A Case for Persistent Caching of Compiled JavaScript Code in Mobile Web Browsers

Lauren Guckert, Mike O'Connor, Satheesh Kumar Ravindranath, Zhuoran Zhao, Vijay Janapa Reddi
Dept. of Electrical and Computer Engineering
University of Texas at Austin
{*lguckert, mike.oconnor, satheesh*}*@utexas.edu, zoranzhao@gmail.com, vj@ece.utexas.edu*

***Abstract*—Over the past decade webpages have grown an order of magnitude in computational complexity. Modern webpages provide rich and complex interactive behaviors for differentiated user experiences. Many of these new capabilities are delivered via JavaScript embedded within these webpages. In this work, we evaluate the potential benefits of persistently caching compiled JavaScript code in the Mozilla JavaScript engine within the Firefox browser. We cache compiled byte codes and generated native code across browser sessions to eliminate the redundant compilation work that occurs when webpages are revisited. Current browsers maintain persistent caches of code and images received over the network. Current browsers also maintain in-memory "caches" of recently accessed webpages (WebKit's Page Cache or Firefox's "Back-Forward" cache) that do not persist across browser sessions. This paper assesses the performance improvement and power reduction opportunities that arise from caching compiled JavaScript across browser sessions. We show that persistent caching can achieve an average of 91% reduction in compilation time for top webpages and 78% for HTML5 webpages. It also reduces energy consumption by an average of 23% as compared to the baseline.**

I. Introduction

JavaScript has permeated into every aspect of the web experience. Over 92% of all webpages rely on JavaScript [7]. With the proliferation of HTML5 and its associated mobile web applications, the world is moving into an age where the majority of webpages involve complex computations and manipulations within the client JavaScript engine. As the number of webpages that provide rich and interactive content (e.g. using HTML5) grows, the demand for fast and efficient JavaScript will also steadily grow. The power required to support JavaScript features is also an emerging concern, since a large fraction of all webpages are increasingly being accessed from mobile, power-constrained handheld devices such as smartphones and tablets [5].

On a mobile system, the computation time required for JavaScript compilation on a given page can be significant. For example, we find that on a dual-core 1.7 GHz ARM Cortex-A15 system typical of a high-end tablet, the ten most popular webpages spend, on average, over 1.8 seconds compiling JavaScript. This accounts for 22% of the total time spent executing JavaScript. Compilation time is higher for HTML5-based webpages, spending one second on average. Webpage load time and JavaScript execution time are both an important concern because 59% of consumers wait at most 3 seconds for a webpage to load on their mobile device before abandoning the webpage [8]. To make matters worse, 46% of e-commerce consumers are unlikely to return to the page if it failed to load fast during their last visit [8]. Therefore, this work looks at an approach to preserve this effort, using a *persistent cache* to retain the compiled JavaScript across browser sessions.

We examine the characteristics of top webpages, as well as examples of emerging, highly-interactive HTML5 webpages in order to understand the characteristics of JavaScript compilation. We use the Mozilla JavaScript engine used by the Firefox web browser for this study. JavaScript compilation in Firefox takes place in several passes that together lead to runtime overheads. The first pass parses the JavaScript text and typically generates interpretable byte codes. A subsequent pass, for code executed more than a few times, takes the byte codes and generates simple native code with few, simple optimizations. Later passes may take hot code and generate highly optimized native code. These passes correspond to the Bytecode compiler, JaegerMonkey compiler, and IonMonkey compiler, respectively, within the Mozilla JavaScript engine used by the Firefox web browser.

On the basis of profiling hot and cold function compilation behavior, we propose persistent caching as a means to improve both performance and energy consumption of web browsers. Functions that are infrequently called are interpreted but are not translated or optimized. The overhead for compilation is incurred each time the function is executed. For functions which are invoked frequently, they are compiled, translated, and stored by the JIT compilers, JaegerMonkey and IonMonkey. Thus, the compilation overhead for frequently-executed functions is better amortized across executions than for less frequent functions. Persistent caching attempts to alleviate this overhead by storing these less-frequent functions across browser sessions. By doing so, the compilation overhead is not incurred for subsequent executions and can be amortized in a similar manner to frequently executed functions.

