m.Site: Efficient Content Adaptation for Mobile Devices

Aaron Koehl, Haining Wang

m.Site: Efficient Content Adaptation for Mobile Devices

Aaron Koehl and Haining Wang

Department of Computer Science
College of William and Mary
Williamsburg, VA, USA

Abstract. Building a mobile user interface can be a time consuming process for web site administrators. We present a novel approach for adapting existing websites to the mobile paradigm. In contrast to existing technologies, our approach aims to provide a trio of functionality, ease of use, and scalability for large web communities. A site administrator visually selects objects within a web page, and assigns one or more attributes to page objects from a rich collection of pre-defined page modifications. Our proposed system then generates code for a multi-session, php-based proxy server to provide dynamic mobile content adaptations based on the attributes selected. The modifications encapsulate complex page interactions and provide a simplified interface to mobile users. The proxy server is augmented with a highly efficient and standards-compliant browser residing on the server to interpose on behalf of a resource-constrained mobile client. Adaptations such as pre-rendering of content can be cached and shared across users to amortize load. We build a prototype and evaluate its efficacy on a complex web application driving a busy online community with nearly 66,000 members.

Keywords: mobile content adaptation, web application proxy

1 Introduction

Web site administrators and content providers continually aim to accommodate an ever-increasing user base, yet doing so requires supporting a diverse set of browsing platforms. As a consequence, site administrators are forced to balance site accessibility and dependability against the costs of supporting multiple platforms. For instance, due to varying DOM (document object model) implementations within popular web browsers, object accesses in JavaScript are often written in such a way that if one function fails because of browser incompatibility, another function must be written to take over, with the idea that eventually a compatible function will be invoked. Such support issues are not limited to scripting. Differences in supported image formats, support for transparency, variation in supported fonts, subtle discrepancies in CSS rendering, incompatibilities caused by user-installed plugins, availability of media extensions such as Flash and SilverLight, and browser quirks modes [6] between versions must all

be taken into consideration to guarantee support for a large audience. Although there are productivity tools that help in this regard, correctly supporting a diverse set of clients is still a time consuming process. Ultimately, it is the content administrators and site owners who suffer revenue loss when a user's browsing experience is compromised.

Support for mobile browsing introduces considerable complexity to the equation, as mobile browsers are limited in their capabilities, and even the extents of those limitations vary greatly between devices. In addition to diverse client software environments, the device's screen size, network bandwidth, and computational ability can compromise the user's browsing experience if disregarded by the site's administrator. Whereas great strides have been made in providing capable mobile architectures, there is a considerable gap between mobile browsing and the richness provided on even low-end desktop platforms. Supporting higher computational power is at odds with the small form factor, heat output, and battery life expected of today's smart phones, such as the BlackBerry, iPhone, and Android.

Currently, site administrators of large and dynamic template-based websites such as online communities often do not have the time, skill, or capability to deploy specialized templates for mobile users, although these websites must consider the demands and needs of the growing mobile market. To tackle the problem faced by site administrators, we propose a cross-cutting approach to content adaptation for mobile browsing. Content adaptation (screen scraping) is an effective way to alter the presentation for resource constrained clients, without involving changes to logic at the database or scripting layer. It is important to emphasize that (1) content adaptation employs a multitude of techniques, and (2) content adaptation techniques do not portend a single correct method, instead we recognize a design space in which content adaptation systems make various tradeoffs.

We therefore develop **m.Site**, a productivity framework that enables site administrators to dynamically adapt content for the mobile web with minimal effort, yet still allows for advanced, programmed customizations. m.Site does not rely on special browsers or remote third party services, is uninvasive with respect to code modification, preserves the platform-independence of the web by not requiring device-specific API's, and provides the site administrator with an efficient and cost-effective way to customize very complicated dynamic web sites.

Our design goal is to make the use of m.Site as simple as possible. We accomplish this by introducing an *attribute* paradigm, where page objects are identified in a visual tool, and attributes are selected and applied from a menu. These attributes embody well-known techniques such as image fidelity transformation, to complex subpage interactions. The visual tool generates php shell code for a server-side proxy, which is responsible for downloading page content, applying page transformations and attributes, managing cookie jars and multiple users, and marshaling interactions between the mobile client and the originating web page. Figure 1 shows the architecture of m.Site at-a-glance. Available to the

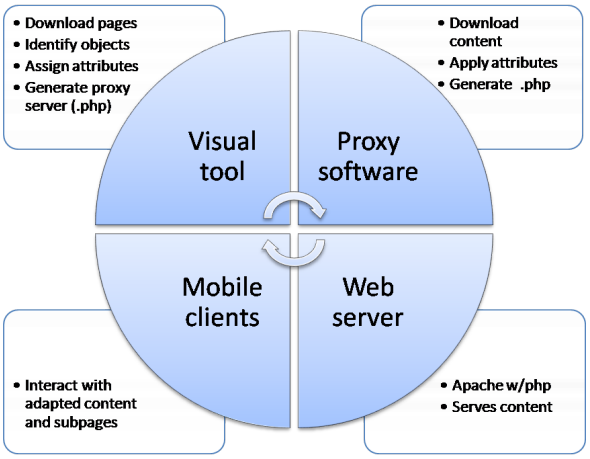

Fig. 1. High-level overview of the m.Site architecture.

server side proxy is an arsenal of web scraping and DOM-manipulation tools, as well as an embedded WebKit [5] browser, which can be used as one of several pre-rendering engines or to execute code. By simplifying the interface and reducing or eliminating the need to write transformation code, we expect administrators will more readily adapt existing web sites for the mobile web.

