JSMeter: Characterizing Real-World Behavior of JavaScript Programs

Paruj Ratanaworabhan
Cornell University
paruj@csl.cornell.edu

Benjamin Livshits
Microsoft Research
livshits@microsoft.com

David Simmons
Microsoft
dsim@microsoft.com

Benjamin Zorn
Microsoft Research
zorn@microsoft.com

December 8, 2009

Microsoft Research Technical Report
MSR-TR-2009-173

Abstract

JavaScript is widely used in web-based applications and is increasing 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 websites and compare this behavior with the benchmarks.

We measure three specific areas of JavaScript runtime behavior: 1) functions and code; 2) heap-allocated objects and data; 3) events and handlers. We find that the benchmarks are not representative of many real websites and that conclusions reached from measuring the benchmarks may be misleading.

Specific examples of such misleading conclusions include the following: that web applications have many loops, that non-string objects in web applications are extremely short-lived, and that web applications handle few events.

We hope our results will convince the JavaScript community to develop and adopt benchmarks that are more representative of real web applications.

To reduce the total length of the paper, we have chosen to shrink our figures substantially, and we recommend that an interested reader view the document in color, preferably with the ability to enlarge the figures as needed. Our intent of including this much raw data is to allow interested readers an opportunity to draw their own conclusions.

1 Introduction

JavaScript is a widely used programming language that is enabling a new generation of computer applications. JavaScript is the scripting language used to program a large fraction of all web content and many important web sites, including Google, Facebook, and Yahoo, rely heavily on JavaScript to make the pages 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" [41], has inspired aggressive new JavaScript implementations based on Just-In-Time (JIT) compilation strategies [14].

Because browser market share is extremely important to all companies competing in the web services mareketplace, an objective comparison of the performance of different browsers is valuable to both consumers and service providers. As a result, JavaScript benchmarks, including SunSpider [40] and V8 [16], are widely used to evaluate JavaScript performance (for example, see [23]). These benchmark results are used to market and promote browers, and the benchmarks significantly influence the design of JavaScript runtime implementations. As a result, performance of JavaScript on the SunSpider and V8 benchmarks has improved dramatically in the past two years. Many technical people, including the benchmark developers themselves, acknowledge that benchmarks have limitations and do not necessarily represent real application behavior.

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 significantly from real web applications might mislead JavaScript runtime developers in the design of their runtimes. By instrumenting the Internet Explorer 8 JavaScript runtime, we measure the JavaScript behavior of 11 important web applications, 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 behaviors, ranging from the mix of operations executed, to the mix of data types allocated, 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.

1.1 Contributions

The contributions of this paper include:

- We are the first to publish a detailed characterization of JavaScript execution behavior in both real web applications and the SunSpider and V8 benchmarks. We measure three specific areas of JavaScript runtime behavior: 1) functions and code; 2) heap-allocated objects and data; 3) events and handlers.

- We conclude that the benchmarks are not representative of real applications in many ways, and that tailoring a runtime to execute the benchmarks efficiently may not result in optimal performance of real applications. Specifically, focusing on benchmark performance may result in overspecialization for benchmark behavior that does not occur in practice, and in missing important optimization opportunities that are present in the real applications but not present in the benchmarks.

- We find that while the benchmarks are compute-intensive and batch-oriented, real web applications are largely event-driven with many thousands of events being handled. Driven to provide high responsiveness, event handlers typically execute for only a few milliseconds. As a result, functions rarely execute many bytecode instructions, and long-running loops are not common.

- We also find that the benchmarks' use of heap-allocated data is quite different from that of real applications. Furthermore, JavaScript web applications use heap data in ways that are quite different from previous reported measurements of Java applications. Specifically, strings and functions represent a major part of all object allocation. We observe that objects and arrays have lifetimes that are often significantly longer than strings, suggesting that collection algorithms that assume most objects will die quickly may be inefficient for JavaScript.

- 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.

1.2 Paper Organization

The rest of the paper is organized as follows. Section 2 provides background on JavaScript execution in web browsers and our approach to measurement. Section 3 describes our experimental design, including how

and what we measured. Section 4 summarizes our main experimental results. Section 5 considers broader implications of our measurements. Finally, Section 6 describes related work and Section 7 concludes.

2 Background

JavaScript is a garbage-collected, memory-safe programming language with a number of interesting properties [12]. Contrary to what one might conclude from their names, Java and JavaScript have many differences. Unlike class-based object-oriented languages like C# and Java, JavaScript is a prototype-based language, influenced heavily in its design by Self [39]. JavaScript became widely used because it is standardized, available in every browser implementation, and tightly coupled with the browser's Document Object Model [5].

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 [20]. 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 [14, 15]. Just-In-Time compilation has also been effectively applied, increasing measured benchmark performance of JavaScript dramatically.

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 [40], and the V8 benchmarks, a collection of seven slightly larger benchmarks published by Google [16]. The benchmarks in both of these suites are relatively small programs; for example, the V8 benchmarks range from approximately 600 to 5000 lines of code. While even the benchmark developers themselves would admit that these benchmarks do not represent real web application behavior, the benchmarks are still used as a basis for tuning and comparing JavaScript implementations, and as a result have an important influence on the effectiveness of those implementations in practice.

This paper is the first paper to characterize both the behavior of the JavaScript benchmarks and the behavior of JavaScript in real web applications. We demonstrate that the behavior of these benchmark suites is unrepresentative of the real applications in fundamental and important ways. Further, we suggest how the benchmarks may mislead designers both by encouraging optimizations that are not important in practice and in missing opportunities that are present in the real web applications but not in the benchmarks.

Weak benchmarks have had a negative impact on language implementations in the past. For example, the SPECjvm98 benchmarks were widely used for many years to evaluate Java implementations [9]. General agreement within the Java community about the weakness of the SPECjvm98 benchmarks led to the creation of the DaCapo benchmark suite, which includes realistic Java programs such as the Eclipse development environment [3]. One of our goals is to provide motivation and insight regarding a similar more realistic set of benchmarks for JavaScript.

3 Experimental Design

In this section, we describe the benchmarks and applications we used in this study and discuss what we measured.

3.1 Web Applications and Benchmarks

Figure 1 lists the 11 real web applications that we used for our study[1]. These sites were selected because they are both important sites that many users visit every day, and 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 1.

[1] 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
cnn	cnn.com	Read the front-page news
ebay	ebay.com	Search for a notebook, sign in, bid, and sign out
economist	economist.com	Read the front-page news, 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
hotmail	hotmail.com	Sign in, check inbox, delete a mail item, sign out

Figure 1: Real web sites visited and actions taken.

The actions taken at each site represent the behavior of a user on a short, but complete, 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. We leave studies of longer user sessions with applications such as gmail and facebook for future work.

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 classes—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

This section describes the instrumentation framework that we used to measure static and dynamic characteristics of JavaScript programs. The framework is summarized and illustrated in Figure 2. Our instrumentation platform 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 applied to other platforms with different browsers as well.

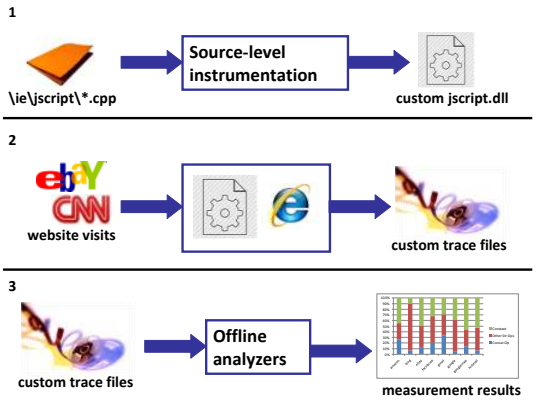


Figure 2: Instrumentation framework for measuring javascript execution using Internet Explorer.