Since our focus is to present a case for persistent caching of compiled JavaScript code, in this paper we present an analytical model to reason about the relative costs of our proposed solution as compared to the current Mozilla JavaScript engine. We then show the performance and power benefits that can be obtained for both sets of benchmarks when persistent

caching is implemented. We show that persistent caching can achieve an average of 91% reduction in compilation time for top webpages and 78% for HTML5 webpages. Our approach also reduces energy consumption by an average of 23% for both the top and HTML5 webpages.

This paper makes the following contributions:

- We characterize JavaScript webpages ranging from common popular webpages to emerging, interactive HTML5-intensive webpages.
- We propose persistent caching to improve the performance and energy consumption of JavaScript webpages.
- We verify the benefits of persistent JavaScript caching for the Mozilla Firefox web browser.

In Section II we motivate persistent caching in web browsers by characterizing webpages in terms of hot and cold code. Section III explains our persistent cache model. Section IV describes an analytical model to study the benefits of our approach. Section V uses the model to evaluate performance benefits, in addition to measuring memory usage and energy consumption improvements. Section VI discusses the related work. We conclude with our future work and summary of our contributions in Sections VII and VIII, respectively.

II. Initial Characterization

In order to analyze the potential opportunity of persistent caching in Firefox's JavaScript Engine, we selected a set of currently top webpages in the Internet, as well as a set of webpages containing forward-looking, highly-interactive HTML5. We measure and characterize the amount of JavaScript code and its execution behavior across the webpages.

A. JavaScript Benchmarks

For the set of common, top webpages, we used the current top 10 US webpages as documented by Alexa—The Web Information Company [1]. The prevalence of these top webpages and frequent access to them suggests that they are important to analyze and understand. In addition to the initial "webpage load," the JavaScript engine is also triggered when interactive events occur, such as on a mouse click. In order to analyze the JavaScript contributions of interactive user behavior, we also perform a specific use case test (beyond the initial load) for each of the top webpages. For a breakdown of the webpages and the specific use case we performed for each see Table I. Note that the Wikipedia webpage does not have a use case due to measurement limitations.

For the HTML5 webpages, we chose 9 of the current top 10 HTML5 webpages as documented by eBizMBA [3]. The tenth caused the mobile benchmarking platform to hang, and was, therefore, not studied. A full list of these webpages is also given in Table I. The use cases for these webpages is just the initial webpage load during which a substantial portion of JavaScript code is executed. We study HTML5-intensive webpages because there has been much speculation that these types of JavaScript webpages are representative of the future, and that webpages can be expected to grow in visual and computational complexity.

TABLE I: Our Webpage Benchmarks

Alexa's Top Webpages		**HTML5**
Webpages	*Use Cases*	*Webpages*
Amazon	Single-word search, click first link	Art of Stars
Bing	Single-word search	CNN Ecosphere
Craigslist	Single-word search, click first link	The Expression Web
Ebay	Single-word search, click first link	360 Langstrasse Zurich
Facebook	Log in and click first profile in News Feed	The Lost World's Fairs
Gmail	Log in and click first email in Inbox	Soul Reaper
Wikipedia	N/A	This Shell
LinkedIn	View most recent updates in news feed	Universeries
YouTube	Single-word search and select first result	The Wilderness Downtown
Yahoo	Log in and click first news item in the feed	
Google	Perform single-word search	

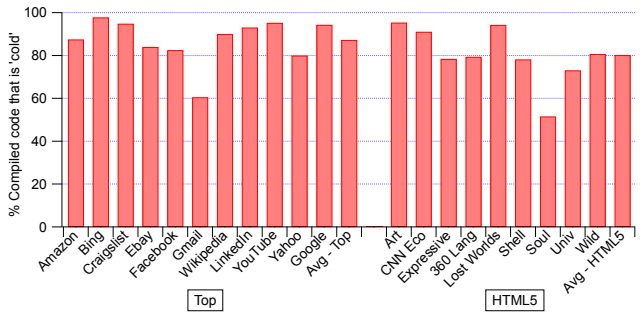
B. Function Execution Characteristics

A key benefit of persistent caching is its ability to amortize the overhead of compiling infrequently executed, or cold, code across many browser sessions. Often, cold code within a single execution is hot across executions as it often consists of initialization functions. Ultimately, persistent caching is most beneficial when cold code contributes heavily to both the JavaScript compiled code and that code's execution time.