Our work is motivated by scalability issues found with previous research in this area. The Highlight [21] system employs a remote control metaphor, in which server-side web browser instances are used to maintain state for each client. The resource consumption makes this approach infeasible for large web communities with thousands of concurrent users.

While providing similar high-level features, we instead generate code for a lightweight proxy that can handle the majority of the content adaptation: page slicing, state management, and DOM manipulation, calling on the web browser only when needed as a graphical rendering engine, or for browser-specific functionality. In this way, we also expose the opportunity for the proxy to cache and amortize rendering costs and general content adaptations across multiple users. Cookie security, session management, manipulation via jQuery, and AJAX requests can all be satisifed independently of a heavyweight browser, providing much of the browser's functionality without the associated scalability issues.

We build a prototype of our system and evaluate its efficacy on a complex web application driving a busy online community with nearly 66,000 members. Summarily, this paper makes the following contributions:

– A code generator that produces a low-overhead, multi-session proxy server to support adapted pages. This proxy server manages sessions, keeping browsing sessions open in a stateful manner, without the overhead of a browser running on the server for each user [21];

- A php-based proxy capable of using a highly efficient and standards-compliant browser running on the server in a disconnected state;
- Server-side caching to amortize rendering costs across many client sessions;
- A visual admin interface that uses a simple *attribute* paradigm to provide site administrators with the ability to perform many complex modifications for both visible and hidden document objects;
- A pluggable content adaptation system that can be extended with multiple rendering engines to produce HTML, static images, PDF, plain text, or Flash content at any point in the rendering process;
- Support for producing thumbnail snapshots of rich media content for resource-constrained devices.

The remainder of this paper is structured as follows. Section 2 surveys related work and existing techniques for adapting mobile content. Section 3 describes the m.Site framework as well as the benefits provided by the attribute system. Section 4 evaluates the efficacy of the framework on a live site, and finally Section 5 concludes.

2 Related Work

m.Site is a productivity framework aimed at allowing site content adaptation post-hoc, for mobile devices. Architecturally, systems that allow content adaptation exist either on the client or as a middleware proxy on the server.

Both client and proxy solutions for content adaptation have been proposed over the years, influenced by varying needs of the user and site administrator, as well as evolving technology in resource-constrained devices. Fudzee and Abawajy [17] provide a high level classification for content adaptation systems, and further argue for their viability as an attractive solution. m.Site is a dynamic, proxy-based content adaptation system colocated on the web server, as our motivation is on the site administrator's need to support as diverse and broad of a user base as possible.

Remote display protocols (e.g. thin clients) are not new [23]. However, several systems have been proposed specifically for mobile devices [16, 14], which offload computation from a mobile device to a more capable server, while sending graphical updates and metadata to the device. While thin clients are a relevant technology, they require the installation of client-side software to manage the interaction. Also similar are specialized accelerated mobile browsers such as Opera [9] and Skyfire [10]. m.Site ascribes to the offloading approach, but proposes lightweight graphical updates to be disseminated using an ordinary, default mobile browser.

Client-side browser plugins [1, 3] can provide the user with many tools to customize a site's layout. A plugin injects Javascript into the downloaded page and manipulates the layout using DOM functions. These systems have trouble with dynamic page changes, as they often use static XPaths and basic heuristics to locate objects on the page. However, there has been research into making

client content adaptation systems more robust [12], allowing customizations to be reused in spite of content changes. Still, Javascript is limited to modifying objects in the DOM tree. m.Site allows for more sophisticated content adaptation techniques in addition to Javascript manipulation.

Systems such as [18, 19] allow content adaptations to persist based on the inputs of a corpus of users. Sharing of scripts within GreaseMonkey communities [2] provides a static analog to this. Unfortunately, client side software solutions all suffer from the same problem when aiming to serve a large user base. Users are reluctant to install or use new browsers and plugins other than the default, and thus site administrators cannot rely upon these techniques for layout and content adaptation, especially of mobile visitors.

Proxy based systems allow more sophisticated content adaptation techniques, extending even to rich multimedia types [26]. FlashProxy [20] allows Flash content to execute remotely on the server yet be displayed on a mobile device. Employing a binary rewriting technique to interpose on behalf of the browser, events trapped on the proxy are sent to the client's browser via a Javascript RPC system, maintaining interactivity. m.Site addresses rich media concerns by allowing snapshots of rich media content to be generated, but leaves the interactivity of Flash, movies, and Silverlight to their respective plugin developers.

A number of proxy-based content adaptation systems have been proposed, which aid in navigating pages on mobile devices [21, 25]. Bickmore and Schilit devise a system [11] to analyze and modify a web page based on heuristics and rules, for instance to adapt all images to a lower fidelity.

Automated techniques for page adaptation are promising but not always widely applicable [24]. Chen et al. propose a system to automatically analyze and split a page into subpages to reduce horizontal scrolling [13]. Xiao et al. extend this approach to allow a page to be split into a hierarchical structure [25]. Tools such as Apple's DashCode [4] can be used to simultaneously author a mobile and web application, avoiding a dual-maintenance scenario, but sites must be rewritten to use such a tool. Automated techniques provide a good starting point for adapting page content, and could be used in conjunction with a framework such as m.Site.

A hybrid approach for enhancing mobile navigation is to use a proxy to generate thumbnail overviews of site content. Annotated thumbnails and page splitting enhance navigation by reducing the input effort of browsing a site [25]. m.Site allows the creation of annotated thumbnails as well as multiple levels of page splitting. Note that a page of low-fidelity thumbnail links can load an order of magnitude faster than rendering complicated site content on a mobile device, by reducing both bandwidth and computational effort.

