AdJust: Runtime Mitigation of Resource Abusing Third-Party Online Ads

Weihang Wang
University at Buffalo
Buffalo, New York, USA
weihangw@buffalo.edu

I Luk Kim
Purdue University
West Lafayette, Indiana, USA
kim1634@purdue.edu

Yunhui Zheng
IBM T.J. Watson Research Center
Yorktown Height, New York, USA
zhengyu@us.ibm.com

***Abstract*—Online advertising is the most critical revenue stream for many Internet companies. However, showing ads on websites comes with a price tag. Since website contents and third-party ads are blended together, third-party ads may compete with the publisher contents, delaying or even breaking the rendering of first-party contents. In addition, dynamically including scripts from ad networks all over the world may introduce buggy scripts that slow down page loads and even freeze the browser. The resulting poor usability problems lead to bad user experience and lower profits. The problems caused by such resource abusing ads are originated from two root causes: First, content publishers have no control over third-party ads. Second, publishers cannot differentiate resource consumed by ads from that consumed by their own contents. To address these challenges, we propose an effective technique, AdJust, that allows publishers to specify constraints on events associated with third-party ads (e.g., URL requests, HTML element creations, and timers), so that they can mitigate user experience degradations and enforce consistent ads experience to all users. We report on a series of experiments over the Alexa top 200 news websites. The results point to the efficacy of our proposed techniques: AdJust effectively mitigated degradations that freeze web browsers (on 36 websites), reduced the load time of publisher contents (on 61 websites), prioritized publisher contents (on 166 websites) and ensured consistent rendering orders among top ads (on 68 websites).**

***Index Terms*—online ads, defective ads, resource abusing, performance degradation, mitigation**

I. Introduction

Modern websites deliver sophisticated contents and provide full-fledged functionalities thanks to the recent advance of web technologies such as HTML5 and JavaScript. Similar to the common practice in traditional software development, web developers make use of third-party utilities that offer dedicated and professional services. They can add various services into their websites by incorporating a few bootstrap snippets provided by the third-party service providers without worrying about the complexity and scalability issues in the development and maintenance of the underlying services.

Among such third-party services, online ads are the most ubiquitous mashups nowadays. Today, over 14.3 million websites use Google's online advertising networks *Google AdSense* [13] and over 2.2 million websites participate Amazon's affiliation marketing program *Amazon Associates* [4] to make money by displaying ads on their web pages. In fact, online advertising has become the most critical revenue stream for many Internet companies. Website owners sell ad space through advertising platforms to monetize their websites and get potential leads. For example, Google provides an ad serving services platform called *DoubleClick* [12], which is widely deployed on many popular websites due to its ease of use and broad coverage. To make use of the services, website developers only need to include `div` elements to specify ad space and a few JavaScript snippets provided by *DoubleClick* in their websites. The additional contents loaded by the scripts will take care of the rest ad delegations on the fly.

Despite the benefits, there is a price to pay if advertisements take excessive resource to display. Since website contents and third-party ads are blended together in browsers, third-party ads may compete with the contents owned by content publishers, delaying the rendering of first-party contents. Additionally, in the current ad networks, an ad slot may go through an intricate chain of delegations until the final ad is delivered. Dynamically including scripts from ad networks all over the world may introduce buggy ad scripts that slow down the page rendering and even freeze the browser. Cybercriminals even target ad networks to inject CPU-draining cryptocurrency mining scripts, which exacerbates the *resource abusing* problems. The resulting poor usability leads to bad user experiences and lower profits: for an e-commerce site making $0.1M per day, a 1-second page delay can potentially cost $2.5M in lost sales each year [18].

We have observed a number of such resource abusing problems on real world websites. To name a few, a news article page on *www.cnn.com* takes around 19 seconds to render the contents including ads, while the article without ads loads within just 5 seconds. The overwhelming majority of the time are spent on ad trackers and advertising networks. A defective video ad on *www.chicagotribune.com* makes tens of thousands of URL requests within 20 seconds and eventually freezes the entire browser.

The problems caused by resource abusing ads are originated from two root causes: First, content publishers have no control over third-party advertisements. Second, publishers cannot differentiate resource consumption of ads from their own contents. In many cases, content publishers even do not have a reliable way to detect such client-side violations. Despite recent advances in improving web page performance and increasing ad bidding efficiency, there is a lack of practical yet systematic techniques to enable content publishers to control

resource consumptions by ads or establish performance-related agreements. Thus, in this paper, we present AdJust as the first step in tackling these challenges. We focus on a content publisher oriented approach, with the goal of effectively mitigating resource abusing ads and consistently delivering better user experiences. In particular, our contributions are as follows:

- We propose a system that allows website developers to specify constraints on resource consumption for third-party ads.
- We develop a novel technique AdJust to monitor and regulate resource abusing ads on the fly.
- We propose a measurement system to quantify resource abusing problems (in particular, *performance degradation* and *priority inversions*) caused by ads.
- Our evaluation on Alexa top 200 news websites shows that AdJust is highly effective in mitigating resource abusing ads that freeze browsers and degrade user experiences.

II. Motivation

We use two real world examples to show (a) how ads delay publisher contents, and (b) how defective ads degrade performance.

A. Ads Delay First-Party Contents

Figure 1 shows the screenshots of loading a baseball video on *www.washingtontimes.com*. The web page contains a handful of media contents. Since the page starts to load, the page header and a video player render at 3.0 second and 4.5 second respectively. The visitor assumes the baseball video will start to play soon. However, about one second later, two Google ads appear first. After another three seconds, an interactive animated ad displays, delaying the video player from playing the video. Eventually, the baseball video starts to play after 14.5 seconds. In this case, the third-party ads compete with first-party contents on computing resources. In reality, many visitors would have already gotten frustrated and left the website to visit alternative sites.

Our Solution: Prioritizing First-Party Content. To mitigate this problem, we propose to prioritize first-party content by instrumenting the web page. Specifically, we delay all ads by intercepting events that send ad requests, only proceeding to request ads once the baseball video has been loaded. As shown in Figure 2, after intentional delays were introduced for low priority ads, the baseball video starts to play at 7.5 second, much earlier than before. Moreover, although all ads were delayed, they were still able to show up in a timely manner (each with a 2 2.5 seconds delay).

B. Defective Ads Freeze the Browser

By including third-party ads, publishers are exposed to potential risks of delivering defective ads to their customers. Figure 3 shows a defective video ad on *www.chicagotribune.com* that makes tens of thousands of URL requests, quickly freezing the browser.

Initially, the video player sends requests to a predefined list of partner servers to request advertisements. Each partner server either responds with an actual ad, or redirects the video player to the next downstream ad server(s). This procedure continues until a valid ad is delivered or a predetermined timer expires. On this website, we observed a buggy ad script intensively issued a large number of requests. Particularly, we visited this website on a modern laptop with 2.8 GHz CPU processor and stable 20 Mbps network speed. The defective ad has sent tens of thousands of URL requests within 20 seconds, which made the CPU usage quickly jump to 100% and the browser freeze within a minute.