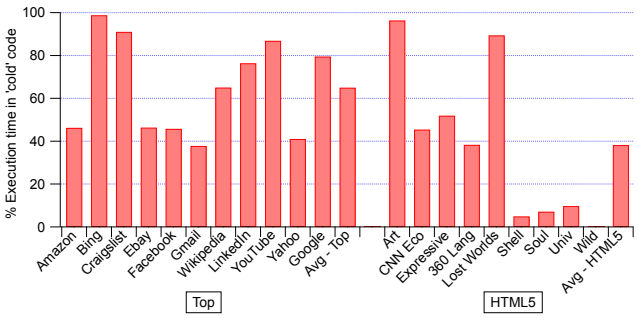
Firefox classifies any function interpreted over 43 times as "hot" at which point the JaegerMonkey compiler generates simple, native machine code. JaegerMonkey invokes the optimizing IonMonkey compiler only when the native code executes more than 10,240 times. For the purposes of persistent caching in this paper, we do not distinguish hot code separately between JaegerMonkey and IonMonkey. All interpreted code is "cold" code and all compiled code is "hot" code.

We performed functional profiling using the Firebug [4] add-on to collect various metrics, such as the number of unique functions, executions per function and the execution time per function. These metrics help us determine the prevalence of cold code, as well as the amount of time spent executing the cold code. Without a high occurrence of these characteristics, the need for persistent caching is not as persuasive.

Figure 1a shows the amount of cold code for the webpages during the initial webpage load. Cold code accounts for an average of 87% of the compiled JavaScript code for the top webpages and 80% for the HTML5 webpages. Even for the webpages with the lowest percentages of cold code, such as Gmail and Soul Reaper, cold code still accounts for over 50% of the compiled code. We attribute the general lower percentages for HTML5 webpages to the fact that these webpages are highly computational and repeatedly call a limited set of JavaScript functions to perform their computations.



(a) Percent of compiled JavaScript code that is cold.



(b) Percentage of execution time spent in infrequent functions.

Fig. 1: Cold code analysis of JavaScript code.

We also collected profiling metrics for each of the top webpages use cases. As we mentioned previously, this is to capture user driven JavaScript activity. While the data is not shown here, the percentage of cold code increases over the initial webpage load to an average of 97%. The prevalence of these colds functions for webpages suggests that persistent caching could produce performance benefits by amortizing compilation overhead across web browser sessions.

Although the amount of cold code is large, if a substantial portion of the execution time is spent in a small number of hot functions, then any relative benefit from persistent caching would be minor. Thus, this infrequently executed cold code must account for a significant portion of execution in order to make a strong case for persistent caching. Figure 1b shows that the percentage of execution time spent in these infrequent functions is also substantial. On average, the top webpages spend 65% of the initial webpage load execution time on executing infrequent, cold functions. In some cases, such as Bing and Craigslist, almost 100% of the execution time is dedicated to the compilation of these functions.

Beyond the initial webpage load, the percentage of execution time increases to an average of 80%. From this observation, we conclude that user driven JavaScript code is typically colder than the non-interactive JavaScript code in a webpage. We do not present this behavior in a graph.

The HTML5 benchmarks have a lower percentage of cold code (as found in Figure 1a), spending an average of 38% of their execution time in cold code. Although the average percentages for the HTML5 webpages are lower than that of the top webpages, some of the HTML5 webpages, such as Art of Stars and Lost Worlds Fairs spend over 80% of execution time in cold code.

The large quantity of cold code and its significant contribution to the overall execution time for both top and HTML5 webpages suggests that significant performance boosts can be obtained from persistent caching. The continuation of these trends beyond the initial webpage load shows the potential benefits extend throughout the browser session.

III. Persistent Caching

We propose persistent caching as an approach to improve both performance and power for JavaScript webpages. As the preliminary profiling results show, both HTML5 and top real-world webpages spend a significant portion of execution time interpreting and compiling functions that are infrequently executed. This observation suggests that persistent caching can lead to a significant reduction in compilation time.