Our measurement system works as follows:

1. We start by instrumentating the C++ code that implements the IE8 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 2).

2. Next, we create a local IE executable copy and bind this executable to the jscript.dll that includes our instrumentation. 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.

3. Finally, we use offline analyzers to process these custom trace files to obtain the results presented here.

The trace files we create range in size to a maximum of 800 megabytes for the instruction traces and 140 megabytes for the object data traces. In Appendix B, we present the trace file sizes for all the applications and benchmarks.

3.3 Behavior Measurements

In studying the behavior of JavaScript programs, we consider three broad dimesions: functions and code, objects and data, 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). Our goal is to measure the logical behavior of JavaScript programs and to avoid as much as possible specific characteristics of the

IE8 implementation. Thus, whenever possible, we consider aspects of the JavaScript source such as functions, objects, and event handlers. As a result, our measurements are mostly machine-independent; however, they are not platform-independent. Because we measure aspects of IE's JavaScript engine, it is unavoidable that some particular characteristics are implementation-specific to that engine (e.g., we count IE8 bytecodes as a measure of execution). Nevertheless, whenever it is possible to untie such characteristics from the engine, we make assumptions that we believe can be generalized to other JavaScript engines as well. The following subsections discuss the specific measurements we make in each of these areas.

3.3.1 Functions and Code

The JavaScript engine in IE8 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 IE8, we cannot measure the execution of JavaScript in terms of machine instructions. The bytecode instruction set implemented by the IE8 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 script functions as well as in its main interpreter loop. Prior work measuring architecture characteristics of interpreters also measures behavior in terms of bytecode execution [34].

3.3.2 Objects and Data

We are interested in the behavior of JavaScript objects that reside in heap memory. We focus our measurements on the four dominant data types: string, object, array, and function.

Strings in JavaScript are immutable; their sizes can never be changed after their creation. Strings are unique in that they can be of both primitive and object type. We are primarily interested in primitive strings, as string objects are essentially wrappers around their primitive values. To obtain properties of strings we are interested in, we hook instrumentation code to points in the JavaScript engine where strings are created. Strings are created either when the source is translated or during the script runtime. The former accounts for all string constants in the source and the latter includes run-time operations that create strings dynamically (e.g., concatenation, slice, `toString`, etc.).

In our measurements, to make our results more independent of the IE8 implementation, we report string size as the number of bytes that the string would occupy in a naive implementation in which the result of every concatenation produced a new independent string. Good implementations can and do improve on this approach, but we chose to provide a measure of a simpler implementation, and as a result we may overestimate the size of strings somewhat. One can infer the opportunities for optimizing the representatation of strings from our measures of string operations presented in Section 4.2.4.

Arrays and objects hold references to other primitive and object types. Therefore, to abstract our measurements away from the IE8 implementation, we assume that the size of each array or object element equals the natural width of a machine pointer where the measurement is performed. In our case, the natural pointer size is four bytes and so we assume objects and arrays contain four bytes per element.

Unlike strings, arrays and objects are mutable and can grow and shrink dynamically. To accurately account for this dynamic behavior, we provide instrumentation hooks at points where arrays and objects are created, and also at points where there are operations that affect their sizes. Note that deleting an element in an array or object does not alter the size of the array or object; the delete operation simply marks that element as undefined. Inserting an element, however, does increase its size. To avoid growing arrays and objects too frequently, some implementations allocate a fixed minimum size to unintialized arrays and objects. We use the IE8 implementation defaults in these cases, which are too numerous to specify in detail here.

One of the important aspects of object behavior is object lifetime. For our measurements, we determine object lifetime in a conservative way, using the time that objects are garbage collected as the time that they are considered dead. While this technique overestimates the lifetime of objects [19] because it inherently ties estimated lifetimes to the frequency of collection, we find that garbage collections occur frequently enough in our implementation that the lifetime estimates we obtain are accurate enough to provide insights.

For functions, we report the memory allocated to functions which includes the size of the source code plus the compiled bytecode. We acknowledge that the size of the bytecode is implementation dependent.

3.3.3 Events and Handlers

Event handling is an aspect of program behavior that is largely unexplored in related work measuring C++ and Java execution (e.g., see [11] for a thorough analysis of Java execution). Most related work considers the behav-

ior of benchmarks, such as SPECjvm98 [10] and SPECcpu2000 [4], that have no interactive component. For JavaScript, however, its current prevailing use is in client-side web applications where such batch processing is mostly irrelevant. In this scenario, the processing is rather event-driven; JavaScript code runs in response to specific user-initiated events such as a mouse click, becomes idle, and waits for another event to process. Therefore, to completely understand behaviors of JavaScript that are relevant to its current usage, we must look into the event-driven programming model of JavaScript as well.

For our measurements, we provide 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 in number of executed bytecode instructions.

4 Evaluation

In this section, we first consider the behavior of the JavaScript functions and code, including the size of functions, opcodes executed, etc. Next we consider the use of heap-allocated objects, including the time-varying composition of the heap and object lifetimes. Finally, we investigate the use of events and event handlers in the applications.

4.1 Functions and Code Behavior

Properties of functions in JavaScript can have a significant effect on implementation decisions, including JIT compilation strategies, code representations, and optimization techniques.

4.1.1 Static and Dynamic Function Size

We begin our discussion by looking at a summary of the functions and behavior of the real applications and benchmarks. Figure 3 summarizes our static and dynamic measurements of JavaScript functions. In the figure, we show the the following information:

- **Static Unique Func.** – The number of unique functions loaded and compiled to bytecode by the application.
- **Static Source** – The size of the uncompressed JavaScript source code.
- **Static Compiled** – The size of the compiled bytecode.
- **Static Global Context** – The number of unique script tags in which JavaScript code appeared in the application (i.e., unique global contexts).
- **Dynamic Unique Func.** – The number of unique functions executed.
- **Dynamic Total Calls** – The total number of JavaScript function calls executed.
- **Dynamic Total Opcodes** – The number of bytecodes executed.
- **Dynamic Opcodes/Call** – The average number of bytecodes executed per function call.
- **Dynamic % Unique Exec. Func.** – The ratio of unique functions executed to unique functions compiled to bytecode.

The real web sites. This high-level characterization of execution behavior already identifies a number of important insights. First, 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 source code associated with these websites 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 [13] or JSmin [8]. This source code is translated to the smaller bytecode representation, which from the figure we see is roughly 60% the size of the source.

Of this large number of functions, in the last column, we see that as many as 35–50% 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 [28] 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 also is 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. These numbers are comparable to the average hardware instructions per call in C++ programs [4]. This low number suggests that most JavaScript functions in these programs do not contain loops.

In considering the behavior of `economist` specifically, we looked at the hottest function in the application and found code which accounts for over 50% of the total bytecodes executed in our visit to the website (see Figure 4). From what we can infer, 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

	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 3: Summary measurements of web applications and benchmarks.

```
function(a,b) {
    var i=0,elem,pos=a.length;
    if (D.browser.msie) {
        while (elem=b[i++])
            if (elem.nodeType!=8) a[pos++]=elem;
    } else {
        while(elem=b[i++]) a[pos++]=elem;
    }
    return a
}
```

Figure 4: Hot function in the `economist` web site.

to compute. An alternative, more efficient implementation might use DOM APIs like `GetElementById` to find the specific elements of interest directly.

In contrast to `economist`, `google` is of particular interest in this study because, being the most visited and lucrative site on the entire web, one can assume that the JavaScript content on that site has received the utmost scrutiny. `google` is also known for its clean home page and responsiveness, suggesting that no effort has been spared in paring out any unnecessary code execution. Despite this fact, we observe that 200 kilobytes of code is delivered during our `google` visit and 400,000 opcodes are executed.

