JSMeter: Comparing the Behavior of JavaScript Benchmarks with Real Web Applications

Paruj Ratanaworabhan
Kasetsart University
paruj.r@ku.ac.th

Benjamin Livshits and Benjamin G. Zorn
Microsoft Research
{livshits,zorn}@microsoft.com

Abstract

JavaScript is widely used in web-based applications and is increasingly popular with developers. So-called browser wars in recent years have focused on JavaScript performance, specifically claiming comparative results based on benchmark suites such as SunSpider and V8. In this paper we evaluate the behavior of JavaScript web applications from commercial web sites and compare this behavior with the benchmarks.

We measure two specific areas of JavaScript runtime behavior: 1) functions and code and 2) events and handlers. We find that the benchmarks are not representative of many real web sites and that conclusions reached from measuring the benchmarks may be misleading. Specific common behaviors of real web sites that are underemphasized in the benchmarks include event-driven execution, instruction mix similarity, cold-code dominance, and the prevalence of short functions. We hope our results will convince the JavaScript community to develop and adopt benchmarks that are more representative of real web applications.

1 Introduction

JavaScript is a widely used programming language that is enabling a new generation of computer applications. Used by large fraction of all web sites, including Google, Facebook, and Yahoo, JavaScript allows web applications to be more dynamic, interesting, and responsive. Because JavaScript is so widely used to enable Web 2.0, the performance of JavaScript is now a concern of vendors of every major browser, including Mozilla Firefox, Google Chrome, and Microsoft Internet Explorer. The competition between major vendors, also known as the 'browser wars" [24], has inspired aggressive new JavaScript implementations based on Just-In-Time (JIT) compilation strategies [8].

Because browser market share is extremely important to companies competing in the web services marketplace, an objective comparison of the performance of different browsers is valuable to both consumers and service providers. JavaScript benchmarks, including SunSpider [23] and V8 [10], are widely used to evaluate JavaScript performance (for example, see [13]). These benchmark results are used to market and promote browers, and the benchmarks influence the design of JavaScript runtime implementations. Performance of JavaScript on the SunSpider and V8 benchmarks has improved dramatically in recent years.

This paper examines the following question: How representative are the SunSpider and V8 benchmarks suites when compared with the behavior of real JavaScript-based web applications? More importantly, we examine how benchmark behavior that differs quite significantly from real web applications might mislead JavaScript runtime developers.

By instrumenting the Internet Explorer 8 JavaScript runtime, we measure the JavaScript behavior of 11 important web applications and pages, including Gmail, Facebook, Amazon, and Yahoo. For each application, we conduct a typical user interaction scenario that uses the web application for a productive purpose such as reading email, ordering a book, or finding travel directions. We measure a variety of different program characteristics, ranging from the mix of operations executed to the frequency and types of events generated and handled.

Our results show that real web applications behave very differently from the benchmarks and that there are definite ways in which the benchmark behavior might mislead a designer. Because of the space limitations, this paper presents a relatively brief summary of our findings. The interested reader is referred to a companion technical report [17] for a more comprehensive set of results.

The contributions of this paper include:

- We are among the first to publish a detailed characterization of JavaScript execution behavior in real web applications, the SunSpider, and the V8 bench-

marks. In this paper we focus on functions and code as well as events and handlers. Our technical report [17] considers heap-allocated objects and data.

- We conclude that the benchmarks are not representative of real applications in many ways. Focusing on benchmark performance may result in overspecialization for benchmark behavior that does not occur in practice, and in missing optimization opportunities that are present in the real applications but not present in the benchmarks.
- We find that real web applications have code that is one to two orders of magnitude larger than most of the benchmarks and that managing code (both allocating and translating) is an important activity in a real JavaScript engine. Our case study in Section 4.7 demonstrates this point.
- We find that while the benchmarks are compute-intensive and batch-oriented, real web applications are event-driven, handling thousands of events. To be responsive, most event handlers execute only tens to hundreds of bytecodes. As a result, functions are typically short-lived, and long-running loops are uncommon.
- While existing JavaScript benchmarks make minimal use of event handlers, we find that they are extensively used in real web applications. The importance of responsiveness in web application design is not captured adequately by any of the benchmarks available today.

2 Background

JavaScript is a garbage-collected, memory-safe programming language with a number of interesting properties [6]. Unlike class-based object-oriented languages like C# and Java, JavaScript is a prototype-based language, influenced heavily in its design by Self [22]. JavaScript became widely used because it is standardized, available in every browser implementation, and tightly coupled with the browser's Document Object Model [2].

Importance of JavaScript. JavaScript's popularity has grown with the success of the web. Scripts in web pages have become increasingly complex as AJAX (Asynchronous JavaScript and XML) programming has transformed static web pages into responsive applications [11]. Web sites such as Amazon, Gmail, and Facebook contain and execute significant amounts of JavaScript code, as we document in this paper. Web applications (or apps) are applications that are hosted entirely in a browser and delivered through the web. Web apps have the advantage that they require no additional installation, will run on any machine that has a browser, and provide access to information stored in the cloud. Sophisticated mobile phones, such as the iPhone, broaden the base of Internet users, further increasing the importance and reach of web apps.

In recent years, the complexity of web content has spurred browser developers to increase browser performance in a number of dimensions, including improving JavaScript performance. Many of the techniques for improving traditional object-oriented languages such as Java and C# can and have been applied to JavaScript [8, 9]. JIT compilation has also been effectively applied, increasing measured benchmark performance of JavaScript dramatically.

