Technical Report
Number: NU-EECS-16-10

June, 2016

Oak: User-Targeted Web Performance

Marcel Flores, Alexander Wenzel, Aleksandar Kuzmanovic

Abstract

Web performance has long proved to be one of the most sought after and difficult to achieve components for the web. With Oak, we address client performance on the individual level, based on data provided directly by each user. Oak provides an automated mechanism by which sites are able to replace resources with those provided by a better performing alternative service for a particular user. We evaluate Oak on experimental and popular existing webpages, and demonstrate its effectiveness.

Keywords

User performance, web performance, HTTP

Oak: User-Targeted Web Performance

Marcel Flores
Northwestern University

Alex Wenzel
Northwestern University

Aleksandar Kuzmanovic
Northwestern University

Abstract

Web performance has long proved to be one of the most sought after and difficult to achieve components for the web. Since the inception of the modern web infrastructure, the situation has been growing in complexity, adding remote hosts and objects, providing everything from computation infrastructure, content distribution capability, and targeted advertising. While many of these components provide improvements for some users, the complexity of the Internet often leaves other users suffering from poor performance. Worse still, much of this poor external performance is hidden from site operators.

With Oak, we propose a system which addresses client performance on the *individual level*, based on data provided directly by each user. Based on this user-reported performance, Oak determines which components of a page are under-performing. Oak further provides an automated mechanism by which sites are able to replace resources with those provided by a better performing alternative service for a particular user. In this work, we demonstrate the prevalence of under-performing services on the web, finding that over 60% of the Alexa Top 500 have at least one under-preforming server. We further evaluate Oak on experimental and popular existing webpages, and demonstrate its effectiveness in making decisions in existing environments and with a distributed user base.

1. INTRODUCTION

Achieving good web performance remains one of the key challenges in the modern Internet. Users want sites to perform quickly for a smooth user experience, and providers want sites to perform well for users to ensure customer satisfaction. Indeed, numerous studies have demonstrated the importance of a rapidly functioning website in cases of e-commerce customer return [6, 22, 28, 29].

This task has been made particularly challenging by the rise in complexity of modern websites. Modern pages load more than simple HTML files from a single server, instead fetching scripts, videos, and images from numerous sources across the Internet [3, 7]. This complexity has come with a cost, as the more complex a site becomes, the more likely it is to encounter challenges with that complexity [33, 34].

Worse still, many of these components are served by third parties, obfuscating their performance from original site operators. In this setting many of the classic approaches for obtaining performance information [1] are not able to provide a complete picture of the user's network performance, complicating auditing and debugging procedures. This prevents site operators from understanding what clients are experiencing in the wild, leaving operators in the dark on the performance of portions of their own sites which are hosted externally.

The details of web performance have been a common consideration in research [7, 8, 11, 20, 30, 33]. Given the importance of web performance to commercial interests and its use in everyday life, it is no surprise that it has proved to be of such interest. However, numerous works have demonstrated that much of the current web is problematic: regularly encountering performance and compatibility issues [7, 8]. A significant body of work has proposed measurement-based solutions [11, 20, 30, 33]. However, such solutions largely address complications in the loading and execution of pages (*e.g.*, load and execution dependencies) from the client perspective. While providing important insights and solutions, they do not generally address a site's management of external resources.

Existing work has examined specific components of web resources, such as the selection of cloud providers [38]. Some have considered post-facto analysis, seeking to provide generalized recommendations to improve future deployments [20, 33, 34]. Others still have sought to provide real-time solutions, but focused on maximizing the performance of existing resources from a fixed set of providers [8].

In this paper, we present Oak, a system which provides insight into the client-observed performance of external resources and provides real-time changes to a site to adapt to the needs of each individual user. In particular, Oak employs a user-profile scheme, to monitor the performance of external objects for each user. This is made possible by direct communication with the client in which the client provides a performance report directly to Oak. Oak then determines if future requests from the client should be served with content from an alternative external server or provider in place of the default.

By employing direct performance information from clients, Oak is able to understand precisely the performance observed by that client, rather than relying on external services or measurements. Furthermore, Oak is able to address performance challenges which may be unique to that user, for example network blind-spots by third party providers, or localized network anomalies. Much in the same way that ad networks provide users with a customized experience that reflects user interest, Oak provides users with a customized version of a page which connects them with known-best providers. Contrary to ad networks, Oak monitors a user's *performance*, rather than their behavior. Since Oak manages outgoing pages on an individual level, it is able to address these issues, providing users with pages which behave best specifically for that user.

Oak chooses providers by way of operator specified *rules*. These rules are designed to allow operators to replace segments of a page which represent abstract objects, *i.e.*, references to an external domain, such as an externally provided advertisement. Operators are then able to express an alternate behavior, such as excluding the object entirely in cases of non-performance, or providing an alternative with a different external provider. In this way, Oak returns control of a site with many external components to the origin server, rather than further outsourcing the task to other services. The origin now has the power to control from where each client loads information based on data directly from the client.

Our chief contributions are the following:

- We introduce user-targeted performance as a new approach for websites to operate with external resources.
- A mechanism for site operators to gain insight into, otherwise hidden, external object performance via direct client performance reports.
- An approach for detecting performance outliers which considers the median absolute deviation as its mile-marker for performance, offering a robust and versatile metric.
- A rule specification mechanism which allows operators to specify portions of pages that can be replaced, and the corresponding alternative content.
- A system which detects the cause-and-effect associated with the aforementioned rules, determining which portions of a page may be responsible for connecting to particular outliers.

We present the design and implementation of the above system, and demonstrate that it is able to correctly detect poorly performing servers and thereby provide significant gains, reducing the median page load time in experimental scenarios on the Internet. In a further evaluation, we consider a prototype implementation of Oak on versions of existing webpages, demonstrating its ability to provide meaningful improvements on current pages, offering faster object downloads in over 80% of considered cases.

This paper is structured as follows: in Section 2 we consider the current state of affairs on the Web, and demonstrate the prevalence of sites with a large number of external objects, many of which perform poorly. We then consider the current approaches in Section 3. In Section 4 we present a careful look at the components of Oak, and in Section 5 we demonstrate its effectiveness in the real world. In Section 6 we discuss our findings as well as potential alternative design components, and we conclude in Section 7.

2. MOTIVATION

Modern web pages have grown increasingly complex, relying on numerous web objects, including JavaScript, CSS, and HTML5 components. From a networking perspective, much of this additional complexity comes in the form of an increasing number of remote hosts. Specifically, when a page is loaded, significant portions of a page originate from remote servers. This remote hosting means that the client must perform fresh DNS look-ups, create new connections, and request the corresponding objects. While many of these services come in the form of CDNs, or other performance enhancing services, they complicate the work done on the user end, and sites are left without a mechanism for directly auditing user performance. Furthermore, many of these services are available from multiple providers, presenting an opportunity for site operators to "shop" for the best performance.

To quantify the frequency of these remote components, we performed a measurement study of the Alexa Top 500 [27]. Our first measurement considered the fraction of components on the page which loaded from outside hosts. We do not consider sub-domains of the original domain to be outside hosts.

Figure 1 presents a CDF of the fraction of external hosts. We see that in the median case, 75% of the objects loaded from a page come from external hosts. This is likely the result of the wide use of CDNs, image and video hosting services, and externally served ads. While they certainly increase the complexity, it is not, however, obvious that the increase of external objects results in slower page loads, or that particular objects are

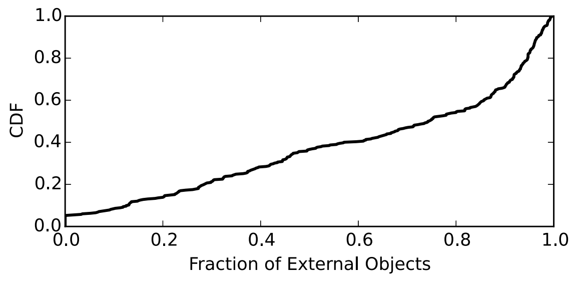


Figure 1: CDF of fraction of objects with non-origin hostnames from the Alexa Top 500.