The Highlight system [21] employs a modified Firefox browser located on a proxy server. A user interacts with modified content sent to the mobile browser, which in turn remotely controls the browser session maintained on the server. While this keeps sessions separate and allows for dynamic content, it does not scale well. In contrast, the m.Site framework uses Apple's WebKit [5] library for server side rendering, but only when absolutely necessary. Most of the DOM

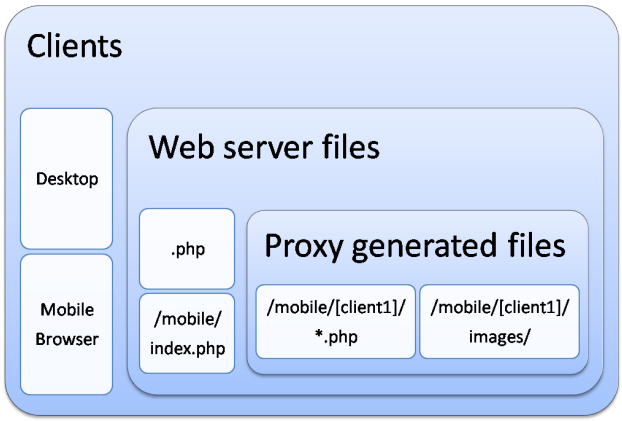

Fig. 2. m.Site organization

manipulation and content adaptation can occur outside of the context of the web browser, while keeping the ability for the proxy to maintain state and sessions for multiple users, and exposing additional cross-session optimizations such as caching of pre-rendered objects.

3 System Architecture

m.Site consists of two major components: a visual tool that the site administrator uses to reshape the site content, and a proxy server that dynamically applies attributes and generates the reauthored pages. Figure 1 provides a high-level overview of the m.Site architecture, while Figure 2 presents how m.Site is organized on the server.

3.1 Site Administrator Tools

In order to be as productive as possible, we develop a visual tool for providing site administrators a live view of the site. Once a page is loaded, the administrator is able to highlight page objects using a point and click approach, to select DOM objects on the page. A separate dock exists for non-visual objects, such as CSS, Javascript functions, head-section content, `doctype` tags, and cookies. The selected objects can be subsequently assigned any number of special attributes that ultimately affect their display on the client. Once a page is downloaded, the proxy system dynamically identifies these objects, applies any defined adaptations, applies any default rules for unidentified objects, generates the appropriate subpages and content, and redirects the user to a newly generated entry page.

It is possible that graphical objects split from the main page cannot be rendered without JavaScript and associated CSS, that is, objects may have intra-page dependencies. These dependencies can be identified in the visual tool. If

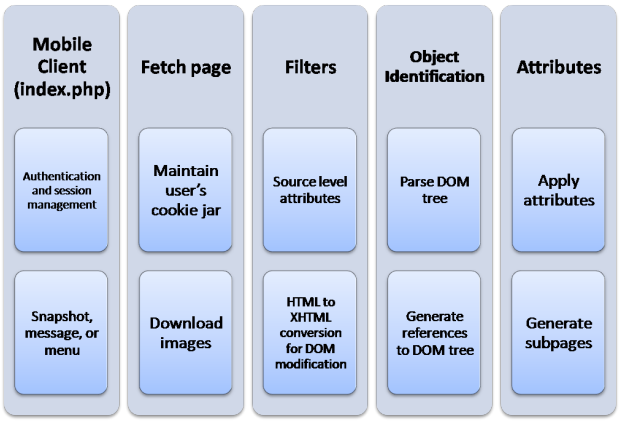

Fig. 3. Role of the Rendering Proxy

a dependent object is to be rendered on the client, the appropriate CSS and JavaScript dependencies can be satisfied by assigning an attribute, which provides the object to the browser.

The typical work flow to mobilize a site is to load a site's page into the tool, visually select relevant objects, and choose attributes to apply (if any). A more advanced work flow delivers more control, and may include matching objects and content with regular expressions, image fidelity manipulation, defining of cacheable objects, and more sophisticated adaptation techniques, such as pre-rendering CSS on the server but rendering text on the client. As with most authoring tools, such techniques will be heavily dependent on the site being adapted.

3.2 Proxy Server

Upon completion, the visual tool generates a php file from shell template code. This shell code becomes a proxy for the originating page, and handles user session authentication, cookie jars, and high-level session administration, such as deletion of cookies. The proxy also handles downloading of the originating page on demand, http authentication on behalf of the client, and any error handling should the page be unavailable. Figure 3 highlights the main tasks performed by the proxy.

After a page is downloaded by the proxy, the attribute system and filters are invoked to apply any attributes defined by the site administrator. This includes locating any objects that need to be modified within the page, and performing any DOM manipulation. The proxy then creates a subdirectory for the user, generates one or more static subpages, and creates any supporting images and files as needed—the contents of which are controlled by the attribute system as shown in Figure 2.

The mobile client begins its interaction with a php file, which contains code responsible for handling authentication and management specific to m.Site sessions, as well as for providing a mobile-friendly entry point (snapshot and menu) into the site. Upon starting a mobile session for the first time, the mobile browser is issued a session cookie for maintaining state on the server. All of the files generated during a user's session are stored in the file system under a (protected) subdirectory created specifically for that user.

If the snapshot does not yet exist, it must be generated. The proxy first loads the user's cookie jar (as determined by the session cookie), and issues a page fetch on behalf of the mobile client for the desired page, which includes downloading any images to be rendered. The cookie jar is necessary as the proxy itself must be authenticated on behalf of the user to view content privy to that user. For publicly accessible forums, this would not be an issue, but typical online communities provide access to private forums and user setting pages that require authentication.