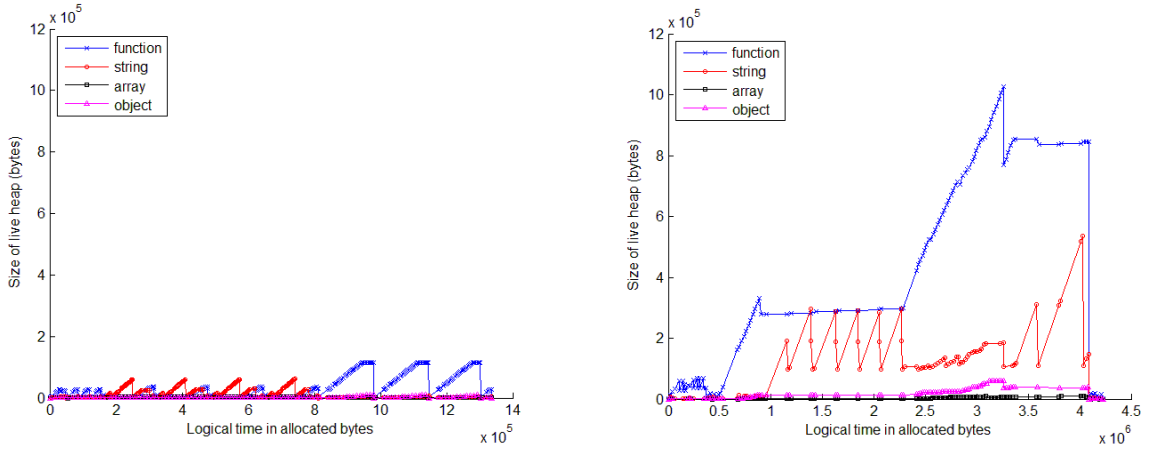
Value of benchmarks. Because browser performance can significantly affect a user's experience using a web application, there is commercial pressure for browser vendors to demonstrate that they have improved performance. As a result, JavaScript benchmark results are widely used in marketing and in evaluating new browser implementations. The two most widely used JavaScript benchmark suites are SunSpider, a collection of small benchmarks available from WebKit.org [23], and the V8 benchmarks, a collection of seven slightly larger benchmarks published by Google [10]. The benchmarks in both of these suites are relatively small programs; for example, the V8 benchmarks range from approximately 600 to 5,000 lines of code.

Illustrative example. Before we discuss how we collect JavaScript behavior data from real sites and benchmarks, we illustrate how this data is useful. Figure 1 shows live heap graphs for visits to the `google` and `bing` web sites[1]. These graphs show the number of live bytes of different types of data in the JavaScript heap as a function of time (measured by bytes of data allocated). In the figures, we show only the four most important data types: functions, strings, arrays, and objects. When the JavaScript heap is discarded, for example because the user navigates to a new page, the live bytes drops to zero, as we see in `google`.

These two search web sites shown offer very similar functionality, and we performed the same sequence of operations on them during our visit: we searched for "New York" in both cases and then proceeded to page through the results, first web page results and then the relevant news items.

We see from our measurements of the JavaScript heap, however, that the implementations of the two applications are very different, with `google` being implemented as a series of visits to different pages, and `bing` implemented as a single page visit. The benefit of the `bing` ap-

[1] Similar graphs for all the real web sites and benchmarks can be found in our tech report [17].



(a) Live heap for `google`.

(b) Live heap for `bing`.

Figure 1: Live heap contents as a function of time for two search applications.

proach is highlighted in this case by looking at the right hand side of each subfigure. In the case of `google`, we see that the contents of the JavaScript heap, including all the functions, are discarded and recreated repeatedly during our visit, whereas in the `bing` heap the functions are allocated only once. The size of the `google` heap is significantly smaller than the `bing` heap (approximately an order of magnitude), so it could be argued that the `google` approach is better. On the other hand, the `bing` approach does not lead to the JavaScript heap being repeatedly recreated.

In conclusion, we note that this kind of dynamic heap behavior is not captured by any of the V8 or SunSpider benchmarks, even though it is common among real web applications. Knowledge about such allocation behavior can be useful when, for example, designing and optimizing the garbage collection systems.

3 Experimental Design

In this section, we describe the benchmarks and applications we used and provide an overview of our measurements.

Figure 2 lists the 11 real web applications that we used for our study[2]. These sites were selected because of their popularity according to Alexa.com, and also because they represent a cross-section of diverse activities. Specifically, our applications represent search (`google`, `bing`), mapping (`googlemap`, `bingmap`), email (`hotmail`, `gmail`), e-commerce (`amazon`, `ebay`), news (`cnn`, `economist`), and social networking (`facebook`). Part of our goal was to understand both the differences between the real sites and the benchmarks as well as the differences among different classes of real web applications. For the remainder of this paper, we will refer to the different web sites using the names from Figure 2.

The workload for each site mimics the behavior of a user on a short, but complete and representative, visit to the site. This approach is dictated partly by expedience — it would be logistically complicated to measure long-term use of each web application — and partly because we believe that many applications are actually used in this way. For example, search and mapping applications are often used for targeted interactions.

3.1 Web Applications and Benchmarks

In measuring the JavaScript benchmarks, we chose to use the entire V8 benchmark suite, which comprises 7 programs, and selected programs from the SunSpider suite, which consists of 26 different programs. In order to reduce the amount of data collected and displayed, for SunSpider we chose the longest running benchmark in each of the 9 different benchmark categories — 3d: `raytrace`, access: `nbody`, bitops: `nseive` – `bits`, controlflow: `recursive`, crypto: `aes`, date: `xparb`, math: `cordic`, regexp: `dna`, and string: `tagcloud`.

3.2 Instrumenting Internet Explorer

Our approach to data collection is illustrated in Figure 3. The platform we chose for instrumentation is Internet Explorer (IE), version 8, running on a 32-bit Windows Vista operating system. While our results are in some ways specific to IE, the methods described here can be

[2]Throughout this discussion, we use the terms web application and web site interchangeably. When we refer to the site, we specifically mean the JavaScript executed when you visit the site.

Site	URL	Actions performed
amazon	amazon.com	Search for the book "Quantitative Computer Architecture," add to shopping cart, sign in, and sign out
bing	bing.com	Type in the search query "New York" and look at resulting images and news
bingmap	maps.bing.com	Search for directions from Austin to Houston, search for a location in Seattle, zoom-in, and use the bird's-eye view feature
cnn	cnn.com	Read the front-page news and three other news articles
ebay	ebay.com	Search for a notebook computer, sign in, bid, and sign out
economist	economist.com	Read the front-page news, read three other articles, view comments
facebook	facebook.com	Log in, visit a friend's page, browser through photos and comments
gmail	mail.google.com	Sign in, check inbox, delete a mail item, sign out
google	google.com	Type in the search query "New York" and look at resulting images and news
googlemap	maps.google.com	Search for directions from Austin to Houston, search for a location in Seattle, zoom-in, and use the street view feature
hotmail	hotmail.com	Sign in, check inbox, delete a mail item, sign out

Figure 2: Real web sites visited and actions taken.

applied to other browsers as well.