On a final note, in column five of Figure 3 we show the number of instances of separate script elements 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`. We speculate that in some cases these different instances of script represent different sources of code content, perhaps representing independent authorship in the form of reusable modules. In any case, this diversity indicates that there are many sources of script code in these applications and that the code is intertwined with the many other DOM elements in the web pages.

The benchmarks. In Figure 3, we also see the summary of the V8 and SunSpider benchmarks. We see immediately that the benchmarks are much smaller, in term 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.

Lessons. We summarize the findings from Figure 3 here:

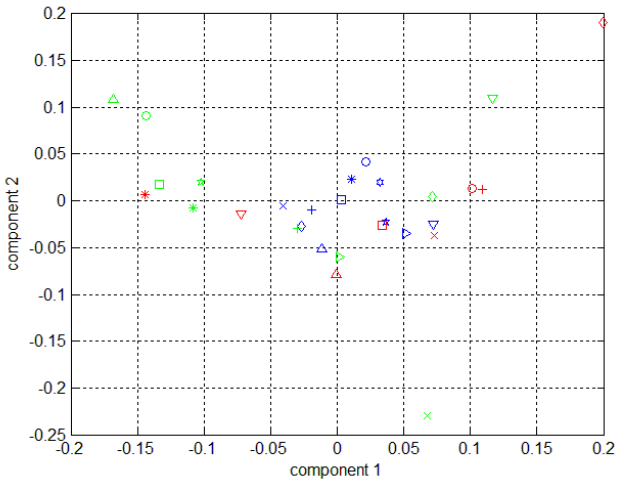
- Real applications have a significant amount of code from many sources.
- Source code is often not executed, in which case runtime efforts to translate that code is wasted.
- Many functions return after executing only tens of bytecodes. This point is supported more fully in Section 4.3.
- The V8 benchmarks are more representative of the real applications than the SunSpider benchmarks, and of those, `earley` is the most representative.

4.1.2 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 that they are themselves quite diverse.

To compare the resulting vectors, we used Principal Component Analysis (PCA) [21] to reduce the 160-dimensional space to 2 principal dimensions. Figure 6 shows the result of this analysis. In the figure, we see the three different program collections (real, V8, and SunSpider) indicated in different colors (blue, red, and green, respectively). The figure shows that the real sites cluster in the center of the graph, showing relatively small variation among themselves. For example, `cnn` 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,



(a) PCA clustering.

Legend:		
Popular Internet Sites (Blue):	V8 (Red):	SunSpider (Green):
+ = amazon	+ = richards	+ = 3d-raytrace
o = bing	o = deltablue	o = access-nbody
* = bingmap	* = crypto	* = bitops-nsieve
x = cnn	x = raytrace	x = controlflow
□ = ebay	□ = earley	□ = crypto-aes
♦ = economist	♦ = regexp	♦ = date-xparb
▲ = facebook	▲ = splay	▲ = math-cordic
▼ = gmail	▼ = V8 aggregate	▼ = regexp-dna
▶ = google		▶ = string-tagcloud
pentagram = googlemap		pentagram = SunSpider aggregate
hexagram = hotmail		

(b) Legend.

Figure 5: Opcode frequency distribution comparison.

`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 of 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. We revisit this point later when we examine the event handling behavior of the applications.

Figure 6 provides another view of this comparison. The figure presents the results of hierarchical clustering applied to all the programs in all the suites. We see the same clustering behavior in this view.

Of the benchmarks, we see that `earley` is again among the most representative. The benchmark `math` − `cordic` also has an instruction mix quite close to the real application cluster. We also see that several of the benchmarks cluster among themselves. For example, `richards` and `deltablue` are quite close.

4.1.3 Function Size Distribution

Figure 7 shows the distribution of static function sizes in the real applications and the benchmarks. The figure shows the cumulative size of all functions plotted on the y-axis, with the individual functions sorted by increasing size on the x-axis. These figures highlight our earlier conclusion that the benchmarks are far smaller than the real applications. Furthermore, we see that there are many small functions in the real applications, with half the total functions fitting within 200 kilobytes across all real applications. Again, `earley` is the most representative of the benchmarks in terms of the distribution of function sizes and total size.

To better understand the sizes of individual functions, we plot the distribution of static function sizes in Figure 8. The figure shows the size of all functions plotted on the y-axis, with the individual functions sorted by increasing size on the x-axis. This figure tells us what percentile of functions are larger than a certain size. From the figure, we see that the distribution of static function sizes is quite similar across the program suites. The median size appears to be around 100 bytes in real applications and in V8. In terms of the larger functions, we see that the 95th percentile range for the real applications is from 500 to 1000 bytes. These results suggest that while the total amount of code is greatly different in the real applications, in both benchmarks and real applications the vast majority of JavaScript functions are relatively small.

4.1.4 Hot Function Distribution

More important than the static function size distribution is the distribution and total size of hot functions in the applications. This characteristic determines how much code needs to be highly optimized and how much memory that code will occupy. Figure 9 shows the distribution of hot functions in the real applications (split into two subfigures, a and b), the V8 benchmarks, and the SunSpider benchmarks. Each figure shows the cumulative contribu-

