Journal of Web Engineering, Vol. 16, No. 3&4 (2017) 311–346

LOAD-TIME REDUCTION TECHNIQUES FOR DEVICE-AGNOSTIC WEB SITES

EIVIND MJELDE[a]
Dept. Information Science and Media Studies, University of Bergen
P.O.Box 7802, N-5020 Bergen, Norway
Eivind.Mjelde@bouvet.no

ANDREAS L. OPDAHL
Dept. Information Science and Media Studies, University of Bergen
P.O.Box 7802, N-5020 Bergen, Norway
Andreas.Opdahl@uib.no

Received June 22, 2016
Revised January 5, 2017

Modern *device-agnostic web sites* aim to offer web pages that adapt themselves seamlessly to the front-end equipment they are displayed on, whether it is a desktop computer, a mobile device, or another type of equipment. At the same time, mobile devices and other front-end equipment with limited processing powers, screen resolutions, and network capacities have become common, making *front-end performance optimisation* in general, and *load-time reduction* in particular, a central concern. The importance of load-time reduction is exacerbated by the proliferation of multimedia content on the web. This paper therefore reviews, evaluates, and compares available load-time reduction techniques for device-agnostic web sites, grouped into techniques that improve client-server communication, optimise UI graphics, optimise textual resources, and adapt content images to context. We evaluate the techniques on a case web site using both desktop and mobile front-ends, in a variety of settings, and over both HTTP/1.1 and HTTP/2. We show that each technique has its pros and cons, and that many of them are likely to remain useful even as HTTP/2 becomes widespread. Most techniques were clearly beneficial under at least one of the conditions we evaluated, but most of them were also detrimental in certain cases — sometimes drastically so. Hence, load-time reduction techniques for device-agnostic web sites must be selected with care, based on a solid understanding both of usage context and of the trade offs between the techniques.

Keywords: Adaptive Images, Clown-Car Technique, content-delivery network, domain sharding, HTML 5.1 pictures, HTTP/2, image sprites, lazy loading, Picturefill, responsive images, responsive web design, symbol fonts, web-browser load time, web-resource compression, web-resource concatenation, web-resource minification.
Communicated by: G-J Houben & M. Norrie

1. Introduction

More and more internetworked devices are being introduced to the consumer and professional markets, ranging from mobile phones and tablets through laptops and desktop computers to smart TVs and other online appliances. The sale of mobile phones passed that of desktop

[a]Current address: Bouvet, Solheimsgaten 15, N-5058 Bergen, Norway

and laptop computers already at the end of 2010, and the proliferation *mobile devices* like smartphones and tablets has been even stronger since [1]. Central actors in the web market, such as Google, Facebook and Adobe, have increased their focus on mobile devices [2]. Today, such mobile devices also play an important role in society as an inexpensive way for hundreds of millions of people worldwide to become global digital citizens. People in developing countries and new immigrants to industrial societies are often *mobile only*, their mobile devices being their sole point of entry into the information society.

In the early days of the web, pages were designed with rigid one-size-fits-all layouts that targetted standard desktop monitor resolutions.[b] As mobile devices with smaller screens became common, many web-site providers responded by offering multiple versions of their web sites, each version targetting a particular type of device. A mobile device would be redirected to a mobile subdomain (a so-called *M Dot* web site [3]), whereas a desktop computer would access the default domain. But today the diversity of front-end equipment has increased further. Tablets and high-end smartphones blur the distinction between device types. They have high-resolution screens and run advanced operating systems like Android and iOS, with browser engines very similar to their desktop counterparts, albeit with weaker processing and networking capabilities. They have made the transition between mobile and desktop front ends seamless, and the versioned approach to serving different front ends has become inadequate.

Instead, modern web sites attempt to be *device agnostic*, offering pages that *adapt to the front-end equipment they are displayed on*, and possibly also to the wider *usage context:* whether the user is in a silent or noisy place, in a light or dark spot, at home or at work, etc. Hence, the same set of pages aim to cater to the needs of all kinds of front-end equipment [4]. This is a positive development because it ensures that mobile-only users can access whole web sites, and not only limited, pre-selected subsets. But offering the same web pages to all types of front-end equipment also creates performance challenges. In particular, users of low-end mobile devices suffer from limited CPU power and memory size resulting, among other things, in slower page rendering and smaller browser caches. Mobile off-loading [5, 6] is not an option because web-page rendering has to respond rapidly to end-user interactions and to changes in usage context. Even high-end mobile devices suffer from the increased latency and reduced bandwidth of mobile-phone (cellular) networks that involve radio communication through base transceiver stations (cell towers).

These performance challenges are exacerbated by the current proliferation of multimedia, in particular of graphics-heavy web sites. Reducing load times has therefore become crucial, in particular on mobile devices. The purpose of this paper is to review, evaluate, and compare the various load-time reduction techniques that are available. We will review the alternatives and evaluate how the techniques affect load times in desktop and mobile settings. Because images often constitute the bulk of a page's size, we will pay them particular attention. We will answer three research questions:

(i) Which techniques are available to reduce the load times of device-agnostic web sites?
(ii) What are the advantages and disadvantages of each technique on desktop and mobile front ends?
(iii) How much does each technique improve the load time of a real web site?

[b] We will use the term *desktop* to denote the screens of both regular desktop and laptop computers.

We think these questions are topical because central web standards and protocols ares currently undergoing major changes. Just as HTML5 and CSS3 have stabilised, HTML 5.1 and HTTP/2 are becoming increasingly common. It is particularly interesting to investigate how these new technologies impact the load times of device-agnostic web sites. For example, do the new solutions offered by HTTP/2 and HTML 5.1 render existing techniques obsolete, or will the existing and new techniques continue to co-exist alongside one another? We consider load times to be a central performance measure because it affects end users most directly. But we readily acknowledge that other measures (e.g., [7]), such as CPU demand and its effect on power consumption, also need to be studied in further work, a point we will return to in the discussion.

The rest of paper is organised as follows: We first present relevant literature on device-agnostic web design. We then review available techniques for reducing the load times of web sites, highlighting their potential advantages and drawbacks. We then describe how we have evaluated the load times of each technique empirically, before presenting the results. Finally, we discuss our findings and conclude the paper, offering suggestions for further research.

2. Theory

Device-agnostic web design is not a completely new idea. Several similar approaches have been proposed and practised over the years.

Graceful degradation is an early practice that focusses on building web sites for the *most advanced browsers first*, while trying to provide an acceptable user experience for older browsers too.

Progressive enhancement instead focusses on *content first*, making sure that content is accessible and usable by anyone, regardless of location, device capabilities, or user disabilities [8]. The term was coined in [9], which promotes web design that targets the "least capable or differently capable devices first, then moves on to enhance those documents with separate logic for presentation, in ways that do not place an undue burden on baseline devices but which allow a richer experience for those users with modern graphical browser software". This is achieved through rich HTML markup which ensures that basic content can be viewed even in browsers that do not support more advanced markup, e.g., using CSS [10].

Mobile first focusses on developing web sites *under the constraints of mobile devices first*. This approach gained increasing attention when the sale of mobile phones passed that of desktop computers at the end of 2010 [11] and Google adopted it as an official development strategy [12]. Typical constraints addressed in mobile first include screen size, hardware capabilities, and other factors such as time and location. Of these, screen size has played a central role from the start, because the first smartphones with standards-compliant browsers had screens that were approximately 80% narrower than the average desktop computer screen [2, Chapter 2].

Responsive web design attempts to consolidate the code bases of web sites through *context-aware* pages that are able to adapt themselves to the front-end equipment they are

displayed on. While the term "responsive" is often used in performance evaluation contexts to mean "short response time", this is not the meaning intended here. Instead, responsive design of web sites "suggests that design and development should respond to the user's behaviour and environment based on screen size, platform and orientation" [13]. The resulting *device-agnostic web sites* remove the need for providing different variants of pages for specific processors, screens, or networks. From the start, responsive web design focusses in particular on screen layout and user interaction (UX), for example on ordering and scaling content for different screen sizes and on making navigation elements, forms and tables usable on small screens. Fluid grids, flexible images and CSS media queries are three important techniques for creating responsive designs [14]. More recently, new CSS rules, such as CSS3's Flexible Box Layout Module, and front-end frameworks like Bootstrap[c] and Foundation[d] have also made this easier. Hence, images is a central, but not the only, concern in responsive web design.

The importance of images has grown as multimedia content has become ubiquitous on the web. In February 2014, images comprised on average 62% — or 1040kB of 1687KB total — when a web page was downloaded to a desktop computer [15], an increase from 59% in November 2010 [16], indicating that the download sizes of graphic content continues to grow compared to other static content.[e] Analysing a collection of 471 responsive web sites [17], images contributed several times more to total download sizes than HTML, scripts, style sheets, and other resources combined. The results are consistent with ones reported in [18]. In the sample of pages surveyed, the numbers of HTTP requests did not vary much across resolutions: of 41 total requests made on average, 23 were for image resources, and the images downloaded to low-resolution screens were only slightly smaller than the ones for higher resolutions [17]. That file sizes and numbers of requests were so similar across different resolutions suggests that the resources are not optimised for different screens and devices. This is surprising, given that adapting image sizes to screen resolutions is a central idea behind the responsive web sites analysed: the situation may be even worse on the non-responsive web.

3. Review of Optimisation Techniques

This section answers our first and second research questions through a review and discussion of existing techniques for optimising the front-end performance of web sites.

3.1. *Method*

We have included all the techniques that satisfy the following two criteria: (1) they must aim to reduce the load times of web sites, and (2) they must be usable on both desktop computers and mobile devices.

Because images constitute a large portion of web resources and pose particular challenges for mobile devices, we have looked for optimisation techniques for images in particular. We have subdivided images into two distinct types that tend to be targetted by different optimisation techniques: *UI graphics* are part of the user interface (UI) of a web page and do not change often. Examples are icons, logos, buttons, thumbnails and background images. UI

[c] `http://getbootstrap.com/`
[d] `http://foundation.zurb.com/`
[e] The word static is used here to differentiate from streaming content such as audio and video.

graphics are typically reused by many of the pages on a site, often in groups, and they are often smaller than 50x50 pixels, but can be larger. In contrast, *content images* are part of the primary content of a web page. Content images are specific to a page, and usually change when the other content of the page changes. They are often larger than, e.g., 50x50 pixels, but can be smaller.

Few academic research papers have been written about front-end performance optimisation for the web. Systematic methods for identifying and reviewing the existing literature, such as the ones offered by Kitchenham [19] for software engineering studies, are therefore less useful for our purposes. Instead, our review draws on a wider selection of white papers, books, articles and blog posts written by organisations and people in the web-development community. In doing so, we attempt provide a first compilation of all the available knowledge in academic form.

In the end, we have identified twelve major techniques, which we have subdivided into the following four groups:

- *Improving server connections:* content-delivery networks (CDNs) and domain sharding
- *Optimising UI graphics:* sprites and symbol fonts
- *Optimising textual resources:* concatenation, minification and compression
- *Adapting content images to context:* Picturefill, Adaptive Images, the Clown-Car Technique, HTML 5.1 pictures, and lazy loading

Some of the techniques will be further combined or refined, so that the evaluation section will compare up to 28 variants of the baseline web page.

3.2. *Improving server connections*

The techniques in this group attempt to improve load times by improving communication between the client (or front end) and the server. We have identified two techniques that aim towards this goal: *domain sharding* and use of a *content-delivery network (CDN).*

3.2.1. *Domain sharding*

Domain sharding increases browser parallellisation by "distributing page resources across multiple domains (servers), allowing browsers to open more parallel connections to download page resources" [20, p. 15]. The technique works around the cap in HTTP/1.1 on simultaneous connections to the same domain: its specification states that "[a] single-user client SHOULD NOT maintain more than 2 connections with any server or proxy" [21, section 8.1.4], meaning that no more than two concurrent TCP connections are allowed at the same time per domain. When HTTP/1.1 became an official standard in 1999, this was considered a reasonable number, but web sites have since grown in both size and complexity, with each page embedding more additional resources. The maximum of two simultaneous connections came to limit web-page performance, and modern web browsers have therefore increased the cap, most of them allowing up to six connections per domain today [22, p. 44][23].