Our measurement approach works as follows: we have instrumented the C++ code that implements the IE 8 JavaScript runtime. For IE, the code that is responsible for executing JavaScript programs is not bundled in the main IE executable. Instead, it resides in a dynamic linked library, `jscript.dll`. After performing the instrumentation, we recompiled the engine source code to create a custom `jscript.dll`. (see Step 1 in Figure 3).

Next, we set up IE to use the instrumented `jscript.dll`. We then visit the web sites and run the benchmark programs described in the previous section with our special version of IE. A set of binary trace files is created in the process of visiting the web site or running a benchmark. These traces typically comprise megabytes of data, often up to 800 megabytes in the case of instruction traces. Finally, we use offline analyzers to process these custom trace files to obtain the results presented here.

3.3 Behavior Measurements

In studying the behavior of JavaScript programs, we focused on three broad areas: functions and code, objects and data (omitted here), and events and handlers. In each of these dimensions, we consider both static measurements (e.g., number of unique functions) and dynamic measurements (e.g., total number of function calls). We measure mostly the logical behavior of

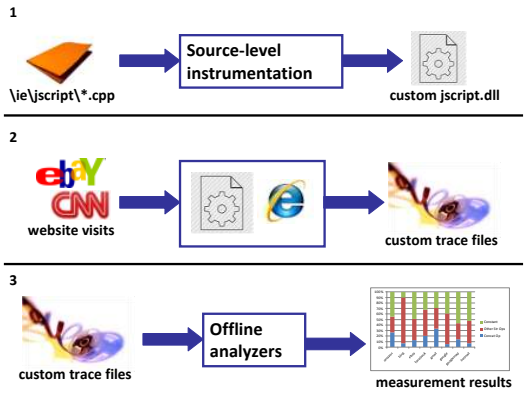


Figure 3: Instrumentation framework for measuring JavaScript execution using Internet Explorer.

JavaScript programs, avoiding characteristics that are browser-dependent. Thus, our measurements are largely machine-independent. However, we also look at specific characteristics of the IE's JavaScript engine (e.g., we count IE 8 bytecodes as a measure of execution) that pertain to interpreter-based engines. We leave measurements for characteristics relevant to JIT-based engines such as those found in Firefox and Chrome for future work.

3.3.1 Functions and Code

The JavaScript engine in IE 8 interprets JavaScript source after compiling it to an intermediate representation called bytecode. The interpreter has a loop that reads each bytecode instruction and implements its effect in a virtual machine. Because no actual machine instructions are generated in IE 8, we cannot measure the execution of JavaScript in terms of machine instructions. The bytecode instruction set implemented by the IE 8 interpreter is a well-optimized, traditional stack-oriented bytecode.

We count each bytecode execution as an "instruction" and use the term bytecode and instruction interchangeably throughout our evaluation. In our measurements, we look at the code behavior at two levels, the function and the bytecode level. Therefore, we instrument the engine at the points when it creates functions as well as in its main interpreter loop. Prior work measuring architecture characteristics of interpreters also measures behavior in terms of bytecode execution [19].

3.3.2 Events and Handlers

JavaScript has a single-threaded event-based programming model, with each event being processed by a non-preemptive handler. In other words, JavaScript code runs in response to specific user-initiated events such as a

Behavior	Real applications	Benchmarks	Implications
CODE AND FUNCTIONS			
Code size	100s of kilobytes to a few megabytes	100s of bytes to 10s of kilobytes	Efficient in-memory function and bytecode representation
Number of functions	1000s of functions	10s to 100s of functions	Minimize per-function fixed costs
Number of hot functions	10s to 100s of functions	10 functions or less	Size hot function cache appropriately
Instruction mix	Similar to each other	Different across benchmarks and from real applications	Optimize for real application instruction mix
Cold code	Majority of code	Minority of code	Download, parse, and JIT code lazily
Function duration	Mostly short	Mostly short, some very long running	Loop optimizations less effective
EVENTS AND EVENT HANDLERS			
Handler invocations	1000s of invocations	Less than 10 invocations	Optimize for frequent handler calls
Handler duration	10s to 100s of bytecodes	Very long	Make common short handler case fast
MEMORY ALLOCATION AND OBJECT LIFETIMES			
Allocation rate	Significant, sustained	Only significant in a few	GC performance not a factor in benchmark results
Data types	Functions and strings dominate	Varies, JS objects dominate in some	Optimize allocation of functions, strings
Object lifetimes	Depends on type, some long-lived	Very long or very short	Approaches like generational collection hard to evaluate with benchmarks
Heap reuse	Web 1.0 has significant reuse between page loads	No heap reuse	Optimize code, heap for reuse case—cache functions, DOM, possibly heap contents

Figure 4: A summary of lessons learned from JSMeter.

mouse click, becomes idle, and waits for another event to process. Therefore, to completely understand behaviors of JavaScript that are relevant to its predominant usage, we must consider the event-driven programming model of JavaScript. Generally speaking, the faster handlers complete, the more responsive an application appears.

However, event handling is an aspect of program behavior that is largely unexplored in related work measuring C++ and Java execution (e.g., see [5] for a thorough analysis of Java execution). Most related work considers the behavior of benchmarks, such as SPECjvm98 [4] and SPECcpu2000 [1], that have no interactive component. For JavaScript, however, such batch processing is mostly irrelevant.

For our measurements, we insert instrumentation hooks before and after event handling routines to measure characteristics such as the number of events handled and the dynamic size of each event handler invocation as measured by the number of executed bytecode instructions.

4 Evaluation

We begin this section with an overview of our results. We then consider the behavior of the JavaScript functions and code, including the size of functions, opcodes executed, etc. Next, we investigate the use of events and event handlers in the applications. We conclude the section with a case study showing that introducing cold code, i.e., code that is never executed, into existing benchmarks has a substantial effect on performance results.

4.1 Overview

Before drilling down into our results, we summarize the main conclusions of our comparison in Figure 4. The first column of the table indicates the specific behavior we measured, and the next two columns compare and contrast results for the real web applications and benchmarks. The last column summarizes the implications of the observed differences, specifically providing insights for future JavaScript engine designers. Due to space constraints, a detailed comparison of all aspects of behavior is beyond the scope of this paper, and we refer the reader to our tech report for those details [17].