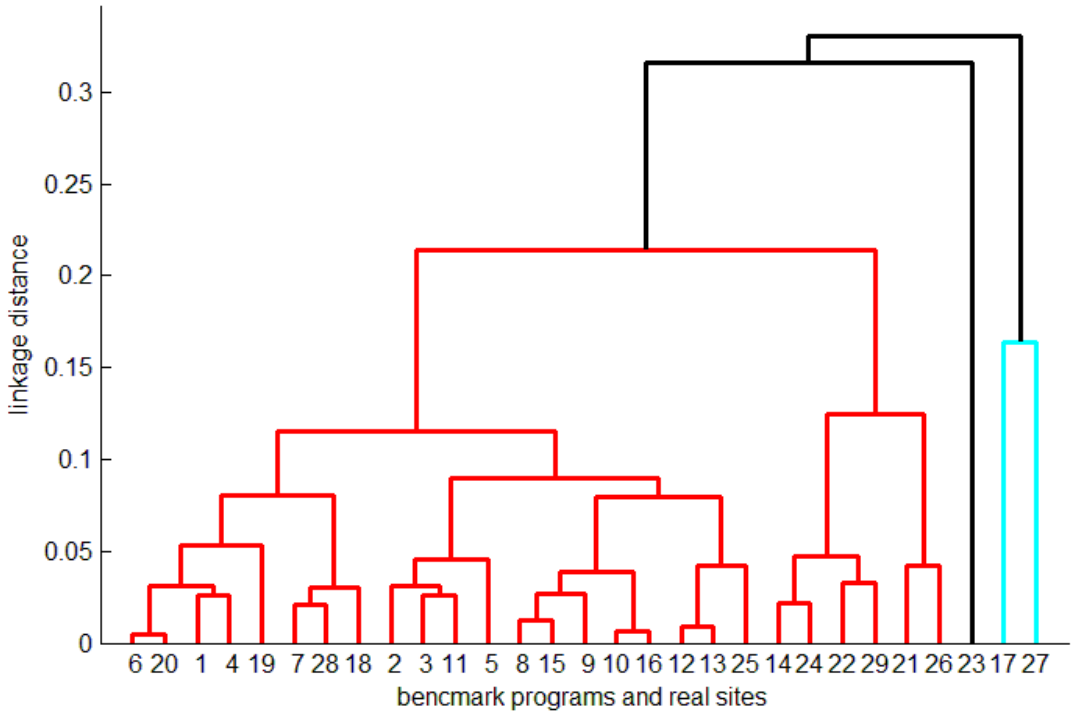
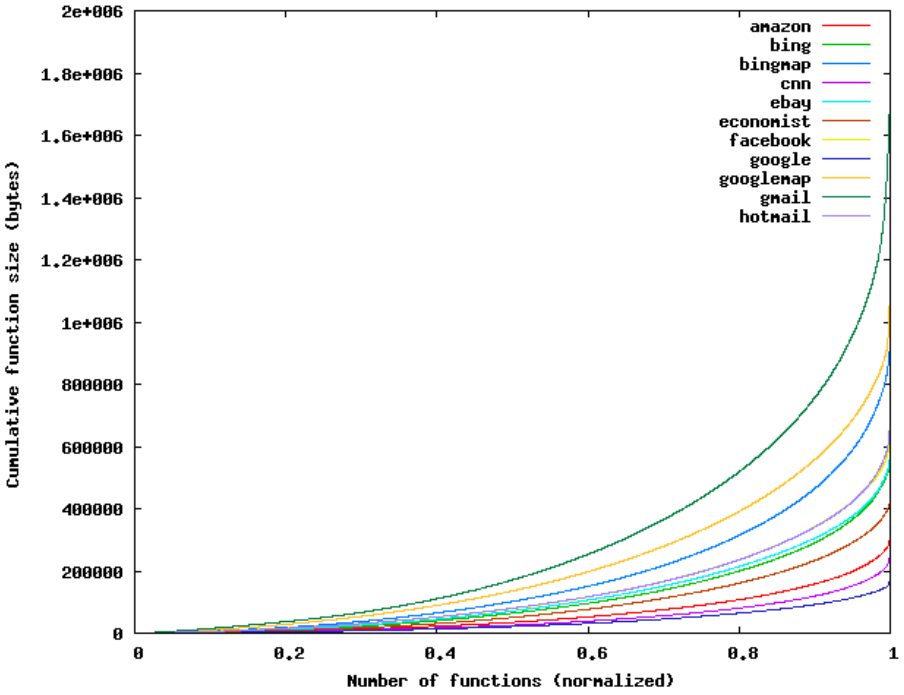
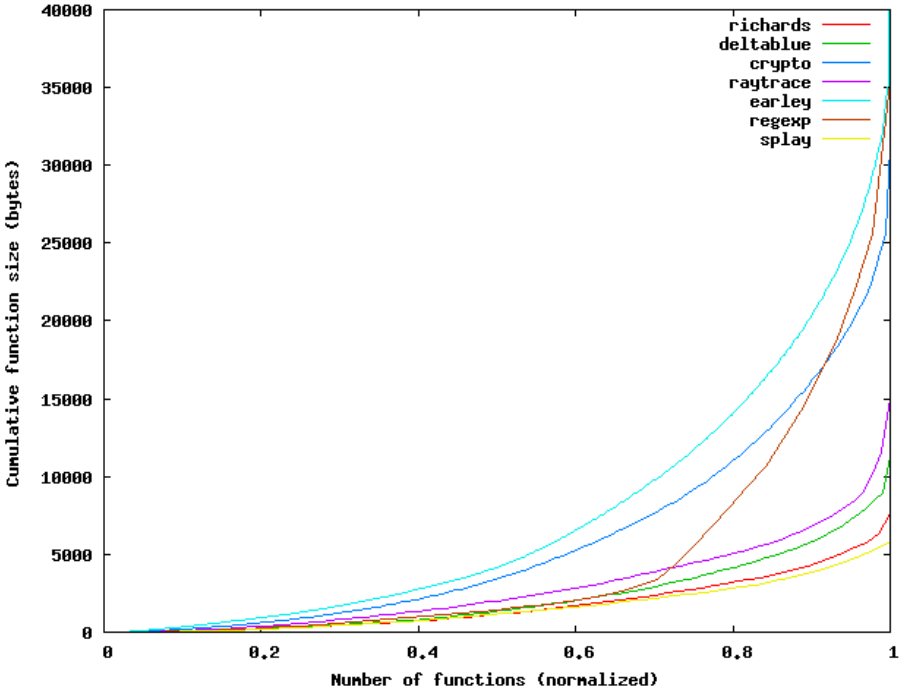


Figure 6: Opcode frequency distribution comparison. Key: **Real web sites** 1:amazon 2:bing 3:bingmap 4:cnn 5:ebay 6:economist 7:facebook 8:gmail 9:google 10: googlemap 11:hotmail **V8** 12:richards 13:deltablue 14:crypto 15:raytrace 16:earley 17:regexp 18:splay 1:earley 19:v8 aggregate **SunSpider** 20: 3d-raytrace 21: access-nbody 22: bitops-nsieve 23: controlflow 24: crypto-aes 25: date-xparb 26: math-cordic 27: regexp-dna 28: string-tagcloud 29: SunSpider aggregate

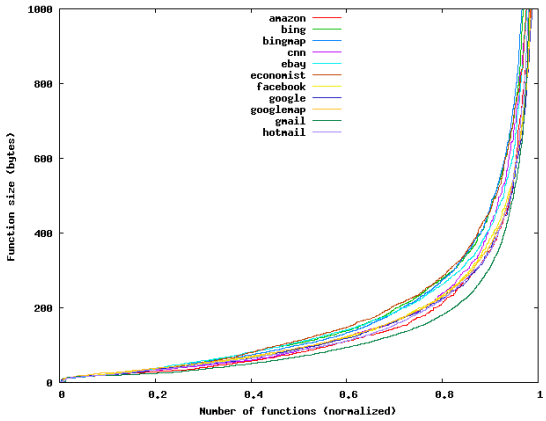


(a) Real web application function size distribution.

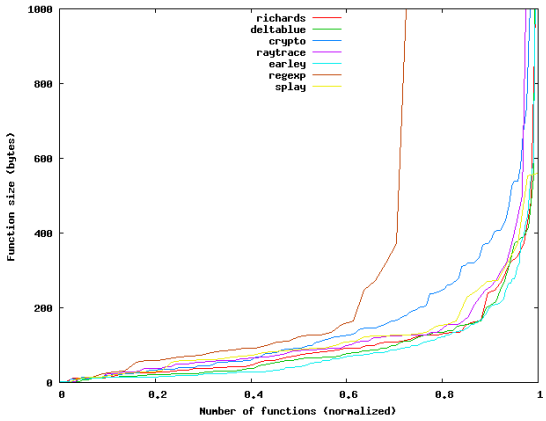


(b) V8 benchmark function size distribution.

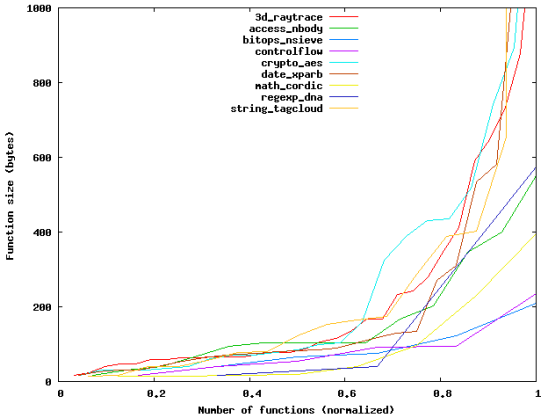
Figure 7: Static cumulative function size distribution.



(a) Real web application function size distribution.

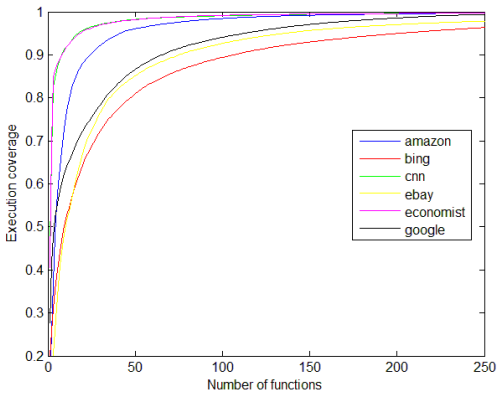


(b) V8 benchmark function size distribution.