As shown in Figure 4, the video ad loads in the following steps:

- *Steps 1–3:* The video player sent ad requests to layer1 servers, one at a time. It first requested ads from server *sx.streamrail.net* [22]. This server did not have a suitable ad, so it replied with an empty ad (①). The player proceeded to other servers until it reached *vid.springserve.com* (②), which replied with a wrapper script `vd0.2.88.1.js` [2] (③), as well as a list of 214 downstream ad servers. This wrapper script is used to handle the messages between the video player and the 214 layer2 servers.
- *Steps 4–6:* The player then requested ads from layer2 servers (④). The 10th server *ads.adaptv.advertising.com* instructed the player to load script `jsvpaid.js` [1] which was supposed to be the wrapper script for another layer of servers (⑤, ⑥). However, in this message, no additional servers were provided.
- *Steps 7–13:* Because the player cannot fetch ads along the chain introduced by Ad Server 10, it moved on to Ad Server 11 (⑦). Recall that in *Step 6*, `vd0.2.88.1.js` invokes functions in `jsvpaid.js` via indirect function calls, in an attempt to request ads from additional servers. However, due to a buggy implementation in `vd0.2.88.1.js`, the call targets were not properly reset. As a result, starting from Ad Server 11, all delegations still went through `jsvpaid.js` (⑧, ⑪), which kept triggering exception events (⑨, ⑫). As such, the browser was busy with handling exceptions and eventually froze within a minute when the CPU usage reached 100%.

Our Solution: Regulating Offensive URL Requests. In this example, a layer1 ad server redirected the video player to hundreds of additional servers. Such many redirections introduced a large number of network requests and intensive JavaScript executions, which eventually halted the entire browser. To mitigate the resource abusing issue, we limit the requests sent by this ad. In particular, given that ads are usually loaded via AJAX requests, we instrument the `open()` method of `XMLHttpRequest` objects. If too many AJAX requests were sent in a short time period, additional requests will be delayed. By limiting the number of URL requests, we were able to browse the website without noticeable delays.

C. Our Approach

As shown in previous examples, resource abusing ads substantially delay first-party contents and significantly degrade user experience. To solve these problems, we propose

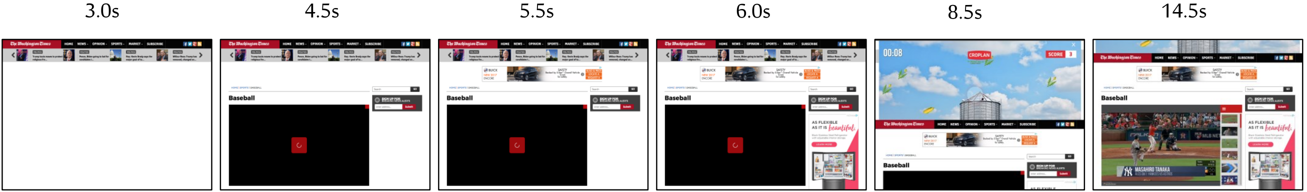


Fig. 1. The timeline of loading a baseball video on *www.washingtontimes.com*. Two Google ads appeared at 6.0s and an animated ad was displayed at 8.5s. Finally, the baseball video was loaded and started to play at 14.5s.

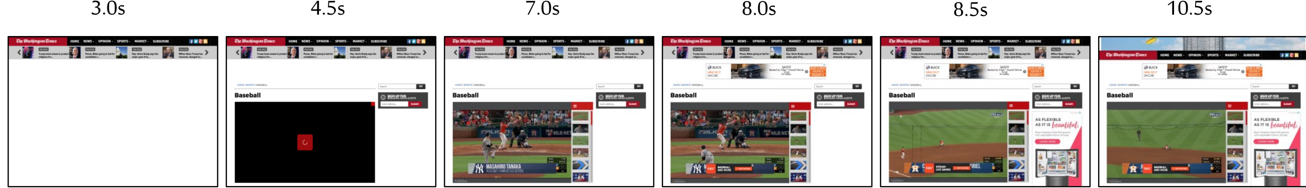


Fig. 2. The timeline of loading a baseball video on *www.washingtontimes.com* with intentional delays. The baseball video started at 7.0s. Then, two Google ads appeared at 8.5. Finally, the animated ad was displayed at 10.5s. First-party contents were rendered faster and loaded before third-party ads.

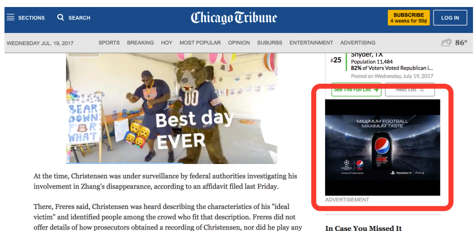


Fig. 3. A defective video ad on *www.chicagotribune.com*.

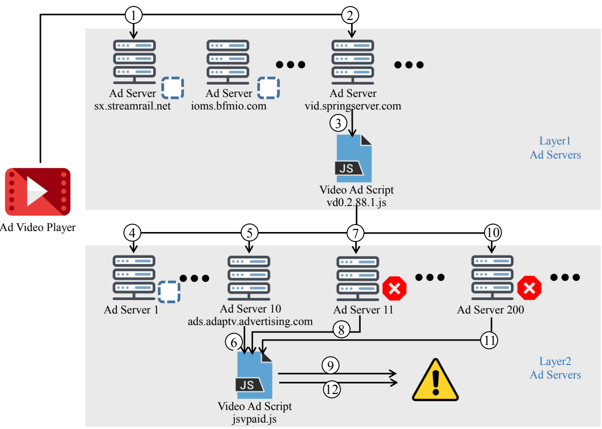


Fig. 4. Video ad loading on *www.chicagotribune.com*.

a novel runtime technique that allows publishers to specify regulation policies on each third-party ad to control resource consumption. Policies can be "load the top ad earlier than the bottom ad" (constraints for priorities between ads) and "the number of URL requests an ad can send per second should be no more than x" (constraints for resource consumption). Once publishers have created regulation policies, they only need to add a tiny JavaScript snippet and transform the web page. The rest of the regulation is handled by the AdJust runtime. On the client side, AdJust runtime enforces the rules on resource consumption and the loading orders of ads. AdJust is persistent and robust in enforcing the rules during the entire life-cycle of third-party ads, including layers of nested frames and dynamically generated contents which are common practices in advertising services.

With AdJust, publishers can (1) define constraints on priorities and execution of resource-consuming events (e.g., HTTP requests, HTML element creations, exceptions caused by third-party ads etc.) and (2) persistently regulate resource consumption caused by ads to mitigate user experience degradations and enforce consistent ads experience to all users.

III. Design

A. Overview

Figure 5 shows the overview of AdJust. It has two phases: *(1) Offline Web Page Transformation* (Figure 5-a) and *(2) Runtime Regulation* (Figure 5-b).

Offline Web Page Transformation. For each ad, content publishers define the expected service quality by specifying *resource constraints*. For instance, a publisher can specify "the maximum AJAX request rate allowed" by annotating resource constraints within an ad slot (normally embedded in `<iframe>`). The resource constraints are then translated to JavaScript code by the *Source Code Transformer*. The source code transformer inserts the compiled constraints into the publisher's web pages, as well as including the *AdJust Runtime* to enforce runtime regulation.