Our persistent cache evaluation leverages the existing Mozilla JavaScript engine framework. We pinpointed key locations in the source which are responsible for interpretation and compilation of JavaScript functions. Each time a function is to be compiled, a 128-bit MD5 hash is computed on the source-code to be compiled (the JavaScript source text in the case of the bytecode compiler, and the bytecodes in the case of the JaegerMonkey and IonMonkey compilers). This hash, coupled with the length of the source, is used as a key to determine if this function has been previously compiled. If the hash ID is identified in the persistent cache, the previously compiled function will be recalled. Otherwise, the compilation will take place and store the resulting compiled function to the persistent cache for future use. *The hash ID ensures that we do not execute stale JavaScript code from the code cache.*

As an optimization, our persistent cache does not cache "small" functions that take less time to recompile than to retrieve from the cache. To find this threshold, we swept across increasing bytecode sizes and measured the average time it took to compile versus retrieve from our cache structure and found the threshold to be 32 bytes. Thus, we persistently cache all compiled bytecode that is greater than this size.

The proposed structure also provides the benefit of inter-webpage amortization. In the current Firefox implementation, if an identical function, for example a library call to jQuery, is called on multiple webpages, each webpage will independently interpret this function. In our proposal, these identical functions will have identical MD5 hashes, and the function will be read from the persistent cache rather than incurring the cost of interpretation and re-translation.

TABLE II: Chromebook Specifications

Samsung Exynos 5250 Dual-core ARM @ 1.7 GHz	Ubuntu Chrome OS
Two ARM Cortex-A15	32 KB I-cache, 32 KB D-cache
64-bit wide interface to 2 GB of LPDDR3 DRAM	1 MB L2 Cache
Class 10 16 GB mounted SD card mounted as swap	16+ GB of onboard flash

We assume a filesystem-based persistent cache and an in-memory hash table for looking up the MD5 hashes. For the purposes of our initial study, we have not placed any restrictions on the maximum size of the persistent cache structures. We acknowledge that our approach with an unbounded cache can have adverse effects on some systems, particularly on mobile platforms with limited memory. We leave refinements and cache management policies as future work.

IV. Methodology

Our implementation is built on top of the existing in-memory Firefox code cache, and it is yet to completely represent the actual timing behavior of a persistent cache. Therefore, we developed an analytical model that allows us understand the potential benefits of persistent caching. The model establishes an upper bound on the improvements our persistent cache model can obtain and the maximum penalty due to the write of the persistent cache after an initial compile.

In order to accurately model a real-system persistent cache, we modified the Mozilla JavaScript engine source code to measure the timing costs for the corresponding accesses to the persistent cache. We use high-resolution timers to measure the time required to perform the hashing, lookup, and file access operations that would be required in a persistent cache system. For each compiled function, we index based on the hash, seek to the appropriate offset within the 256 MB file system-based persistent cache file, and read or write the corresponding amount of bytecode. We measure the timing behavior of both the write and read accesses to determine both the cost for storing the compiled code on a first access and reading it back in the case of a persistent cache hit.

V. Results

To understand the potential benefits and consequences of our persistent cache, we collect measurements on a mobile system in terms of execution time, power, and memory usage. We used a dual-core ARM Samsung Chromebook. The system's specifications are given in Table II. Our results show that the persistent cache model achieves significant compilation time and energy improvements while requiring manageable in-memory and persistent memory footprints.

A. Performance Speedup

We evaluate two scenarios of persistent caching. In the first scenario, we flush the persistent cache prior to each website being loaded. In the second scenario, which is based on a more realistic use-case behavior, the persistent cache is already partially populated. We populate the cache by visiting each of the webpages three times each, so that the model also captures persistent caching benefits across webpages.