4.2 Functions and Code Behavior

We begin our discussion by looking at a summary of the functions and behavior of the real applications and benchmarks. Figure 5 summarizes our static and dynamic measurements of JavaScript functions.

The real web sites. In Figure 5a, we see that the real web applications comprise many functions, ranging from a low of around 1,000 in `google` to a high of 10,000 in `gmail`. The total amount of JavaScript

	Static				Dynamic				
	Unique Func.	Source (bytes)	Compiled (bytes)	Global Context	Unique Func.	Total Calls	Total Opcodes	Opcodes / Call	% Unique Exec. Func.
amazon	1,833	692,173	312,056	210	808	158,953	9,941,596	62.54	44.08%
bing	2,605	1,115,623	657,118	50	876	23,759	1,226,116	51.61	33.63%
bingmap	4,258	1,776,336	1,053,174	93	1,826	274,446	12,560,049	45.77	42.88%
cnn	1,246	551,257	252,214	124	526	99,731	5,030,647	50.44	42.22%
ebay	2,799	1,103,079	595,424	210	1,337	189,805	7,530,843	39.68	47.77%
economist	2,025	899,345	423,087	184	1,040	116,562	21,488,257	184.35	51.36%
facebook	3,553	1,884,554	645,559	130	1,296	210,315	20,855,870	99.16	36.48%
gmail	10,193	2,396,062	2,018,450	129	3,660	420,839	9,763,506	23.20	35.91%
google	987	235,996	178,186	42	341	10,166	427,848	42.09	34.55%
googlemap	5,747	2,024,655	1,218,119	144	2,749	1,121,777	29,336,582	26.15	47.83%
hotmail	3,747	1,233,520	725,690	146	1,174	15,474	585,605	37.84	31.33%

(a) Real web application summary.

	Static				Dynamic				
	Unique Func.	Source (bytes)	Compiled (bytes)	Global Context	Unique Func.	Total Calls	Total Opcodes	Opcodes / Call	% Unique Exec. Func.
richards	67	22,738	7,617	3	59	81,009	2,403,338	29.67	88.06%
deltablue	101	33,309	11,263	3	95	113,276	1,463,921	12.92	94.06%
crypto	163	55,339	31,304	3	91	103,451	90,395,272	873.80	55.83%
raytrace	90	37,278	15,014	3	72	214,983	5,745,822	26.73	80.00%
earley	416	203,933	65,693	3	112	813,683	25,285,901	31.08	26.92%
regexp	44	112,229	35,370	3	41	96	935,322	9742.94	93.18%
splay	47	17,167	5,874	3	45	678,417	25,597,696	37.73	95.74%

(b) V8 benchmark summary.

	Static				Dynamic				
	Unique Func.	Source (bytes)	Compiled (bytes)	Global Context	Unique Func.	Total Calls	Total Opcodes	Opcodes / Call	% Unique Exec. Func.
3d-raytrace	31	14,614	7,419	2	30	56,631	5,954,264	105.14	96.77%
access-nbody	14	4,437	2,363	2	14	4,563	8,177,321	1,792.09	100.00%
bitops-nsieve	6	939	564	2	5	5	13,737,420	2,747,484.00	83.33%
controlflow	6	790	564	2	6	245,492	3,423,090	13.94	100.00%
crypto-aes	22	17,332	6,215	2	17	10,071	5,961,096	591.91	77.27%
date-xparb	24	12,914	5,341	4	12	36,040	1,266,736	35.15	50.00%
math-cordic	8	2,942	862	2	6	75,016	12,650,198	168.63	75.00%
regexp-dna	3	108,181	630	2	3	3	594	198.00	100.00%
string-tagcloud	16	321,894	55,219	3	10	63,874	2,133,324	33.40	62.50%

(c) SunSpider benchmark summary.

Figure 5: Summary measurements of web applications and benchmarks.

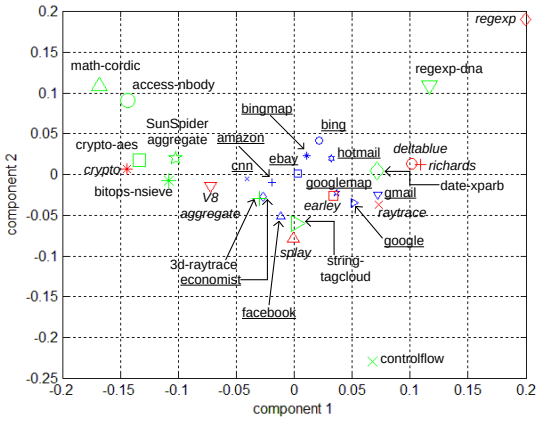


Figure 6: Opcode frequency distribution comparison.

source code associated with these web sites is significant, ranging from 200 kilobytes to more than two megabytes of source. Most of the JavaScript source code in these applications has been "minified", that is, had the whitespace removed and local variable names minimized using available tools such as JScrunch [7] or JSmin [3]. This source code is translated to the smaller bytecode representation, which from the figure we see is roughly 60% the size of the source.

In the last column, which captures the percentage of static unique functions executed, we see that as many as 50–70% are *not* executed during our use of the applications, suggesting that much of the code delivered applies to specific functionality that we did not exercise when we visited the sites. Code-splitting approaches such as Doloto [15] exploit this fact to reduce the wasted effort of downloading and compiling cold code.

The number of bytecodes executed during our visits ranged from around 400,000 to over 20 million. The most compute-intensive applications were `facebook`, `gmail`, and `economist`. As we show below, the large number of executed bytecodes in `economist` is an anomaly caused by a hot function with a tight loop. This anomaly is also clearly visible from the opcodes/call column. We see that `economist` averages over 180 bytecodes per call, while most of the other sites average between 25 and 65 bytecodes per call. This low number suggests that a majority of JavaScript function executions in these programs *do not* execute long-running loops. Our discussion of event handler behavior in Section 4.6 expands on this observation.

Because it is an outlier, the `economist` application deserves further comment. We looked at the hottest function in the application and found a single function which accounts for over 50% of the total bytecodes executed in our visit to the web site. This function loops over the elements of the DOM looking for elements with a specific node type and placing those elements into an array. Given that the DOM can be quite large, using an interpreted loop to gather specific kinds of elements can be quite expensive to compute. An alternative, more efficient implementation might use DOM APIs like `getElementById` to find the specific elements of interest directly.