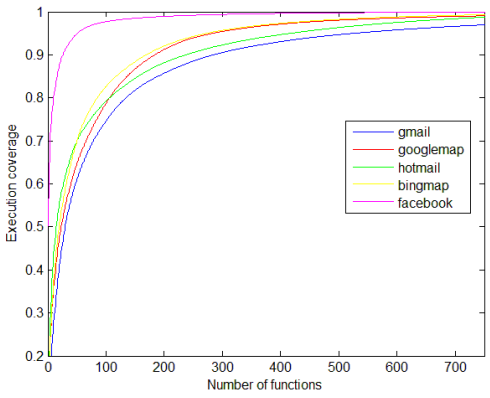


(c) SunSpider benchmark function size distribution.

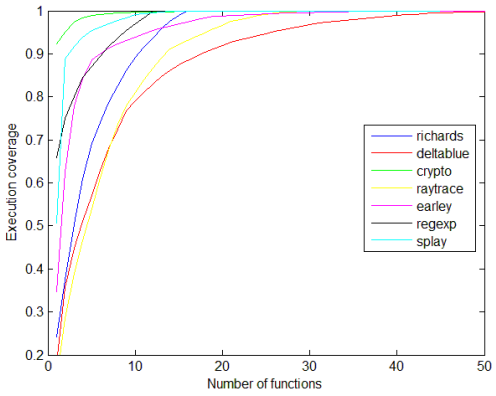
Figure 8: Static function size distribution.



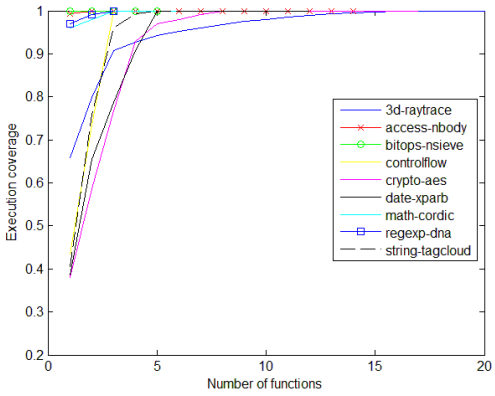
(a) Real web application hot function distribution (1).



(b) Real web application hot function distribution (2).



(c) V8 benchmarks hot function distribution.



(d) SunSpider benchmarks hot function distribution.

Figure 9: Hot function distribution.

tion 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) in some cases to get a better view of the left end of the curve. The figures show that all programs, both 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. Some of the real applications are even more highly skewed. As mentioned, `economist` has a single function that accounts for more than 50% of total execution. The `facebook` benchmark also has some very hot functions, which we consider in more detail when discussing event handling.

4.1.5 Code Behavior Discussion

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 elements. 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. JITing ten kilobytes of JavaScript source will happen much faster than JITing two megabytes. 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. When we discuss events in Section 4.3, we revisit this point.

Third, we observe that each individual function execution in the real applications is relatively short, which mirrors previous measurements of object-oriented programs. 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 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.

4.2 Object Allocation Behavior

We now consider the allocation behavior and object lifetimes in the JavaScript programs. JavaScript is a type-safe language with garbage collection, which can have significant memory overheads and CPU costs. Complicating this management is the fact that the DOM interface in JavaScript can create numerous cross domain references to objects (DOM to JavaScript and vice versa) leading to memory leaks in many commercial browsers. In this study, we omit the DOM interaction and focus entirely on objects in JavaScript, leaving studying that interaction for future work.

We first consider the number and distribution of types allocated by JavaScript programs. We then drill down on the String type, which is especially important, as we show. Next we consider the evolution of the heap over time in our applications, and finally, we consider object lifetimes.

4.2.1 Object Allocation Rates

One of the most important aspects of object behavior in JavaScript is the allocation rate. Here we compare the allocation rate of the real applications and the benchmarks.

Real web sites. Figure 10 summarizes the object allocation behavior of the real applications and benchmarks. Each row shows the total bytes allocated to specific types by that application. In Appendix C, we show the same data in terms of objects allocated (instead of bytes allocated).

The figure shows that our real applications allocate a significant amount of memory, ranging from one to almost

	Script Func	Arrays	String	Native Func	Date	Objects	Others	Total
amazon	3,092,649	877,060	6,815,942	14,848	50,240	6,442,936	18,272	17,311,947
bing	1,712,152	12,876	2,400,828	4,800	50,400	88,432	1,440	4,270,928
bingmap	8,614,419	186,224	9,243,964	2,848	40,576	1,834,048	6,016	19,928,095
cnn	854,749	28,724	2,107,978	13,984	4,640	368,044	13,280	3,391,399
ebay	7,576,009	127,320	3,132,490	16,800	11,648	994,412	14,976	11,873,655
economist	980,027	2,092,148	5,656,640	7,616	4,736	776,844	57,312	9,575,323
facebook	3,197,424	180,100	9,459,048	15,104	10,688	2,120,732	73,536	15,056,632
gmail	2,987,271	289,068	5,294,964	4,480	34,144	1,121,408	2,528	9,733,863
google	656,552	43,340	582,148	8,192	1,728	55,152	5,888	1,353,000
googlemap	3,602,156	450,048	4,626,230	2,496	186,848	4,188,932	15,392	13,072,102
hotmail	1,384,202	22,632	1,382,342	11,296	3,840	166,096	4,096	2,974,504

(a) Object allocation summary in real applications (bytes allocated).

	Script Func	Arrays	String	Native Func	Date	Objects	Others	Total
richards	7,656	472	448	416	128	2,404	64	11,588
deltablue	11,359	4,488	1,772	480	64	80,968	64	99,195
crypto	31,313	38,236	13,162	640	352	74,752	64	158,519
raytrace	15,751	44	478	640	64	3,192,276	64	3,209,317
earley	2,867,672	12,692	442,332	992	192	6,833,032	64	10,156,976
regexp	35,373	868,076	13,187,868	576	64	432,536	64	14,524,557
splay	5,874	11,091,216	25,586,804	352	288	4,943,620	64	41,628,218

(b) Object allocation summary in V8 benchmarks (bytes allocated).

	Script Func	Arrays	String	Native Func	Date	Objects	Others	Total
3d-raytrace	7,419	131,360	36,520	288	96	1,384	64	177,131
access-nbody	2,864	80	154	128	64	740	64	4,094
bitops-nsieve	564	20,000	154	96	64	32	32	20,942
controlflow	564	0	158	64	64	32	32	914
crypto-aes	11,759	353,344	47,640	512	96	64	64	413,479
date-xparb	5,341	124	238,662	544	96	640,300	32	885,099
math-cordic	862	48	154	96	128	32	32	1,352
regexp-dna	630	576	8,019,186	224	64	712	0	8,021,392
string-tagcloud	55,219	50,004	2,584,560	640	64	30,168	64	2,720,719

(c) Object allocation summary in SunSpider benchmarks (bytes allocated).

Figure 10: Summary of object allocations by type.

twenty megabytes of data, in the relatively short interactions we had with them. As with bytecode execution behavior, google is the most lean of the applications, while bingmap and amazon allocate the most data. Of the real applications that have the most application-like characteristics, bingmap, facebook, gmail, and googlemap, we see that allocating megabytes of data in a short period of time is common.

