africa. African Journal of Information and Communication ’08.

[19] S. Chandra, C. S. Ellis, and A. Vahdat. Application-Level Differentiated Multimedia Web Services using Quality Aware Transcoding. IEEE Selected Areas in Communications ’00.

[20] S. Chava, R. Ennaji, J. Chen, and L. Subramanian. Cost-Aware Mobile Web Browsing. IEEE Pervasive Computing ’12.

[21] M. Chetty, R. Banks, A. Brush, J. Donner, and R. Grinter. You’re Capped: Understanding the Effects of Bandwidth Caps on Broadband Use in the Home. CHI ’12.

[22] C.-H. Chi, J. Deng, and Y.-H. Lim. Compression Proxy Server: Design and Implementation. USITS ’99.

[23] CNN. Mobile Apps Overtake PC Internet Usage in U.S. http://money.cnn.com/2014/02/28/technology/mobile/mobile-apps-internet/.

[24] B. de la Ossa, J. A. Gil, J. Sahuquillo, and A. Pont. Web Prefetch Performance Evaluation in a Real Environment. IFIP ’07.

[25] G. Detal, B. Hesmans, O. Bonaventure, Y. Vanaubel, and B. Donnet. Revealing Middlebox Interference with Tracebox. IMC ’13.

[26] P. Deutsch. GZIP File Format Specification, May ’96. RFC 1952.

[27] J. Erman, V. Gopalakrishnan, R. Jana, and K. Ramakrishnan. Towards a SPDY’ier Mobile Web? CoNEXT ’13.

[28] T. Flach, N. Dukkipati, A. Terzis, B. Raghavan, N. Cardwell, Y. Cheng, A. Jain, S. Hao, E. Katz-Bassett, and R. Govindan. Reducing Web Latency: The Virtue of Gentle Aggression. SIGCOMM ’13.

[29] A. Fox, S. D. Gribble, Y. Chawathe, and E. A. Brewer. Adapting to Network and Client Variation Using Infrastructural Proxies: Lessons and Perspectives. IEEE Personal Communications ’98.

[30] R. Han, P. Bhagwat, R. LaMaire, T. Mummert, V. Perret, and J. Rubas. Dynamic Adaptation in an Image Transcoding Proxy for Mobile Web Browsing. IEEE Personal Communications ’98.

[31] B. C. Housel and D. B. Lindquist. WebExpress: A System for Optimizing Web Browsing in a Wireless Environment. MobiCom '96.

[32] International Telecommunications Union. The World in 2014: ICT Facts and Figures. http://www.itu.int/en/ITU-D/Statistics/Documents/facts/ICTFactsFigures2014-e.pdf.

[33] U. Irmak and T. Suel. Hierarchical Substring Caching for Efficient Content Distribution to Low-Bandwidth Clients. WWW '05.

[34] B. Livshits and E. Kiciman. Doloto: Code Splitting for Network-Bound Web 2.0 Applications. SIGSOFT/FSE '08.

[35] D. Lymberopoulos, O. Riva, K. Strauss, A. Mittal, and A. Ntoulas. PocketWeb: Instant Web Browsing for Mobile Devices. SIGARCH '12.

[36] M. Meeker. Internet Trends 2014. http://pathwaypr.com/must-read-mark-meekers-2014-internet-trends.

[37] L. A. Meyerovich and R. Bodik. Fast and Parallel Webpage Layout. WWW '10.

[38] J. Mickens. Silo: Exploiting JavaScript and DOM Storage for Faster Page Loads. WebApps '10.

[39] J. C. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy. Potential Benefits of Delta Encoding and Data Compression for HTTP. SIGCOMM '97.

[40] B. D. Noble and M. Satyanarayanan. Experience with Adaptive Mobile Applications in Odyssey. Mobile Networks and Applications '99.

[41] V. N. Padmanabhan and J. C. Mogul. Using Predictive Prefetching to Improve World Wide Web Latency. SIGCOMM '96.

[42] K. Park, S. Ihm, M. Bowman, and V. S. Pai. Supporting Practical Content-Addressable Caching with CZIP Compression. ATC '07.

[43] A. Patro, S. Rayanchu, M. Griepentrog, Y. Ma, and S. Banerjee. Capturing Mobile Experience in the Wild: a Tale of Two Apps. ENET '13.

[44] S. Radhakrishnan, Y. Cheng, J. Chu, A. Jain, and B. Raghavan. TCP Fast Open. CoNEXT '11.

[45] C. Reis, S. D. Gribble, T. Kohno, and N. C. Weaver. Detecting In-Flight Page Changes with Web Tripwires. NSDI '08.

[46] N. Sambasivan, P. Lee, G. Hecht, P. M. Aoki, M.-I. Carrera, J. Chen, D. P. Cohn, P. Kruskall, E. Wetchler, M. Youssefmir, et al. Chale, How Much It Cost to Browse?: Results From a Mobile Data Price Transparency Trial in Ghana. ICTD '13.

[47] S. Savage, T. Anderson, A. Aggarwal, D. Becker, N. Cardwell, A. Collins, E. Hoffman, J. Snell, A. Vahdat, G. Voelker, et al. Detour: Informed Internet Routing and Transport. IEEE Micro '99.

[48] A. Sivakumar, V. Gopalakrishnan, S. Lee, and S. Rao. Cloud is Not a Silver Bullet: A Case Study of Cloud-based Mobile Browsing. HotMobile '14.

[49] W3Techs. Usage of SPDY for Websites. http://w3techs.com/technologies/details/ce-spdy/all/all.

[50] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. Demystify Page Load Performance with WProf. NSDI '13.

[51] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. How speedy is SPDY? NSDI '14.

[52] X. S. Wang, H. Shen, and D. Wetherall. Accelerating the Mobile Web with Selective Offloading. MCC '13.

[53] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. Image Quality Assessment: From Error Visibility to Structural Similarity. IEEE Image Processing '04.

[54] Z. Wang, F. X. Lin, L. Zhong, and M. Chishtie. How far can client-only solutions go for mobile browser speed? WWW '12.

[55] Z. Wang, Z. Qian, Q. Xu, Z. Mao, and M. Zhang. An untold story of middleboxes in cellular networks. SIGCOMM '11.

[56] N. Weaver, C. Kreibich, M. Dam, and V. Paxson. Here Be Web Proxies. PAM '14.

[57] X. Xu, Y. Jiang, E. Katz-Bassett, D. Choffnes, and R. Govindan. Investigating Transparent Web Proxies in Cellular Networks. Technical Report 007-90818, University of Southern California.

[58] D. Zhang. Web Content Adaptation for Mobile Handheld Devices. CACM '07.

[59] K. Zhang, L. Wang, A. Pan, and B. B. Zhu. Smart Caching for Web Browsers. WWW '10.