On a final note, in column five of Figure 5 we show the number of instances of separate matching $<$ `script` $>$ tags that appeared in the web pages that implemented the applications. We see that in the real applications, there are many such instances, ranging to over 200 in `ebay`. This high number indicates that JavaScript code is coming from a number of sources in the applications, including different modules and/or feature teams from within the same site, and also coming from third party sites, for advertising, analytics, etc.

The benchmarks. In Figure 5, we also see the summary of the V8 and SunSpider benchmarks. We see immediately that the benchmarks are much smaller, in terms of both source code and compiled bytecode, than the real applications. Furthermore, the largest of the benchmarks, `string` − `tagcloud`, is large not because of the amount of code, but because it contains a large number of string constants. Of the benchmarks, `earley` has the most real code and is an outlier, with 400 functions compared to the average of the rest, which is well below 100 functions. These functions compile down to very compact bytecode, often more than 10 times smaller than the real applications. Looking at the fraction of these functions that are executed when the benchmarks are run, we see that in many cases the percentage is high, ranging from 55–100%. The benchmark `earley` is again an outlier, with only 27% of the code actually executed in the course of running the benchmark.

The opcodes per call measure also shows significant differences with the real applications. Some of the SunSpider benchmarks, in particular, have long-running loops, resulting in high average bytecodes executed per call. Other benchmarks, such as `controlflow`, have artificially low counts of opcodes per call. Finally, none of the benchmarks has a significant number of distinct contexts in which JavaScript code is introduced (global scope), emphasizing the homogeneous nature of the code in each benchmark.

4.3 Opcode Distribution

We examined the distribution of opcodes that each of the real applications and benchmarks executed. To do this, we counted how many times each of the 160 different opcodes was executed in each program and normalized

these values to fractions. We then compared the 160-dimensional vector generated by each real application and benchmark.

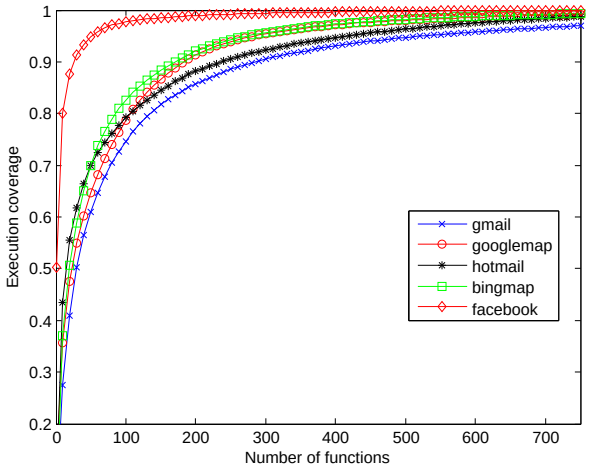
Our goal was to characterize the kinds of operations that these programs perform and determine how representative the benchmarks are of the opcode mix performed by the real applications. We were also interested in understanding how much variation exists between the individual real applications themselves, given their diverse functionality.

To compare the resulting vectors, we used Principal Component Analysis (PCA) [12] to reduce the 160-dimensional space to two principal dimensions. This dimension reduction is a way to avoid the curse of dimensionality problem. We found that components after the third are insignificant and chose to present only the two principal components for readability. Figure 6 shows the result of this analysis. In the figure, we see the three different program collections (real, V8, and SunSpider). The figure shows that the real sites cluster in the center of the graph, showing relatively small variation among themselves.

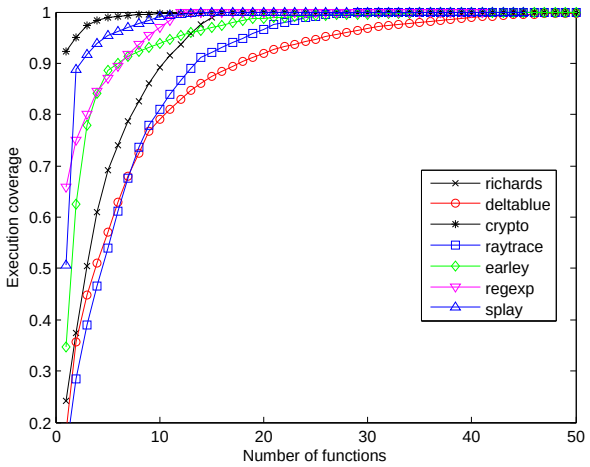
For example, `ebay` and `bingmap`, very different in their functionality, cluster quite closely. In contrast, both sets of benchmarks are more widely distributed, with several obvious outliers. For SunSpider, `controlflow` is clearly different from the other applications, while in V8, `regexp` sits by itself. Surprisingly, few of the benchmarks overlap the cluster of real applications, with `earley` being the closest in overall opcode mix to the real applications. While we expect some variation in the behavior of a collection of smaller programs, what is most surprising is that almost all the benchmarks have behaviors that are significantly different than the real applications. Furthermore, it is also surprising that the real web applications cluster as tightly as they do. This result suggests that while the external functionality provided may appear quite different from site to site, much of the work being done in JavaScript on these sites is quite similar.

4.4 Hot Function Distribution

We next consider the distribution of hot functions in the applications, which tells us what code needs to be highly optimized. Figure 7 shows the distribution of hot functions in a subset of the real applications and the V8 benchmarks (full results, including the SunSpider benchmarks are included in [17]). Each figure shows the cumulative contribution of each function, sorted by hottest functions first on the x-axis, to normalized total opcodes executed on the y-axis. We truncate the x-axis (not considering all functions) to get a better view of the left end of the curve. The figures show that all programs, both



(a) Real web application hot function distribution.



(b) V8 benchmarks hot function distribution.

Figure 7: Hot function distribution.

real applications and benchmarks, exhibit high code locality, with a small number of functions accounting for a large majority of total execution. In the real applications, 80% of total execution is covered by 50 to 150 functions, while in the benchmarks, at most 10 functions are required. `facebook` is an outlier among the real applications, with a small number of functions accounting for almost all the execution time.