Runtime Regulation. AdJust runtime interposes JavaScript APIs that generate resource-consuming events. When such events are invoked, they are not executed immediately. Instead, they are enqueued into the *Event Queue* (①) which maintains multiple queues for events with different priorities. *Resource Regulator* is the core component of AdJust runtime. It repeatedly fetches events from the event queue (②) and ensures that each fetched event does not violate the resource constraints specified by the publisher. If no violation is observed, the regulator allows the fetched event to be executed (③). Otherwise, the event will be delayed until the constraints are satisfied (③).

B. Resource Constraints

To specify resource constraints for ads, content publishers leverage three types of constraint primitives (Table I).

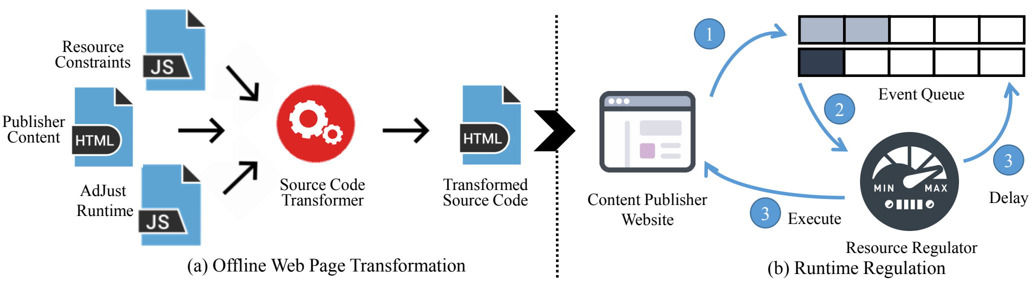


Fig. 5. System Overview.

(1) *Priority primitives* specify the priority level (numbers 0-9 with smaller number denoting higher priority) of an ad. Ads with higher priority will get loaded first.
(2) *Total quota primitives* are designed to regulate resource consumption during the entire ad life-cycle. For example, a publisher can refuse to load an ad with more than 10 layers of delegations with the primitive "$delegations[AD] \leq 10$".
(3) *Time-slot quota primitives* focus on resource consumption in a short period of time. For example, publishers can use them to regulate the behavior of aggressively sending requests within one second.

TABLE I
PRIMITIVES FOR RESOURCE CONSTRAINTS.

Type	Primitive	Description (Example)
Priority	pri	Priority of an ad slot • e.g., '$pri[AD] = 1$' (Set priority to 1)
Total quota	$totalTime$	Cumulative time spent • e.g., '$totalTime[AD] < t$' (Should be loaded within t seconds)
	$repeatRequest$	# of repetitive requests on the same URI • e.g., '$\forall repeatRequests[AD] < r$' (No more than r requests with the same URI)
	$delegations$	# of delegations • e.g., '$delegations[AD] < d$' (Should be loaded before d delegations)
	$ajaxRequests$	# of Ajax requests • e.g., '$ajaxRequests[AD] < a$' (At most a Ajax requests can be sent)
	$uriRequests$	# of URI requests • e.g., '$uriRequests[AD] < u$' (At most u URI requests can be sent)
	$elements$	# of elements created • e.g., '$elements[AD] < e$' (At most e HTML elements can be created)
Time-slot quota	$delegationsTS$	# of delegations per second • e.g., '$delegationsTS[AD] <= dr$' (At most dr delegations per second)
	$ajaxRequestsTS$	# of Ajax requests per second • e.g., '$ajaxRequestsTS[AD] <= ar$' (At most ar Ajax requests per second)
	$uriRequestsTS$	# of network requests per second • e.g., '$uriRequestsTS[AD] <= ur$' (At most ur URI requests per second)
	$elementsTS$	# of HTML elements created per second • e.g., '$elementsTS[AD] <= er$' (At most er elements can be created per sec.)

C. Source Code Transformer

Once content publishers annotate resource constraints on their web pages, *Source Code Transformer* will compile the resource constraints into JavaScript snippets and inject them into the web pages. Specifically, it first parses the web page to a DOM tree using htmlparser [25]. Then, it traverses the tree to encode resource constraints on the node that encloses ads (e.g., `<iframe>`). Along with the encoded resource constraints, it also inserts AdJust runtime support into the web page. The transformed source code will be executed in web browsers to regulate resource-consuming events at runtime.

D. AdJust Runtime

AdJust's runtime support has two components: *Persistent Runtime Installer (PRI)* and *Resource Event Shim (RES)*.

Persistent Runtime Installer (PRI). To mitigate resource abusing ads throughout the entire ad delivery process, AdJust's runtime support must be persistently active along the delegation chain. During each ad delegation, third-party ads create new inner `<iframe>` elements that create a completely new and isolated execution context from their parents. Thus, AdJust must install itself into the newly created `<iframe>` elements so that it can be persistently monitoring and regulating resource consumptions in the new inner elements. Otherwise, AdJust loses control over the `<iframe>` elements. To do so, *Persistent Runtime Installer* (PRI) ensures the entire system, including itself, is persistently installed throughout the entire ad delivery process by intercepting JavaScript APIs that can be used to dynamically create `<iframe>`. For example, `document.write` and `document.createElement` are the most common ways to create a DOM element dynamically. AdJust intercepts such APIs and injects JavaScript code snippets that can install AdJust in the newly created `<iframe>`.

Resource Event Shim (RES). Resource Event Shim instruments JavaScript APIs that create resource consuming events to transparently intercept their invocations at runtime. These events include sending HTTP requests, loading images, DOM creations, etc. We categorize resource-consuming events associated with ads into two event types: (1) *Events causing HTTP requests (ET1)* and (2) *Events causing new HTML elements inserted into a web page (ET2)*. Without loss of generality, there are four ways to dynamically generate resource consuming events: (a) writing HTML source into a web page, (b) creating HTML elements via the `createElements()` API, (c) modifying the `src` property of HTML elements, and (d) creating AJAX requests. Note that all four ways may initiate HTTP requests. (a) and (b) will also insert new HTML elements. Next, we discuss each type of the interfaces in detail and explain how AdJust intercepts these interfaces.

(a) *Writing HTML source into a web page.* Developers may call the `document.write()` or set the HTML content of an element by assigning its `innerHTML` property. These interfaces may insert new elements (ET2) and generate HTTP requests (ET1) if a newly inserted element supports URL attributes. AdJust intercepts the interfaces and puts a resource consuming event with its event type (e.g., ET1 or ET2) into the event queue. In particular, if a newly created element supports a URL value, the event is annotated with ET1 type and then enqueued. Otherwise, it is annotated with ET2 and then enqueued.
(b) *Creating HTML elements.* The most common way to inject elements into a web page is calling `createElement()` function to create an element, followed by `appendChild()` or `insertBefore()`. AdJust instruments functions `appendChild()` and `insertBefore()` to intercept their invocations. When invoked, AdJust checks whether a `src`/`url`/`href` property is supported. If so, the event with ET1 event type is enqueued to the event queue. Otherwise, the event is treated as ET2 events and gets queued.
(c) *Modifying the* `src` *property of HTML elements.* Modifying an existing element's `src` property by calling `setAtrribute()` or directly assigning a new value may generate a URL request. For instance, changing the `src` property of an image element (`<img>`) to the link of an image file results in downloading a new image file. To intercept such events, AdJust overrides `setAttribute()` and the setter of `src` property. When they are called, AdJust enqueues the event with ET1 event type to the event queue.
(d) *Sending AJAX requests.* AJAX requests are commonly used to refresh contents without reloading the entire page. AJAX requests can be initiated by calling the `send()` method of an `XMLHttpRequest` object. AdJust overrides the `send()` method so that AdJust enqueues the event with ET1 event type onto the event queue.