Benchmarks. The overall allocation of the benchmark programs is highly variable, with many benchmarks hardly allocating any data at all (e.g., richards deltablue, controlflow, math − cordic, etc.) and others allocating ten or more megabytes (e.g., earley, splay, and regexp). Only six of the benchmarks allocate more data than google, the real application that allocates the least data. The SunSpider benchmarks, in particular, have total allocation behavior that is highly unrepresentative of the real applications, and as a result, performance comparisons based on them will be highly skewed to the performance of code execution without regard to the efficieny of the object representation or memory management. Some of the V8 bechmarks are more representative in this regard. Benchmarks such as splay were probably chosen specifically to test the performance of the JavaScript engine's allocation performance, and they do allocate an amount of data that is comparable to that of our real applications. Nevertheless, based on total allocation alone, one can conclude that results from the benchmark suites are likely to mislead JavaScript engine designers about the importance of memory allocation in their implementations.

Lessons. One conclusion that can be reached across both the real applications and the benchmarks is that the only object types of significance are script functions, strings, arrays, and objects. The other types rarely, if ever contribute substantively to the overall memory allocation of the applications.

Another conclusion we can reach from the real web applications is that many make substantial use of all four major data types, with the mix of types varying between the applications.

4.2.2 Object Type Distribution

We now consider the mix of types allocated by the programs in more detail. Figure 11 illustrates the fraction of total objects allocated to different types for all the programs considered.

Real web sites. The graphs show that in the real applications the most frequently allocated data types are strings and script function objects. Recall that in many real applications the script functions often require one to two megabytes of source code, and as a result, represent a major fraction of total heap allocation in our scenarios. Strings represent the bulk of the remaining data allocated in most real applications, with objects contributing a significant fraction in amazon and googlemap.

Strings are particularly important in JavaScript because a major function of JavaScript is to rewrite DOM elements for rendering in the browser. Surprisingly, for the most part, arrays and objects play a relatively minor role in total allocation. Arrays are only prominent in economist, and we hypothesize that this heavy use is caused by the same hot function listed above. These results suggest the runtime representation and garbage collection of strings and script functions will have a major impact on overall JavaScript engine performance.

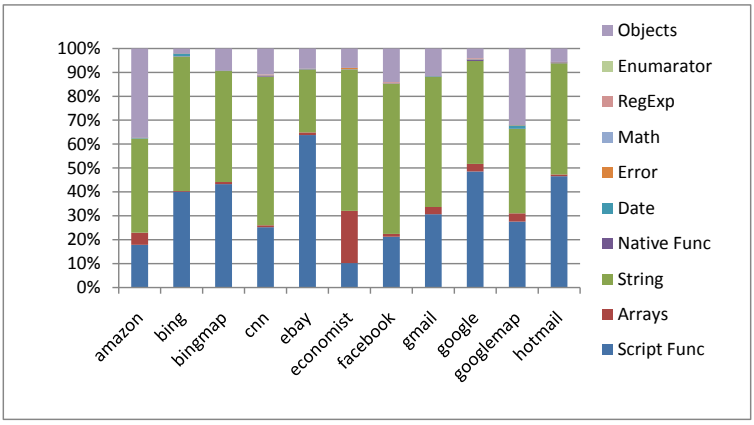
If we were to consider longer interaction sequences with these applications, we might imagine that the relative importance of script functions would diminish. This would happen because over time, once all the code for the application has been loaded, no further script functions need to be allocated, whereas strings, objects, and arrays would continue to be. This may well be the case for some of these applications, but, as we show when we consider the size of the live heap, many JavaScript heaps don't live long enough to allocate a substantial quantity of non-script data. This is especially true for applications based on a Web 1.0 design, where page transitions happen frequently when you interact with the application (see the discussion in Section 5.6).

Benchmarks. By contrast, the benchmark programs have quite different mixes of object types. Considering the five benchmarks with more than one megabyte of allocation (string − tagcloud, regexp − dna, raytrace, earley, regexp, and splay), we see that object allocation in these programs is highly bimodal, with almost all allocations being either strings (in string − tagcloud, regexp, regexp − dna, and splay) or objects (in raytrace and earley). Script functions represent a significant fraction of total allocation only in earley. Arrays play a significant role only in splay.

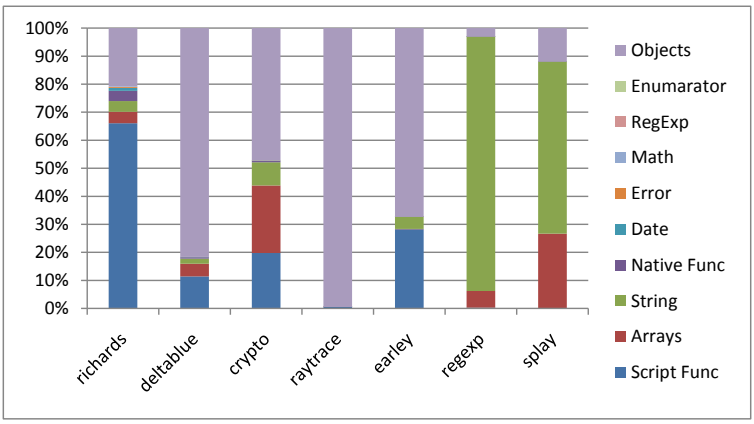
4.2.3 Type Distribution Clustering

In Figure 12 we apply the same analysis we applied to the distribution of opcodes to the distribution of data types. The figure again shows that the real applications are clustered tightly together, indicating that the mix of object types from one real application to the next is quite similar, as we have seen. As with the opcode frequency distribution, the benchmark programs are much more diverse and skewed.

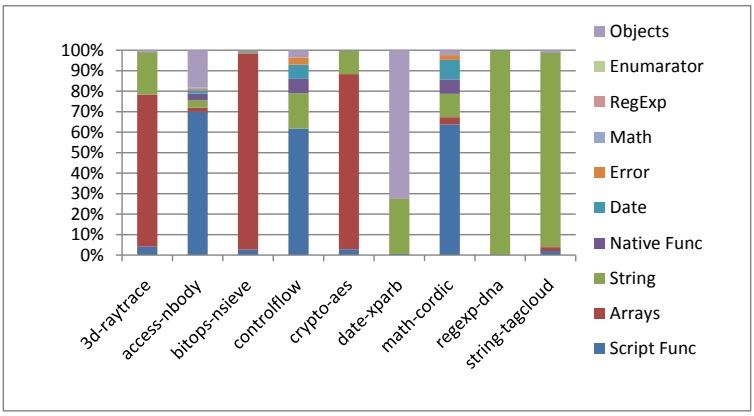
In this case, none of the benchmarks is very close to the real applications, with earley diverging significantly



(a) Object allocation distribution in real applications.

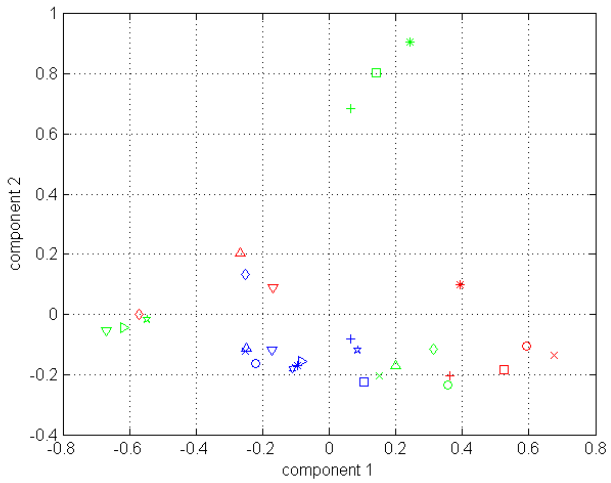


(b) Object allocation distribution in the V8 benchmarks.



(c) Object allocation distribution in the SunSpider benchmarks.

Figure 11: Distribution of objects allocated by type.



(a) PCA clustering.