4.5 Implications of Code Measurements

We have considered static and dynamic measures of JavaScript program execution, and discovered numerous important differences between the behaviors of the real applications and the benchmarks. Here we discuss how these differences might lead designers astray when building JavaScript engines that optimize benchmark performance.

First, we note a significant difference in the code size of the benchmarks and real applications. Real web applications have large code bases, containing thousands of functions from hundreds of individual < `script` > bodies. Much of this code is never or rarely executed, meaning that efforts to compile, optimize, or tune this code are unnecessary and can be expensive relative to what the benchmarks would indicate. We also observe that a substantial fraction of the downloaded code is not executed in a typical interaction with a real application. Attempts to avoid downloading this code, or minimizing the resources that it consumes once it is downloaded, will show much greater benefits in the real applications than in the benchmarks.

Second, we observe that based on the distribution of opcodes executed, benchmark programs represent a much broader and skewed spectrum of behavior than the real applications, which are quite closely clustered. Tuning a JavaScript engine to run `controlflow` or `regexp` may improve benchmark results, but tuning the engine to run any one of the real applications is also likely to significantly help the other real applications as well. Surprisingly, few of the benchmarks approximate the instruction stream mix of the real applications, suggesting that there are activities being performed in the real applications that are not well emulated by the benchmark code.

Third, we observe that each individual function execution in the real applications is relatively short. Because these applications are not compute-intensive, benchmarks with high loop counts, such as `bitops` − `nsieve`, distort the benefit that loop optimizations will provide in real applications. Because the benchmarks are batch-oriented to facilitate data collection, they fail to match a fundamental characteristic of all real web applications — the need for responsiveness. The very nature of an interactive application prevents developers from writing code that executes for long periods of time without interruption.

Finally, we observe that a tiny fraction of the code accounts for a large fraction of total execution in both the benchmarks and the real applications. The size of the hot code differs by one to two orders of magnitude between the benchmarks and applications, but even in the real applications the hot code is still quite compact.

4.6 Event Behavior

In this section, we consider the event-handling behavior of the JavaScript programs. We observe that handling events is commonplace in the real applications and almost never occurs in the benchmarks. Thus the focus of this section is on characterizing the handler behavior of the real applications.

	# of events	unique events	executed instructions	
			handler	total
richards	8	6	2,403,333	2,403,338
deltablue	8	6	1,463,916	1,463,921
crypto	11	6	86,854,336	86,854,341
raytrace	8	6	5,745,817	5,745,822
earley	11	6	25,285,896	25,285,901
regexp	8	6	935,317	935,322
splay	8	6	25,597,691	25,597,696

Figure 9: Event handler characteristics in the V8 benchmarks.

Before discussing the results, it is important to explain how handlers affect JavaScript execution. In some cases, handlers are attached to events that occur when a user interacts with a web page. Handlers can be attached to any element of the DOM, and interactions such as clicking on an element, moving the mouse over an element, etc., can cause handlers to be invoked. Handlers also are executed when a timer times out, when a page loads, or when an asynchronous `XMLHttpRequest` is completed. JavaScript code is also executed outside of a handler context, such as when a < `script` > block is processed as part of parsing the web page. Often code that initializes the JavaScript for the page executes outside of a handler.

Because JavaScript has a non-preemptive execution model, once a JavaScript handler is started, the rest of the browser thread for that particular web page is stalled until it completes. A handler that takes a significant amount of time to execute will make the web application appear sluggish and non-responsive.

Figures 8 and 9 present measures of the event handling behavior in the real applications and the V8 benchmarks[3]. In both tables, unique events are defined as follows. Events are nominally unique when they invoke the same sequences of handler instructions with the same inputs. Our measurements in the figures only approximate this definition. We associate each event with three attributes: name, the set of handler functions invoked, and the total number of instructions executed. If the two events have the same three attributes, we say that they are unique.

We see that the real applications typically handle thousands of events while the benchmarks all handle 11 or fewer. In all the benchmarks, one `onload` event (for loading and, subsequently, running the benchmark program) is responsible for almost 100% of all JavaScript execution. We will see shortly that this is in stark contrast to the behavior seen in the real applications. Even though real web sites typically process thousands of events, the unique events column in the figure indicates that there are only around one hundred unique events per application. This means that a given event is likely to be repeated and

[3] SunSpider results are similar to V8 results, so we omit them here.

	# of events	unique events	executed instructions handler	executed instructions total	% of handler instructions	handler size average	handler size median	handler size maximum
amazon	6,424	224	7,237,073	9,941,596	72.80%	1,127	8	1,041,744
bing	4,370	103	598,350	1,226,116	48.80%	137	24	68,780
bingmap	4,669	138	8,274,169	12,560,049	65.88%	1,772	314	281,887
cnn	1,614	133	4,939,776	5,030,647	98.19%	3,061	11	4,208,115
ebay	2,729	136	7,463,521	7,530,843	99.11%	2,735	80	879,798
economist	2,338	179	21,146,767	21,488,257	98.41%	9,045	30	270,616
facebook	5,440	143	17,527,035	20,855,870	84.04%	3,222	380	89,785
gmail	1,520	98	3,085,482	9,763,506	31.60%	2,030	506	594,437
google	569	64	143,039	427,848	33.43%	251	43	10,025
googlemap	3,658	74	26,848,187	29,336,582	91.52%	7,340	2,137	1,074,568
hotmail	552	194	474,693	585,605	81.06%	860	26	202,105

Figure 8: Event handler characteristics in real applications.

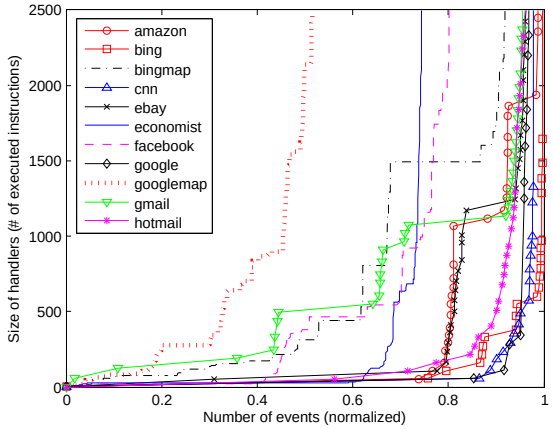


Figure 10: Distribution of handler durations.

handled many times throughout the course of a user visit to the site.