At runtime, depending on the event type, the resource regulator will check different constraints to determine if the event should be executed immediately or delayed. For example, ET1 type events are checked against network related resource constraints while ET2 type events are checked against DOM resource constraints (e.g., # of DOM element creations).

E. Resource Event Queue

Resource Event Queue is a multi-level queue where each queue contains resource-consuming events with the same priority. Specifically, we maintain 10 internal queues: 0-9 with a smaller number denoting higher priority. Incoming resource events are enqueued accordingly. If no priority is specified, an event will be equeued to the queue with least priority (priority 9). When AdJust dequeues an event, it first checks the priority queue with highest priority. If no event is available to schedule, it dequeues an event from the next priority queue.

F. Resource Regulator

Resource regulator repeatedly dequeues events from the Resource Event Queue and decides whether each event should be executed right away or delayed. Specifically, given an event, it checks the event against the resource constraints to see if any of the constraints are violated. If any are violated, the regulator puts the event back into the queue and delays its execution. Otherwise, the event will be dispatched and executed immediately.

Resource Constraint Checking. Before dispatching an event, the Resource Regulator checks relevant resource constraints. Depending on the type of resource constraints, the Resource Regulator handles the event accordingly:
(1) For priority resource constraints, the Resource Regulator dequeues events in their priority orders. In particular, it first checks the priority queue with the highest priority. If no event is available to schedule in this queue, the Resource Regulator dequeues an event from the next priority queue and so on.
(2) For total quota resource constraints, once an ad exceeds the quota, no further event will be allowed, because the constraint is enforced for the entire life-cycle of an ad. For instance, suppose a developer specifies the resource constraint "$elements[AD] \leq 100$" to only allow ads that inject less than 100 HTML elements into the web page. Based on the value of runtime counter, once 100 elements have already been inserted, AdJust will discard additional element creation event.
(3) For time-slot quota primitives, resource constraints are temporary. If an event violates time-slot quote constraints, the Resource Regulator will put this event back to the event queue and retry this event at a later time. Consider a resource constraint "$elementsTS[AD] \leq 10$" and assume that there were already 10 elements injected during the last second. If there is a new element creation event, the resource constraint is violated. However, since "$elementsTS$" represents the allowed number of elements injected within a second, a new element can be inserted after the counter is reset.

Runtime Counters. Runtime counters maintain the current state of resource consumption. At runtime, the Resource Regulator compares the runtime counter values with resource constraints to decide whether an event should be executed or delayed. Maintaining the runtime counters is straightforward: AdJust updates the associated runtime counters whenever a resource event is processed and dispatched. For instance, when an HTML creation event is executed, the counter for "# of elements created" is increased by 1. The runtime counters for each primitive are maintained as follows.
- *totalTime:* When an event is executed, AdJust records timestamps when it starts and ends to obtain an elapsed time for the event and accumulates the time to *totalTime*.
- *repeatRequest:* For each url, if there are multiple requests on the same url, AdJust counts the number of repeated requests including AJAX requests.

- *delegations:* We consider creating an inner <iframe> as a delegation. When an inner <iframe> is created, *delegations* are increased by 1.
- *ajaxRequests, urlRequests, elements:* They are increased by 1 on every AJAX request, URL request (e.g., loading an image in <img>), and HTML element creation respectively.
- *delegationsTS, ajaxRequestsTS, urlRequestsTS, elementsTS:* These are time-slot quota primitives. AdJust counts the number of events that occurred during the last 1 second.

IV. Empirical Studies

To better understand how resource abusing problems affect user experience in practice, we have conducted a one-month empirical study on Alexa top 200 news websites that contain third-party ads. In total, 166 websites contain at least one ad. We quantify these problems and classify resource abusing ads based on their effects.

A. Experimental Methodology

We conduct the data collection in three steps.

Step 1: Collecting website speed metrics. We collect the following data from the Alexa top 200 news websites to measure: (1) the page load time, the visually complete time, speed index, and time to interactive; (2) the CPU time spent on each individual HTML element/JavaScript/<iframe>; (3) the network time spent on URL requests. Also, for each element and network request, we identify the ad generated that element or request. To achieve that, we leverage website testing service WebPageTest [14] to measure page-level metrics. The service can work with Chrome DevTool to provide an in-depth view of the CPU and network requests at the element level. The results are collected from page tests at Dulles, VA by following the guidelines of empirical studies presented in [16], [19], [30]. We chose 10/1 Mbps as the connection rate because this is the rate half of Internet users in United States use as of Q1 2017 [26]. Of the 200 news websites, 166 websites contained at least one third-party ad.

Step 2. Annotating third-party ad scripts. To study the CPU and network time on individual ads, we separate the result for third-party ad scripts from publisher content. As most third-party ads are enclosed within an frame (<iframe>) and can be identified effectively by ad blockers, we use an ad blocker, AdBlock Plus Chrome extension [10], to locate <iframe> with ads. In particular, we visit a website twice using Chrome browser with and without the Adblock Plus extension. We saved the web page code for both visits and leveraged an HTML parser [25] to compare the DOM structures.

Step 3: Data analysis. To measure the time spent on each ad, we identify the top-level <iframe> for an ad from the profiler in Chrome DevTool, based on the DOM tree comparison result. Then, starting from the top-level <iframe>, we check whether nested inner frames are enclosed by examining the referrer field of entries that request new pages. In this way, we can discover a chain of delegations and record each nested inner <iframe> that belongs to an ad. Finally, we add up the time for URL requests within all of the recorded frames to obtain the network time for this ad. Similarly, we sum up the time of the HTML parsing, the script evaluation and the layout update that belong to the ad as its CPU time.

B. Resource Abusing Ads

We have observed two common resource abusing problems: *performance degradation* and *priority inversions*.

1) Performance Degradation: Website performance degradation can be the result of bad programming practices or defects that consume excessive computation resources. They drag down the browser performance and introduce unnecessary delays in user-perceived page load time. We define the following indices to measure the performance degradation.

Slow Response Time (SRT). We are interested in the response time that JavaScript code takes to react to user actions, as it is one of the most critical factors affecting user perceptions. Slow responses are usually caused by long-running JavaScript that blocks the main thread from serving user events. As suggested, 0.1 seconds is *the limit for having users feel the system is reacting without delays* [24]. Therefore, we say a page is *unresponsive* when the browser main thread is blocked for more than 0.1 seconds and collect such *unresponsive windows*. Intuitively, we define third-party ads cause *slow response time* problem, if they introduce more unresponsive windows than publisher contents. Formally, if we use w' and w to denote unresponsive windows caused by ads and publisher contents, a page suffers from the SRT issue when $\sum w'_i > \sum w_i$.