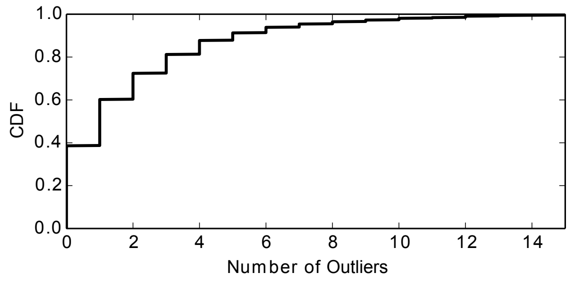


Figure 2: CDF of the number of outliers for the Alexa Top 500, from 25 vantage points.

loading poorly. However in previous work [7], the number of external hosts has been found to correlate with high page load times.

To understand the specific user-experienced behaviors encountered as a result of this added complexity we conduct a further analysis. Here, we again load the Alexa Top 500 from 25 vantage points distributed around the world. For each page, we consider the relative performance of objects, labeling performance outliers using a method we will discuss more fully in Section 4.2.1.

Figure 2 shows that over 60% of sites in this set feature at least a single performance outlier, and 20% of sites feature at least 4 such outliers. Such a figure is not surprising given the large number of external servers that sites contact: it becomes quite likely that at least one of them will be under-performing.

Oak is able to offer visibility of such outliers and allows the the creation of user-targeted pages which sidestep poorly performing external hosts. By basing the provider decision on observed performance, Oak is further agnostic to the cause of such poor performance. If a server is slow to deliver an object relative to others seen by the client, Oak will attempt to switch providers if possible. Oak can therefore manage issues caused by everything from short term congestion to long-term network misconfiguration. Since many site operators already contract with multiple CDNs and take advantage of numerous external services [3], the potential burden of finding a replacement is lowered.

Site	Category
facebook.com	Social Networking
stats.g.doubleclick.net	Ads/Analytics
sp.analytics.yahoo.com	Ads/Analytics
s-static.ak.facebook.com	Social Networking
analytics.twitter.com	Social Networking
counter.yadro.ru	Ads/Analytics
www.dsply.com	Analytics
d31qbv1cthcecs.cloudfront.net	Analytics
rtb-ap.vizury.com	Ads/Analytics
ib.adnxs.com	Ads/Analytics

Table 1: The most frequently seen outliers and their categories. Advertisements, social networking, and analytics dominate.

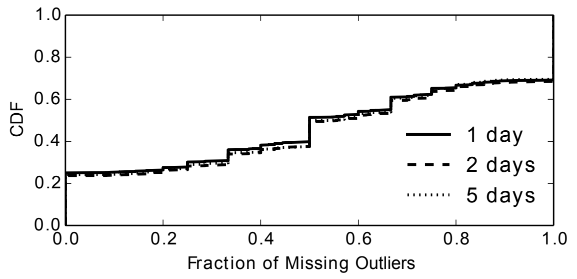


Figure 3: Fraction of outliers which vanished after varying intervals.

2.1 Outlier Characterization

Table 1 shows the most frequently seen outliers across all our measurements in Figure 2. We further provide a determination of the category of the resources fetched from that domain. External advertising, analytics, and social networking components appear to dominate these outliers. Specifically, outliers are characterized by being external additions to sites, rather than just externally hosted assets, increasing their potential for flexibility.

We further wish to understand the potential improvement that Oak may have on a site's operation. In the case of dynamic pages, providers may change overtime, and therefore replacing certain components may have long term effects, while others will offer primarily short term gains. To this end, we consider the consistency of outliers, that is, for how long a given server appears as an outlier for a particular client.

Figure 3 shows the fraction of outliers which changed over time, relative to the results shown in Figure 2, considering 1 day, 2 days, and 5 days. In the first day, there is a significant drop off, with 52% of outliers changing after a single day in the median case. However, on subsequent days the set of re-occurring outliers remains consistent, remaining nearly unaltered after 5 days. Therefore, while some performance outliers are ephemeral, vanishing over short time scales and likely the result of temporary network conditions, about half of them are consistent, appearing reliably. This suggests that Oak must be prepared to handle both types of performance deviation.

It is important to understand that the presence of such outliers does not *necessarily* mean that those outliers are on the critical path in a site's loading. As such, they may not directly increase the page load time. However, the objects loaded from outliers are loading slowly, suggesting components of a site, which the site operator saw fit to include in the first place, are performing poorly. This potentially diminishes their value and damages the user experience. Moreover, the performance of these outliers is hidden from the site operators, making them difficult to detect and address in traditional settings.

3. RELATED WORK

Web performance has long been a concern at the forefront of Internet research. Individuals want their Internet experience to proceed smoothly and quickly and businesses and commercial interests aim to make their sites appealing to consumers [6, 29]. However, the complex nature of this challenge has increased significantly since the dawn of the web, with the vast majority of sites pulling from many servers, generating dynamic content on the fly, aided by Javascript and other web scripting technologies [7, 20, 30, 33]. Others have explored the performance of web systems and various protocol upgrades and environments [12, 34, 36]. Oak builds on many of these observations, and seeks to incorporate aspects of these measurements into its live client-based observations. Many of these works have further explored the nature of JavaScript loading and execution dependencies. Oak focuses on a potentially orthogonal problems and instead is only concerned with which components of a page caused connections to particular remote servers.

Various solutions have been proposed to try and reduce the latency associated with web access, including object prefetching [9, 13, 25, 26], web page parallelization [23, 24], higher granularity cacheing [35], and redundancy [10, 32]. However these approaches generally propose changes to client behavior, rather than adapting the behavior of the service to observed performance. In Costlo [39], the authors further explored the most effective combinations of redundancy, but don't incorporate past client feedback. With Oak, servers actively adapt to communicated client performance, which could likely be augmented with many of the above systems and approaches.

Other systems have explored how to choose between cloud providers, which may overlap with choosing between components on a page. Measurement studies have been performed on cloud services and the performance of different workloads [5, 16, 19]. Rather than provide static analysis, Oak aims to measure and apply changes to problematic providers in real time. Additionally systems which use multiple cloud systems for security, performance, and configuration benefits have been developed [2, 4, 17, 21]. Conductor [37] and SPANstore [38] provide systems which balance cost and latency when choosing cloud providers. C3 [31] reduces tail latency in a databases running on EC2 by incorporating server feedback about current load. C3 [15] presents a deployed system for optimizing CDN selection for streaming video. Oak is designed to be general, considering live client performance on all aspects of a page, such as advertising and general CDN performance, making it fundamentally different from CDN brokering, replica selection, and application specific techniques.

Another approach to improve user experience is to alter the way in which the site loads. In Klotski [8], users specify portions of a site to prioritize by way of utility functions which describe the important portions of the page, and the system attempts to deliver them within a time budget. While similar in its approach to altering the download behavior, Oak does so in response to client feedback, and directly from the server itself, altering where content is downloaded from, rather than its ordering.

Some have considered web performance a network challenge, asking how the networking side of web interactions could be improved to push through more data, faster [11, 18]. Others have pursued deeper changes to the lower layers of the Internet to try and optimize web performance [14]. Oak is intended to be an application layer change, enabling additional communication between clients and servers, and is entirely compatible with such improvements.