Once the page is fully downloaded, the HTML rendering engine can be employed to generate a snapshot of the page, save a low-fidelity version of that snapshot to an image file, and generate an appropriate HTML/Javascript overlay to use the snapshot as a menu to other subpages. At the end of this phase, the snapshot image and HTML can be sent to the client browser while the rest of the subpages are generated. The user should be satisfied with seeing a familiar screen shot and branding from the desired site.

For subpages, any attributes that need to be applied at the raw source level can be applied at this point, which we refer to as the filter phase. This can include extremely simple filters such as changing the `doctype` and `title`, or blanketly removing `css` and `script` tags. Slightly more complex filters would include rewriting all images to reference a low-fidelity image cache or different server. The page could be completely adapted after just a few simple filters, avoiding a DOM parse altogether, and assuming the snapshot is served from the cache, the work of the proxy could be done at this point.

For more complex modifications, a DOM parse is necessary. The m.Site framework has the capabilities of the popular source formatting tool HTML Tidy [22] compiled in. This library is applied at the filter phase, and is used to convert HTML to XHTML, which enables parsing by the wide array of XML/DOM manipulation tools available, as most of the XML-based tools won't handle malformed XML. The m.Site framework is modular enough to allow different libraries to be employed for DOM parsing (and subsequent filters based on the DOM tree), though for the next phase it is tightly integrated with the DOM parse provided by WebKit.

At the end of the attribute phase, all newly-allocated subpages are written to the file system in the client's session directory (see Figure 1). Any pre-rendered images are written to the client's image subdirectory, and any newly-generated shared images are written to the public cache. Some of the more powerful attributes call for m.Site to generate the server side php code to manage any

interactions required as a result of the custom attributes, for instance, to satisfy AJAX requests.

Mobile client detection Detection of a mobile device can be accomplished in a number of ways, but common practice is to use a set of heuristics that are kept up-to-date with new browsers and devices[1]. For our purposes, it is assumed that the client is already identified, and has either been automatically redirected to the proxy, or has explicitly chosen to use the proxy service for a particular page. Note that not all pages require a proxy to be mobile-friendly.

Object identification As a page is loaded for the first time, the proxy server must have a way to identify objects on the page, so that attributes and content adaptations can be applied.

The m.Site framework supports multiple object identification techniques, including source-level rules and heuristics. As in other systems [12, 1], a DOM-based approach is supported using XPath. Similarly, objects can be identified using new CSS 3 selector support, since the framework integrates a server-side port of the popular jQuery [8] DOM manipulation library. Page modifications can be made directly to a parsed DOM. Likewise, modifications can also be made at the source level, rather than by manipulating the DOM tree, which can expose some optimizations.

3.3 Attribute System

The power of the m.Site framework originates from the very rich attribute system, which makes it possible to customize a site's layout and adapt its content for mobile browsers. The attributes provide a site administrator with fine-grained control over the rendering of pages, and also provides new adaptation techniques not available in other systems, such as partial pre-rendering, a subset of which is described below.

Pre-rendering. For complex pages, considerable time is spent in a mobile browser downloading content, parsing, rendering CSS and HTML, and fetching additional images. A page, subpage, object, or object group can be marked to be completely rendered on the server side into a single graphic, saving much computational effort on the mobile device. Additional attributes allow the rendered image's fidelity to be lowered, reducing network bandwidth. In the index page of our test site, this technique can reduce wall-clock load time by a factor of 5. Pre-rendered objects can be dynamically linked to subpages, creating a mobile-friendly menu.

Page splitting. Any object, object group, or page can be split and set to render in its own separate HTML file, thus creating a subpage. If the subpage is combined with the pre-rendering attribute, it will be made up of simple pre-rendered images. Otherwise, the HTML making up that object will still be intact,

[1] See [7] for more information about the detection of a mobile device.

and will be delivered to and rendered on the client's browser. For instance, a long column of links may be identified and moved to its own page.

Sub-subpages. Subpages can also be further split into more subpages. When a subpage is split, it allows for a hierarchical navigation reminiscent of that provided by [25].

Object dependencies. When a subpage is set to be rendered in its entirety on the client side (HTML and CSS rather than a pre-rendered graphic), certain objects such as scripts that are needed to render the subpage may only exist in the master document or other subpages. By identifying these dependencies in the visual tool, we allow Javascript, CSS, and other objects to be pulled into the subpage as needed. This allows both non-visual and visual contents to be repeated in multiple subpages. The approach taken in other systems is to repeat head content on all subpages [25]. Unfortunately, this approach misses cases, where Javascript and other functionality are located in the body of pages. m.Site allows scripts and other content to be pulled from any portion of the page, and duplicated on as many subpages as is desired. Similarly, content such as ads, and navigational aids such as jump-menus can be made to appear on every subpage. Since any object can be duplicated on any subpage, this provides superior control over regular page-splitting approaches.

Javascript insertion / removal. Javascript functions can be dynamically inserted into the HTML source before rendering on the server, as well as after rendering. For instance, to modify how the server renders, one script can be used to manipulate the DOM tree to control certain layout elements, akin to [1]. For the client, a second script can be inserted to create a mobile-friendly navigational menu from the rendered elements. This is sort of modification cannot be realized by using Javascript-based content adaptation systems alone, such as [1, 12].