Result. Of the 166 websites contained ads, *36 websites suffered from slow response time issue.* Figure 6 shows a partial result on 20 websites. The red bars are the accumulated unresponsive time caused by ads, and the blue bars are the accumulated unresponsive time caused by publisher contents. *On average, ads contribute 12.5 seconds to the unresponsive time, which is 3.1 times of publisher contents.*

Load Time Bloating (LTB). The requests initiated by third-party ads and publisher contents are dispatched in a first-come-first-serve manner, as browsers only allow a limited number of concurrent requests. However, third-party ads usually make a substantial number of HTTP requests to load resources at runtime, which may delay or block the delivery of publisher contents. It is well known that slow website load time costs money, as it makes visitors switch to competitor websites. Moreover, search engines recently incorporate page load time into its ranking algorithm to downgrade websites with slow loading time [23]. To this end, we define *load time bloating* to measure the delay introduced by ads. Suppose we use T and T' to represent the page load time with and without ads, a website suffers from the LTB issue if $T > 2 * T'$.

Result. We have found *61 websites experienced load time bloating problem*. Figure 7 shows the partial result on 20 websites. The red bars are the time to load ads, and the blue bars are load time of publisher contents. *The average time to fully load the publish contents is 22.91 seconds, where ads contribute 15.09 seconds.*

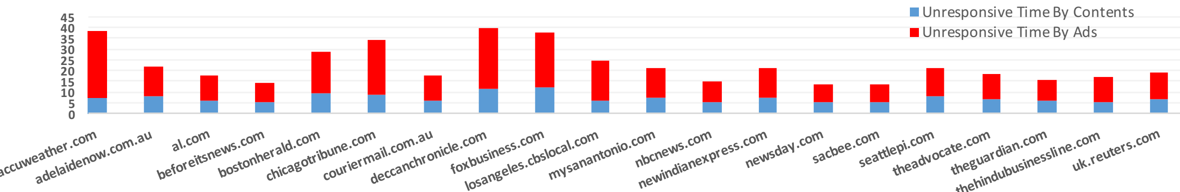


Fig. 6. 20 news websites show SRT: browser experience more unresponsive periods due to ads.

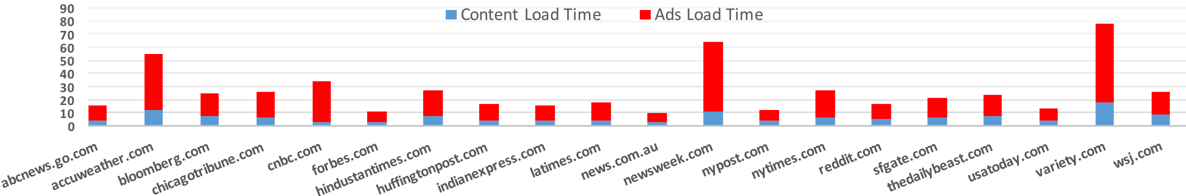


Fig. 7. 20 news websites experience LBT: web page load time dramatically increased due to ads.

2) Priority Inversions: In general, publisher contents are expected to be delivered with higher priority than ads. However, we observe that, on many websites, ads are loaded before publisher contents. Moreover, off-screen ads may delay contents or ads displayed in the current screen. As we can generally infer developer's preference from the order suggested by HTML tags, we define two types of priority inversions to capture unintended load sequences.

Content Priority Inversion (CPI). Website developers usually optimize the critical rendering path to promptly deliver their contents. However, ads may compete for the resources and get rendered first. We say *content priority inversion* happens if a higher priority publisher content loads after an ad, or formally, $\exists\ i \quad s.t. \quad T > \ T_i$, where T and T_i denote when publisher contents and ad i are loaded.

Result. *98 websites exhibited content priority inversion issue at least once during our one-month test.* These websites either have many ads or contain complex ads that needs a long time (a few seconds) to finish.

Ad Order Inversion (AOI). Publisher developers may want to specify the loading order between two ads. Although most ad serving platforms allow them to do so, the actual execution order may be different. We use *ad order inversion* to denote the scenario when two ads are loaded in a different order comparing to the order specified in the HTML source code. Formally, $r_1 \prec r_2 \quad and \quad T_1 > T_2$, where the request r_1 initiated by ad_1 is sent out before the one r_2 by ad_2 but ad_1 loaded at T_1 is rendered after ad_2 loaded at T_2 .

Result. We identified *68 websites experienced ad order inversion issue* by preloading off-screen ads prior on-screen ads. Preloading off-screen ads prior on-screen ads contradicts the order defined by developers. It is more likely developers prefer to prioritize the ads in the current view over the ads off the screen.

V. Evaluation

We evaluate how AdJust mitigate the resource abusing ads, particularly focusing on the following five research questions.

- *RQ1: Can AdJust mitigate the SRT problem and improve web page response time?*
- *RQ2: Can AdJust mitigate the LTB problem and reduce the time to load publisher contents?*
- *RQ3: By prioritizing publisher contents, does AdJust help on loading publisher contents faster?*
- *RQ4: Can AdJust ensure advertisements are rendered in the consistent order as developers expected?*
- *RQ5: How much performance overhead does AdJust introduce?*

A. Experimental Results

RQ1: Can AdJust mitigate the SRT problem and improve web page response time?

As we have shown in Section IV, out of the Alexa top 200 news websites, 36 news websites suffer from the SRT problem. To mitigate the SRT problem, we set the maximum speed of URL requests initiated by ads to be less than 20 per second by using the resource constraint "$urlRequests \leq 20$".

To ease the comparison, we show the unresponsive time before and after AdJust side by side in Figure 8. In particular, the first bar represents the unresponsive time before AdJust (red denotes ads, blue denotes publisher contents), while the second bar represents the unresponsive time after AdJust (dark blue denotes ads, grey denotes publisher contents).

Findings. After the mitigation with AdJust, on average, the total unresponsive time drops to 8.2 seconds where ads contribute 3.9 seconds. By setting a frequency cap for the URL requests initiated by ads, the main thread is blocked by ads less frequently. As the main thread is less busy with third-party ads, it will be able to process publisher contents and user interactions more smoothly.

RQ2: Can AdJust mitigate the LTB problem and reduce the time to load publisher contents?

We have identified 61 sites on which publisher contents substantially delayed due to third-party ads. By setting the maximum number of elements can be created by ads to 50 per second, the average time to load publisher contents is decreased from 22.9 seconds to 10.3 seconds. Figure 9 shows

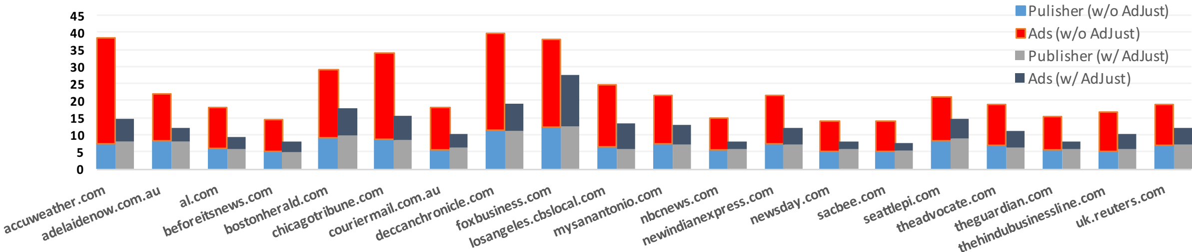


Fig. 8. Mitigation of Slow Response Time.

the result of 20 representative sites. For each site, the first bar and second bar denote the duration to load publisher contents before and after AdJust.