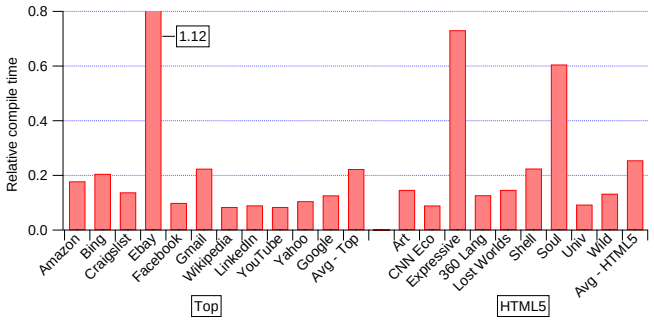


Fig. 2: Relative compile time for initially empty scenario.

The first scenario represents the worst-case overhead for populating the persistent cache and for hash lookup misses, giving us an upper bound on expected persistent caching penalties. The second scenario begins with a "warmed-up" cache, reducing the number of cache stores and increasing the likelihood of a hash lookup hit. In both the scenarios, we assume that the same set of functions is called during each visit to a webpage. Thus, the first time a webpage is accessed, any function not existing in the persistent cache incurs an initial overhead for storing its compiled bytecode. Each subsequent visit to the webpage performs zero compilation, instead reading the bytecode from the persistent cache. This establishes an upper bound on the improvements from a persistent cache.

For the initially empty cache scenario, the average overhead for the first visit to a webpage is 7% for both top and HTML5 webpages. For each subsequent visit the compilation time reduces to an average of 22% and 26% of the baseline version. Figure 2 summarizes the reduction in compilation time across all the webpages when they are visited subsequent times (after the first call which incurs the overhead). For example, Facebook, Wikipedia, LinkedIn, YouTube, CNN Ecosphere, and Universeries all improve in compilation time to less than 10% of the original baseline Firefox. Ebay is the only webpage that increased in compilation time. We suspect that this is because Ebay calls many functions a single time, and thus does not amortize the initial overhead of storing to the persistent cache.

For the "warmed-up" persistent cache scenario, the average overhead for the first visit to a webpage is 4% and 5% for the top and HTML5 webpages, respectively. The overhead reduces because many of the functions are already present in the persistent cache. The reductions in compilation time for subsequent webpage visits are also more significant for this "warmed-up" scenario, reducing to an average of 9% and 22% of the baseline for the top webpages and for the HTML5 webpages, respectively.

Figure 3 shows the relative compilation times for the sub-

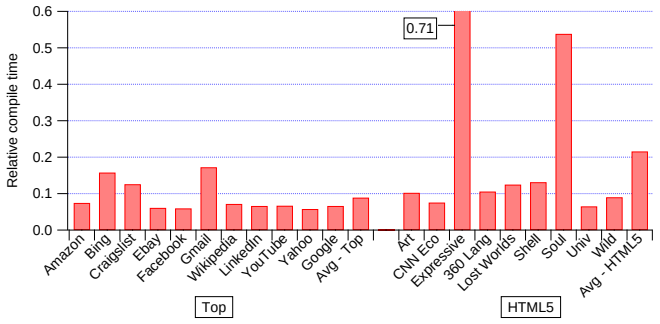


Fig. 3: Relative compile time for "warmed-up" scenario.

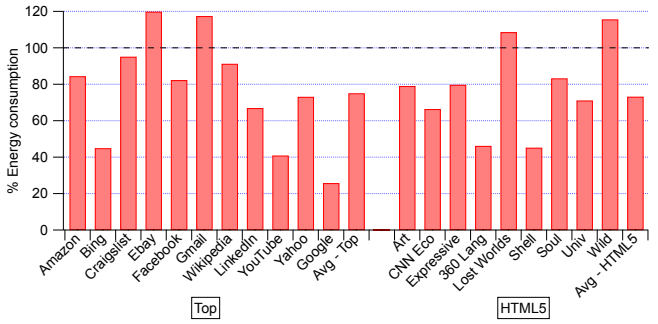


Fig. 4: Relative energy consumption over the original Firefox.

sequent webpage visits for this "warmed-up" scenario. The improved performance of this scenario over the previous scenario is due to the presence of multiple webpages' compiled bytecode within the persistent cache. The subsequent visits to a website can leverage precompiled functions from both the current webpages as well as past webpages. The fact that this results in greater performance improvements than the initially empty scenario suggests that many functions are in fact shared across webpages. To confirm this, we measured the reuse of functions across webpages (using previously described techniques) and found that 539 functions were called on more than one site, implying that a persistent cache has additional benefits for inter-webpage scenarios.