Object insertion, removal, relocation, and replacement. When adapting a mobile layout, we allow HTML, CSS, and Javascript to be manipulated by the proxy. Objects can be inserted, for instance, to support adding an ad to the bottom or a breadcrumb navigational element at the top of each subpage. Objects can be hidden (via CSS style properties) when it arrives on the client, or stripped out of the source completely. Objects can also be relocated or duplicated into disparate subpages. Lastly, objects can be replaced entirely. For instance, if a mobile-friendly version of a client Javascript API exists, the desktop-based library can be replaced outright. Another example is the replacement of a logout button with a get parameter, which allows cookies to be cleared on the proxy.

Partial CSS rendering. A complicated CSS design can take much time to render on a mobile device. Sometimes, it is desirable to take a portion of CSS code, replace the text with stretched one-pixel placeholders (to allow the layout engine to properly size the object), and take a snapshot of the rendered object. We call this partial pre-rendering. The proxy takes responsibility for rendering the graphical component, but uses Javascript to render the text on the device. Thus, the rendered object can then be used as a background in a static subpage, while the device only needs to draw text in the proper location.

Image fidelity. As one would expect of content adaptation systems, objects can be passed to a post-processor before being made available to the client, allowing for manipulations in image fidelity and cropping. The attribute system is used to supply parameters to the post processor. For instance, when a full page is rendered into a high-fidelity png, it can consume upwards of 600K. This would take considerable time and bandwidth to send to the device. A post-processor can produce a reduced-fidelity jpg at 25-50k. When displaying a zoomed-out overview page on a small device screen, the lowered image fidelity is not noticeable, and only results in a faster load and rendering time.

Search. Search functionality is inherently lost when a web page is rendered on the server side. Although restructuring the mobile layout into subpages reduces the need to search, sometimes searchability is desired even on subpages, despite the associated costs. Thus, we allow an attribute to be defined as "searchable". At rendering time, a sorted word index is built on the server from the textual content read from the web page. The rendered location of each word is stored in a Javascript array along with the word list, and the ordered search index is then inserted into the subpage along with a Javascript binary search function. In order for the client to make use of the search functionality, the site administrator must define an HTML element (button or link) to make the initial Javascript call. Thus, the search attribute effectively allows pre-rendered images to be searched.

Object caching. Certain areas of a site may be defined as cachable across sessions, amortizing the initial pre-rendering cost across many users. Once a cacheable object is rendered, it is placed into a pre-render cache on the server and can be used by the attribute system as needed. Using the properties of the cache attribute, for instance, a cached snapshot of the main page of a site can be set to expire after an hour.

Sometimes it is necessary to be able to maintain interactivity for portions of a site. For instance, some areas of the site may be protected with HTTP authentication. If the proxy comes across a page that requires user input, the client is redirected to a lightweight HTTP authentication page. Once authenticated, the proxy stores this information and uses it on behalf of the client. Authentication information is stored and maintained separately across users. HTTP authentication can be set with the application of a single attribute.

Overall, the m.Site framework leverages these rich attributes to provide site administrators with as much control over the mobilization of the site as possible.

4 Evaluation

In this section, we describe how the m.Site framework can be applied in a real-world setting—a complex, template-driven dynamic web site. We present the modification of the various content elements on the site's main page as well as those attributes that we ultimately select for deployment on the site. Finally, we show our experimental results.

4.1 Anticipated load

The site used for testing runs the popular vBulletin [15] forum software for online communities. As of 2012, the site receives an average 2.2 million hits per day with as many as 1200 users online at a time, and with a historical doubling of traffic every 18 months. Like many catering to a growing and diverse community of users understand, the site's membership has grown large enough to expect streamlined mobile access. Hence, this load drives the need for a scalable and cost-effective mobilization solution. As vBulletin encompasses an active and broad community of site administrators with varying skills and capabilities, it is essential to provide a framework that is both accessible and useful, yet to be so it must be scalable, cost-effective, and have minimal deployment requirements.

4.2 Target usage

Figure 4 shows the main page of the test site rendered from a desktop machine at its native resolution. The site starts with a logo and leader board banner advertisement, followed by a box of navigational links and a login form. Below this is a transient box used for announcements, followed by a long list of about 30 forum descriptions (clipped for space) and links to each forum's most recent post. Underneath the forum listing is a display showing which members are logged in, with links to each online member's public profile. Toward the bottom is a box of site statistics, a list of birthdays, public calendar entries, and finally some additional navigational links. This layout is a nearly unmodified default template reminiscent of thousands of online forums, and as such serves as a suitable test candidate.

The entry page of the test site requires a total of 224,477 bytes to be received from the network, inclusive of all images, external Javascripts (of which there are about 12), and CSS files. On the BlackBerry Tour smart phone (528 MHz processor), wall clock rendering time for this forum listing page is 20 seconds. For a grounded comparison, a modern desktop browser renders the page in about 1.5 seconds. Over WiFi, a 3rd-generation iPod Touch (600 MHz) using the WebKit-based Safari renders the page in 4.5 seconds, and 9 seconds over 3G.

Over time, the page has grown more complex to suit the desktop user. For what is tantamount to a magazine's table of contents, 20 seconds can be a burdensome wait. Table 1 draws a comparison. By using m.Site to render a snapshot of the page on the server side, the user perceives a significant reduction in latency, and unlike text-based content adaptation, the site administrator still delivers a branded look. The snapshot is overlayed using an image map with links to content areas defined with the subpage attribute.