4. OAK

In this section, we present the design of Oak, exploring the details of how Oak receive performance information from individual users, how this information can be used by Oak to activate rules which modify the pages, and how further insights can be drawn from this data.

Oak consists of two main components: a server and a client. The server operates side-by-side a site's web server, modifying outgoing pages according to decisions made based on client reported performance and a set of operator-determined actions. The server is further responsible for processing incoming client data and recording the resulting changes. The client runs on end user machines, likely as a browser modification, is responsible for measuring the performance when loading pages and objects, and reporting these measurements back to

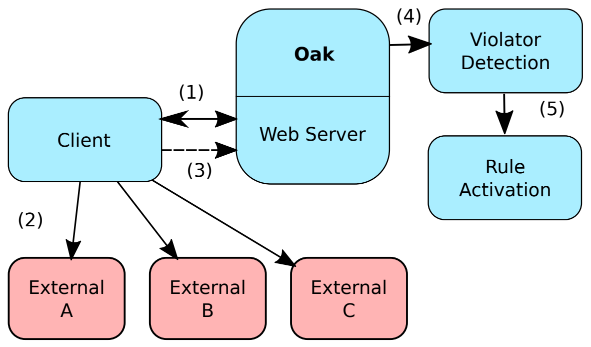


Figure 4: The Oak page performance analysis interaction.

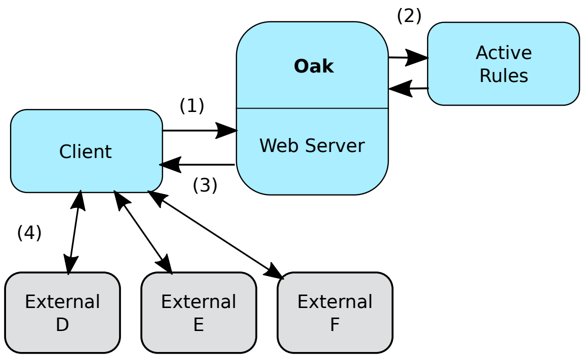


Figure 5: After Oak has detected violators, it provides alternative external servers.

the server.

Oak performs two primary actions: *performance analysis*, which processes incoming performance data and *page modification*, which alters outgoing pages for each individual. We now consider each of these interactions in greater depth.

Performance Analysis Figure 4 presents an overview of this process. First, the client loads the default page from the Oak enabled server and the server responds with the default version of the requested page and an identifying cookie (1). As a result, the client will likely fetch content from a number of external services (2). Since the client is Oak enabled, it generates a *performance report*, which it sends back to the server via HTTP POST, along with its identifying cookie, allowing the server connect its performance with the particular client (3). This report contains information on which external servers the client communicated with, the size of the objects loaded from each of those servers, and download times for each loaded object.

When the Oak Server receives this report, it analyzes the performance, and determines which, if any, servers were operating outside of acceptable bounds (4), a process which we discuss in detail in Section 4.2.1. Any such servers are labeled as *violators*. These violators are then checked against a set of operator specified *rules*, activating any rules which pertain to objects on the violating servers (5). We discuss the specification and use of the rules in greater detail in Section 4.1. We discuss the process of rule activation in Section 4.2.2.

Page Modification Figure 5 outlines the page modification process. When a request for a page (with an identifying cookie) arrives at the web server (1), the HTTP response to the user's request is further processed by Oak. In particular, Oak considers a set of currently active rules for that user (2). These rules alter the page so as to change the loading of particular external resources, either by eliminating them from the page, or providing an alternate server or provider. Next the rule-modified page is delivered to the client (3), and the client contacts external servers (4), which have been selected as alternatives to the previous external servers.

Both of these processes are performed at the *user level*. Each client submits its own performance information, which is then considered against its own history. Rules are then activated on a per-client basis, meaning that outgoing pages are modified based on user-perceived performance. This focus on client level behavior allows Oak to address performance issues that may not be visible on the aggregate, and make decisions that best benefit each individual client.

4.1 The Rules

Here, we present a detailed look at the rules Oak employs to act on operator specified alternatives. The core idea is that the rules allow for the abstract description of resources and their potential alternatives. By activating a rule, Oak can prevent a client from connecting to a server with known poor performance, and instead connect to a faster server or provider. For example, if a rule specifies an advertisement from a particular provider with which a client sees poor performance, it can switch to an alternate server or provider. Since these rules will potentially alter the appearance of a page, as well as the set of hosts and third parties that clients contact, they must be generated directly be operators and cannot be generated automatically.

In order for Oak to alter the operation of a site in a meaningful way, it performs actions specified by a set of operator defined *rules*. These rules consist of: A rule type, a block of text representing a default object, a block of text representing an alternative object, a time to live, a scope, and a potential list of sub-rules. For each user, each rule can then be *activated*. The result of the activation, and how the rule is activated, depends on the type of rules. The time to live indicates how long the rule should remain activated before it is deactivated, and the scope is a path or regular expression which indicates to which pages within a site a rule should be applied.

Oak allows for three types of rules. The first, Type 1, indicates that the block of text representing the default object should be removed if the rule is activated. No alternative object is necessary for Type 1 rules. A Type 2 rule indicates that the specified object can be replaced with the same object at an alternative source, specified by the alternative object text. Oak will simply replace occurrences of the default object text with the alternative object text. Finally, Type 3 indicates that the default object can be replaced with a non-identical object specified by the alternative object text.

By specifying alternatives to Oak using this format, operators authoring rules are able to easily single out both directly included objects (*e.g.*, <img> tags), or larger more abstract objects, such as inline scripts, or even collections of objects that have some abstract form (*e.g.*, many scripts and objects together). We discuss the specifics of how rules are activated and applied in Section 4.2.2.

Rules may also load *sub-rules*. These rules are simple replacements which occur only if the parent rule is activated. These allow operators to specify potentially more elaborate changes to a page in the event of the initial rule activating, without requiring the trigger of more full-fledged rules.

As an example, consider the following rule, which replaces an externally included JavaScript in a script tag with the same JavaScript at an alternative source:

```
(2, #Replacement Type
 "<script src="http://s1.com/jquery.js">",
 "<script src="http://s2.net/jquery.js">",
 0, # Never Expire
 *) # Site wide
```

This rule is given a rule type 2, as the alternative provides an identical object. The replaced blocks replace script tags which refer to an alternate server. In this example, we'd like our rule to be permanent, and therefore set a TTL of 0. Finally, as this particular object is used site wide, we set the scope to be the entire site, allowing Oak to direct the user to a better server for all sub pages. While the example demonstrates a simple tag replacement, the rules could describe much larger objects, for example in-lined JavaScript, or a number of tags.

4.2 Performance Analysis

When a user submits performance information, Oak must process this information in a way that allows it to meaningfully measure the performance of external providers. This task is made challenging by the large amount of noise that HTTP servers regularly demonstrate, as well as the variation in file size, and therefore the relative cost of overhead.

To develop a sane measure in such an environment, Oak begins by grouping all objects by the IP address to which the client ultimately connected, keeping track of all related domain names. We then consider the average time for small objects, and the average throughput for large objects. Small objects are defined to be any object less than 50 KB. We measure the average time to download each of these small objects. For larger objects, in excess of 50 KB, we consider the throughput achieved in downloading each object and then compute the average throughput for all objects collected from that IP.

In this way, Oak is able to generate a server-oriented report of the performance a client achieved, considering the average performance of small and large objects for each. These reports make no decisions on what objects may need to be acted on, but instead stores the raw information about the observed performance.

4.2.1 Violator Detection

Next, using the provided report, Oak performs a page analysis. Specifically, it will determine which of the external servers it contacted are under-performing relative to the others, as seen by the client. Since clients may be widely distributed over networks and geography, it is critical that this determination be made in a way that is specific for this user, and relative to the performance of other servers. For example, users on narrow-bandwidth long-haul links will likely see low performance no matter which servers they are communicating with, and Oak need not waste its time with such cases.