After the mitigation, we are interested in if the webiste with AdJust takes longer time to load ads and if so, how much additional time AdJust introduced on ads. Table II shows the load time of the top two ads above the fold on the 20 sites. Specifically, the second column shows the number of HTML elements associated with publisher contents. The third column shows publisher contents load time before and after the mitigation. Similarly, the number of HTML elements and load time are given for the first ad (columns 4-5) and the second ad (columns 6-7).

Findings. The average HTML elements created by the top two ads are 119 and 206 (if exists). By regulating HTML creations speed, it takes slightly longer to load the first ad (14.91s $\rightarrow$ 16.02s) and the second ad (17.87s $\rightarrow$ 18.63s). However, the improvement on publisher contents is significant: publisher contents are able to be loaded 2.2x faster (22.9s $\rightarrow$ 10.3s).

RQ3: By prioritizing publisher contents, does AdJust help on loading publisher contents faster?

To prioritize publisher contents over third-party ads in a web page, we assign higher priority to publisher contents and the same lower priority to all ads. Out of the top 200 news websites, 166 sites contain at least one ad tag. By prioritizing publisher contents, on average, the load time of publisher contents is significantly improved (15.25s $\rightarrow$ 8.09s). To measure the performance overhead on third-party ads, we measure the slowdowns for the top two ads in the current view.

Findings. As shown in Figure 10(a) and Figure 10(b), the load time of the top two ads are slightly increased: On 158 out of 166 sites (95%), the first ad is slowed down less than 5 seconds; for the second ad, 45 out of 49 sites (92%) are slowed down less than 5 seconds.

RQ4: Can AdJust ensure advertisements are rendered in the consistent order as developers expected?

To evaluate the effectiveness of AdJust in regulating ad loading orders, we test 68 news websites that preload the ads below the current view. Preloading ads off the screen is not an optimized strategy, because it competes network and CPU resources and delays the contents in the current view. Hence, web developers may wish to prioritize the ads in the current view over the ads off the screen.

Findings. We assign higher priority to ads above the fold and lower priority to remaining ads. As shown in Figure 11, with AdJust, the ads in the current view on all websites are loaded faster. Some of the ads are even delivered 10s earlier. In the meanwhile, by deprioritizing off screen ads, they are loaded slower on all tested websites.

RQ5: How much performance overhead does AdJust introduce?

When reducing the HTML creation rate, we observed a slight increase in time to load the top two ads (7.4% and 4.2% respectively). However, the publisher content loaded significantly faster (220%). Similarly, by limiting the ad requests speed, the top two ads were slowed down for less than five seconds.

Findings. In summary, the only runtime overhead is maintaining resource usage counters per ad iframe to delay ad executions. *No overhead is added to the publisher content.*

B. Case Study

In this section, we discuss how a defective video ad on *www.accuweather.com* caused the web page respond slowly, and demonstrate the mitigation.

The simplified code of loading the video ad is shown in Figure 12. Specifically, the website used *Google DoubleClick* to sell the ad space (line 1), which further resold this ad space to another advertising platform *PubMatic* (line 2). *PubMatic* instructed *DoubleClick* to load a video player `OvaMediaPlayer.js` to request ads (line 6). The `<div>` element with class "`_cm-video-ad`" was created to hold the video ad (line 8). During ad delegations, several ad files were considered insecure and thus blocked by the browser. In particular, the ad files used *self-signed SSL certificates* which were not verified by a trusted certificate authority. Due to this error, each ad request further incurred 307 additional URL requests on average. As a result, the website experienced slow response time for user events.

Figure 13(a) shows the page responsiveness for the first 25 seconds. For simplicity, we omit unresponsive windows shorter than 0.5 seconds. The red windows represent the unresponsive windows longer than 0.5 seconds but shorter than 1 second, while the dark red windows are unresponsive time longer

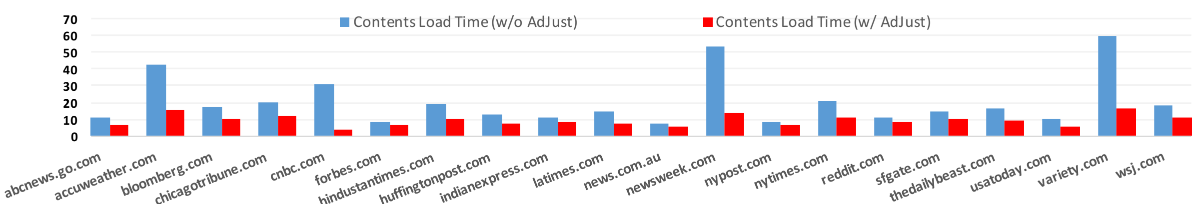
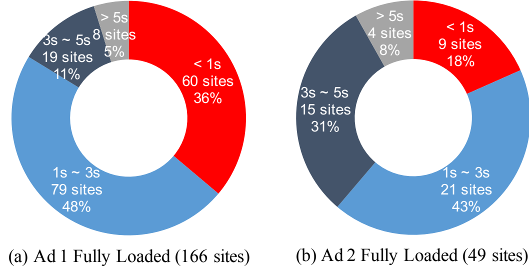


Fig. 9. Mitigation of Load Time Bloating.

TABLE II
EFFECT ON THE FIRST TWO ADS ABOVE THE FOLD.

Website	Publisher Content		First Ad		Second Ad	
	Elements	Load Time Before / After AdJust (s)	Elements	Load Time Before / After AdJust (s)	Elements	Load Time Before / After AdJust (s)
abcnews.go.com	6163	11.036 / 6.891	26	9.817 / 9.296	-	-
accuweather.com	5107	42.728 / 15.826	210	19.72 / 22.287	285	37.688 / 41.573
bloomberg.com	5077	17.034 / 10.097	67	12.785 / 13.290	-	-
chicagotribune.com	4328	20.145 / 11.727	191	13.573 / 18.621	330	18.449 / 23.782
cnbc.com	9344	30.513 / 4.292	118	15.222 / 17.403	-	-
forbes.com	3949	8.233 / 6.421	52	6.109 / 6.059	-	-
hindustantimes.com	6223	19.491 / 10.345	162	14.715 / 16.513	141	20.702 / 22.031
huffingtonpost.com	4493	12.929 / 7.599	159	9.887 / 11.354	-	-
indianexpress.com	4174	11.471 / 8.490	127	9.691 / 11.559	148	14.982 / 18.603
latimes.com	2878	14.688 / 7.100	125	12.651 / 14.270	268	13.774 / 17.006
news.com.au	3194	7.482 / 5.694	63	6.843 / 7.099	253	10.808 / 13.295
newsweek.com	2363	53.281 / 14.034	307	27.763 / 34.992	-	-
nypost.com	6038	8.661 / 6.404	256	6.983 / 10.421	-	-
nytimes.com	9776	20.68 / 10.824	189	15.002 / 16.693	-	-
reddit.com	2880	11.324 / 7.943	62	5.157 / 5.922	174	10.821 / 12.117
sfgate.com	11215	14.908 / 10.002	195	11.748 / 14.972	240	16.528 / 19.393
thedailybeast.com	1754	16.101 / 8.999	240	9.474 / 13.103	-	-
usatoday.com	5377	10.025 / 5.881	268	8.988 / 12.512	-	-
variety.com	5595	59.678 / 16.326	101	59.542 / 62.390	-	-
wsj.com	3878	17.793 / 11.498	139	14.529 / 16.788	-	-