According to [22], a web page downloads resources from 16 different domains on average, many of them coming from third party providers of ads, widgets, and analytics. The majority of web sites download 39 or more resources from the domain they rely most heavily on (the average is 50 resources from the most used domain), and the median number of resources

downloaded from a single domain is 39 [22]. A browser limited to six parallel connections to a single domain will need a sequence of seven requests per connection to download all resources from such a domain, making it a good candidate for sharding because splitting the requests over two domains or more increases parallellisation. However, there is a trade-off involved, because splitting resources across additional domains induces additional time to set up new connections (DNS resolution, three-way handshake, increasing the congestion window, etc.) and consumes processing resources on the client and server sides.

Because HTTP/2 multiplexes request-response pairs over the same TCP connection, we might expect HTTP/2 to benefit less — or not at all — from domain sharding.

Advantages The main advantage of domain sharding is *increased browser parallellisation*. For pages with many embedded resources hosted on a single domain, domain sharding can improve download times by increasing the number of resources downloaded in parallel. Faster downloading of scripts and style sheets is also beneficial for web-page rendering times, because web-page rendering can be *blocked* by both. Domain sharding is also *easy to implement*. It is not necessary to split content across servers in order to shard a domain. Creating domain aliases (using CNAMEs) is sufficient, as the browser only limits connections to host names, not IP addresses [22].

Drawbacks Research indicates that, in most cases, resources should not be sharded across more than two domains [22, p. 541][24, 25]. This is because aggressive sharding induces *performance costs of DNS lookups* for each new domain, as well as *increased CPU and memory costs* on both client and server [22, p. 44][23]. This can impact performance negatively and increase *network congestion* [24, p. 44][23]. The *penalty of opening additional connections* might not warrant the increase in parallellisation.

Successful domain sharding *requires thorough analysis* of which resources that are frequently requested around the same time, to determine which resources that should be split into different shards. Caching may make the need for sharding *difficult to assess*: some pages might request images that are used by many other pages on the same web site (and which are therefore likely to be cached by the browser), whereas other pages may request mostly unique images (which are unlikely to be cached).

3.2.2. *Content-delivery networks*

One way of reducing request and response times between client and server is to reduce the distance between them, where distance is measured in *topological proximity*, which is "an abstract metric that considers a variety of criteria, such as physical distance, speed, reliability, and data transmission costs" [26], encompassing both physical signal distances and numbers of intermediate routers. A content-delivery network (CDN) reduces topological proximity through a network of *surrogate servers* distributed around the world. Their job is to cache contents from the origin server, which remains the main server through which the network gets updated. When content is cached at multiple locations around the world, requests for a particular page can be directed by the domain-name system (DNS) to the topologically closest IP address. CDNs are most often used to deliver static content, such as scripts, style sheets, and images [27, p. 20].

Advantages The primary advantage of CDN is that it *reduces round-trip times between client and server*. By bringing resources topologically closer to clients, the network allows resources to be downloaded faster by reducing round-trip times. By distributing load, it also *reduces load* both on the origin and each surrogate server, possibly decreasing their response times. It also increases dependability by providing *multiple fallback locations for resources*: if one surrogate server becomes unavailable, the CDN may redirect requests to another. And by distributing resources across domains, CDNs also be advantageously combined with *domain sharding*.

Drawbacks CDN *increases the complexity of resource distribution*, because the mechanics for distributing resources to surrogate servers need to be rigorous and fast, and the caching mechanisms between origin and surrogate servers must be reliable. These tasks are therefore often left to dedicated CDN providers, resulting in *lack of control* for the content provider. Another resulting disadvantage is *cost*, because the most used networks are commercial (although there are free alternatives, such as `CloudFlare.com`). Finally, like domain sharding, CDNs induce *additional DNS lookups and connection delays* when resources are split across domains.

3.3. *Optimising textual resources*

Textual resources include — but are not limited to — HTML code, scripts, and style sheets. This section considers three techniques for optimising textual resources: *concatenation*, *minification* and *compression*, which are often used in combination.

3.3.1. *Concatenation*

The number of HTTP requests required to load the resources in a web page can have a big impact on performance. This issue is exacerbated on mobile-phone (cellular) networks because of the delays inherent in connecting to base stations. The number of HTTP requests should therefore be reduced as much as possible.

Concatenation combines multiple resources into a single, larger file, reducing the numbers of HTTP requests and of subsequent server round trips needed to fetch the resources. [28] notes that "[t]he best way to combat latency is for a web site or application to use as few HTTP requests as possible. The overhead of creating a new request on a high-latency connection is quite high, so the fewer requests made to the internet, the faster a page will load." Concatenation of textual resources is most frequently applied to JavaScript and CSS files. It can be done both in production, before the resources are uploaded to the server, or dynamically on the server.

Advantages The advantage of concatenation is that it *reduces the number of HTTP requests* needed to download a set of resources.

Drawbacks Concatenated files have *longer download and parse times*, which can *delay page rendering* when the concatenated file contains CSS or blocking JavaScripts. Concatenation also *breaks modularity* by merging several resources and thus *impacts caching negatively*.

When several files are combined, *maintenance becomes more complex* because editing a single file requires the concatenated file to be recreated [23]. Concatenation should therefore be reserved for files that change infrequently and are loaded together. Concatenated textual resources also incur *memory and CPU costs* [23].

3.3.2. *Minification*

Minification removes whitespace, comments, and characters that are not critical for execution of code. The term is also used for making code smaller, for example by shortening variable names in JavaScript and rewriting CSS to use shorthand properties[f]. Minification can be automated, both on the server side (e.g., Google's PageSpeed Module[g]) and by locally installed applications (e.g., Grunt[h] and Compass[i]).

Advantages Minification *reduces download sizes* by removing characters and code segments that are not necessary for execution. JavaScripts can be reduced as much as 20% through minification [30, p. 39]. For standard libraries like jQuery, file-size reductions can be an order of magnitude.

Drawbacks *File-size reductions may be small* unless the sources themselves are large. If the files are already compressed (e.g., using `gzip`) when they are downloaded from the server, the *load-time reduction can be negligible compared to that of compression.* [31, p. 8] explains that minification of scripts also causes *code obfuscation*, which can be considered beneficial in some circumstances, but which reduces readability and *makes debugging more difficult*, unless the original non-minified scripts remain available. To remedy this problem, *source mapping* is available in many browsers, so that additional files are downloaded when debugging to recreate the original source code from its minified version [32].

3.3.3. *Compression*

Load times can be improved by compressing the resources that are sent from the server to the client. Compression is much used because it is broadly support both on the server and client sides. The most common schemes are `gzip` and `deflate` (zlib), which both use variants of the lossless LZ77 (Lempel-Ziv 1977) compression algorithm, which looks for repeating string patterns in text. When occurrences of the same string are encountered, the repeating occurrences of the string are replaced by a pointer to the first occurrence, using a pair that contains the distance to the original string along with its length [33]. HTML, JavaScript, and CSS are good candidates for compression because they tend to contain many repeated strings [31, 34].

[35] shows that pre-compression is always preferable to real-time compression. One should therefore always try to pre-compress as many static resources as possible. But this is not always achievable, because many modern web sites generate content dynamically.

[f] "Shorthand properties are CSS properties that let you set the values of several other CSS properties simultaneously. Using a shorthand property, a Web developer can write more concise and often more readable style sheets, saving time and energy" [29]
[g] `https://developers.google.com/speed/pagespeed/module`
[h] `http://gruntjs.com/`
[i] `http://compass-style.org/`

```
<!-- CSS -->
<style>
.arrow  { background-image: url(arrows.png);
          height: 10px;
          width: 10px; }
.up     { background-position: 0 -10px }
.down   { background-position: 0 -20px }
.left   { background-position: 0 -30px }
.right  { background-position: 0 -40px }
</style>

<!-- HTML -->
<ul>
        <li class="arrow up"></li>
        <li class="arrow down"></li>
        <li class="arrow left"></li>
        <li class="arrow right"></li>
</ul>
```

Fig. 1. Simple use of an image sprite.

Advantages The advantage of compression is that it *reduces the file sizes* of textual resources such as HTML code, scripts, and style sheets. Because it is built into modern web servers and browsers, it is one of the *simplest load-time reduction techniques to implement*, and it *gives good results* [36]. Gzip typically reduces the size of textual files by 60–80%.

Drawbacks A disadvantage is that compression standards like `gzip` and `deflate` *only work well on textual resources.* Multimedia content such as images, video, and audio are usually stored and transferred in formats that are already compressed.

3.4. *Optimising UI graphics*

As we explained earlier, UI graphics are image elements that belong to the user interface of a web page, such as icons, logos, buttons, thumbnails, and background images. They are usually smaller than content images, change less frequently, and tend to be reused by many pages of the same web site, often in groups.

This section considers two techniques for optimising UI graphics: *image sprites* and *symbol fonts*. Both aim to reduce numbers of HTTP requests by concatenating multiple UI-graphical resources into a single file. Although they can thus be considered subtypes of concatenation — as discussed in the previous section — we treat them separately because they are more specialised techniques to which specific considerations apply.

3.4.1. *Sprites*

Sprites concatenate several smaller images into a single larger one. When the sprite is downloaded, the browser picks out the original images again using their coordinates inside the sprite in combination with CSS styling. Figure 1 shows CSS and HTML code that defines and uses four 10 by 10 pixel arrow icons contained in the same sprite file, `arrows.png`. Instead of loading each image separately using four HTTP requests, the browser downloads all four icons in a single file, but treats the four areas inside it as distinct icons. While sprites can in principle be used with arbitrarily large images, caching and bandwidth issues make them most effective for *UI graphics* because: they are small (so that relative overhead per file is

larger), they are often used together (so that unused images are not downloaded too often as part of the sprite), and they seldom change (so that caching problems are reduced).

Advantages The advantage of sprites is that it leads to *fewer HTTP requests*. The more images that can be combined into an image sprite, the more HTTP requests can potentially be saved.

Drawbacks Sprites increase *development and maintenance complexity* because they require file concatenations and coordinate mappings that can be time consuming, although tools like Compass[j] can automate much of the process. Sprites *only work with background images*, not with inline image elements (`<img>`), meaning that it cannot be made clear from HTML markup alone whether a sprite is primary content on a page (this is another reason why we suggest sprites primarily for UI graphics). Because sprite coordinates are explicitly stated in pixel values, they are also *harder to implement in responsive designs*. Fluid sizing is harder to implement for sprites, and loading differently-sized sprites under different conditions would increase complexity.

Sprites also inherit the disadvantages of concatenation in general: they incur *CPU decoding and memory costs* and, because an image sprite combines several images into a single file, *images cannot be cached individually*. Editing a single image in the sprite would therefore require the entire sprite to be re-cached. Optimal use of sprites thus *requires images that change infrequently* and that are *frequently loaded together*. If images in the same sprite are never used on the same pages, there is no performance gain. Sprites should therefore be reserved for frequently concurring images.

3.4.2. *Symbol fonts*

Symbol fonts have been around since the early 90-ies when Microsoft released its Wingdings font, a collection of what is known in typography as *dingbats* — ornamental characters used in typesetting for symbols and shapes instead of alphabetical and numerical characters. More recently, symbol fonts have been used as a load-time reduction technique to concatenate several vectorised graphics images into a single font file. Symbol fonts are often used for UI-graphics elements, such as buttons and navigation bars.