Though page load performance will be less of an issue as more modern, standards-compliant mobile browsers become the norm, the site administrator can still take advantage of content adaptation to mitigate the small form factor, and facilitate quick access to information on-the-go. Even with the incredibly responsive zoom capability of the iPod Touch, for many core site requests, only a small amount of information is needed from the web page. For instance, looking

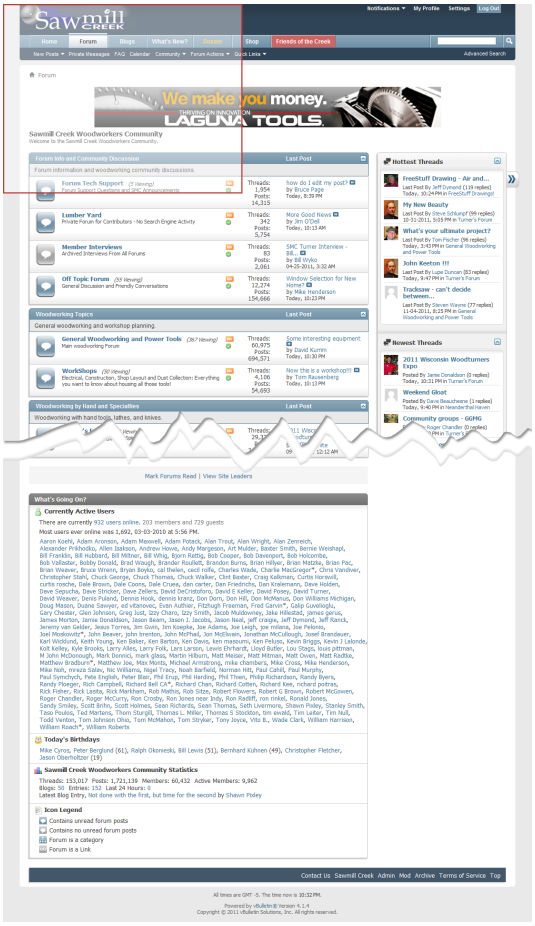

Fig. 4. SawmillCreek.org Test site rendered at full resolution.

up flight cancellations in an airport usually only requires a small subset of the functionality provided by most airlines on their web pages.

Fully zoomed in its native resolution, the BlackBerry Tour (480x325 browser area) displays only a small window into the normal site, as shown by the upper left box drawn in Figure 4. Such a small viewing window requires considerable scrolling to read, both vertically and horizontally. Indeed, this is not even wide enough to display a common leader board banner ad of 728 pixels wide, and obviates the need to adapt this banner by replacing it with a mobile-specific version. Ideally, this is done by selecting the ad and applying an attribute that directs its replacement at the source level.

Just as an HTML page can take many forms, m.Site attributes can be applied in many different ways, depending on the needs of the site administrator. A mobile visit to an online weather site or movie theater should probably focus

Device	Wall-clock Time
BlackBerry Tour browser page load	20 sec.
Snapshot page generation	2 sec.
Cached snapshot page to Blackberry	5 sec.
iPhone 4 via 3G	20 sec.
iPhone 4 via WiFi	4.5 sec.
Desktop browser page load	1.5 sec.

Table 1. Comparison of wall-clock time from initial request to browsable page.

on providing local weather or show times as quickly as possible, then perhaps national forecasts or box office descriptions. Recognizing that a mobile visit to an online woodworking community is akin to reading a magazine, the focus of our content adaptation on the entry page is to connect the reader with interesting threads as efficiently as possible, while maintaining the site's branding. Such a decision is an important factor in determining which content to display more prominently, while it should not cause functionality to be hidden on that basis alone. Thus, even though we will employ attributes to emphasize the forum listing, other functionality on the page will still be accessible to the user via subpages (rather than removed altogether).

4.3 Applying attributes to the test site

A user will typically perform one of two actions when visiting the main page: either logging in to access the site's private areas, or browsing the forum listing for interesting topics. Whereas the structure remains the same, the links on the forum listing page continually change content as new discussion threads are added. We detail how both of these areas are adapted for a mobile user as follows.

Upon visiting the site, the mobile user is presented with a quick-loading, cached snapshot of the entire site. Application of this attribute gives the user the satisfaction of an immediate response upon visiting the site. The snapshot is pre-rendered, saved at low fidelity, and stored in a public cache for 60 minutes. The image itself is also scaled down to prevent the user from having to zoom in before clicking. The main idea is to present the user with the site's overall branding and an efficient means of diving into the desired site content.

The subpage attribute allows document fragments to be moved into subpages, along with dependent CSS and Javascript snippets. As shown in Fig. 5, we have applied the subpage attribute to the login form. Clicking the snapshot, where the login form would have been, links the user to the login form subpage. The login form elements have multiple dependencies in the original HTML source, including CSS and Javascript, which are satisfied by inserting the dependent scripts underneath the head tag in the subpage using a copy attribute.

The logo box (table and image) is also copied (rather than moved) to the top of the login subpage, but the `src` attribute of the image is set to a mobile-specific

Fig. 5. SawmillCreek.org login form subpage rendered as a result of applying page-splitting, image replacement, and css injection attributes.

version of the logo. Figure 5 shows a screen shot of the adapted login subpage rendered on a BlackBerry Storm.

All of the defined subpage attributes contribute to an image map overlay, which is automatically generated for the main page snapshot. For each subpage generated, the coordinates and extents of the original document elements must be queried from the DOM, (in this case, the top left corner, height, and width), and are used to draw clickable rectangular image map regions on the snapshot. Each region links to its corresponding subpage. The queried coordinates map to the original-size document, but since the snapshot is scaled down, the m.Site framework implicitly translates the coordinates as well.

The site navigation links below the login box in the original site do not scale down at all. When viewed on a small display, the result is a single horizontal line of links (constructed as a table) that necessitates a horizontal scrollbar. To mitigate, we apply an attribute to transform the DOM, stripping the links from the segment and rewriting the HTML to list the links vertically, into two columns.