To this end, we compute the Median Absolute Deviation (MAD) for the small objects timings and the large object throughputs. The MAD gives the median value of the deviation from the median of a population, providing a measure of variance that is less effected by outliers than a standard deviation. We compute the MAD for the loading of a site with object times (or throughputs) x_i as

$$\text{MAD} = \text{median}_i(|x_i - \text{median}_j(x_j)|).$$

We then label all servers whose performance was worse than the median (*i.e.*, longer time, lower throughput) by more than twice the MAD as being potential violators. Formally, the server x_i is a violator if

$$\text{time}(x_i) > \text{median}_{time}(x_i) + 2 * \text{MAD}_{time},$$

or

$$\text{tput}(x_i) < \text{median}_{tput}(x_i) - 2 * \text{MAD}_{tput}.$$

In the event that a server has both small and large objects, a violation of either type will result in the server being labeled as a violator. We explore the sensitivity of such a criteria in Section 5.

4.2.2 Rule Activation

Next, Oak must determine if the client-reported performance warrants the activation of an operator speci-

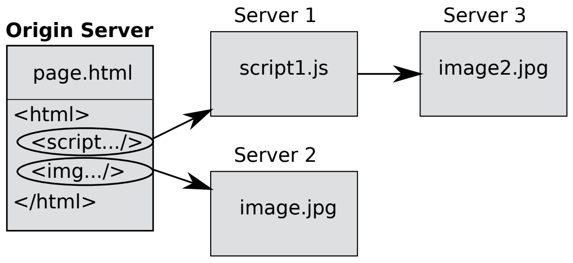


Figure 6: A simple example of connection dependence. The client must contact each of the downstream servers for some piece of content.

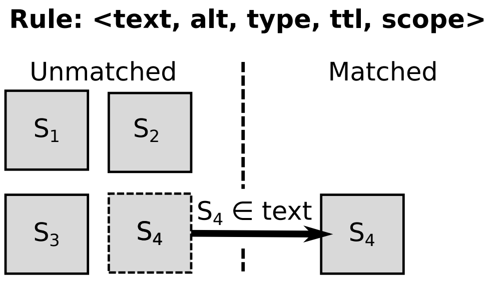


Figure 7: If a server matches a rule, that rule is then activated.

fied rule. In particular, this means that Oak must determine which specified rules should be activated for the current user based on the performance of individual servers.

To this end, Oak considers each of the servers which it determined to be in the set of violators. Then for each rule, the rule is activated for that user if any of the following conditions are met:

- Did the rule contain a reference to an explicit object hosted on a domain that resolved to the violating server?
- Did traffic from the violating server include any domain names which appear in the default object text of the rule?
- Did the rule contain external JavaScript which satisfies either of the above?

We now consider a detailed breakdown of the above conditions. It is important to note that Oak is not concerned with the execution or loading dependencies as previous systems have been [8, 20, 33, 34]. Instead, Oak simply needs to know if a block on a page (*i.e.*, a rule) caused the connection to an external server, a weaker condition, which we call a *connection dependency*.

Figure 6 gives a simple example of connection dependence. In the example, a user loads *page.html* from the origin server, which includes an external script, which it then loads from Server 1. This script, in turn, causes the loading of another object, *image2.jpg*. As described, Oak isn't concerned with any execution, ordering, or load dependencies: Oak need only determine that the original script tag on the HTML page resulted in the connections to Server 1 and 3.

Direct Inclusion We refer to the first two of the above conditions as *direct inclusion*. In particular, these are objects which directly appear on a page that we can tie to a violating server. The first condition scans the rule for *src* attributes in HTML tags. If it finds a source which matches a domain that lead to a violating server, we know that tag lead to the loading of an object from that server, and the rule should be activated.

In addition to *src* attributes, a page may include inline scripts which refer to external objects. These scripts may populate existing portions of the page's DOM tree and otherwise load objects from external sources. However, these scripts often do not contain well formed URLs, and instead construct the final URL programatically. In this case, we perform a regular expression search of the rules for the domains associated with each violator. If such a domain exists in the script, it is likely responsible for the ultimate connection to that server. Figure 7 shows an example where a server, S_4 is explicitly included in the text of a rule, and therefore moved from the set of unmatched servers to the matched servers.

External JavaScript However, not all JavaScript on a page is inline: a common model is to load scripts from external sources, which may then load objects from additional external servers. Indeed, our first two conditions would fail to detect the connection between a violating server and a rule which contains such scripts, as seen in Figure 6.

We therefore consider the following additional procedure. When the user submits object timings, Oak considers all external scripts loaded by the user. It then checks the source domain of each of these external scripts against the rules under consideration. If the script domain appears in a rule (using either of the above conditions), the script is then labeled as being activated by that rule. Oak then collects the labeled JavaScripts, loading them directly from the external sources. Next, Oak reconsiders the first two conditions on each violating server and rule, but now searches not only the text of the rule but the text of any external scripts that the rule loads. In this way Oak is able to extend its view to externally loaded objects. Importantly, Oak does *not* modify these external scripts, it simply uses them to expand the surface to which a rule might match.

To demonstrate that this methodology is able to capture the performance of a significant fraction of the behavior of a page using the above methodology, we consider the following experiment. On a local machine, we

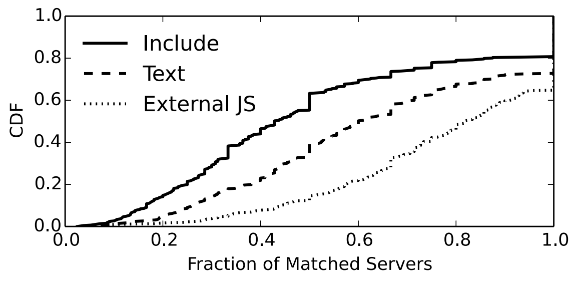


Figure 8: CDF of fraction of servers that were matched for each of the Alexa Top 500.

load the Alexa Top 500, once with the Oak client, generating a record of all contacted servers, and once with a simple wget command, which downloads the initial index page. We'd like to know what fraction of servers can be mapped to a potential rule that describes a section of the collected indices. We therefore treat the entire index page as a single rule, and attempt to match each server to it. Any servers which do not match therefore represent objects that are loaded as the result of scripts or other methods which mask the origin from Oak.

Figure 8 shows a CDF of the match rates for each of the three levels. When considering only strict includes, we are only able to match 42% of servers in the median case, meaning that at this level we may fail to activate a significant set of rules. By adding in text matches, this number improves significantly, with the median case now matching 60% of servers. Finally, by including the first layer of included JavaScript objects, we see that the median performance further increases to 81%, allowing rules to specify a significant fraction of most servers. We note that this process could be continued to an additional layer of external inclusion, however, the payoff is rapidly diminishing. The remaining contacted servers generally consist of servers likely contacted by way of dynamic scripts, which decide on a server on the fly, or in response to external inputs, for example internal page management, or external tracking and analytics scripts.

4.2.3 *Rule History*

In the course of operation, it may occur that Oak has activated a rule, but the alternate server is later deemed a violator. In such cases, Oak must determine whether or not to leave the rule active, or to deactivate it. To address this, when Oak activates a rule, it records the difference between the median performance and the performance of the violator responsible for the rule activation. If, at a later time, the alternate becomes a violator, Oak considers the alternate's distance from the median. Oak then chooses the action which minimizes this distance, attempting to retain rules which outperform the default.