Advantages The primary advantage of symbol fonts is again that *multiple vector graphics can be stored in a single font file*, reducing the number of HTTP requests needed. Symbol fonts represented as vector images are also *infinitely scalable*, which is useful for responsive designs because fonts can easily be resized for different screen sizes. They are also *easily customisable* because they can use the same styling as other CSS fonts, making it easy to change their colour and size. *Special font properties*, such as `text-shadow` [37], can be applied, making symbol fonts versatile. Symbol fonts *work on many older browsers*, and fallback fonts may be available whenever vector formats are not supported [37, Chapter 3]

Drawbacks Symbol fonts *depend on CSS parsing*. They also need to be declared in CSS using the `@font-face` rule. Either the CSS needs to be downloaded and parsed before the

[j] `http://compass-style.org/reference/compass/helpers/sprites/`

symbol font can be downloaded, or the symbol font must be inlined in the CSS. In both cases, *rendering is delayed*. Although vector fonts are becoming available that support colour through the OpenType initiative[k] most vector fonts still have *limited colour options*, so that each use of a symbol font graphic (or grapheme) must be monochromatic. As for sprites, there are the usual concatenation drawbacks, including *processing and memory cost* of decoding the full symbol font and storing it in memory, even when only a few of the images are rendered on the screen, and graphemes *cannot be cached individually*. If the UI graphics are available as raster images, they *must be converted to vector graphics format* before they can be included in a vector font, a process that is time consuming and may reduce graphics quality.

3.5. *Adapting content images to context*

W3C defines a responsive image as "an image that adapts in response to different environmental conditions: adaptations can include, but are not limited to, changing the dimensions, crop, or even the source of an image" [38]. Responsive-image techniques aim to select image resources that match device properties, such as pixel density, orientation, and screen size; that match a medium, such as `screen` or `print`; or that match particular browsing conditions.

Because adapting images to context incurs an overhead — in the least deciding which among several images to load — it is most effective for larger images. We therefore discuss responsive images primarily as a technique for what we previously have called content images (as opposed to UI graphics), i.e., to larger images that are part of the primary content of a web page and that usually change when the other content of the page changes. However, responsive-image techniques can be used for UI graphics too: a particularly common technique is *responsive sprites*, i.e., downloading different sprites depending on screen size and type.

An early, straightforward approach to responsive images is *flexible images*, which uses the simple CSS2.1 rule `img { max-width: 100%; }` to ensure that an `<img>` element will never stretch beyond the full width of its parent element. In most cases, the dimensions of the image will also be maintained independently of the dimensions of other elements in the layout. Unfortunately, this technique encourages use of the same (large) images for all layouts, a practice that punishes load times on low-end devices severely.

This section therefore discusses five better-performing techniques for responsive images: *Picturefill*, *Adaptive Images*, *the Clown-Car Technique*, *HTML 5.1 pictures*, and *lazy loading*.

3.5.1. *Picturefill*

Picturefill is a client-side JavaScript-based technique that enables responsive images, proposed in 2012 by Scott Jehl [39]. This and the following two techniques can all be seen as workarounds that address problems to which HTML, at least until the arrival of HTML 5.1 pictures, did not offer native solutions.

Picturefill loads images dynamically based on *CSS3 media queries*, which extend the media type rules introduced in CSS2. They let different style rules take effect and different resources be downloaded depending on context, such as which media types the browser supports, whether the page is for viewing or printing, whether screen orientation is portrait or landscape, and whether the screen is bigger or smaller [40]. The screen size is controlled through

[k] See for example `http://www.w3.org/community/svgopentype/`.

```
<span data-picture
      data-alt="A stone face at The Bayon temple in Angkor Thom">
    <span data-src="small.jpg"></span>
    <span data-src="medium.jpg"     data-media="(min-width: 400px)"></span>
    <span data-src="large.jpg"      data-media="(min-width: 800px)"></span>
    <span data-src="extralarge.jpg" data-media="(min-width: 1000px)"></span>

    <!-- Fallback content for non-JavaScript browsers.
         Same img src as the default Picturefill img -->
    <noscript>
        <img src="small.jpg"
             alt="A stone face at The Bayon temple in Angkor Thom">
    </noscript>
</span>
```

Fig. 2. Basic HTML markup for Picturefill (from [39]).

```
<span data-src="specific.jpg"
      data-media="(min-width: 568px) and
                  (min-device-pixel-ratio: 2.0) and
                  (orientation: landscape)">
</span>
```

Fig. 3. Chained media query specificity.

the browser's *viewport width*, i.e., the width of the visible part of a web page inside the browser window. In particular, media queries let web designers specify *breakpoints*, so that different images can be loaded for different viewport ranges (media queries do not support smooth image scaling) [41]. For example, the rule `@media screen and (max-device-width:480px)` can be used to load a resource only on small screens.

Figure 2 shows the necessary markup to make the script work. All the responsive-image code is contained inside a `<span>` element marked with a `data-picture` attribute. The outer `<span>` element contains several inner `<span>` elements, each with a `data-src` attribute that stores the path to an image resource and possibly a `data-media` attribute that stores a media condition (a CSS3 media query) that triggers download of that image. These and the other `data-*` attributes are hooks for the `Picturefill.js` script. As soon as a media condition stored in a `data-media` attribute evaluates to true, the corresponding image is downloaded. The script then replaces the corresponding `<span>` with an `<img>` element, which is annotated with the correct `src` and `alt` attributes. In this example, `small.jpg` is loaded as a default if none of the specified media conditions are evaluated to true, or if JavaScript is disabled in the browser, in which case the markup inside the `<noscript>` element is used. If the media condition were to change after the initial page load, for example because the browser's viewport or the device's orientation is changed, download of an alternative image can be triggered, replacing the already loaded image.

`<img>` elements are not used in Figure 2 because an `<img>` element without a `src` attribute is invalid HTML and because an `<img>` element with a specified `src` will always trigger a download, regardless of media conditions (unless it is the child of a `<noscript>` element, in which case it will only download if scripting is disabled).

Advantages A major advantage of Picturefill is that it is *versatile*, supporting many of the use cases for responsive images described by the W3C [38]. It owes much of its versatility to *media queries*. Each main `<span>` element can contain as many child `<span>` elements (image

```
<script>
    document.cookie='resolution='+window.innerWidth+'; path =/';
</script>
```

Fig. 4. Writing the AI session cookie (according to [43]).

containers) as needed. In addition, each `data-media` attribute can *chain multiple media queries* to increase specificity. For example in Figure 3, the browser would only download an image on a mobile or tablet device held in landscape orientation mode with a minimum viewport-width of 568 pixels and a device-pixel ratio (DPR) of two or higher. Picturefill's media-query support means that it *supports several types of image selection*: both resolution-based, viewport-based, and DPR-based. The fact that different images can be specified for each media condition means that *art direction is supported*.

Drawbacks Disadvantages of Picturefill include *extra processing time* and an *additional HTTP request* needed for the `Picturefill.js` script, which also *increases download size*. The technique is *JavaScript-dependent*, making it *necessary to set fallback image sources* using the HTML `<noscript>` element. Making images script dependent also *hinders indexing by web crawlers*. Because the responsive image logic is dependent on the `Picturefill.js` JavaScript, *image downloads must wait on script downloading, parsing and execution*, so that images cannot be detected by speculative parsing.

Picturefill also requires *time-consuming image creation and referencing* by the content provider if done manually, when each image needs to be duplicated, scaled or cropped, its path needs to be referenced, and a media query written. Of course, when art direction is not needed, all or most of the process can be automated, for example by a content management system (CMS). Picturefill *breaks the "Don't Repeat Yourself" (DRY) principle*, which states that "every piece of system knowledge should have one authoritative, unambiguous representation" [42], because breakpoints may have to be declared twice: both in HTML and CSS.

3.5.2. *Adaptive Images*

Adaptive Images (AI) is a server-side technique for responsive images written in PHP, proposed by Matt Willcox in 2011. Adaptive Images was created to offer a responsive-image technique that could be automated on the server side.

The first thing that needs to be in place for AI to work, is a method that can identify the screen resolution of the device to the server. This is done by writing the resolution to a session cookie using JavaScript. Figure 4 shows one way of doing this, using viewport width as a measure of resolution. A snippet like this needs to be included in all HTML documents that use AI, preferably as early as possible in the markup (for example at the beginning of the `<head>` element). Next, the web server must be instructed that requests for certain resources — typically image files with extensions like `.jpg`/`.jpeg`, `.gif`, and `.png` — should be handled in a particular way. Specifically, the image requests should be passed on to the `adaptive-images.php` script. The OpenLiteSpeed server we have used offers a web interface for defining rules to *rewrite* requests that match a specified pattern. Other web servers, such as Apache 2, have similar ways to define redirections (Apache uses `.htaccess` files).

When the `adaptive-images.php` script is invoked for the redirected request, it compares

the width of the requested image (passed as a parameter) with the resolution (in our case the viewport width) stored in the session cookie. If the viewport width from the cookie is smaller than the width of the requested image, the script looks up a predefined array of supported image widths to find the largest width that fits inside the viewport. It then checks the `ai-cache/` folder to see if a version of the requested image has already been generated for this width (it uses different subfolders for different image widths, i.e., `ai-cache/500/` for images 500 pixels wide). If not, it uses PHP's GD library to add an appropriately resized image to the cache. Finally, it returns the image to the client.

Advantages A major advantage of Adaptive Images is that it *automates all the work of scaling and referencing* the correct images for different screen resolutions. The technique is thus very suitable for smaller web sites that are not supported by a content-management system. The technique is also easier to use because it requires *no additional HTML markup*, making the HTML document more readable.

Drawbacks A major disadvantage of Adaptive Images is that it requires a *session cookie* to store device widths. Of course, not all users accept cookies and, even when they do, there is no guarantee that the cookie will be written and sent along with the first image request on the page, because speculative parsing may pre-fetch images before the cookie has been processed [44]. Adaptive Images therefore offers a fallback solution which reads the User-Agent (UA) string[l] for content negotiation in case the cookie is not yet stored. If the UA string indicates a mobile device, the smallest defined image is loaded. If not, it is assumed that the device is a desktop computer, and the requested image itself is loaded [43].

Another issue with AI is that it can *only fetch images from the same server as the main HTML page*, due to the same-origin restriction for cookies. This precludes, for example, using Adaptive Images with sharding or over a content-delivery network. AI introduces a *delay on first image load* whenever there is no appropriately-sized image that matches the viewport width in the session cookie. The AI script then has to resize and deliver the new image on the fly. This is a minor drawback, because the image would only need to be requested once in order to be resized and written to the `ai-cache/` folder. Although AI images can be cached in the browser, the technique uses the same URL for all the differently scaled versions of an image.

Finally, a downside of the server-side automation is that Adaptive Images *only supports image scaling, not other forms of art direction.* AI also *only works with screen sizes*, it is therefore unable to load different images for changes in other conditions, such as orientation. Also, using AI is more cumbersome when the layouts for larger screens are designed to require smaller images than layouts for lower resolution screens, for example when using multiple-column layouts on large screens.

3.5.3. *Clown-Car Technique*

Like AI, the Clown-Car Technique (CCT) removes the image-selection logic from HTML and CSS and, like Picturefill, it leverages CSS3 media queries. In addition, CCT combines

[l] A text field in an HTTP request header that contains the name, version, as well as other features of the web browser.

```
<svg xmlns="http://www.w3.org/2000/svg"
     viewBox="0 0 300 329"
     preserveAspectRatio="xMidYMid meet">
    <title>Clown-Car Technique</title>
    <style>
        svg { background-size: 100% 100%;
              background-repeat: no-repeat;
        }
        @media screen and (max-width: 400px) {
            svg { background-image: url(images/small.png); }
        }
        @media screen and (min-width: 401px) and (max-width: 700px) {
            svg { background-image: url(images/medium.png); }
        }
        @media screen and (min-width: 701px) and (max-width: 1000px) {
            svg { background-image: url(images/big.png); }
        }
        @media screen and (min-width: 1001px) {
            svg { background-image: url(images/huge.png); }
        }
    </style>
</svg>
```

Fig. 5. SVG file used in by CCT (according to [45]).

the *Scalable Vector Graphics* (SVG) format with the `<object>` element "to serve responsive images with a single request" [45]. SVG "is a language for describing two-dimensional graphics in XML. SVG allows for three types of graphic objects: vector graphic shapes (e.g., paths consisting of straight lines and curves), images and text" [46]. Hence, in addition to vector graphics, SVG can also define raster images and text, and it can be styled using XSL or CSS. CCT leverages these abilities of SVG by using media queries to selectively load appropriately-sized raster images as CSS-background images.