We see the diversity of the collection of handlers in the results comparing the mean, median, and maximum of handler durations for the real applications. Some handlers run for a long time, such as in `cnn`, where a single handler accounts for a significant fraction of the total JavaScript activity. Many handlers execute for a very short time, however. The median handler duration in `amazon`, for example, is only 8 bytecodes. `amazon` is also unusual in that it has the highest number of events. We hypothesize that such short-duration handlers probably are invoked, test a single value, and then return.

These results demonstrate that handlers are written so that they almost always complete in a short time. For example, in `bing` and `google`, both highly optimized for delivering search results quickly, we see low average and median handler times. It is also clear that `google`, `bing`, and `facebook` have taken care to reduce the duration of the longest handler, with the maximum of all three below 100,000 bytecodes.

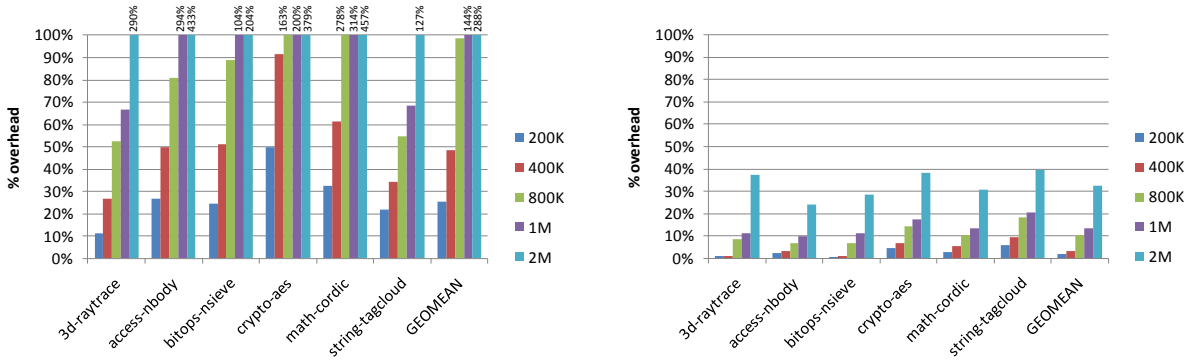
Figure 10 illustrates the distribution of handler durations for each of the applications. The x-axis depicts the instances of handler invocations, sorted by smallest first and normalized to one. The y-axis depicts the number of bytecodes executed by each handler invocation. For example, in the figure, approximate 40% of the handlers in `googlemap` executed for 1000 bytecodes or less.

Figure 10 confirms that most handler invocations are short. This figure provides additional context to understand the distribution. For example, we can determine the 95th percentile handler duration by drawing a vertical line at 0.95 and seeing where each line crosses it. The figure also illustrates that the durations in many of the applications reach plateaus, indicating that there are many instances of handlers that execute for the same number of instructions. For example, we see a significant number of `bingmap` instances that take 1,500 bytcodes to complete.

4.7 Cold Code Case Study

Our results show that real web applications have much more JavaScript code than the SunSpider and V8 benchmarks and that most of that code is cold. We were curious how much impact the presence of such cold code would have on benchmark performance results. Based on our understanding of the complexity and performance overhead of code translation, especially in a JIT-compiler, we hypothesized that simply increasing the amount of cold code in existing benchmarks would have a significant non-uniform impact on benchmark results. If this hypothesis is true, then a simple way to make results from current benchmarks more representative of actual web applications would be to add cold code to each of them.

To test this hypothesis, we selected six SunSpider benchmarks that are small and have mostly hot code. To each of these benchmarks, we added 200 kilobytes, 400 kilobytes, 800 kilobytes, 1 megabyte and 2 megabytes



(a) Impact of cold code in Chrome.

(b) Impact of cold code Internet Explorer 8.

Figure 11: Impact of cold code using a subset of the SunSpider benchmarks.

of cold code from the jQuery library. The added code is never called in the benchmark but the JavaScript runtime still processes it. We executed each benchmark with the added code and recorded its performance on both the Google Chrome and Internet Explorer browsers[4].

Figure 11 presents the results of the experiment. It shows the execution overhead observed in each browser as a function of the size of the additional cold code added in each benchmark. At a high level, we see immediately that the addition of cold code affects the benchmark performance on the two browsers differently. In the case of Chrome (Figure 11a), adding two megabytes of cold code can add up to 450% overhead to the benchmark performance. In Internet Explorer (Figure 11b), cold code has much less impact.

In IE, the addition of 200 to 400 kilobytes does not impact its performance significantly. On average, we observe the overhead due to cold code of 1.8% and 3.2%, respectively. With 1 megabyte of cold code, the overhead is around 13%, still relatively small given the large amount of code being processed. In Chrome, on the other hand, even at 200 kilobytes, we observe quite a significant overhead, 25% on average across the six benchmarks. Even between the benchmarks on the same browser, the addition of cold code has widely varying effects (consider the effect of 1 megabyte of cold code on the different benchmarks in Chrome).

There are several reasons for these observed differences. First, because Chrome executes the benchmarks faster than IE, the additional fixed time processing the cold code will have a greater effect on Chrome's overall runtime. Second, Chrome and IE process JavaScript source differently, and large amounts of additional source, even if it is cold code, will have different effects on runtime. The important takeaway here is not that one browser processes cold code any better than another, but that results of benchmarks containing 1 megabyte of cold code will look different than results without the cold code. Furthermore, results with cold code are likely to be more representative of browser performance on real web sites.

[4]We use Chrome version 3.0.195.38 and Internet Explorer version 8.0.6001.18865. We collected measurements on a machine with a 1.2 GHz Intel Core Duo processor with 1.5 gigabytes of RAM, running 32-bit Windows Vista operating system.

5 Related Work

There are surprisingly few papers measuring specific aspects of JavaScript behavior, despite how widely used it is in practice. A concurrently submitted paper by Richards et al. measures static and dynamic aspects of JavaScript programs, much as we do [18]. Like us, their goals are to understand the behavior of JavaScript applications in practice, and specifically they investigate the degree of dynamism present in these applications (such as uses of eval). They also consider the behavior of JavaScript benchmarks, although this is not a major focus of the research. Unlike us, they do not consider the use of events in applications, or consider the size and effect of cold code.