Whereas the default action for a subpage attribute is to render into a separate HTML file, setting one more attribute can allow the subpage to be loaded asynchronously and on demand into a `div` element in the current page. That is, any subpage can set to render into the current document using an asynchronous http request (AJAX). The m.Site framework injects the needed Javascript functions and creates appropriate `div` containers to enable this functionality on those pages that require it. The container is hidden and empty by default. When displayed, it can be centered in the viewport. Thus, it gives the appearance of being able to "activate" otherwise static portions of the pre-rendered snapshot, all without reloading the page. This has the added advantage of saving bandwidth and latency by not having to reload and parse large amounts of CSS and Javascript. The site's navigation links are loaded asynchronously through this method.

4.4 AJAX Support

Consider for a moment, the most typical use of asynchronous Javascript calls on a given website: a user clicks on a link, causing data to be retrieved into a DIV element, circumventing the cost of a full page load. At the low level, a user clicks on a link, triggering a Javascript onClick event, which in turn instantiates an asynchronous call to the server (usually a GET request), whose response is then marshaled to another Javascript function serving as a handler to populate a DIV element.

On mobile devices that support AJAX, such as Apple's iPad, iPhone, and Google Droid phones, no content adaptation is needed to maintain the original interactivity of the website. That is, the original asynchronous calls can be employed on these mobile devices, saving full-page rendering costs as is the case on desktop platforms. However, the next subsection shows how content adaptation can be used for these devices.

For non-AJAX capable devices, like the Blackberry's browser, content adaptation can be employed to restore AJAX-like interactivity. Previous work highlights a "remote browser in a proxy" metaphor [21] as a solution, but unfortunately, this solution does not scale well. How then, can AJAX interactivity be maintained without a remote browser?

As it turns out, the solution is simple—rewrite the link that gets sent to the device, and embed an additional function for the proxy to satisfy the request. For example, the following `onClick` handler for a "Show Picture" thumbnail loads a larger picture version when clicked:

```
$("#picframe").load('site.php?do=showpic&id=1')
```

The original site has a server-side AJAX request handler invoked when the action `showpic` and an `id` are supplied to the script. Upon validation (proper session, security, and accessible id), the desired image is displayed. This link would be adapted, using server-side jQuery, with a static call to the proxy, as follows:

```
proxy.php?action=1&p=1
```

This illustration is invariably simple, but is easily extended. When a site is integrated with the Google API and Yahoo (YUI) DOM API's, the link translation is more complex, but just as easily performed by the framework. Why not replace the link with a direct call to `site.php`? In more complex instances the returned result is rarely a simple picture, and often contains XML or JSON and must be massaged via Javascript. This can be handled easily and efficiently in the php-based proxy augmented with server-side jQuery. Using a CSS3-style pattern allows the content adaptation to be more robust to changes. The proxy's `action` is no more than a function, and the parameter `p` is its parameter representing the `id` in the original call.

4.5 AJAX Evaluation

Many popular "apps" for Apple's iPad and iPhone platforms are site-specific content-adaptation applications, which make navigation of data and page intensive sites more convenient for mobile users, in spite of the fact that these devices do already support AJAX, Javascript, and many HTML5 features.

To evaluate our approach, we choose to adapt a portion of the popular classified listing engine (CraigsList.com) using our proxy. Craigslist users browse pages of classified listings organized by category and sorted by date; clicking on a link brings the user to a new page with the contents of the selected ad. The evaluation device is a 1st-generation iPad and we want to take advantage of its extra screen real estate, to help the user locate desired information faster.

Craigslist does not ordinarily require any AJAX requests, which for a mobile device means an overuse of the browser's tiny back button, and continual reloading of pages. Rather than designing a platform specific application through the Apple developer network, we develop a browser-based content adaptation application for Craigslist, which simplifies navigation by *adding* asynchronous data loads.

Figure 6 shows the before-and-after results using our prototype. On the top left is the original site rendered in Google Chrome containing a page of links to classified ads. The second and third snapshots show the links and text identified in the administrator tool, and the proxy code. The last illustration is the result of content adaptation applied to the original page.

The adapted site is split into two DIV panes, with the left pane containing the list of classified listings, and the right pane containing the detailed classified listing. When an ad in the left pane is clicked, an AJAX call is dispatched to the proxy in the manner previously described. The proxy checks the cache for the downloaded page, and if it does not exist, fetches the page from CraigsList, performs the content adaptation, and outputs it to the iPad as an AJAX response. The result is a much more enjoyable browsing experience on the mobile device.

4.6 Limits to Scalability

As mentioned previously, our work is motivated by acknowledged scalability issues with the approach used in [21]. In that system, a costly browser instance is required for every client request. The core of our approach is to mitigate this cost, by (1) amortizing rendering costs across multiple clients where possible, and (2) only using a full-scale browser instance when absolutely necessary for server-side graphical rendering. In most cases, the server-side browser metaphor is maintained by our proxy as a lightweight and scalable substitute.

To illustrate the improvement offered by our approach, we conduct a series of tests to measure the throughput (i.e., the number of satisfied requests) under various load conditions. We simulate repeated client requests for a remote site, while we vary the percentage of requests that require instantiation of a full browser instance. Our tests are performed on commodity dual-core hardware running Windows Vista, Qt, and WebKit, and do not make use of a thread

Fig. 6. Adding AJAX calls to enhance Craig's List for the iPad.

pool of browser instances. Using a browser pool can potentially violate security assumptions if shared by multiple clients.