Figure 5 shows an SVG file that acts as a container for all the image-selection logic. This file can be embedded in HTML as follows:

```
<object data="responsive-image.svg"
        type="image/svg+xml">
</object>
```

The `<object>` element is used instead of the more specific `<img>` element because of inconsistencies in how different browsers handle SVGs referenced in `<img>`. Some browsers do not allow external images to be loaded, whereas others fail to preserve the aspect ratio of the image set in the SVG file, when SVG is loaded through `<img>`. None of these issues occur when using `<object>` [45].

Selective image loading in CCT is similar to using Picturefill, but with some important distinctions. The media queries specified in the SVG file are not relative to the browser, but to its parent element. For example, if a CCT `<object>` is declared in a `<div>` element, image selection depends on the properties of the `<div>`, such as its width. Unlike Picturefill, which loads inline images, images specified in the SVG in CCT are loaded as background images, and background images need their dimensions explicitly set to become visible. In SVG, aspect ratio and scaling therefore have to be defined by the `viewbox` and `preserveAspectRatio` attributes. Declaring `background-size: 100% 100%;` on the SVG element itself ensures that the image stretches to the full width of the object element.

```
<picture width="500" height="500">
    <source srcset="large.jpg" media="(min-width: 45em)">
    <source srcset="med.jpg" media="(min-width: 18em)">
    <source srcset="small.jpg">
    <img src="small.jpg" alt="Fallback image">
    <p>Accessible text.</p>
</picture>
```

Fig. 6. The syntax for the `<picture>` element [48]. Full use of the `srcset`-attribute is not shown.

Also unlike Picturefill, CCT does not use HTML markup or CSS to select images. This logic instead resides in the style definitions in an SVG file that retrieves images as backgrounds at different breakpoints. The separation of image-selection logic from HTML and CSS can be seen as a feature, but at the same time necessitates a separate SVG file for each image to be displayed. The separate SVG file needs to be downloaded and parsed before an image can be downloaded, adding latency and an additional HTTP request for each image. An alternative that avoids the extra resource request is to embed SVG markup, such as the one in Figure 5, directly inside the `<object>` element in the main HTML file. It is also possible to inline the SVG file using base64 encoding.

Advantages An advantage of the Clown-Car Technique is that it does *not require JavaScript*. It works with scripting disabled and also avoids problems with blocking behaviours. CCT also *removes image-selection logic from HTML markup*, which becomes less cluttered as a result, and it is *more modular* because its *images adapt automatically to the size of their parent elements*. Indeed, CCT and HTML 5.1 pictures are the only responsive-image technique discussed in this paper with breakpoints that are not relative to the size of the screen or viewport. As with Picturefill, CCT *supports many of the responsive image use cases* defined by the W3C [38], but it supports parent-element instead of viewport-based selection of images.

Drawbacks A major disadvantage of CCT are the *additional HTTP requests* needed to download the SVG files before downloading the image files (unless the SVG code is directly embedded in the main HTML file, when *its size increases* instead). Also, *SVG is not supported in some older browsers*, such as IE prior to version 9.0 and Android's browser prior to version 3.0 [47]. Although the technique requires little additional HTML markup, *the SVG markup required is quite extensive*, calling for automation by a content-management system. Replacing the `<img>` element with the `<object>` elements and loading images as CSS backgrounds make it impossible to distinguish primary content from UI graphics. Because SVG images are loaded as backgrounds, they *need to have explicit aspect ratios declared* as `viewbox` attributes to ensure that image dimensions are displayed correctly, severely limiting the benefit of parent-scaling. Also, although CCT avoids problems with JavaScript-blocking, parent-scaling makes image-selection logic dependent on style calculations, which therefore have to wait on CSS parsing. CCT also *breaks the DRY ("Don't Repeat Yourself") principle* when the same breakpoints have to be declared in both CSS and SVG. And, because the SVG-breakpoints are parent-based, they will most likely have different coordinates from the CSS breakpoints.

3.5.4. *HTML 5.1 pictures*

The responsive-image techniques we have presented so far "rely on either JavaScript or a server-side solution (or both), which adds complexity and redundant HTTP requests to the development process. Furthermore, the script-based solutions are unavailable to users who have turned off JavaScript. [...] standardisation of a browser-based solution can overcome these limitations" [38]. Very recently, W3C's HTML 5.1 Recommendation [49])) has offered such a standardised `<picture>` element as a browser-based and HTML-native technique for responsive images.

Figure 6 shows an example: a `<picture>` element contains *several* `<source>` elements, one of which is loaded conditionally depending on a media query specified in its `media` attribute [50]. The `<picture>` element thus includes a variant of the `srcset` attribute for the `<img>` element [51] and the earlier `src-N` attribute proposal. A regular `<img>`-element is provided as a fallback. This syntax is quite similar to that of the `<video>` element, which has been available in HTML5 for some time.

`<picture>` elements are already implemented in popular web browsers such as Chrome and Firefox. `<picture>` elements thus have the potential to replace the workarounds we have presented earlier in this section, making it interesting to compare their load times.

Advantages HTML 5.1's `<picture>` element offers a *HTML-native technique for responsive images*, so that image selection happens during HTML parsing and thus *allows images to download in parallel with other resources*. This is unlike current script-dependent techniques, which have to wait for script downloading, parsing and execution before image downloading can start. The `<picture>` element also *does not rely on JavaScript*, thus avoiding blocking issues. Through its *use of media queries*, it *supports a wide range of use cases* for responsive images as defined by W3C [38].

Drawbacks The `<picture>` element is new and and may be *implemented differently by different browsers. Older browsers that do not implement* the `<picture>` element may prefetch the fallback image before the media queries are processed. It remains dependent on *CSS parsing and CSS object-model (CSSOM) building.*

3.5.5. *Lazy loading*

A final way to reduce page load times is to delay loading of non-critical resources. Lazy loading is a technique that initially only loads those resources that are displayed inside the browser viewport. Loading of other embedded resources (so-called *below-the-fold* content) is delayed until a placeholder element that represents the un-downloaded image is scrolled into (or near) the browser viewport. Although this paper only discusses lazy loading of images, the technique can be used for any resource that is fetched asynchronously. Although lazy loading is thus not a responsive-image technique in itself, it can be straightforwardly applied to images. We will refer to the non-lazy loading responsive-image techniques as *eager* techniques in the rest of this paper.

Lazy loading of images is usually done through client-side scripting. A common approach is to store the actual image path in a `data` attribute of the `<img>` element (thus turning it into a placeholder element):

```
<img class="lazy"
```

```
data-original="img/lazy.jpg"
alt="Lazy loaded image">
```

A JavaScript event handler detects when the user scrolls an `<img>` element of the `lazy` class into (or close to) the browser viewport. The event handler then inserts the path stored in `data-original` into the `src` attribute of the `<img>` element, triggering image downloading. Omitting the `src` attribute from an `<img>` element is invalid HTML, but this can be circumvented by using a tiny (e.g., 1x1 pixel, transparent) placeholder image that is shared by all the image placeholder elements and that is replaced by the proper image when lazy loading is triggered.

Advantages The central advantage of lazy loading is that it causes *fewer HTTP requests on initial page load*, because requests will only be made for images inside the viewport (other requests are deferred until the placeholder is scrolled inside the viewport). In consequence, lazy loading also *reduces page size on initial page load*, because images comprise the bulk of average web page sizes. Smaller page size can in turn lead to *faster load times*, especially on low-bandwidth connections such as mobile-phone (cellular) networks. Also, lazy loading *avoids downloading images that are never viewed*, which is an issue when paying per downloaded byte over mobile-phone networks.

Drawbacks A major disadvantage of lazy loading is that it *may trigger content reflow* (recalculating the positions of DOM elements) because, by default, the image placeholder does not reserve space for the lazy-loaded image, causing other content to move around when it is loaded later. Because the image should be dynamically resized using `max-width:100%`, space cannot be reserved by hard-coding `width` and `height` attributes in the `<img>` element. The combination of recalculation of DOM elements and image sizes together *may add considerable delays in rendering*. The reflow of elements in the layout may also result in a degraded user experience. [52] concludes that image reflows were a big challenge for user experience and usability, whereas [53] states that "[r]eflows are very expensive in terms of performance, and is one of the main causes of slow DOM scripts, especially on devices with low processing power, such as phones. In many cases, they are equivalent to laying out the entire page again".

[52] proposes a solution to this problem: the *padding-bottom hack*, which combines CSS and intrinsic ratios. The hack adds a class `img-container` to an element (e.g., a `<span>` or a `<div>`) that wraps around a lazy-loaded image. It also adds a `padding-bottom` ratio attribute to reserve an area for the image. It thus inherits the reserved width from its parent element, and its reserved height becomes the product of its width and the padding-bottom ratio. This hack requires the aspect ratio of each image to be defined in advance, and it runs the risk of stretching images beyond their native sizes.

Another disadvantage of lazy loading is that *scrolling is slowed down* and *becomes jagged* when images-below-the-fold are lazy loaded because of scrolling. The problem is exacerbated when TCP connections have been closed in the meantime, or the radio has gone to sleep on mobile devices — when power consumption also becomes a factor. Establishing new connections has an inherent delay making it slower than reusing existing connections. An *additional HTTP request* is also needed to download the lazy-loading script, which *depends on JavaScript*, repeating the problems we have mentioned earlier with *blocking*, *fallbacks*, and

crawler indexing. Lazy loading thus makes most sense for large web pages with many images located towards the bottom.

4. Performance Evaluation

We will answer our third research question through empirical performance evaluation of the techniques we have identified, using a real web site as case.

4.1. *Method*

Choice of case study We first investigated the basic load-time reduction techniques in a pilot study that used artificial examples. To increase validity of our results, we decided to perform a more realistic evaluation using a real web site as a case. We wanted to harness a web page from the case web site and measure how each technique would impact its load times under realistic, yet controlled conditions, on the live internet, and using remote servers with many simultaneous users.

Choice of case web site We wanted to produce, as much as possible, repeatable, generalisable and communicatable results and to highlight the strengths and weaknesses of each technique. We therefore defined the following criteria to select a live web site as a case for our evaluations: The site would have to be in English. It would have to be much visited, and not contain adult material. It would have to have a limited size, instead of being more or less infinitely downwards scrollable. It would have to contain actual HTML-marked content inside the main file, instead of being completely or almost completely web-service based. It should contain many pictures, preferably large and external to the main page, and many small (i.e., thumbnail-size or smaller) pictures (e.g., more than twelve of each). It should contain many non-trivial JavaScripts, style sheets, and other resources (e.g., more than six of each), preferably external to the main page and not compressed, minimised or concatenated. A site intended for both desktop and mobile browsing would be preferable.

Around midnight on March 1st 2015, we inspected the main pages of the top 100 sites on the web according to Alexa[m]. We used Firefox' built-in *store complete web page*-function to harness a short list of candidate sites. After closer inspection, we found IMDb's main page `www.IMDb.com` to best match our criteria. (Although IMDb also offered a smaller-size page for mobile devices at the time, it linked directly to the main page, suggesting that the full-size page was intended at least as an alternative for mobile devices.)