4.2.4 *Policy*

In addition to the technical components of the rule activations described above, Oak can also be configured with operator specified policies. Such policies are necessary to accommodate the often complex relationships between a site operator and their third party providers. For example, in the case of a CDN, using an alternative provider may represent a significant expense, and should only be done sparingly. In such cases, Oak can tighten the restrictions on when a rule can be activated, for example by only activating a rule after 3 violations.

Oak further allows for the specification of multiple alternatives in each rule. By default, Oak progresses through the list linearly with each activation, however this can further be configured via a selection policy. Since Oak is able to observe clients directly via the web server, it could further discriminate the activation of rules based on client information, for example by IP subnet, or other network level features.

The scope parameter of rules further allows for the implementation of a rule-specific policy. Rules can be set with very wide scope, for example to include all the pages on a particular site. In such a situation the information Oak learns when a users first navigates to a site could be effectively implemented on all subsequent pages. This could be especially useful when a problematic provider is used throughout a site.

4.3 Page Modification

Finally, we discuss the alterations Oak makes to a page before finally delivering it to a client. When the client connects, it presents Oak with a unique user ID via HTTP cookie. Oak then loads the requested page via the existing web server. Oak applies the user's active rules, removing text where specified by Type 1 rules, and replacing text for Type 2 and 3 rules. Finally, Oak serves this customized page to the user and the process repeats.

In the case of a type 2 rule in which an external host is replaced with another that serves an identical object, there is potential for interference with traditional browser chaching. In particular, since the location of a resource was altered, the browser may re-fetch an identical object, ignoring a usable copy in its cache and therefor incurring fresh download costs. In order to avoid such pathological behavior, Oak informs the browser of objects subject to type 2 rules by way of a custom HTTP response Header. The client is then able to determine if an object in its cache is potentially still applicable.

Critical to Oak's operation is the fact that rule activation and subsequent changes to the page are done on a user-by-user basis. Therefore any changes that a user observers are in direct response to the performance that the user reported. This per-user approach allows users to only induce changes that effect them. Oak works to provide a user customized experience to such users, aiming to optimize page performance in terms of that user alone, rather than aiming for best general-case solutions.

5. EVALUATION

Here we consider the performance of the Oak system and demonstrate a number of the underlying component behaviors.

Implementation The client implementation of Oak consists of modified versions of the WebKit browser and PhantomJS which collect and send page reports, using infrastructure designed for use with outputting HAR files. Our page reports includes only a limited set of fields: the loaded URL, the size of the loaded object, and the timing information of that object.

The server is implemented as a multi-threaded server in Python, which serves a dual purpose as both the web server and the Oak server platform. We use regular expressions in order to apply active rules, allowing for straight forward and rapid replacement of text before each page is served.

In addition to the information necessary for Oak to perform, the server also maintains log information on the objects downloaded from particular servers, the activation and removal of rules, as well as aggregate site performance. For the majority of our experiments, Oak is run on a single machine on our campus network, allowing it the full bandwidth of the available connection. All other servers in this experiment are either production servers beyond our control, or multi-threaded Python servers which simply respond to requests for files and employ HTTP 1.1 with no modification.

Unless otherwise stated, when considering file servers, we consider servers spread across North America on Planet Lab nodes selected for geographic diversity. Our clients consist of 25 Planet Lab nodes, half of which are in North America, and the remainder evenly spread between Europe and Asia (including Oceania).

In order to evaluate the functionality of Oak, we first consider Oak's ability to respond to degrading performance of a single server. Next, we demonstrate Oak's ability to converge to the best performing set of pages when loading from multiple external services. Finally, we demonstrate that Oak is able to make appropriate decisions and improve resource download time on replicated versions of existing sites.

5.1 Sensitivity

First, we demonstrate Oak's ability to respond to degrading performance using a *relative performance mechanism*. In particular, we'd like to show that in cases where clients experience strong performance in general, Oak is able to detect servers that begin to perform poorly as soon as they begin to degrade. On the other hand, servers which see a generally worse performance have a higher threshold for detection.

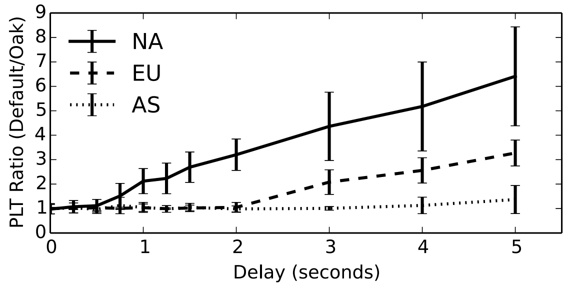


Figure 9: PLT Ratio between the default and Oak for increasing injected delays.

We consider a scenario in which a page is loaded from a client who loads objects of varying sizes from 5 external servers. We repeatedly load this page. With each subsequent load, a single external host adds a small delay before responding to HTTP requests. For each iteration, we perform this process once with Oak configured with an alternate for that server, and once with the default server. We repeat the entire experiment for 20 iterations with 11 different delays ranging from 250ms to 5s. We further consider 3 clients in North America ("NA"), Europe ("EU"), and Asia ("AS").

Figure 9 shows the average ratio of page load times (PLT) between the default and Oak cases, where error bars show a standard deviation. We see that in the case of North America, when all servers are traversing short paths and performing similarly, Oak detects the delay early on, with delays as small as .75 seconds. On the other hand, in the case of Europe, where paths become longer, and therefore the user-observed performance more spread, delays reach above 2 seconds before Oak begins to respond. In the extreme cross-global case, delays go unnoticed until they reach 5 seconds.

This varied sensitivity demonstrates the value of Oak's relative performance mechanism. By only reacting to poorly performing servers relative to other servers at the same time, Oak avoids activating rules inappropriately. While here we use geographic distance to vary performance this principle applies in other scenarios of reduced functionality, for example when using a mobile device.

5.2 Benchmark Detection

Next, we show that Oak's MAD-based selection criteria results in improved performance. Specifically, we'd like to demonstrate that Oak is able to identify underperforming servers and make decisions that drive it towards improved user performance. To test this, we consider a simple website which consists of 6 sets of simple

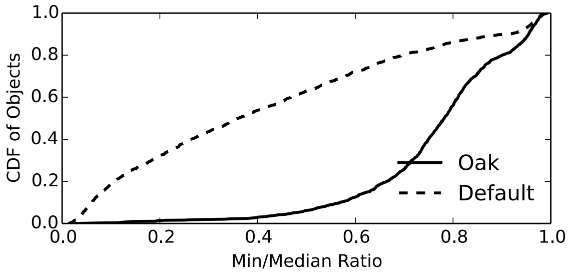


Figure 10: Min/Median ratio for both Oak and default loads.

objects. Each set consists of files sized 30, 50, 100, and 500KB. The first set of these objects are hosted on the same machine as the page index. Each of the remaining 5 sets are hosted on different external servers, in this case 5 randomly selected North American Planet Lab nodes. Each object is provided with headers to prevent cacheing.

An additional 5 sets of the same objects are created on another randomly selected set of 5 servers. A rule is created for each of the original sets that specifies one of the second set as an alternative using only Type 2 rules. We thereby pair each of the 5 sets with an alternative that serves an identical set of objects. No modification was done to the servers to ensure differences in the pairs hosting performance: they were selected randomly. We then host the entire page on a web server at our local institution running Oak with the described configuration.

Next, we load the page with references to all 6 sets of objects from 25 client locations around the world, once with Oak enabled, allowing the system to apply any rules which are active, and once with all rules disabled, *i.e*, the default page. The pages are reloaded every 30 minutes for 72 hours.