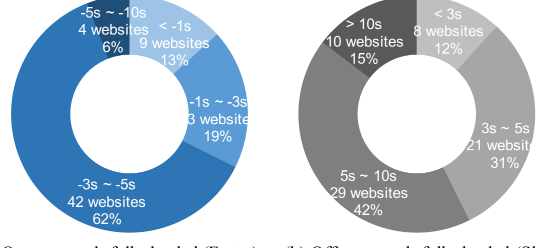


(a) Ad 1 Fully Loaded (166 sites) (b) Ad 2 Fully Loaded (49 sites)

Fig. 10. Load time differences of top two ads after mitigation with AdJust.

```
1.  <iframe src="tpc.googlesyndication.com/..." sandbox="allow-same-origin ...">
2.    <script src="ads.pubmatic.com/AdServer/js/showad.js"></script>
3.    <iframe id="aswift_0">
4.      ...
5.      <iframe src="pagead2.googlesyndication.com//show_ads_impl.js">
6.        <script src= "cdn.cmeden.com/.../v6.6.33/OvaMediaPlayer.js"></script>
7.        ...
8.        <div class="_cm-video-ad">
9.          <!-- video ad -->
10.       </div>
11.     </iframe>
12.   </iframe>
13. </iframe>
```

Fig. 12. Code snippet of a defective video ad on *accuweather.com*.



(a) On-screen ads fully loaded (Faster) (b) Off-screen ads fully loaded (Slower)

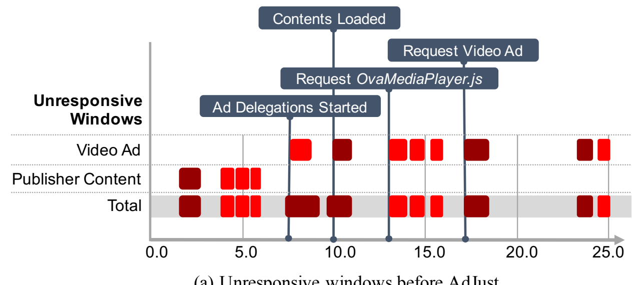
Fig. 11. Load time differences of on-/off-screen ads.

than 1 second. Initially, the web page requested ad from *DoubleClick* at 7.82 second. After two layers of delegations, the video player was requested at 13.04 second. Then the player executed JavaScript to initialize and set up event handlers for playing/pausing actions. After 17.03 seconds, the video player started to request video ads every 5 seconds, with a 200-second timeout. Since then, unresponsive windows (longer than 0.5 seconds) were observed every 5 seconds. The publisher contents loaded at around 10 second.

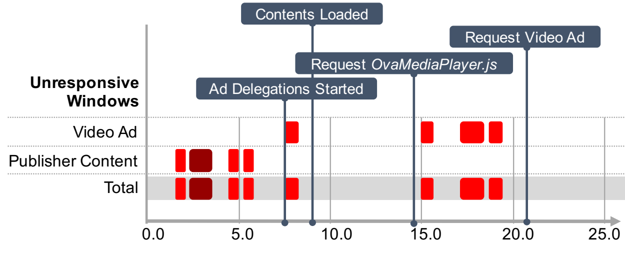
Mitigation. We deployed AdJust on this web page by setting the maximum URL requests rate to 20 requests per second. The result is shown in Figure 13(b). Because AdJust had constrained the frequency of URL requests, the video player was requested with slight delays (13.04 seconds $\rightarrow$ 14.61 seconds). After 20.62 seconds, the video player finished the initialization and started to request ads. As can be seen, the web page was rarely blocked by requesting ads and would be able to quickly respond to user events. In addition, publisher contents loaded faster due to less competitions from ads (10.02 seconds $\rightarrow$ 8.71 seconds).

VI. Limitation and Future Work

AdJust presents an effective approach for publishers to regulate resource abusing ads. Despite its efficacy, there is room for improvement.



(a) Unresponsive windows before AdJust



(b) Unresponsive windows after AdJust

Fig. 13. Website *www.accuweather.com* unresponsive windows.

AdJust requires publishers to specify resource constraints manually. While developers are able to do so without much difficulty in practice, AdJust can be extended to provide a more user-friendly experience such as automated resource consumption monitorings and recommendations. In fact, our empirical studies provide a guideline for automated measurements. Measuring client-side machine capacities to automatically set resource constraints is our ongoing work.

Another current limitation which we leave for future work is conducting user evaluations on AdJust. By deploying AdJust on websites with resource abusing ads and collecting user evaluations, we will be able to measure how effective AdJust is in improving user experience and may discover other measures that can lead to further improvements.

To reduce the threats to validity, we have carefully included and measured the most critical factors we believe truly affect user experience. We have also made efforts to maintain consistency and avoid bias. However, non-deterministic and uncontrollable factors, such as the accuracy of the measurement tool being used and the time of the day measurements were performed, may affect the validity of our empirical study. Conducting measurements under various settings to handle inaccuracy and avoid bias is an intriguing direction for our future research.

VII. Related Work

The line of work most closely related to ours are the solutions to speed up web page loads. WProf [28] is proposed to produce a dependency graph for page load activities. Shandian [29] restructures the page load process based on dependencies and optimizes what portion of the page should be loaded first. WebGaze [17] prioritizes loading objects that exhibit collective fixation to improve user experience. It leverages eye gaze to identify most attractive portions for users and optimize the loading procedure correspondingly. Klotski [9] prioritizes the content delivery based on high-utility contents or invariant dependency structures. Our work also restructures the load procedure. However, we leverage a set of orthogonal features, i.e., the source of the contents. So, the optimizations based on dependencies among page load activities, most attractive page portions or high-utility content may not directly apply to the performance issues caused by third-party ads.

Lahaie et al. [20] proposed an expressive auction design including a language for the bidding process in order to enable flexible controls over advertisements. Similarly, Lang et al. [21] propose an algorithm to improve the effectiveness of an auction process. AdJust allows publishers to control over the ads at runtime to solve resource related issues and provide better user experiences to publishers and website visitors.