Overall, our results show that the savings that we can achieve using persistent caching are as great as 94%. The initial overheads for our persistent cache are less than 7% over the baseline. At first glance, this persistent cache overhead may seem high. However, it is likely that such overhead can be effectively masked from the user. For example, the persistent cache can be written "behind the scenes" during the browser shutdown phase. In both cases (i.e., overheads and savings), the real-world scenario shows greater performance over the initial scenario. These findings indicate that cold functions are indeed getting reused in our persistent caching model.

B. Memory Characterization

The main consequence of persistent caching model is the increased memory usage to index and store the additional compiled code. Persistent caching is only beneficial if the improvements in power and performance are not overwhelmed by the cost of the additional memory.

Our persistent cache implementation requires two primary structures. First, an in-memory hash table, with each element consisting of a 160-bit key (128-bit hash plus 32-bit source length), 32-bit file index, 32-bit length, and 32-bit "next" pointer (to support chaining when there are hash bucket collisions). This data structure is also stored and restored from the filesystem on each browser launch and exit. Second, we have a filesystem-based persistent cache file containing the native code for the cached functions.

In practice, the in-memory structures are quite small. The entire persistent cache hashtable in our studies accumulated across many runs of the full benchmark suite and other webpages was less than 336 KB. The in-filesystem persistent cache that holds the retained compiled code was modeled as a 256 MB file, and is roughly the size of the total compiled code for all the benchmarks together.

C. Power Characterization

Our persistent cache model shows energy reductions up to 74% compared to the baseline Firefox browser. We measured hardware power values on a ARM Cortex A9 Pandaboard, which is a development board. It is not possible to collect fine-grained power measurements on the Chromebook. Although we are performing the power measurements on a different setup than the timing measurements, we found that the timing improvements on the ARM A9 are similar to those collected for the Chromebook, averaging around 20% relative compilation time over the baseline.

In order to establish reliable power measurements, we only measure the power supplied to the A9 SoC itself. Our sampling rate was set to 200 KS/s. Also, in order to compare between top webpages and HTML5 webpages, we only measure the energy for the initial webpage load of each webpage.

Figure 4 shows the measured energy savings that our persistent cache model achieves over the baseline version. On average, energy consumption reduces to 77% of the baseline for both types of webpages. However, many webpages experience much higher improvement. For instance, Google reduces to nearly 25% of the baseline while Bing, YouTube, and This Shell all consume less than 50% of the baseline's energy consumption. These dramatic improvements for mobile search engines are key since half of all local searches occur from mobile systems [5]. Note that there were cases where a webpage experienced higher power usage for the persistent cache version but none were beyond 20% of the baseline. We speculate that these webpages perform more cache accesses to small bytecode chunks, which results in higher energy consumption than interpreting the bytecode.

The large number of webpages that benefit in terms of energy

reduction outweigh the small number of webpages that suffer from energy increases. This suggests that persistent caching can lead to an overall energy reduction.

VI. Related Work

Recent works, such as v8 [2] and SunSpider [6], have focused on developing and characterizing benchmark suites and showing their superiority over other suites. JSMeter from Ratanaworabhan [11] characterizes the v8 and SunSpider suites alongside real-world webpages. This work performs memory and functional execution characterization of JavaScript webpages but does not investigate HTML5 webpages. BBench [10] also proposes a representative web benchmark suite. However, this work studies microarchitectural characterization of webpages and limits the scope to interactive webpages on mobile platforms. These works did not propose a methodology to achieve performance boosts for webpages.

Previous work on persistent caching has focused on specific dynamic compilation tools. Reddi et al. [12] showed the potential improvements of a persistent code cache within and across executions of an application. However, they limit their study to the binary instrumentation tool PIN, which does not perform staged compilation as in JavaScript. Similarly, Bruening and Kriansky [9] show the speedup and memory improvements that persistent caching can achieve for their dynamic instrumentation engine, DynamoRIO. Serrano et. [13] developed a "quasi-static" compiler for Java Virtual Machines (JVM) systems. Unlike all these works, we explore the effectiveness of persistent caching in real-world JavaScript compilers. More specifically, our approach is a purely online mechanism that is geared towards JavaScript execution inside of web browsers.