Figure 10 shows the CDFs of the Min/Median ratio for both the Oak loads and the default loads across all clients. In the case of a poorly performing server, we expect a higher occurrence of lower ratios, *i.e.*, a smaller Min/Median ratio. In the case of more consistent performance, we expect a higher ratio, as the minimum will sit closer to the median. We see that Oak has provided exactly this difference, increasing the median ratio from .3 to .7, and pushing 90% of loads to a ratio of above .5. We emphasize that the magnitude of the performance gains are a function of the network state and the quality of the providers and reiterate that the value of Oak lies in its ability to detect these cases of poor performance, regardless of their underlying cause, and move towards a well behaved configuration for each user.

During this experiment, we found that 2 of the Planet Lab servers were performing significantly worse than the others. As these 2 servers were used in the default

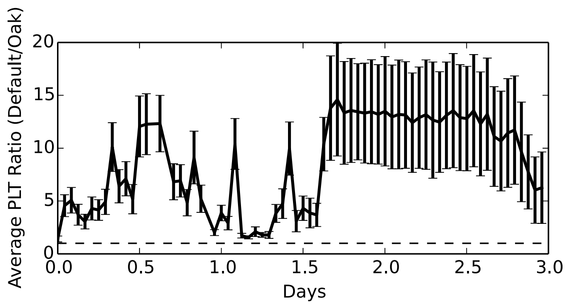


Figure 11: The average PLT ratio over 3 days. Regular variations can be seen in the performance gains.

page, Oak was readily able to detect them and switch to the alternative providers, which provided consistently improved performance. To quantify the improvement seen overall by avoiding these non-performant servers, we consider the ratio of total page load time between the default and Oak loads over time.

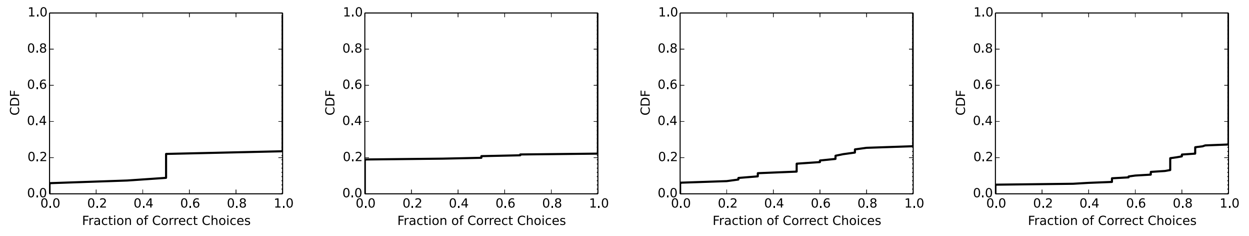
Figure 11 shows the average ratio over all 25 clients for the duration of the experiment, where the error bars show a standard deviation. We see that during the night, Oak performance was near that of the default. As the default providers became busy during the day, Oak was able to significantly improve the total page load time. While in these cases Oak was able to improve the load times by over 10×, these gains are exactly proportional to the delays incurred at the poorly performing servers. Oak is able to avoid incurring poor performance after the first load, as long as a suitable alternative is available.

5.3 Performance on Existing Sites

While our previous evaluation demonstrated the ability of Oak to detect non-performant servers on real Internet links, and our analysis in Section 2 showed that there exist non-performant objects on real websites, we'd like to demonstrate Oak's ability to detect and correct such issues on existing sites. We therefore consider the following experiment built on this premise.

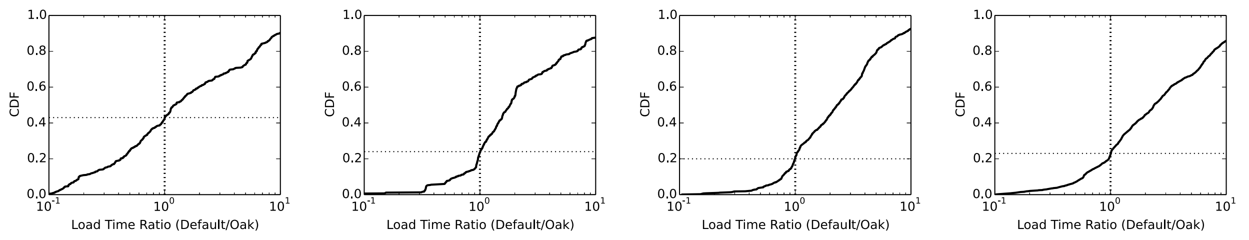
Here, we replicate existing sites by copying them onto a server in our control which is running Oak. Externally hosted objects are still hosted at third-party production servers that we do not control. We then load the site from external clients and demonstrate that Oak is able to identify the violating servers to which the site directs clients, and switch to viable alternatives when available.

Selecting Sites First, we select a set of sites to replicate. We again consider the Alexa Top 500. Necessarily, there exist two main categories of sites: (*i*) Low-expectation sites, which have a small to moderate number of external servers, hence their performance is less likely to be improved by Oak. (*ii*) High-expectation sites, which utilize a large number of external servers,



(a) (b) (c) (d)

Figure 12: The fraction of correct rule choices made by Oak for (a) H1-Close, (b) H1-Far, (c) H2-Close, and (d) H2-Far



(a) (b) (c) (d)

Figure 13: The ratio of load time of default objects to Oak-enabled objects for Oak protected objects with active rules for (a) H1-Close, (b) H1-Far, (c) H2-Close, and (d) H2-Far

H1	**H2**
youtube.com	ok.ru
msn.com	flipkart.com
wordpress.com	qunar.com
naver.com	hulu.com
adcash.com	xhamster.com

Table 2: Selected sites for low and high expected improvement.

hence their performance is likely to be improved by Oak. We emphasize, however, that *all* sites would benefit from implementing Oak, since there may still exist clients in circumstance which would benefit from using alternate servers. We indeed show that this is the case.

To select sites from the first category we consider 5 sites with more than 5 but fewer than 15 external hosts. We label these *H1* sites. Likewise, we take 5 sites with more than 15 external hosts and label them as *H2* sites. In both cases, we select sites which were able to achieve the highest rule-activation match rate. Table 2 shows the selected sites.

Generating Rules To run Oak on such sites, we must configure Oak with a set of rules, providing alternatives for each site. To this end, we consider every external domain contacted during a normal load of each site. We then generate a type 2 replacement rule for every observed domain, allowing Oak the greatest opportunity to offer each site improvement, though we note that not all of these rules will necessarily be activated during the course of the experiment.

Alternative Servers In order to offer alternative servers which offer improved performance, we replicate all external objects to 3 web servers: one in each of North America, Europe, and Asia. Each client is then directed to its closest alternative when a rule is activated. This setup allows us to emulate an alternate provider, which may present clients with reasonably close replicas.

Performance We load each page from each of the clients 15 times with three conditions: the default page without Oak, Oak with all rules activated, and Oak with the usual rule behavior. Since a portion of our sites come from each North America, Europe, and Asia, we have clients which load resources from servers at varying distances. We are thus able to consider two classifications: *close* and *far*. Therefore, we consider 4 possible conditions: H1 sites with close and far clients, and H2 sites with close and far clients. In our evaluation, we consider cases in which rules were activated, *i.e.* we ignore cases in which no rule was ever activated, as the performance matches the default exactly.

First, we wish to explore how often Oak correctly chose to activate or deactivate a rule. We therefore must first determine a correct value for each rule. To make this determination, we consider the timing results of our loads without Oak (*i.e.*, the default server) and with all rules forced on. If the default server is faster for the majority of objects within a rule, the correct setting is to disable the rule. If the active-rule value is faster, the correct setting is to enable the rule and use the alternate.