Preparing the baseline page We debugged our copy of the IMDb site to remove errors that resulted from harnessing, downloading additional resources as needed. To control interference between the optimisation techniques and the rendering order of style sheets and scripts, we followed the standard advice of placing all style sheets in the `<head>` element of the main page and all JavaScripts at the end of the `<body>` element (except for a few shorter scripts that could not be moved easily). Because we wanted repeatable measures of the interactions between a client (or front end) and a single server, we attempted to eliminate

[m] www.alexa.com/topsites

page elements that communicated with third parties, such as Facebook and Twitter frames, advertisements and click trackers embedded in the JavaScripts. The result was a *baseline page* of 114.3 kBytes that contained 1505.0 kBytes of other resources in 52 additional files: 9 JavaScripts, 5 CSS files, 30 JPEGs, 6 PNGs, 1 HTML and 1 text file. When fully rendered, the baseline page comprised more than 1200 DOM elements.

The following subsection will explain how we proceeded to create further variants of the baseline page for evaluation purposes. Our focus when creating the baseline and its variants was not to make perfect visual and interactional replicas of `www.IMDb.com`, but to create pages that could be used for systematic evaluation and that appropriately reflected the impact each optimisation technique would have on the load times of the original site. Nevertheless, we made sure that the baseline closely resembled and behaved like the original page. For example, most of the links, menus and buttons worked as before. We have made the baseline page and its variants are available at `http://hdl.handle.net/1956/15343`.

Choice of web server To have full control of web server conditions and to ensure they remained constant during evaluation, we chose to host our own server in the cloud instead of relying on a web-page hosting service. We used a virtual server (a `t2.micro`) running in Amazon's elastic cloud (EC2) on a physical machine cluster located in the Oregon area in North-Western USA. The virtual server ran Ubuntu Linux. It had a single 2.5 GHz Intel Xeon CPU, a 1 GB memory and an EBS disk. Although our virtual server was thus computationally rather weak, the hardware in the underlying EC2 cluster was more powerful. And although our virtual server was dedicated to serving only our variant pages, it had to compete with other virtual servers running on the same hardware for CPU, memory, network and other resources. Hence we consider our web server set-up sufficiently realistic for the purposes of our evaluation.

Because we wanted to evaluate both HTTP/1.1 and HTTP/2, we chose OpenLiteSpeed 1.4.7 as our web server. Of the more widely used servers (W3Techs[n] ranked it 4th), this was the one that supported the most recent versions of HTTP/2 when we started our evaluations. Apart from the few parameters we deliberately modified for evaluation purposes, we used out-of-the-box settings for each server. Later, We will describe the additional servers we used for domain sharding (`.xyz` domains registered through `Namecheap.com`) and content-delivery networking (`www.cdn77.com`).

Evaluation setup We ran all our tests using WebPagetest [54, 55], a project that develops and maintains open-source software for performance evaluation of web sites[o]. The project runs several test servers of its own, among which we selected servers located in Dulles/VA in Eastern USA for all our evaluations. A WebPagetest server works as follows: The user supplies the URL of a web page to evaluate. The user also chooses one of the supported clients, such as Chrome or Firefox running on a desktop computer, or a mobile browser running on a mobile device. The user can also set a variety of test parameters, such as screen size, network bandwidth and round-trip time (RTT), the number of tests to run, whether to ignore SSL warnings, and so on. The WebPagetest server then instructs the chosen client to

[n] `http://w3techs.com/technologies/overview/web_server/all`
[o] `http://www.WebPagetest.org/`

Table 1. Clients used in the evaluations, all located in Dulles, Virginia, Eastern USA.

Label	Device	Browser	Conn-ection	Bandw. (Mbps)	RTT (ms)	Screen size[p]
DESK	Generic desktop	Chrome	Cabled	5/1	28	1377x614
MOBI	Emulated mobile	Chrome Mobile	3G	1.6/0.768	300	611x870
MOTO	Motorola Moto G	Chrome Mobile	3G	1.6/0.768	300	360x511

load and re-load the page a specified number of times, while collecting a rich set of measures. The results can be accessed as HTML pages or downloaded from the test server in various formats, including JSON. Tests can be initiated manually through a web interface or scripted through a RESTful API.

Choice of client We ran all our evaluations with the desktop and mobile versions of Google's Chrome browser as provided by WebPagetest.org. This was the clearly most popular browser on both desktop and mobile front ends when we ran our evaluations [q]. We used the following three front ends (or clients):

- A generic desktop computer (DESK), configured to emulate a screen size of 1366x768 pixels, the most common screen resolution for desktop computers when we started our evaluations (according to popular sites like `w3schools.com`).
- An emulated mobile device (MOBI), configured to emulate a screen size of 600x1024 pixels, the most common screen resolution for Android devices when we started our evaluations (according to the Android Developer Community[r]).
- A physical mobile device (MOTO), a Motorola Moto G phone made available by the WebPagetest project, with a screen size of 360x640.[s]

The desktop and mobile front ends are summarised in Table 1.

Variables The independent variables in our evaluations were:

- page: either the baseline page or one of the variants we will describe in the next section;
- server: either the EC2 cloud host running over HTTP/1.1, HTTPS/1.1 or HTTPS/2, or the content-delivery network (CDN);
- client: either the generic desktop computer (DESK), the emulated Android device (MOBI) or the Motorola Moto G phone (MOTO) as shown in Table 1;
- encryption: either cleartext (i.e., no encryption) or OpenSSL version 1.0.1m — for HTTP/2 we always used encryption (i.e., we used HTTPS/2);
- compression: either compressed or uncompressed data transmission.

Of the extensive set of data generated by WebPagetest, our analysis will focus on the following dependent variables:

[q] According to `https://www.w3counter.com/trends`.
[r] `developer.android.com`, "Android Developers > About > Dashboards" and "Develop > API Guides > Supporting Multiple Screens"
[s] In addition to the Moto G mobile phone, we also evaluated an Android One phone but, because the results were similar, we decided to focus on the Moto G.

- load time: the mean time from the first GET request is sent by the client until the *load event*, which the browser dispatches when all the resources in a web page have been downloaded, parsed, and rendered;
- number of HTTP requests: the number of HTTP requests sent from the client to the server;
- number of HTTP connections: the number of HTTP connections established between the client and server;
- download size: the number of K bytes downloaded.

We have made the full data set available at `http://hdl.handle.net/1956/15343`.

Running the tests We scripted the testing through WebPagetest's RESTful API, running tests throughout 2016 between 1000 and 1300 CET, corresponding to 0400-0700 in Eastern and 0100-0400 in Western USA. Internet traffic watchers[t] suggest that this is a time of low and stable internet traffic in North America.

4.2. *Implementing the techniques*

4.2.1. *Improving server connections*

Domain sharding (SHA2, SHA3, and SHA10) We implemented domain sharding by registering 10 alias domains on a domain-hosting service (`.xyz` domains registered through `Namecheap.com`). With six simultaneous TCP connections allowed and less than 60 resources to download, we considered this sufficient to maximise domain sharding. We then modified the baseline page to load the various scripts, images and other resources not from the EC2 OpenLiteSpeed server, but from the alias domains in a round-robin fashion, so that the next file to load would usually be loaded from another domain. As a result the browser was tricked into increasing the number of concurrent TCP connections, although all the files were in the end stored on the same EC2 server. We created three variants of sharding pages: using two, three, and ten domains, respectively. We wanted to test sharding across two domains because this is recommended as the maximum by some authors [22, p. 541][24, 25]. We wanted to test three domains to see whether load times would increase if we went above the recommended two-shards maximum. We wanted to test ten domains because this was the maximum sharding achievable with less than 60 resources.

Content-delivery network (CDN) We implemented content-delivery networking through a commercial CDN provider (`http://www.cdn77.com/`) to mirror our web pages on their North-American server network, with our EC2 server as origin. We then modified the baseline page to load the various scripts, images and other resources not from the EC2 OpenLiteSpeed origin server, but from the CDN. All the scripts, images and other resources were thus downloaded from the content-delivery network, not only the main page. This CDN provider has a worldwide distribution network consisting of more than 34 data centres.

[t] Such as `http://www.internettrafficreport.com/namerica.htm`.

Domain-sharded content-delivery network (CDN2, CDN3, CDN10) We implemented domain sharding over the content-delivery network by registering 10 additional alias domains from the hosting service. We then modified the CDN page to load the various scripts, images and other resources not directly from the CDN, but from the alias domains in a round-robin fashion, so that the next file to load would usually be loaded from another domain. As a result the browser was tricked into increasing the number of concurrent TCP connections, although all the files were in the end stored on the same content-delivery network.

4.2.2. *Optimising textual resources*

Textual-resource reference page (TRREF) To prepare for evaluating the textual-resource techniques, we created a variant page to serve as a reference (or *pre-test*) for three of the optimisation techniques we will discuss next: concatenation, minification, and the two combined. Because IMDb's original page already used minification, we wanted to create a page variant with scripts and style sheets that more closely resembled human, hand-written code, to be able to assess the full effect of minification later. Hence, the textual-resource reference page differed from the baseline page described in the method section only by beautifying the scripts and style sheets used in the baseline (IMDb's original page did not use concatenation of scripts and style sheets).

We used the three largest search engines on the web (Bing, Google and Yahoo) with search strings ‘‘`beautify JavaScript online`’’ and ‘‘`prettyprint JavaScript online`’’, and similar for CSS, to find suitable beautifiers. After some trial and error, the highly ranked sites `http://www.jsbeautifier.com` and `http://prettyprinter.de/index.php` turned out to suit our needs. We beautified both the external JavaScripts and CSS files and the contents of `<script>...</script>` and `<style>...</style>` elements in the main file. None of the image files were affected by this or the other textual optimisations we made.

The resulting textual-resource reference page was used to derive the three other page variants for textual resources: minification, concatenation, and the two combined.

Minification (MIN) We implemented minification in a similar way, this time using the search strings ‘‘`minify JavaScript online`’’ and ‘‘`minify CSS online`’’ to find suitable minifiers. Both top suggestions, `http://www.jscompress.com` and `http://www.cssminifier.com` turned out to work well. We compressed both the external JavaScripts and CSS files and the contents of `<script>...</script>` and `<style>...</style>` elements in the main file. The resulting page became very similar to the baseline, because most of its scripts had already been minified by IMDb. We did not minify the HTML code, because it was already compact and minifying it further would have made the page difficult to work with.

Concatenation (CAT) We implemented concatenation by including all the external JavaScript and CSS files as `<script>...</script>` and `<style>...</style>` elements embedded in the main file.

Minification and concatenation combined (MINCAT) Accordingly, we also created a page variant that combined minification with concatenation by taking the concatenated variant and minifying all the JavaScripts and style sheets embedded in its `<script>...</script>` and `<style>...</style>` elements.

Compression We did not create a specific variant page for compression, because this is a standard web server option. Instead, we ran all our tests on servers both with and without compression.

4.2.3. *Optimising UI graphics*

UI-graphics reference page (UGREF) To prepare for evaluating the UI-graphics techniques, we created a variant page to serve as a reference (or *pre-test*) for the two optimisation techniques we will discuss next: sprites and symbol fonts. Because IMDb's original page already used a few sprites, we wanted to create a page variant that stored all the icons in separate files, to be able to assess the full effect of sprites later. Hence, the UI-graphics reference page differed from the baseline page described in the method section only by splitting the sprite files used in the baseline into separately loaded icon files (IMDb's original page did not use symbol fonts). The resulting minification/concatenation reference page was used to derive the two other page variants for UI graphics: sprites and symbol fonts.

Image sprites (SPRITE) We implemented image sprites by collating all the UI images of the UI-graphics reference page into a single sprite file. Most of these images had already been combined into smaller sprites by `IMDb.com`. We then changed the sprite coordinates in the style sheets to reflect the new positioning of each visible image inside the collated sprite.