Our work is also related to user experience measurements. There are large scale user studies [8], [27] that measure performance of real-world web pages. Bocchi et al. [6] surveyed state of the art metrics for web user experience quality measurements and proposed metrics relevant and practical. Google Ad Experience Report [3] helps developers locate annoying ads (e.g. popups). However, Google Ad Experience does not address the performance issues caused by third-party ads. Besides, Google Ad Experience Report is an offline tool and does not fix problematic ads. Developers have to manually fix violating ads by changing or removing ads once observed. AdJust regulates resource abusing ads and prevents over-consumption at runtime.

Researchers studied the impact of budget management strategies and pricing [5], [15] as well as the impact of ad-blocking software and restrictions on third-party tracking [7]. Goldstein et al. [11] studied relationship between ad exposure time and the effectiveness of ad. However, they do not focus on resource related issues hence are not applicable to quantify the user experience degradation cause by ads.

VIII. Conclusion

In this paper, we present AdJust, a system that allows publisher developers to specify resource constraints for third-party ads. AdJust is able to monitor and regulate resource abusing ads by transparently intercepting key JavaScript APIs. AdJust features a runtime scheduler, which can regulate resource consuming events based on the resource constraints specified by publisher developers. We also propose a system of measurement that can be used to quantify the performance degradation and priority inversion issues caused by ads. AdJust's runtime support persistently enforces the regulations by propagating through the delegations along the entire ad delivery chain. Our evaluation of Alexa top 200 news websites shows that AdJust is highly effective in mitigating resource abusing ads.

IX. Acknowledgements

We would like to thank the anonymous reviewers for their valuable inputs and constructive comments.

REFERENCES

[1] Ad Script jsvpaid.js.
http://redir.adap.tv/redir/javascript/jsvpaid.js.

[2] Ad Script vd0.2.88.1.js.
http://cdn.springserve.com/vd/vd0.2.88.1.js.

[3] Google Ad Experience Report API.
https://developers.google.com/ad-experience-report/.

[4] Amazon. Amazon.com Associates: The web's most popular and successful affiliate program.
https://affiliate-program.amazon.com/.

[5] Santiago Balseiro, Anthony Kim, Mohammad Mahdian, and Vahab Mirrokni. Budget Management Strategies in Repeated Auctions. In *Proceedings of the 26th International Conference on World Wide Web*, WWW '17, pages 15–23, Republic and Canton of Geneva, Switzerland, 2017. International World Wide Web Conferences Steering Committee.

[6] Enrico Bocchi, Luca De Cicco, and Dario Rossi. Measuring the Quality of Experience of Web Users. In *Proceedings of the 2016 workshop on QoE-based Analysis and Management of Data Communication Networks*, pages 37–42, Florianopolis, Brazil, December 2016. ACM.

[7] Ceren Budak, Sharad Goel, Justin Rao, and Georgios Zervas. Understanding Emerging Threats to Online Advertising. In *Proceedings of the 2016 ACM Conference on Economics and Computation*, EC '16, pages 561–578, Maastricht, The Netherlands, 2016. ACM.

[8] Michael Butkiewicz, Harsha V. Madhyastha, and Vyas Sekar. Understanding Website Complexity: Measurements, Metrics, and Implications. In *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference*, IMC '11, pages 313–328, Berlin, Germany, 2011. ACM.

[9] Michael Butkiewicz, Daimeng Wang, Zhe Wu, Harsha V. Madhyastha, and Vyas Sekar. KLOTSKI: Reprioritizing Web Content to Improve User Experience on Mobile Devices. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, pages 439–453, Oakland, CA, USA, 2015. USENIX Association.

[10] Eyeo GmbH. Adblock Plus.
https://adblockplus.org/.

[11] Daniel G. Goldstein, R. Preston McAfee, and Siddharth Suri. The Effects of Exposure Time on Memory of Display Advertisements. In *Proceedings of the 12th ACM Conference on Electronic Commerce*, EC '11, pages 49–58, San Jose, California, USA, 2011. ACM.

[12] Google. DoubleClick – Digital Advertising Solutions.
https://www.doubleclickbygoogle.com/.

[13] Google. Google AdSense – Make Money Online through Website Monetization.
https://www.google.com/adsense/.

[14] Google. WebPagetest – Website Performance and Optimization Test.
https://www.webpagetest.org.

[15] Hoda Heidari, Mohammad Mahdian, Umar Syed, Sergei Vassilvitskii, and Sadra Yazdanbod. Pricing a Low-Regret Seller. In *Proceedings of the Thirty-Third International Conference on Machine Learning (ICML 2016)*, New York, NY, USA, 2016.

[16] Natalia Juristo and Ana M. Moreno. *Basics of Software Engineering Experimentation*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[17] Conor Kelton, Jihoon Ryoo, Aruna Balasubramanian, and Samir R. Das. Improving User Perceived Page Load Times Using Gaze. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 545–559, Boston, MA, 2017. USENIX Association.

[18] Kissmetrics. How Loading Time Affects Your Bottom Line.
https://blog.kissmetrics.com/loading-time/, 2011.

[19] Barbara A. Kitchenham, Shari Lawrence Pfleeger, Lesley M. Pickard, Peter W. Jones, David C. Hoaglin, Khaled El Emam, and Jarrett Rosenberg. Preliminary Guidelines for Empirical Research in Software Engineering. *IEEE Trans. Softw. Eng.*, 28(8):721–734, August 2002.

[20] Sébastien Lahaie, David C. Parkes, and David M. Pennock. An Expressive Auction Design for Online Display Advertising. In *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 1*, AAAI'08, pages 108–113. AAAI Press, 2008.

[21] Kevin J. Lang, Benjamin Moseley, and Sergei Vassilvitskii. Handling Forecast Errors While Bidding for Display Advertising. In *Proceedings of the 21st International Conference on World Wide Web*, WWW '12, pages 371–380, Lyon, France, 2012. ACM.

[22] StreamRail Ltd. Streamrail Ad Server.
https://sx.streamrail.net/spot?scid=182172&p=http%3A%2F%2Fchicagotribune.com&cb=51425144694422030000&ctrl=true&muted=false.

[23] moz.com. Page Speed.
https://moz.com/learn/seo/page-speed.

[24] Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann Publishers Inc., San Francisco, CA, 1993.

[25] npm Software. Htmlparser2.
https://www.npmjs.com/package/htmlparser2.

[26] Akamai Technologies. Akamai's [state of the internet] Q1 2017 Report.
https://www.akamai.com/fr/fr/multimedia/documents/state-of-the-internet/q1-2017-state-of-the-internet-connectivity-report.pdf.

[27] M. Varela, L. Skorin-Kapov, T. Mäki, and T. Hoßfeld. QoE in the Web: A Dance of Design and Performance. In *2015 Seventh International Workshop on Quality of Multimedia Experience (QoMEX)*, pages 1–7, May 2015.

[28] Xiao Sophia Wang, Aruna Balasubramanian, Arvind Krishnamurthy, and David Wetherall. Demystifying Page Load Performance with WProf. In *Presented as Part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI'13)*, pages 473–485, Lombard, IL, 2013. USENIX Association.

[29] Xiao Sophia Wang, Arvind Krishnamurthy, and David Wetherall. Speeding up Web Page Loads with Shandian. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI'16)*, pages 109–122, Santa Clara, CA, 2016. USENIX Association.

[30] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Bjöorn Regnell, and Anders Wesslén. *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.