Legend:		
Popular Internet Sites (Blue):	V8 (Red):	SunSpider (Green):
+ = amazon	+ = richards	+ = 3d-raytrace
o = bing	o = deltablue	o = access-nbody
* = bingmap	* = crypto	* = bitops-nsieve
x = cnn	x = raytrace	x = controlflow
□ = ebay	□ = earley	□ = crypto-aes
♦ = economist	♦ = regexp	♦ = date-xparb
▲= facebook	▲= splay	▲= math-cordic
▼= gmail	▼= V8 aggregate	▼= regexp-dna
►= google		►= string-tagcloud
pentagram = googlemap		pentagram = SunSpider aggregate
hexagram = hotmail		

(b) Legend.

Figure 12: Data type distribution comparison.

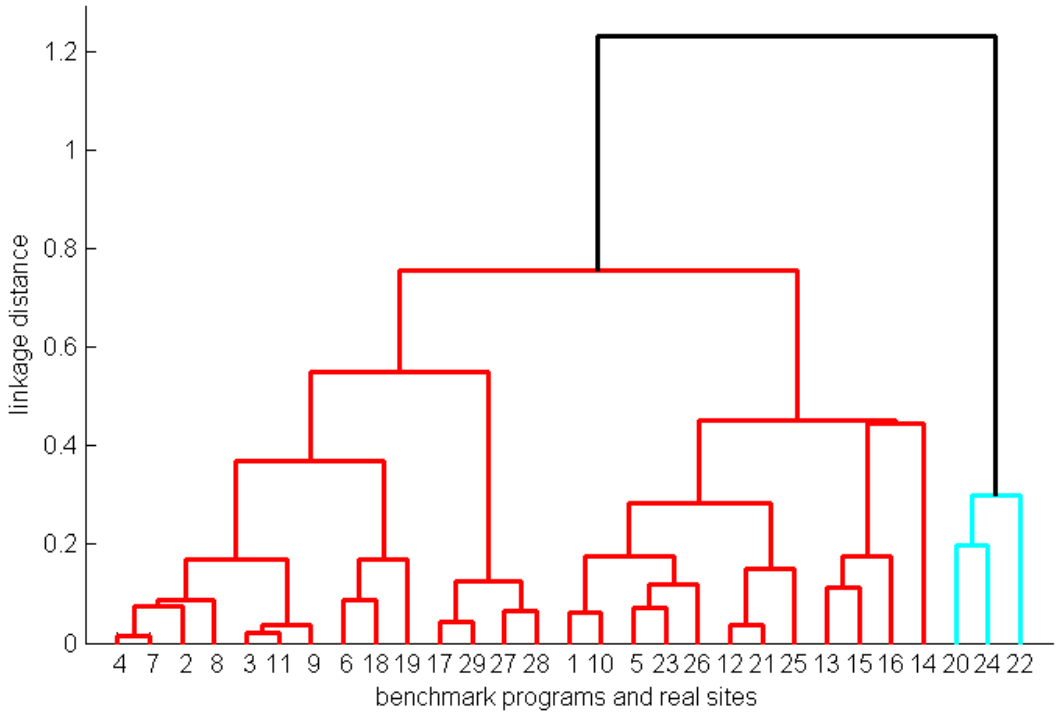


Figure 13: Type Frequency Distribution Comparison. Key: **Real websites** 1:amazon 2:bing 3:bingmap 4:cnn 5:ebay 6:economist 7:facebook 8:gmail 9:google 10: googlemap 11:hotmail **V8** 12:richards 13:deltablue 14:crypto 15:raytrace 16:earley 17:regexp 18:splay 1:earley 19:v8 aggregate **SunSpider** 20: 3d-raytrace 21: access-nbody 22: bitops-nsieve 23: controlflow 24: crypto-aes 25: date-xparb 26: math-cordic 27: regexp-dna 28: string-tagcloud 29: SunSpider aggregate

because of its relative overuse of object data and lack of strings. Figure 13 shows the same data in a hierarchical clustering. Both results show that the type frequency of the benchmark aggregates is closer the the real applications than most of the individual benchmarks.

4.2.4 Creation and Use of Strings

Because strings are such an important data type in JavaScript, we drill down further to understand how these strings are created. In Figure 14, we identify how the allocated strings observed in our benchmarks were created. The sources we considered include having the string appear as a constant in the source, being the concatenation of two strings, and other operations (substring, find character, etc. The figure shows that in the real applications, a significant fraction of all strings are constant strings (20-50%), while concatenation is a significant source in some applications, but not all. Of the real applications, only `google` fails to allocate less than a megabyte of string data. Of the four benchmarks that allocate more than a megabyte of string data (`string` − `tagcloud`, `regexp` − `dna`, `regexp`, and `splay`), we see that concatenation plays a major role only in `string` − `tagcloud`, and string constants play almost no role in `regexp`.

4.2.5 Live Heap Contents

We now consider how the contents of the heap vary as a function of time and data type. As already discussed, the only types that account for a substantial fraction of allocation are functions, strings, objects, and arrays, and we show the number of live bytes of these types in the following figures. Because we are only able to tell if an object is dead when it is garbage collected, our results are accurate only up to the granularity of garbage collection events, which are visible in the figures as a drop in the "combined" line. In this section, we focus on specific applications to illustrate important observations. We include figures for all the applications and benchmarks in Appendix A.

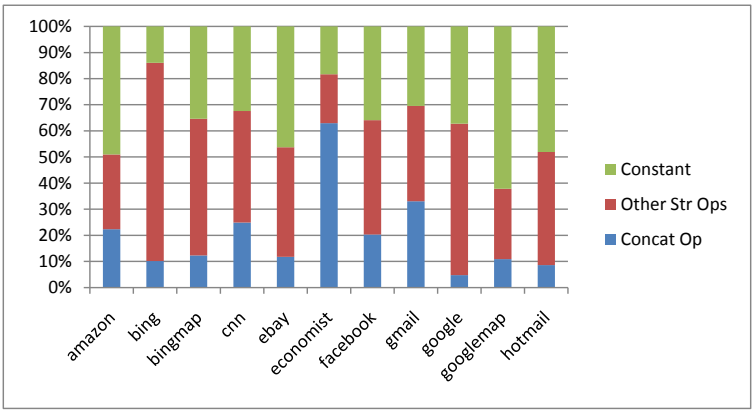
Real web sites. We start by considering the live heap in `gmail`, `facebook`, and `amazon` (Figure 15. Each figure shows live bytes of data as a function of time. For these results, we measure time as a function of bytes allocated by the applications. Thus, the figure for gmail shows a steady growth of the heap over time, with occasional reductions, indicating the onset of garbage collections. In addition to the combined line, we include lines indicating live bytes of each individual type. We can learn much about how an application is implemented by examining these graphs. `gmail`, for example, is implemented for the most part as a single web page in which JavaScript execution rewrites the page to present additional data as the user interacts with it. Amazon, on the other hand (somewhat out of necessity, since some of the interactions need to be secured), is implemented as a series of pages that the user interacts with in sequence. As a result, in `amazon`, we see a number of instances where the entire JavaScript heap is completely discarded and a new heap created immediately. Facebook represents a combination of these two behaviors, in which there is one page interaction with significant heap activity (likely caused by the login page), after which that heap is discarded and another heap is created and used for the rest of the session.

We now drill down on the results for each application. In `gmail`, we see that the heap grows overall, but that garbage collections reduce the size substantially on several occasions. We also see that the major data reclaimed by these collections is string data, and that the amount of live function and object data does not change much after a collection. Both function and object data increase gradually during the session, contributing to the overall growth of the heap, but string data represents a large fraction of data throughout the execution.