Symbol fonts (SYMFO) We implemented symbol fonts by creating a single TrueType (ttf) font file with glyphs (or characters) corresponding to each of the UI images in the UI-graphics reference page. For example, the new symbol font had one glyph for the black-on-yellow IMDb logo next to the search bar, another glyph for the magnification glass on the search button, a third glyph for the play button layered over movie-trailer images, etc. We used Inkscape on Windows to generate the first version of the font. We then used FontForge on Ubuntu to improve glyph positioning somewhat. Finally, we made a variant page where, whenever a UI graphic was included as an `<img>` element in the original file, it was replaced with a single glyph from the symbol font. Although multichromatic fonts are becoming available through the OpenType initiative, our SVG-based fonts were coarser monochromatic approximations of the original icons.

4.2.4. *Adapting content images to context*

Responsive-image reference page (RIREF) To prepare for evaluating the responsive-image techniques, we created a variant page to serve as a reference (or *pre-test*) for the responsive-image optimisation techniques we will discuss next. Because IMDb's original page

used a fixed-width layout and rather small content images, we wanted to create a page variant with larger pictures, to be able to assess the full effect of the other techniques. Hence, the responsive-image reference page differed from the baseline page from the method section in two ways: (1) One row of three images placed side-by-side was replaced by a single wider image, collected from the same position on the IMDb.com page from a later date, in order to let the effect of a larger image weigh in on the results. (2) The fixed-width layout of the baseline was changed to a variable-width layout that filled the browser viewport, in order to capture the full effect of viewport sizing. The resulting responsive-image reference page was used to derive all the other page variants for content images, including the lazy-loading variants.

Picturefill (PIC) We implemented Picturefill by replacing the `<img>` elements for the main content images (i.e., all the large, visual JPEG images on the page) in the image baseline with Picturefill markup similar to that shown in Figure 2. We used the following breakpoints, also suggested for Adaptive Images[u]: 480px, 768px, 992px and 1200px. If all the images in our image baseline were displayed in full-width, we would thus have generated four correspondingly scaled versions for each image. However, our baseline page instead contained a mixture of images arranged in rows of one to five images per row, and even the widest single-column image covered only 70% of the viewport width. We therefore scaled each image as follows:

$$w_N = \text{round}(0.7w_B - 15(c-1))/c$$

where w_B is a breakpoint width (480, 768, 992, or 1200), w_N is the width of the image after scaling to the breakpoint, c is the number of images in the same row ($1 \ldots 5$), and 15 is the margin in pixels between pictures in the same row. The spreadsheet available at `http://hdl.handle.net/1956/15343` shows the exact image widths calculated for each breakpoint using this equation.

Before testing, we used this equation to generate 4 appropriately scaled versions for each main content image on the server, using Linux' ImageMagick tools. To make the implementations as directly comparable as possible, we used the same image widths also for the Picturefill, Adaptive Image and Clown-Car Technique implementations described in the following sections.

Adaptive Images (AI and AIGEN) We implemented Adaptive Images on the server side by downloading the `adaptive-images.php` script to the same directory as our main images. We used OpenLiteSpeed's rewrite rules to redirect each attempt to download a main content image to this script, which retrieved the viewport width from a browser cookie, as described earlier. The script then calculated the required image width using the same equation as for Picturefill. If it was not already present in the `ai-cache/` folder, the script would resize it on the spot using PHP's GD module and place it in the cache folder. Finally, the script would return the cached image.

To test the effect of image caching on the server side, we also created a variant page that was identical to AI, but which simulated an always-empty cache, so that any required image

[u]They were coded into the `adaptive_images.php` script we had downloaded.

Table 2. Techniques/page variants in each group along with their reference pages.

Group	**Techniques/page variants**	**Reference**
Server connections	SHA2, SHA3, SHA10, CDN, CDN2, CDN3, CDN10	BASE
Textual resources	MIN, CAT, MINCAT	TRREF
UI graphics	SPRITE, SYMFO	UGREF
Adapting content images	PIC, HT5PIC, AI, AIGEN, CCT LAZYRIR, LAZYPIC, LAZYHT5, LAZYAI LAZYCCT	RIREF

would always have to be resized on the spot. We will call this variant of Adaptive Images AIGEN ("Generating Adaptive Images").

Clown-Car Technique (FCCT, ECCT) We implemented the Clown-Car Technique by replacing the `<img>` elements for all the main content images in the image baseline with `<object>` elements. CCT with separate files (FCCT for "file-CCT") generates many additional HTTP requests because it includes SVG code stored in separate files. To make the comparison with other responsive-image techniques fairer, we therefore also implemented a embedded CCT (ECCT) variant that included the SVG code directly inside the `<object>` elements in the main HTML file. We used SVG code similar to the one shown in Figure 5, using the same media queries as for Picturefill.

HTML 5.1 Pictures (HT5PIC) We implemented HTML 5.1 pictures by replacing the `<img>` elements for the main content images in the image baseline with `<picture>` elements as shown in Figure 6. We used the same breakpoints and image widths as for the Picturefill page.

Lazy Loading We implemented lazy-loading versions both of the responsive-image reference page (LZYRIR) and of the other responsive-image techniques, i.e, lazy-loading variants of Picturefill (LZYPIC), Adaptive Images (LZYAI, LZYAIG), the Clown-Car Technique (LZYFC, LZYEC) and HTML 5.1 pictures (LZYH5), by adapting available JavaScripts.

For example, it took only minor modifications of Picturefill and addition of a few JavaScript functions to allow it to lazy-load images. This was done by replacing the `data-picture` attribute (necessary to trigger Picturefill) with a `lazyload` class on the image container element. The modified script would then monitor if an element with the class `lazyload` was scrolled into the browser viewport. If so, the script would reapply the `data-picture` attribute, triggering the execution of the original Picturefill script and loading an appropriately-sized image. The other lazy loads all used variants of this theme: a `class="lazyload"` to mark lazy-loading elements, obfuscation of their element and/or attribute names to ensure that they are not automatically rendered when first loaded, a JavaScript function to call back whenever a lazy-loading element should become visible (enters or nears the browser viewport), and JavaScript to change the obfuscated element and/or attribute names back into the intended, renderable HTML markup when this happens.

5. Results

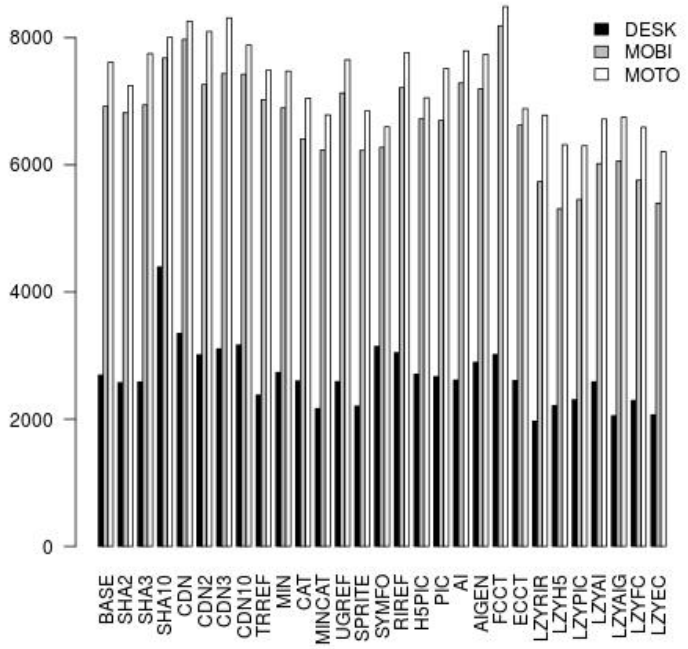

Fig. 7. Comparison of the median HTTP/1.1 load times with compression, but no encryption.

Prior to analysis, we discarded some of the responses we had received from WebPagetest. We removed 30 test runs as outliers and to avoid interference, because they had load times longer than 30 seconds and we had run our tests with only 30 seconds between them. We also removed test runs that had resulted in screen dumps with other viewports than we specified. For these reasons, the numbers of test runs for each technique vary, but there are always more than 60 and usually around a hundred.[v]

We analysed the measurements using the statistics package R [56]. For each combination of server, client and technique (page variant), we calculated the median load times *(Med.)* and the median numbers of HTTP requests *(Req.)*, HTTP connections *(Conn.)*, and K bytes downloaded *(Byt.)*. We calculated the standard deviations of the load times *(St.d)* and the relative changes of median load time *(Cha.)* for each page variant compared to its reference page (see Table 2). We used Cohen's d to calculate the effect *(Eff.)* of each technique compared to its reference and one-tailed Mann-Whitney U tests to calculate its significance *($p <$)*. Finally, we counted the numbers of valid responses for each test run *(n)*.

From here on, by *significance* we mean $p < 0.001$-significance, unless otherwise stated.

5.1. *HTTP/1.1 with compression, but no encryption*

[v]There is one exception: because the MOTO client had problems with tenfold sharding, there are very few measures for that particular combination of treatments.

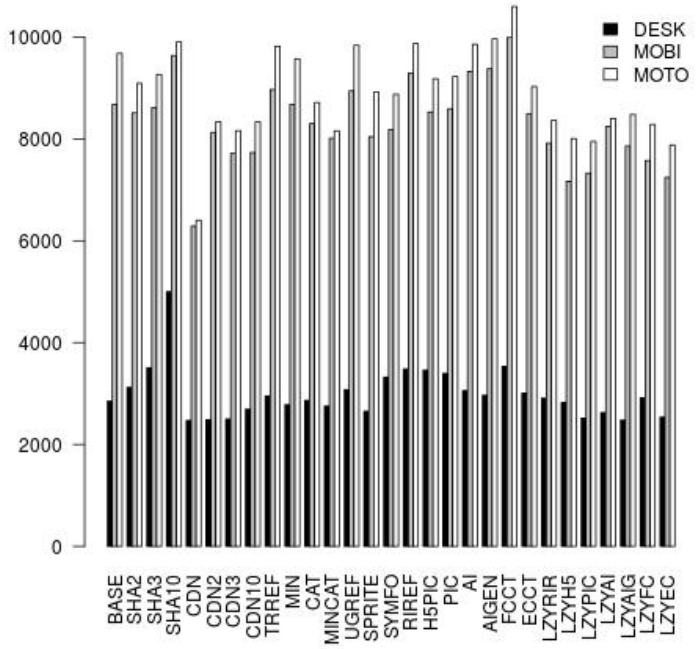

Fig. 8. Comparison of the median HTTPS/1.1 load times with encryption and compression.

Figure 7 shows the median load times when accessing the page variants over HTTP/1.1 with compression, but no encryption. The detailed results are provided in the appendix available at `http://hdl.handle.net/1956/15343`. SHA2 works well for all three clients, SHA3 a bit less so, whereas SHA10 is oversharded and detrimental. CDN also increases load times, even when combined with sharding. MINCAT always reduces load times significantly. Used separately, CAT and MIN also reduce load times on the mobile devices, but not on the desktop client. Not surprisingly, given that server compression is already used, CAT performs better than MIN. For UI graphics, SPRITE produces significant improvements in all the clients. SYMFO is detrimental on the desktop but significantly beneficial on the mobiles. For the eager responsive-image techniques, ECCT, H5PIC and PIC all reduce load times on all clients, significantly so on the mobiles, ECCT the most and PIC the least. The results are mixed for AI/AIGEN. FCCT is always detrimental. Unsurprisingly, the lazy responsive-image techniques perform even better. They all significantly reduce load times on all three clients. Plain LZYRIR works best on the desktop, whereas lazy-loaded HTML 5.1 pictures (LZYH5) and embedded CCT (LZYEC) work best overall.