Figure 12 presents the fraction of correct choices over all collections for each of our 4 groups. In a perfect scenario (no errors), we would expect all the figures to show a vertical line at $x = 1.0$. We note that all conditions feature cases with incorrect choices. A portion of these are due to Oak's experiential approach: Oak must use a server before it has information about that server. It must necessarily activate rules which are later deactivated when the alternate was non-performing. In the H1 cases, nearly 80% of choices are entirely correct, *i.e.*, as shown in Figures 12(a) and (b). In the H2 case, approximately 74% of choices are always correct, as shown in Figures 12(c) and (d). Notably, there are more rules in the H2 cases, as there are more external domains, creating the more varied results.

Figure 13 presents the resulting difference in object performance: here we consider the ratio of the default object timing to the timing for the choice Oak made (either the default or an alternate). With the H1-Close condition, we see close-to-even-spread performance, as the alternates and default server provided very similar performance, though Oak's choice was an improvement for 57% of cases. In the remaining three conditions, Oak's choices offered improvement for significant fractions of users: 66% for H1-Far, 80% for H2-Close, and 77% for H2-Far. In nearly all cases where the default performs better, the difference is within normal variations.

The performance variation across conditions demonstrates the scenarios where Oak is most readily able to offer gains: when a site is well provisioned with few external objects (H1-Close), Oak generally offers no standard gains, but will protect against performance dips. In cases where clients see varied performance (H1-Far, H2-Far), Oak is able to correctly switch to alternatives. Finally, in cases of complex sites with many external providers (H2-Close), Oak is able to detect underperforming providers and activate possible alternatives.

Individual vs. Common Problems Here, we explore if Oak resolves common problems, seen by many clients accessing a site, or individual problems, seen by only a small fraction of users. The idea is to see which types of problems Oak is solving: specific issues that may be dependent on the particular conditions of a client, or more general issue with external providers. To this end, we examine how often each of our rules is activated across users and sites.

Figure 14 presents a CDF of rules over the fraction of users which activated the rule on a site. We see that

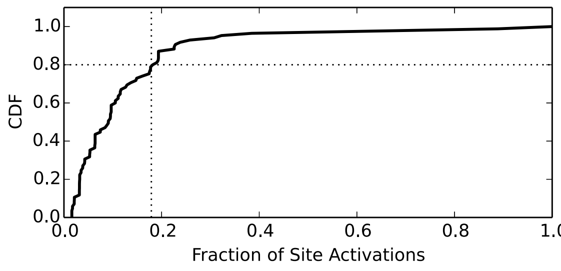


Figure 14: Cumulative activation by fraction of rules.

Individual	Common
vdp.mycdn.me	fonts.googleapis.com (88%)
img1.qunarzz.com	insights.hotjar.com (63%)
i.ytimg.com	ad.doubleclick.com (38%)
ut06.xhcdn.com	ads1.msads.net (32%)
img1a.flixcart.com	pubads.g.doubleclick.net (31%)

Table 3: Examples of providers domains from individual (< 18% of activations) and commonly (> 18%) activated rules.

80% of rules never account for more than 18% of their sites activations: most rules are only activated by a few users and may be the result of a client-oriented condition, for example a resource always being in a distant location from the user. The first column of Table 3 presents examples of such sites. These domains generally seem to be associated with externally hosted site resources, such as images, and may likely reflects the regional nature of the sites (*e.g.* Resources for Chinese travel site qunar.com performs poorly only for clients outside of China).

As we see in the figure, the remaining 20% of rules account for higher fractions of activations, with the most active rule, an object from Google's public font API, accounting for 88% of wordpress.com rule activations. The larger fraction taken by each of these external domains suggest that a potentially significant number of clients are seeing sub-par performance from those advertising providers.The second column of Table 3 shows that ad providers are some of the most frequent common offenders. Here, the percentages of activations are per site, so all rules add up to greater than 100%. The presence of such frequent violators, as well as the high rate of less-commonly activated rules, emphasizes the importance of Oak's user-oriented approach, which can simultaneously address both scenarios.

6. DISCUSSION

While Oak could employ absolute conditions of performance, for example a maximum time or minimum throughput for a specific object, we chose to focus on relative performance. In particular, we use the median

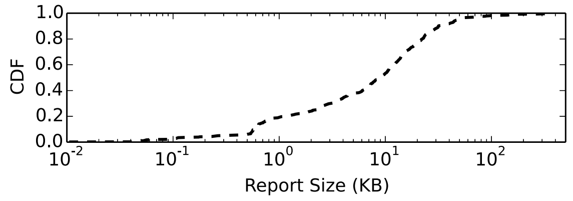


Figure 15: Report sizes from the Alexa Top 500. Most reports are below 10KB.

and the MAD to determine which external providers are not performing well. By doing so Oak is able to accommodate clients who may encounter generally poor performance. Using a relative measure further eliminates the need to perform parameter selection that might otherwise require regularly updated measurements.

Oak's client-reported approach further enables it to readily function with dynamic pages. By performing its analysis on only those resources which the client observed, Oak is able to make generic determinations about their performance, regardless of what might be expected, *i.e.*, operators need not specify which servers and providers a client will contact before-hand in order for Oak to analyze the performance. Operators could therefore indicate a large scale set of rules, providing alternatives for any component of the site which may ultimately damage user performance.

Furthermore, by bringing client-observed performance directly into the process of selecting external servers, Oak creates incentive for third-party services to improve their performance. Indeed, if these services perform poorly, they will actively lose business, as Oak selects better performing options. Examining which rules are being activated by clients enables site operators to determine which components of their sites are performing poorly, effectively using the performance reports of Oak as an offline auditing tool.

Overhead The added communication performed by Oak is not without overhead. Indeed, communicating the performance report back to the server means clients must explicitly send data back. The size of these reports is further dependent on the total number of objects fetched while loading the page. As described in Section 4, Oak uses a report format similar to a HAR file, but with a limited set of information. Figure 15 presents the distribution of report file sizes when an Oak client loads the Alexa Top 500. We see that these values remain entirely within manageable ranges for all clients. In the median case reports are below 10KB, and in the worst-case only 345KB. Because performance reports are uploaded to a server *after* the page has been downloaded, this process does not affect the user-perceived performance.

Alternative Mechanisms In addition to reports which rely on browser modification, other mechanisms could plausibly be used. In particular, the JavaScript Resource Timing API provides a mechanism by which client-side JavaScript can report network performance information back to servers [1]. However, for the resource timing API to function with external objects, which is the purpose of Oak, the external provider must explicitly include an authorizing header. This opt-in behavior means many providers are not visible with the API, rendering Oak less effective. We therefore believe that client modification is the best solution at present.

7. CONCLUSIONS

In this paper, we presented Oak, a system that provides a mechanism by which site operators can take control of the complexities of modern websites and provide users with targeted resources. Indeed, by using performance feedback directly from individual users, Oak can understand the full performance of a site and optimizes the end-user experience for each user individually, providing user-targeted HTTP performance. Oak does so by focusing on limiting exposure to user-reported performance outliers. Moreover, Oak provides a straightforward rule specification scheme which allows site operators to specify precisely how they would like Oak to respond to such outliers. We demonstrated that Oak is successfully able to detect outliers in our benchmark experiments, where it was able to avoid servers which were causing clients to experience poor page load performance. We further demonstrated that Oak is able to detect such outliers on existing websites. When given the opportunity to provide clients with an alternative, Oak was able to improve performance. Our chief contribution lies in providing a system that empowers sites with an exceptional auditing mechanism; the mechanism helps sites fundamentally regain control over the performance seen by their clients, while tailoring content for each client individually.

8. REFERENCES

[1] Resource Timing. `http://www.w3.org/TR/resource-timing/`.

[2] Abu-Libdeh, H., Princehouse, L., and Weatherspoon, H. Racs: A case for cloud storage diversity. In *Proceedings of SoCC '10* (2010), pp. 229–240.