In current open-source browsers, WebKit (used by Safari and Chrome) and Mozilla (Firefox), when the browser is exited, the work performed to parse and compile the JavaScript is discarded. These browsers maintain persistent caches of HTML, JavaScript source, and images received over the network, but do not retain the compiled JavaScript code. Current web browsers also maintain in-memory caches of recently accessed webpages (WebKit's Page Cache or Firefox's Back-Forward cache). However, they do not persist across browser sessions. These in-memory caches are effectively just the data structures generated when viewing a webpage, not code.

As far as our research effort is concerned, we believe that we are the first to propose and study a persistent cache for the Firefox web browser. We are also the first to perform a preliminary characterization comparing real-world and HTML5 webpages for a browser implementing a persistent cache. Finally, we are the first to analyze the power impact of these webpages, both with and without a persistent cache.

VII. Future Work and Limitations

In this work, we took an opportunistic approach to investigating the benefits of persistent caching in a JavaScript engine. We did not enforce restrictions on memory usage and limited our scope to Firefox's JavaScript engine and its specifications. In the future, we plan to implement the persistent caching model under more realistic constraints and evaluate the practical power and performance advantages of a persistent cache in mobile web browsers.

VIII. Conclusions

As webpages become more complex and interactive, it becomes increasingly vital for JavaScript to be efficient and performant. Also, as the majority of web usage transitions to the mobile realm, the power and memory consumption of these JavaScript features must remain low. In this work, we propose the use of a persistent cache in order to reduce compilation time and energy usage. We prototyped a model of our implementation in the Mozilla JavaScript engine within the Firefox browser and studied the impact of our model on both top webpages as well as emerging HTML5 webpages. We find that persistent caching can achieve an average of 91% reduction in compilation time for top webpages and 78% for HTML5 webpages and it reduces energy consumption by an average of 23% compared to the baseline version.

References

[1] Alexa the web information company. http://www.alexa.com/topsites/countries/US, Accessed May 12, 2013.

[2] Chrome v8. https://developers.google.com/v8/benchmarks, Accessed May 12, 2013.

[3] ebizmba the ebusiness knowledgebase. http://www.ebizmba.com/articles/best-html5-websites, Accessed May 12, 2013.

[4] Firebug. http://getfirebug.com, Accessed May 12, 2013.

[5] Hubspot. http://blog.hubspot.com/blog/tabid/6307/bid/33314/23-Eye-Opening-Mobile-Marketing-Stats-You-Should-Know.aspx, Accessed May 12, 2013.

[6] Sunspider 1.0 javascript benchmark. http://www.webkit.org/perf/sunspider/sunspider.html, Accessed May 12, 2013.

[7] W3techs. http://w3techs.com/technologies/details/cp-javascript/all/all, Accessed May 12, 2013.

[8] What Users Want from Mobile. http://www.gomez.com/wp-content/downloads/19986_WhatMobileUsersWant_Wp.pdf, Accessed May 8, 2013.

[9] Derek Bruening and Vladimir Kiriansky. Process-shared and persistent code caches. In *Proceedings of the ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2008.

[10] Anthony Gutierrez, Ronald G. Dreslinski, Thomas F. Wenisch, Trevor Mudge, Ali Saidi, Chris Emmons, and Nigel Paver. Full-system analysis and characterization of interactive smartphone applications. In *Proceedings of the IEEE Intl. Symp. on Workload Characterization*, 2011.

[11] Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin G. Zorn. Jsmeter: comparing the behavior of javascript benchmarks with real web applications. In *Proceedings of the 2010 USENIX conference on Web application development*, 2010.

[12] V.J. Reddi, D. Connors, R. Cohn, and M.D. Smith. Persistent code caching: Exploiting code reuse across executions and applications. In *Proceedings of the IEEE Intl. Symp. on Code Generation and Optimization*, 2007.

[13] Mauricio Serrano, Rajesh Bordawekar, Sam Midkiff, and Manish Gupta. Quicksilver: a quasi-static compiler for java. *SIGPLAN Not.*, 35(10), October 2000.