5.2. *HTTPS/1.1 with encryption and compression*

To prepare for comparison with HTTP/2, which strongly encourages encryption, we also accessed the page variants over HTTPS/1.1, i.e., with both encryption and compression.

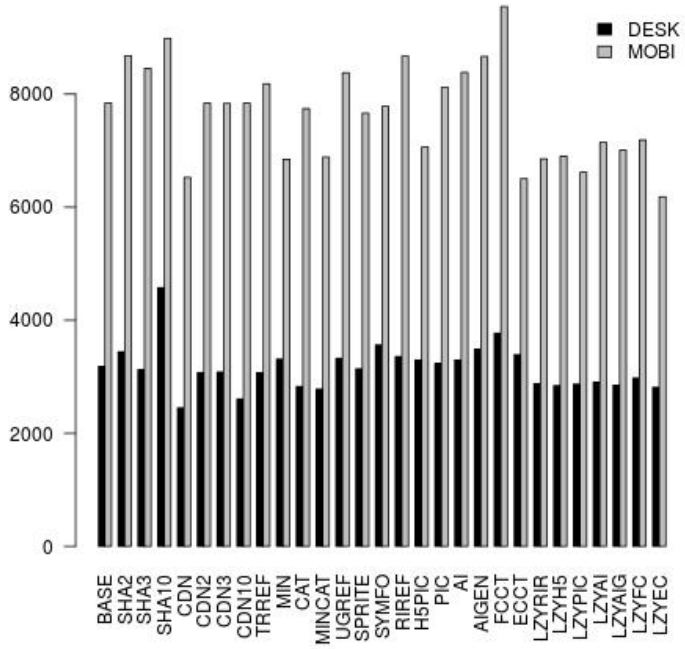

Fig. 9. Comparison of the median HTTPS/2 load times with encryption and compression.

Figure 8 shows the median load times. The detailed results are provided in the appendix available at `http://hdl.handle.net/1956/15343`. Compared to the results for HTTP/1.1, SHA2 and SHA3 now work less well on the desktop client, but still reduce load times on the mobile devices. SHA2 again reduces load times more than SHA3, whereas SHA10 remains oversharded and detrimental. CDN reduces load times significantly on all three clients, but combining CDN with sharding does not produce further improvements. Both MINCAT, MIN ($p < 0.005$) and CAT significantly reduce load times for all clients, always most with MINCAT. For UI graphics, the results are similar to HTTP/1.1. SPRITE reduces load times significantly on all clients, whereas SYMFO is again detrimental on the desktop but significantly beneficial on the mobiles. Several of the eager responsive-image techniques are now weakly detrimental on the desktop, for which AI/AIGEN and ECCT work best. ECCT, H5PIC and PIC again reduce load times significantly on the mobiles, ECCT the most and PIC the least. Indeed, embedded CCT (ECCT) reduces load times significantly on all clients ($p < 0.05$ on the desktop). As in the unencrypted tests, all the lazy responsive-image techniques significantly reduce load times. LZYEC is the most effective lazy-loading technique overall.

5.3. *HTTPS/2 with encryption and compression*

Finally, we accessed the page variants over HTTPS/2 with both encryption and compression. Figure 9 shows the median load times. We only ran these tests for DESK and MOBI,

because the mobile client did not support HTTPS/2 when we began our tests. Sharding is mostly detrimental, whereas CDN reduces load times significantly for both clients, although combining CDN and sharding does not reduce load times further. MINCAT again reduces median load times significantly. CAT works best on the desktop, MIN on the emulated mobile. For UI graphics, SPRITE reduces load times on the desktop, both SYMFO and SPRITE on the mobile. Among the eager responsive-image techniques, only FCCT is detrimental to load times. None of them are significant on the desktop, whereas ECCT, H5PIC and AI ($p < 0.005$) are significant on the mobile. As before, all the lazy image techniques reduce load times significantly, more on the mobile than on the desktop. LAZYEC is most effective on both clients.

The detailed results are provided in the appendix available at `http://hdl.handle.net/1956/15343`. We see that, whereas the HTTP/1.1 tests tended to use all six available connections (albeit more for sharding), the HTTP/2 tests used fewer due to multiplexing, and often only one.

5.4. *Tests without compression*

We also ran tests *without compression* for both HTTP/1.1, HTTPS/1.1, and HTTPS/2, but excluding the CDN tests. The detailed results are provided in the appendix available at `http://hdl.handle.net/1956/15343`. The results for the three servers resemble both one another and their respective compressed counterparts.

Sharding remains detrimental. SHA2 is mostly better than SHA3, and SHA10 is worst. Without compression, MIN becomes a more effective technique than CAT. MINCAT and MIN always significantly reduce load times. CAT is sometimes significantly beneficial on the desktop client, but can be detrimental on mobile devices, perhaps because of the large concatenated main file that has to be downloaded. On HTTP/1.1 without compression, MIN actually outperforms MINCAT. For UI graphics, SPRITE is slightly beneficial and SYMFO detrimental on the desktop. On the mobiles, SYMFO always reduces load times significantly, and SPRITE is also beneficial, several times significantly. For content images, the results are similar to those with compression: on the desktop, the techniques are often detrimental whereas, on the mobiles, H5PIC and ECCT always reduce load times significantly ($p < 0.005$ for MOBI on HTTPS/1.1). The lazy-loading techniques are often significantly beneficial, and tend to reduce median load times most on the smallest screen (i.e., on MOTO), although the results are mixed.

6. Discussion

6.1. *Summary of the results*

An overall picture has emerged from the analyses, which we have summarised in Table 3 as practical recommendations for front-end developers.

At least for our case web site and choice of client, sharding did not consistently improve load times. Two- (SHA3) and threefold (SHA3) sharding worked consistently better than tenfold (SHA10). In a few cases, threefold sharding worked better than twofold, despite suggestions that resources should not be sharded across more than two domains [22, p. 541][24,

Table 3. Summary of main results.

Technique	**Summary**
SHA2	Only effective on HTTP(S)/1.1 with encryption.
SHA3	Mostly worse than SHA2.
SHA10	Oversharding always increases load times.
CDN	Significantly reduces load times on HTTPS/1.1 and /2 with compression. Detrimental on HTTP/1. Not tested without compression.
CDN2, CDN3, CDN10	Less beneficial than CDN on HTTPS/1.1 and /2. Less detrimental on HTTP/1.1. Not tested without compression.
MIN	Significantly reduces load times on mobiles, and often on the desktop. Much stronger effect without compression.
CAT	Significantly beneficial with compression and on the desktop with encryption. Sometimes detrimental on mobiles without compression.
MINCAT	Always reduces load times significantly: more on mobiles than on the desktop, and more without compression than with.
SPRITE	Always significantly beneficial on the desktop with compression. Always reduces load times on mobiles, often significantly so on HTTP(S)/1.1. With compression, mostly more effective than SYMFO.
SYMFO	Always detrimental on the desktop. Always reduces load times on mobiles, often significantly so on HTTP(S)/1.1. Without compression on mobiles, sometimes more effective than SPRITE.
H5PIC	An effective eager image loading technique. Always significantly reduces load times on mobiles. Sometimes a little detrimental on the desktop.
PIC	With compression, similar to H5PIC, but less effective. Mostly detrimental without compression.
AI	Sometimes beneficial, sometimes detrimental. Few strong effects.
AIGEN	Resizing images on the fly does not greatly increase load times compared to cached AI.
FCCT	Almost always detrimental.
ECCT	The most effective eager image loading technique overall. Always reduces load times with compression. Always significant on mobiles.
LZYRIR	Always significantly reduces load times with compression, and often without compression too. Stronger effect on mobiles than on the desktop.
LZYH5	An effective lazy-loading technique. Resembles LZYH5, but the effects are usually weaker.
LZYPIC	An effective lazy-loading technique. Resembles H5PIC overall. Also resembles LZYH5, but the effects are usually weaker.
LZYAI	Always significantly beneficial with compression, and often without. Less effective than LZYEC, H5PIC and PIC overall.
LZYAIG	Resizing images on the fly does not greatly increase load times compared to cached, lazy-loading AI.
LZYFC	With compression, resembles LZYEC, but the effects are weaker. Mixed results without compression.
LZYEC	The most effective lazy-loading technique overall. Almost always reduces load times significantly: mostly stronger effects on mobiles than on the desktop, and stronger effects with compression than without.

25]. The content-delivery network (CDN) worked well on the secure connections we tested it on, but combining CDN with sharding (CDN2, CDN3, CDN10) did not reduce load times further.

For textual resources, minification and concatenation (MINCAT) together always reduced load times Minification tended to reduce load times more than concatenation. Minification alone (MIN) always reduced load times when compression was not used but, when combined with compression, it was sometimes detrimental on the desktop. Concatenation alone (CAT) tended to be beneficial, and reduced load times slightly even on HTTPS/2 although, on mobile devices without compression, it often increased load times.

For UI graphics, sprites (SPRITE) usually improved load times on the desktop, whereas symbol fonts (SYMFO) were detrimental. On the mobile devices, both sprites and symbol fonts tended to reduce load times. Sprites worked best on MOBI, symbol fonts on MOTO.

Among the eager responsive-image techniques, HTML 5.1 pictures (H5PIC) were mostly beneficial, more clearly so on mobiles than on the desktop. With compression, Picturefill (PIC) always reduced load times, slightly more than HTML 5.1 pictures on the desktop, but less on the mobiles. Adaptive images (AI) with cached images tended to reduce load times with compression. Having to reisze images on the fly when using adaptive images (AIG) did not punish load times severely. The embedded clown-car technique (ECCT) reduced load times more than the separate-file variant (FCCT), which tended to be detrimental.

With compression and on HTTP/2, all the lazy-loading techniques were beneficial. LZYEC, LZYH5, and LZYPIC were most effective, roughly in that order. On HTTP/1 without compression, the results were more often beneficial than detrimental. On HTTPS/2, the differences between the lazy-loading techniques were small.

In the introduction, we asked whether HTTP/2 and HTML 5.1 would render obsolete many of the existing load-time reduction techniques. It turns out that both CDN, MIN/MINCAT, SPRITE, and lazy responsive-image loading turn out to remain beneficial for HTTPS/2. Importantly, however, HTML 5.1's new `<picture>` element competes well with the existing responsive-image techniques, suggesting that it indeed offers a viable long-term replacement for the earlier workarounds.

6.2. *Validity and reliability*

We have taken care to make our results as valid and reliable as possible. In the review part of the paper, we have defined explicit inclusion criteria for the types of optimisation techniques we wanted to cover. And we have spent a lot of time reviewing an unusually wide range of information sources — including white papers, books, articles and blog posts — until reaching saturation, i.e., until reviewing further sources and conducting additional searches no longer revealed techniques we were not already aware of. We have taken care to present each technique using their originators' own terms and types of examples.

In the evaluation part of the paper, we have used explicitly defined criteria to select a much-used case web site, `IMDb.com`, as the starting point for our tests. We kept all the modifications we made to this site as light as possible — sometimes at the cost of visual fidelity — to preserve realism and make the variants we created more closely comparable to one another. We have taken care to implement the optimisation techniques neutrally and, to the extent possible, exactly as described in the original sources, making only the smallest

modifications necessary for them to work with our case web site. Although the resulting baseline page loads a little more slowly than the original, highly optimised IMDb page, this may actually make it *more* representative of the majority of sites out there, which lack the performance-optimisation resources and competencies of IMDb. We have chosen a much used cloud-hosting system, Amazon EC2, to run our web servers, and we have used a comprehensive and proven online system, `WebPagetest.org`, for automated performance testing. Because we have run all our tests on a live server on the live internet, our measurements are not fully repeatable. But we have taken care to run the tests during the same quiet time of the night to limit the impact of large fluctuations in internet traffic. We have also spread our tests over almost a year to cancel out day-to-day fluctuations. Finally, we have increased repeatability by making our page variants and detailed measurements available in electronic form at `http://hdl.handle.net/1956/15343`.