[3] Bermudez, I. N., Mellia, M., Munafo, M. M., Keralapura, R., and Nucci, A. DNS to the Rescue: Discerning Content and Services in a Tangled Web. In *Proceedings of ACM IMC '12* (2012), IMC '12, pp. 413–426.

[4] Bessani, A., Correia, M., Quaresma, B., André, F., and Sousa, P. Depsky: Dependable and secure storage in a cloud-of-clouds. *Trans. Storage 9*, 4 (Nov. 2013).

[5] Bodik, P., Fox, A., Franklin, M. J., Jordan, M. I., and Patterson, D. A. Characterizing, modeling, and generating workload spikes for stateful services. In *Proceedings of SoCC '10* (2010), pp. 241–252.

[6] Brutlag, J. Speed Matters for Google Websearch. `http://services.google.com/fh/files/blogs/google_delayexp.pdf`.

[7] Butkiewicz, M., Madhyastha, H. V., and Sekar, V. Understanding website complexity: Measurements, metrics, and implications. In *Proceedings of 2011 ACM IMC 11* (2011), IMC '11, pp. 313–328.

[8] Butkiewicz, M., Wang, D., Wu, Z., Madhyastha, H. V., and Sekar, V. Klotski: Reprioritizing web content to improve user experience on mobile devices. In *Proceedings of NSDI '15* (Oakland, CA, May 2015), pp. 439–453.

[9] Chen, X., and Zhang, X. A popularity-based prediction model for web prefetching. *Computer 36*, 3 (Mar. 2003), 63–70.

[10] Dean, J., and Barroso, L. A. The tail at scale. *Commun. ACM 56*, 2 (Feb. 2013), 74–80.

[11] Dhawan, M., Samuel, J., Teixeira, R., Kreibich, C., Allman, M., Weaver, N., and Paxson, V. Fathom: A browser-based network measurement platform. In *Proceedings of ACM IMC '12* (2012), IMC '12, pp. 73–86.

[12] Erman, J., Gopalakrishnan, V., Jana, R., and Ramakrishnan, K. K. Towards a spdy'ier mobile web? In *Proceedings of ACM CoNEXT '13* (2013), pp. 303–314.

[13] Fan, L., Cao, P., Lin, W., and Jacobson, Q. Web prefetching between low-bandwidth clients and proxies: Potential and performance. In *Proceedings of ACM SIGMETRICS '99* (1999), pp. 178–187.

[14] Flach, T., Dukkipati, N., Terzis, A., Raghavan, B., Cardwell, N., Cheng, Y., Jain, A., Hao, S., Katz-Bassett, E., and Govindan, R. Reducing web latency: The virtue of gentle aggression. *SIGCOMM CCR 43*, 4 (Aug. 2013), 159–170.

[15] Ganjam, A., Siddiqui, F., Zhan, J., Liu, X., Stoica, I., Jiang, J., Sekar, V., and Zhang, H. C3: Internet-scale control plane for video quality optimization. In *Proceedings of NSDI '15* (Oakland, CA, May 2015), pp. 131–144.

[16] He, K., Fisher, A., Wang, L., Gember, A., Akella, A., and Ristenpart, T. Next stop, the cloud: Understanding modern web service deployment in ec2 and azure. In *Proceedings of IMC '13* (2013), pp. 177–190.

[17] Kotla, R., Alvisi, L., and Dahlin, M. Safestore: A durable and practical storage system. In *Proceedings of the USENIX ATC'07* (2007).

[18] Kreibich, C., Weaver, N., Nechaev, B., and Paxson, V. Netalyzr: Illuminating the edge network. In *Proceedings of ACM IMC '10* (2010), pp. 246–259.

[19] Li, A., Yang, X., Kandula, S., and Zhang, M. Cloudcmp: Comparing public cloud providers. In *Proceedings of IMC '10* (2010), pp. 1–14.

[20] Li, Z., Zhang, M., Zhu, Z., Chen, Y., Greenberg, A., and Wang, Y.-M. Webprophet: Automating performance prediction for web services. In *Proceedings of USENIX NSDI '10* (2010), NSDI'10.

[21] Liu, H. H., Wang, Y., Yang, Y. R., Wang, H., and Tian, C. Optimizing cost and performance for content multihoming. In *Proceedings of SIGCOMM '12* (2012).

[22] Lohr, S. For impatient web users, an eye blink is just too long to wait. *New York Times* (Feb 2012).

[23] Mai, H., Tang, S., King, S. T., Cascaval, C., and Montesinos, P. A case for parallelizing web pages. In *4th USENIX HotPar* (2012).

[24] Meyerovich, L. A., and Bodik, R. Fast and parallel webpage layout. In *Proceedings of WWW '10* (2010), pp. 711–720.

[25] Nanopoulos, A., Katsaros, D., and Manolopoulos, Y. A data mining algorithm for generalized web prefetching. *IEEE Trans. on Knowl. and Data Eng. 15*, 5 (Sept. 2003), 1155–1169.

[26] Padmanabhan, V. N., and Mogul, J. C. Using predictive prefetching to improve world wide web latency. *SIGCOMM CCR 26*, 3 (July 1996), 22–36.

[27] Alexa. `http://www.alexa.com/`.

[28] Sitaraman, R. Network performance: Does it really matter to users and by how much? In *Communication Systems and Networks (COMSNETS), 2013 Fifth International Conference on* (2013), pp. 1–10.

[29] Souders, S. Velocity and the Bottom Line. `http://radar.oreilly.com/2009/07/velocity-making-your-site-fast.html`.

[30] Sundaresan, S., Feamster, N., Teixeira, R., and Magharei, N. Measuring and mitigating web performance bottlenecks in broadband access networks. In *Proceedings of ACM IMC '13* (2013), IMC '13, pp. 213–226.

[31] Suresh, L., Canini, M., Schmid, S., and Feldmann, A. C3: Cutting tail latency in cloud data stores via adaptive replica selection. In *Proceedings of NSDI '15* (Oakland, CA, May 2015), pp. 513–527.

[32] Vulimiri, A., Godfrey, P. B., Mittal, R., Sherry, J., Ratnasamy, S., and Shenker, S.

Low latency via redundancy. In *Proceedings of CoNEXT '13* (2013), pp. 283–294.

[33] Wang, X. S., Balasubramanian, A., Krishnamurthy, A., and Wetherall, D. Demystifying page load performance with wprof. In *Proceedings of USENIX NSDI '13* (2013).

[34] Wang, X. S., Balasubramanian, A., Krishnamurthy, A., and Wetherall, D. How speedy is spdy? In *Proceedings of USENIX NSDI '14* (Seattle, WA, Apr. 2014), pp. 387–399.

[35] Wang, X. S., Krishnamurthy, A., and Wetherall, D. How much can we micro-cache web pages? In *Proceedings of IMC '14* (2014), pp. 249–256.

[36] Wang, Z., Lin, F. X., Zhong, L., and Chishtie, M. Why are web browsers slow on smartphones? In *Proceedings HotMobile '11* (2011), HotMobile '11, pp. 91–96.

[37] Wieder, A., Bhatotia, P., Post, A., and Rodrigues, R. Orchestrating the deployment of computations in the cloud with conductor. In *Proceedings of NSDI'12* (2012), pp. 27–27.

[38] Wu, Z., Butkiewicz, M., Perkins, D., Katz-Bassett, E., and Madhyastha, H. V. Spanstore: Cost-effective geo-replicated storage spanning multiple cloud services. In *Proceedings of SOSP '13* (2013), pp. 292–308.

[39] Wu, Z., Yu, C., and Madhyastha, H. V. Costlo: Cost-effective redundancy for lower latency variance on cloud storage services. In *Proceedings of NSDI '15* (Oakland, CA, May 2015), pp. 543–557.