Figure 7 shows our results. The tests are performed three times per data point, each over a one minute measurement window. The interarrival times between full-scale rendering requests are randomly distributed. A U[0,1] random number is assigned to each request; if the number exceeds the percentage being tested, the request is marked as not requiring a browser instance. As the figure depicts, by limiting the number of requests requiring a graphical render, we are able to increase the number of satisifed requests from 224 to 29038, two orders of magnitude. We expect similar results on non-commodity server hardware as well. For many sites like our test site, rendering the main snapshot is only required once per hour and can be shared by multiple users. Caching and amortizing rendering costs over thousands of clients makes the cost negligible.

Fig. 7. Increased scalability with addition of lightweight proxy for majority of requests.

5 Conclusion

As more and more users access the Web via mobile devices, it becomes essential for site administrators to adapt content for mobile users. However, mobilizing existing content through templates and custom redesign is a costly, tedious, and time-consuming process. Thus, tools to streamline the process are in great demand. In this paper, we have presented the m.Site framework, a powerful set of tools that bolster productivity and provide site administrators with the ability to adapt web content for their mobile users. With a visual tool, a familiar *attribute* paradigm, and an extensible server-side framework, a site administrator can quickly generate code for content adaptation proxies that streamline site functionality. By building a pluggable framework and only calling on a browser instance when absolutely necessary, we have improved the scalability issues from previous work. We have built a prototype of m.Site and validated its effectiveness as a content adaptation tool on real online community websites.

Acknowledgements: We are grateful to the anonymous referees for their insightful feedback. This work was partially supported by NSF grant 0901537 and ARO grant W911NF-11-1-0149.

References

1. Greasemonkey. `www.greasespot.net`, 2009.
2. Greasemonkey user scripts. `www.userscripts.org`, 2009.
3. Platypus firefox extension. `platypus.mozdev.org`, 2010.
4. Apple dashcode. `http://developer.apple.com/tools/dashcode/`, 2012.
5. Apple webkit html engine. `http://webkit.org`, 2012.

6. Browser compatibility information. `www.quirksmode.org`, 2012.
7. Detect mobile browsers. `http://detectmobilebrowsers.mobi`, 2012.
8. jquery, the write less, do more, javascript library. `www.jquery.com`, 2012.
9. Opera-mini browser. `www.opera.com`, 2012.
10. Skyfire mobile browser. `www.skyfire.com`, 2012.
11. T. Bickmore and B. Schilit. Digestor: Device-independent access to the world wide web. In *Proc. WWW-6*, pages 655–663, Santa Clara, CA, 1997.
12. N. Bila, T. Ronda, I. Mohomed, K. Truong, and E. de Lara. Pagetailor: Reusable end-user customization for the mobile web. In *ACM MobiSys'07*, San Juan, Puerto Rico, June 2007.
13. Y. Chen, W.-Y. Ma, and H.-J. Zhang. Detecting web page structure for adaptive viewing on small form factor devices. In *Proceedings of the 12th international conference on World Wide Web*, New York, NY, USA, 2003.
14. L. Deboosere, B. Vankeirsbilck, P. Simoens, F. De Turck, B. Dhoedt, P. Demeester, M. Kind, F. Westphal, A. Taguengayte, and T. Plantier. Mobithin management framework: design and evaluation. In *3rd international Workshop on Adaptive and Dependable Mobile Ubiquitous Systems*, London, United Kingdom, July 13 - 17 2009.
15. I. B. Inc. vbulletin forum software. `www.vbulletin.com`, 2012.
16. J. Kim, R. Baratto, and J. Nieh. pthinc: a thin-client architecture for mobile wireless web. In *15th international Conference on World Wide Web (WWW)*, Edinburgh, Scotland, 2006.
17. M. Md Fudzee and J. Abawajy. A classification for content adaptation systems. In *10th international Conference on information integration and Web-Based Applications & Services*, Linz, Austria, 2008.
18. I. Mohomed, J. Cai, and E. de Lara. Urica: Usage-aware interactive content adaptation for mobile devices. In *1st ACM European Conference on Computer Systems (EuroSys'06)*, Leuven, Belgium, 2006.
19. I. Mohomed, A. Scannell, N. Bila, J. Zhang, and E. de Lara. Correlation-based content adaptation for mobile web browsing. In *ACM/IFIP/USENIX international Conference on Middleware*, Newport Beach, CA, 2007.
20. A. Moshchuk, S. Gribble, and H. Levy. Flashproxy: transparently enabling rich web content via remote execution. In *6th international Conference on Mobile Systems, Applications, and Services (Mobisys)*, Breckenridge, CO, 2008.
21. J. Nichols, Z. Hua, and J. Barton. Highlight: a system for creating and deploying mobile web applications. In *21st Annual ACM Symposium on User interface Software and Technology (UIST '08)*, Monterey, CA, 2008.
22. D. Raggett. Html tidy. `http://tidy.sourceforge.net`.
23. T. Richardson, Q. Stafford-Fraser, K. Wood, and A. Hopper. Virtual network computing. In *IEEE Internet Computing 2, 1*, pages 33–38, January 1998.
24. B. Schilit, J. Trevor, D. Hilbert, and T. Koh. m-links: An infrastructure for very small internet devices. In *7th Annual international Conference on Mobile Computing and Networking (Mobicom'01)*, Rome, Italy, 2001.
25. X. Xiao, Q. Luo, D. Hong, H. Fu, X. Xie, and W. Ma. Browsing on small displays by transforming web pages into hierarchically structured subpages. In *ACM Trans. Web 3, 1*, pages 1–36, January 2009.
26. Y. Zhang, X. Guan, T. Huang, and X. Cheng. A heterogeneous auto-offloading framework based on web browser for resource-constrained devices. In *International Conference on Internet and Web Applications and Services*, pages 193–199, 2009.