The results of our evaluations underline that web page testing is not an exact science. As can be expected, load times vary widely with server and network conditions, and scripts and style sheets often download different versions of resources for different clients. But even when tests are repeated with identical parameters, we observe not only that load times vary widely, but also that numbers of connections, requests, and bytes downloaded differ. Possible reasons are that small fluctuations in server and network conditions affect the speed with which JavaScripts and style sheets are downloaded, which in turn lead to differences in script execution and rendering order, which can in turn cause different additional resources to be loaded under nearly identical conditions.

6.3. *Limitations of the research method*

There are of course limitations to our study. Most importantly, we have only evaluated the techniques on a single case web site, `IMDb.com`. Evaluating them on a broader range of web sites to increase generality of our findings remains an obvious path for further work. Further tests should also investigate broader ranges of front-end equipment, of web browsers, and of servers and server set-ups. In particular, the presented results are valid only for Chrome, the currently most popular browser on both desktop and mobile front ends. We have run a limited set of tests that suggest that the techniques indeed have significantly different effects when run on different browsers (Chrome, Firefox and Internet Explorer 11) on a desktop client and a HTTPS/2 server with compression. In this particular setting, sharding tended to work significantly better on Chrome than on Firefox and IE 11. The textual techniques and sprites worked better on Firefox, whereas symbol fonts worked best on IE 11. Picturefill worked best on Chrome, but the other image techniques, including the lazy-loading ones, worked best on Firefox and Internet Explorer. The current evaluation therefore needs to be repeated with additional clients.

Different combinations of client and server locations also need to be tested, and our tests should be repeated and expanded using other tools than `WebPagetest.org`. Yet we argue that there is much to learn from studying a real and, in many ways, typical web site like `IMDb.com` in great detail as we have done.

Our analysis has focussed on load times, but the rich data set produced by `WebPagetest.org` can also be analysed in many other ways and additional measures collected by other means. For example, *handling JavaScripts* can be a major factor contributing to the load times of web

pages. Although the download sizes of JavaScripts were much smaller than of images both in our case web site and in the pages sampled by [17], the scripts are likely to demand more than proportional amounts of processing power in the browser. Investigating their impact on front-end performance thus remains an important topic for further work, e.g., the effects of JavaScript and different JavaScript interpreters on load times in general and how they interact with different optimisation techniques in particular. We would like to investigate how techniques for DOM manipulation, script-loading order and dependencies, and uses of synchronous and asynchronous JavaScript interact with front-end performance optimisation.

Also, the tests we have presented have not taken *browser caching* into account. Research performed by Yahoo [57] show that the number of users arriving at their site with an empty cache stabilised below 50%, and that only 20% of user clicks had a cache-miss penalty. In a live setting, caching is likely to have a major impact on load times for repeating visitors because it removes the need for resources to be downloaded from the network. Yet some of our optimisation techniques — such as concatenation — hinders or reduces the effect of caching, whereas others — such as compression — does not affect caching at all. This is likely to have a large effect on our optimisation techniques in practice, but it is an effect we can say little about in this paper. Indeed, because different optimisation techniques affect caching differently, the case-miss ratio will be highly different for different techniques. Further evaluations with live users are needed to investigate this important problem in more depth.

An expected development is broader use of content negotiation through HTTP Client Hints. According to [23, p. 49], "optimal implementations of header-compression strategies, prioritisation, and flow-pretest logic both on the client and server, as well as the use of server push" are important areas for further research on front-end performance optimisation for HTTP/2.

In addition to our automated tests, we would also like to conduct *real-user measurements* [54], e.g., using scripting to collect detailed performance statistics from live user interactions with a production web site, such as `IMDb.com`. We would also like to investigate how differently optimised web sites are *perceived* by live users. For example, we could ask users to complete different tasks under varying conditions to evaluate whether and how much better web performance improves task completion, and we could interview them about their experiences afterwards.

7. Conclusion

We have reviewed, evaluated, and compared load-time reduction techniques for device-agnostic web sites, i.e., for web sites that use the same context-aware HTML pages and other resources to support a wide variety of front-end equipment. We have focussed on two front ends that are common today: desktop (and, in practice, also laptop) computers and mobile devices. We have divided the techniques we have found into four categories: improving client-server communication, optimising UI graphics, optimising textual resources, and adapting content images to context. Because images constitute a large portion of web resources, and because they pose particular challenges for mobile devices, we have spent much time and space evaluating the performance effects of image optimisation techniques, with focus on techniques for responsive web design.

We have shown that all the techniques have their pros and cons, and demonstrated that

they can all be implemented on both desktop and mobile front ends. Most of them are clearly beneficial under at least one of the conditions we have evaluated, but most of them are also clearly detrimental in certain cases — sometimes drastically so. MINCAT/MIN, SYMFO, and the lazy responsive-image techniques remain the safest bets. The eager responsive-image techniques also improve load times on mobile devices, but the improvements are offset by a small load-time penalty on desktop computers. Among them, HTML 5.1's new `<picture>` element performs well. Our results suggest that many of the load-time reduction techniques will remain relevant as the web continues to move from HTTP/1.1 to HTTPS/2. We conclude that front-end optimisation techniques must always be chosen with utmost care, based on a solid understanding of anticipated server, network and client conditions, and supported by a keen understanding of the benefits and drawbacks of the available techniques.

Acknowledgements

We thank the anonymous reviewers of this paper for their highly detailed and insightful comments. We also thank `WebPagetest.org` for providing the online testing tool we have used in this paper.

References

1. Cisco. Visual networking index – global mobile data traffic forecast update, 2014-2019, 2015.
2. Luke Wroblewski. *Mobile first.* A Book Apart, New York, 2011.
3. Katie Fehrenbacher. M dot: Web's answer to mobile. *Gigaom*, May 11, 2007.
4. S Mohorovicic. Implementing responsive web design for enhanced web presence. In *Information & Communication Technology Electronics & Microelectronics (MIPRO), 2013 36th International Convention on*, pages 1206–1210. IEEE, 2013.
5. Sehoon Park, Qichen Chen, Hyuck Han, and Heon Y Yeom. Design and evaluation of mobile offloading system for web-centric devices. *Journal of Network and Computer Applications*, 40:105–115, 2014.
6. Minhaj Ahmad Khan. A survey of computation offloading strategies for performance improvement of applications running on mobile devices. *Journal of Network and Computer Applications*, 56:28–40, 2015.
7. Kayce Basques. How to use the timeline tool, December 15, 2016. Retrieved 2016-12-16.
8. Aaron Gustafson and Jeffrey Zeldman. *Adaptive Web Design: Crafting Rich Experiences with Progressive Enhancement.* Easy Readers, 2011.
9. Steven Champeon and Nick Finck. Inclusive web design for the future, 2003.
10. Aaron Gustafson. Understanding progressive enhancement. *A List Apart*, October 2008.
11. Seth Weintraub. Industry first: Smartphones pass pcs in sales, 2011.
12. Ed Hardy. Google adopts a new strategy: Mobile first, 2010.
13. Kayla Knight. Responsive web design: What it is and how to use it, 2011.
14. Ethan Marcotte. Responsive web design. *A List Apart*, 306, May 25, 2010.
15. The HTTP Archive. Interesting stats (desktop), February 1, 2014.
16. The HTTP Archive. Interesting stats (desktop), November 15, 2010.
17. Guy Podjarny. What are responsive websites made of?, April 29, 2013.
18. Guy Podjarny. Performance implications of responsive design, July 11, 2012.
19. Barbara Kitchenham. Procedures for performing systematic reviews. *Keele, UK, Keele University*, 33(2004):1–26, 2004.
20. Yehia Elkhatib, Gareth Tyson, and Michael Welzl. The effect of network and infrastructural variables on spdy's performance. *arXiv preprint arXiv:1401.6508*, 2014.

21. W3C. Hypertext transfer protocol – HTTP/1.1, June 1999.
22. Steve Souders. Domain sharding revisited, September 5, 2013.
23. Ilya Grigorik. Making the web faster with http 2.0. *Commun. ACM*, 56(12):42–49, December 2013.
24. William Chan. Network congestion and web browsing, May 20, 2013.
25. G. Mineki, S. Uemura, and T. Hasegawa. Spdy accelerator for improving web access speed. In *Advanced Communication Technology (ICACT), 2013 15th International Conference on*, pages 540–544, Jan 2013.
26. A. Vakali and G. Pallis. Content delivery networks: status and trends. *Internet Computing, IEEE*, 7(6):68–74, Nov 2003.
27. Steve Souders. *High performance web sites - essential knowledge for frontend engineers: 14 steps to faster-loading web sites.* O'Reilly, 2007.
28. Nicholas C. Zakas. The evolution of web development for mobile devices. *Queue*, 11(2):30:30–30:39, February 2013.
29. Mozilla. Shorthand properties, 2013.
30. Steve Souders. High-performance web sites. *Commun. ACM*, 51(12):36–41, December 2008.
31. Alex Nicolaou. Best practices on the move: Building web apps for mobile devices. *Queue*, 11(6):30:30–30:41, June 2013.
32. Sayanee Basu. Source maps 101. *Tutsplus*, January 16, 2013.
33. Jean-loup Gailly and Mark Adler. What is gzip?, 1999.
34. Andy Davies. *Pocket Guide to Web Performance.* Five Simple Steps, Penarth, UK, 2013.
35. JunLi Yuan, Li Xiang, and Chi-Hung Chi. Understanding the impact of compression on web retrieval performance. *The Eleventh Australasian World Wide Web Conference*, 2005.
36. Ilya Grigorik. *High Performance Browser Networking: What Every Web Developer Should Know about Networking and Web Performance.* O'Reilly Media, Inc., 2013.
37. Brian Suda. *Creating Symbol Fonts.* Five Simple Steps, Penarth, UK, 2013.
38. W3C. Use cases and requirements for standardizing responsive images, November 7, 2013.
39. Scott Jehl, Mat Marquis, and Shawn Jansepar. Picturefill documentation, 2014.
40. Bert Bos, Tantek Çelik, Ian Hickson, and Håkon Wium Lie. Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification, 2009.
41. Ethan Marcotte. *Responsive Web Design.* A Book Apart, New York, 2011.
42. Bill Venners. Orthogonality and the DRY principle – a conversation with andy hunt and dave thomas, part ii, March 10, 2003.
43. Matt Wilcox. Adaptive images documentation, 2012.
44. Yoaw Weiss. Preloaders, cookies and race conditions, September 28, 2011.
45. Estelle Weyl. Clown car technique: Solving adaptive images in responsive web design. *Smashing Magazine*, June 2, 2013.
46. W3C. Scalable vector graphics (SVG) 1.1 (second edition), August 16, 2011.
47. The *Can I use... ?* web site. Can i use svg?, 2014.
48. W3C. The picture element – an HTML extension for adaptive images, 2013.
49. W3C. Html 5.1 — w3c recommendation, November 1, 2016.
50. Ian Devlin. Responsive HTML5 video, August 20, 2012.
51. W3C. The srcset attribute – an HTML extension for adaptive images, 2013.
52. Anders Andersen and Tobias Järlund. Addressing the responsive images performance problem: A case study. *Smashing Magazine*, September 16, 2013.
53. Mark Wilton-Jones. Efficient javascript. *Dev.Opera*, November 2, 2006.
54. Patrick Meenan. How fast is your website? *Communications of the ACM*, 56(4):49–55, 2013.
55. Rick Viscomi, Andy Davies, and Marcel Duran. *Using WebPageTest: Web Performance Testing for Novices and Power Users.* " O'Reilly Media, Inc.", 2015.
56. Nina Zumel, John Mount, and Jim Porzak. *Practical data science with R.* Manning, 2014.
57. Tenni Theurer. Performance research, part 2: Browser cache usage – exposed!, January 4, 2007.