One closely related paper focuses on the behavior of interpreted languages. Romer et al. [19] consider the runtime behavior of several interpreted languages, including Tcl, Perl, and Java, and show that architectural characteristics, such as cache locality, are a function of the interpreter itself and not the program that it is interpreting. While the goals are similar, our methods, and the language we consider (JavaScript), are very different.

Dieckmann and Hölzle consider the memory allocation behavior of the SPECJVM Java benchmarks [4]. A number of papers have examined the memory reference characteristics of Java programs [4, 14, 16, 20, 21] specifically to understand how hardware tailored for Java ex-

ecution might improve performance. Our work differs from this previous work in that we measure JavaScript and not Java, we look at characteristics beyond memory allocation, and we consider differences between benchmarks and real applications.

Dufour et al. present a framework for categorizing the runtime behavior of programs using precise and concise metrics [5]. They classify behavior in terms of five general categories of measurement and report measurements of a number of Java applications and benchmarks, using their results to classify the programs into more precise categories. Our measurements correspond to some metrics mentioned by Dufour et al., but we consider some dimensions of execution that they do not, such as event handler metrics, and compare benchmark behavior with real application behavior.

6 Conclusions

We have presented detailed measurements of the behavior of JavaScript applications, including commercially important web applications such as Gmail and Facebook, as well as the SunSpider and V8 benchmark suites. We measure two specific areas of JavaScript runtime behavior: 1) functions and code and 2) events and handlers. We find that the benchmarks are not representative of many real web sites and that conclusions reached from measuring the benchmarks may be misleading.

Our results show that JavaScript web applications are large, complex, and highly interactive programs. While the functionality they implement varies significantly, we observe that the real applications have much in common with each other as well. In contrast, the JavaScript benchmarks are small, and behave in ways that are significantly different than the real applications. We have documented numerous differences in behavior, and we conclude from these measured differences that results based on the benchmarks may mislead JavaScript engine implementers.

Furthermore, we observe interesting behaviors in real JavaScript applications that the benchmarks fail to exhibit. Our measurements suggest a number of valuable follow-up efforts. These include working on building a more representative collection of benchmarks, modifying JavaScript engines to more effectively implement some of the real behaviors we observed, and building developer tools that expose the kind of measurement data we report.

Acknowledgments

We thank Corneliu Barsan, Trishul Chilimbi, David Detlefs, Leo Meyerovich, Karthik Pattabiraman, David Simmons, Herman Venter, and Allen Wirfs-Brock for their support and feedback during the course of this research. We thank the anonymous reviewers for their feedback, and specifically Wilson Hsieh, who made a number of concrete and helpful suggestions.

References

[1] B. Calder, D. Grunwald, and B. Zorn. Quantifying behavioral differences between C and C++ programs. *Journal of Programming Languages*, 2:313–351, 1995.

[2] W. W. W. Consortium. Document object model (DOM). `http://www.w3.org/DOM/`.

[3] D. Crockford. JSMin: The JavaScript minifier. `http://www.crockford.com/javascript/jsmin.html`.

[4] S. Dieckmann and U. Hölzle. A study of the allocation behaviour of the SPECjvm98 Java benchmarks. In *Proceedings of European Conference on Object Oriented Programming*, pages 92–115, July 1999.

[5] B. Dufour, K. Driesen, L. Hendren, and C. Verbrugge. Dynamic metrics for Java. *SIGPLAN Not.*, 38(11):149–168, 2003.

[6] ECMA International. ECMAScript language specification. Standard ECMA-262, Dec. 1999.

[7] C. Foster. JSCrunch: JavaScript cruncher. `http://www.cfoster.net/jscrunch/`.

[8] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 465–478, 2009.

[9] Google. V8 JavaScript engine. `http://code.google.com/apis/v8/design.html`.

[10] Google. V8 benchmark suite - version 5. `http://v8.googlecode.com/svn/data/benchmarks/v5/run.html`, 2009.

[11] A. T. Holdener, III. *Ajax: The Definitive Guide.* O'Reilly, 2008.

[12] I. T. Jolliffe. *Principal Component Analysis*. Series in Statistics. Springer Verlag, 2002.

[13] G. Keizer. Chrome buries Windows rivals in browser drag race. `http://www.computerworld.com/s/article/9138331/Chrome_buries_Windows_rivals_in_browser_drag_race`, 2009.

[14] J.-S. Kim and Y. Hsu. Memory system behavior of Java programs: methodology and analysis. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, pages 264–274, 2000.

[15] B. Livshits and E. Kiciman. Doloto: code splitting for network-bound Web 2.0 applications. In *Proceedings of the International Symposium on Foundations of Software Engineering*, pages 350–360, 2008.

[16] R. Radhakrishnan, N. Vijaykrishnan, L. K. John, A. Sivasubramaniam, J. Rubio, and J. Sabarinathan. Java runtime systems: Characterization and architectural implications. *IEEE Trans. Computers*, 50(2):131–146, 2001.

[17] P. Ratanaworabhan, B. Livshits, D. Simmons, and B. Zorn. JSMeter: Characterizing real-world behavior of JavaScript programs. Technical Report MSR-TR-2009-173, Microsoft Research, Dec. 2009.

[18] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of JavaScript programs. In *Proceedings of the ACM SIGPLAN 2010 Conference on Programming Language Design and Implementation (PLDI'10)*, pages 1–12, 2010.

[19] T. H. Romer, D. Lee, G. M. Voelker, A. Wolman, W. A. Wong, J.-L. Baer, B. N. Bershad, and H. M. Levy. The structure and performance of interpreters. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 150–159, Oct. 1996.

[20] Y. Shuf, M. J. Serrano, M. Gupta, and J. P. Singh. Characterizing the memory behavior of Java workloads: a structured view and opportunities for optimizations. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, pages 194–205, 2001.

[21] T. Systä. Understanding the behavior of Java programs. In *Proceedings of the Working Conference on Reverse Engineering*, pages 214–223, 2000.

[22] D. Unger and R. B. Smith. Self: The power of simplicity. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 227–242, Dec. 1987.

[23] WebKit. Sunspider JavaScript benchmark, 2008. `http://www2.webkit.org/perf/sunspider-0.9/sunspider.html`, 2008.

[24] Wikipedia. Browser wars. `http://en.wikipedia.org/wiki/Browser_wars`, 2009.