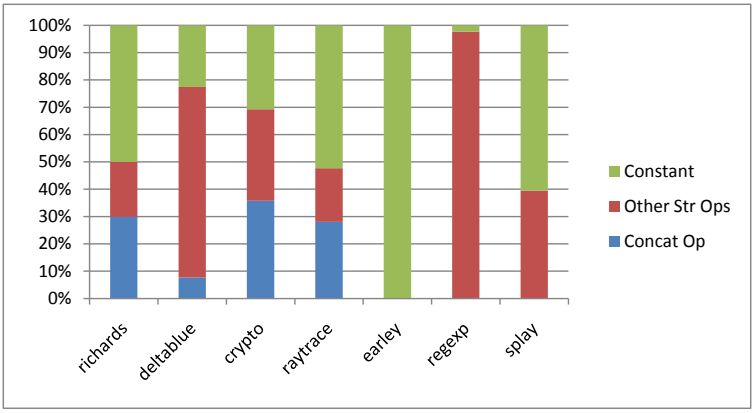
The `amazon` application is quite complicated and very different than `gmail`. We see that our session involved a number of different page visits, each of which required a new heap to be reconstructed from scratch. In some cases the size of the heap that is created and discarded is relatively large (more than a megabyte). There are several instances where the heap spikes with quite similar content distributions, suggesting that there is a significant amount of overlap in content between the page visits.

Unlike with `gmail`, objects represent a substantial fraction of all allocation in `amazon`, comparable in size to strings. Arrays are also quite important, at least on one of the pages visited. `amazon` illustrates that JavaScript is also heavily used in the web sites that are not exactly "applications" in the way that we think of `gmail`, `facebook`, or `googlemap`. A key difference between `amazon` and `gmail` is that the heaps in `amazon` are relatively short-lived. In such an application, the traditional way of thinking about object lifetimes (short versus long-lived) does not necessarily make any sense at all given that the entire heap is short-lived. In such cases, avoiding collection completely is possibly the best strategy. The fact that `amazon` appears to reconstruct the same or similar heap state again and again suggests that there are interesting implementation opportunities that have not been considered or proposed in the literature. Specifically, caching compiled functions across page loads might benefit sites like `amazon`. Caching data state across loads may also be possible and beneficial.

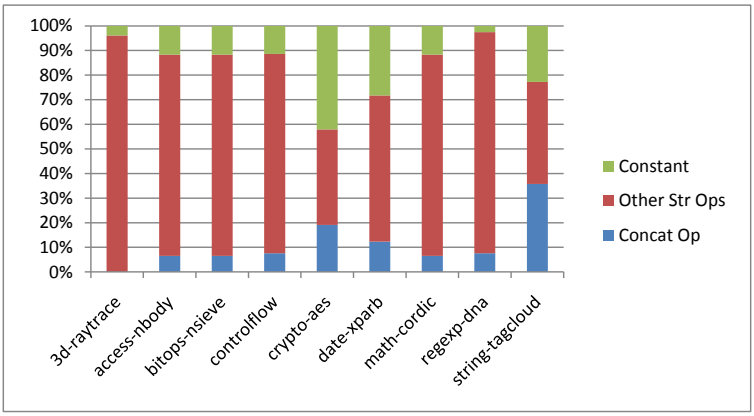
The `facebook` live heap graph shows that this application combines elements we see both in `amazon` and `gmail`. Much of the interation takes place on one page, and then a page transition occurs, such as visiting



(a) String Usage in Real Applications

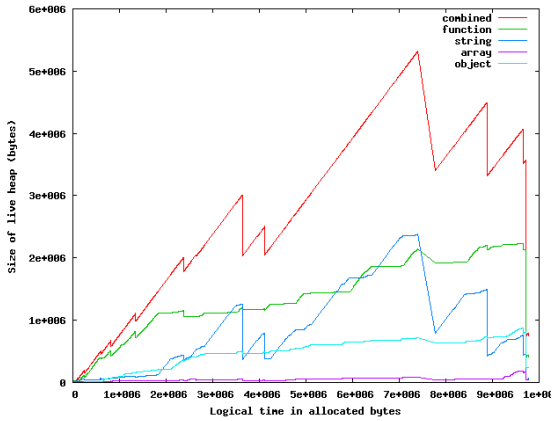


(b) String Usage in the V8 Benchmarks

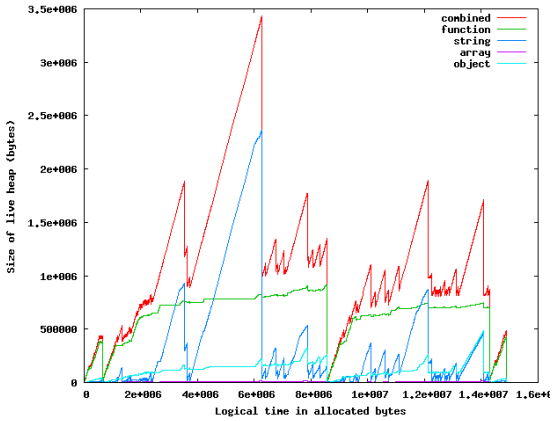


(c) String Usage in the SunSpider Benchmarks

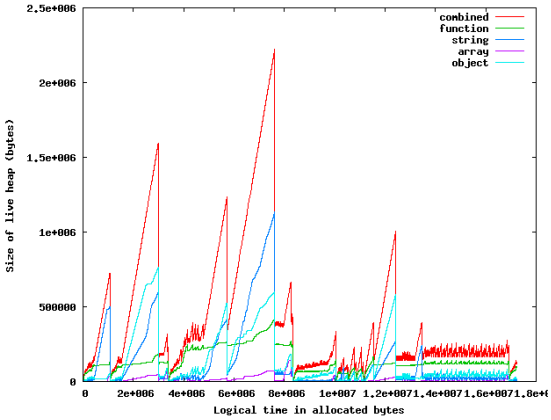
Figure 14: Distribution of String Creation Operations



(a) Live heap for gmail



(b) Live heap for Facebook



(c) Live heap for Amazon

Figure 15: Live heap contents as a function of time in real applications.

```
function GeneratePayloadTree(depth, key) {
  if (depth == 0) {
    return {
      array  : [ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 ],
      string : 'String for key ' + key + ' in leaf node'
    };
  } else {
    return {
      left:  GeneratePayloadTree(depth - 1, key),
      right: GeneratePayloadTree(depth - 1, key)
    };
  }
}
```

Figure 17: Major source of memory allocation in `splay`

a friend's page, and another complex interaction takes place. In `facebook`, we see that strings are the most volatile data type, with objects and functions contributing little to the space reclaimed during garbage collection. Like `amazon`, there appear to be similarities in the amount of code allocated in the two main pages that we observe in the figure, suggesting that caching the compiled code across the page transitions would avoid effort. Given the similarity of the live heap figures between `gmail` and `facebook`, we believe that this pattern is probably typical of many modern Web 2.0 applications and that JavaScript engines should be tuned to execute this mix of type allocations efficiently.

Benchmarks. We show the live heap graphs for four of the allocation-intensive benchmarks in Figure 16 The figure shows that the evolution of the live heap in these allocation-intensive benchmarks is very different than that in the real applications. `earley` has a two-phase behavior with a function-intensive first phase, and then an object-intensive second phase. While `earley` does have short-lived objects (for example, functions) we can see from the growing heap that many of the objects allocated in the second phase are long-lived. `regexp` and `raytrace` both have highly uniform allocation-intensive behavior focusing almost entirely on short-lived objects. Finally, `splay` allocates a significant amount of memory in a very uniform pattern, but does not free it. The source of this allocation is a single recursive function in the benchmark (see Figure 17). We also note that `splay` is to most allocation-intensive of all the V8 and SunSpider benchmarks. The ideal garbage collector for `splay` would be one that never runs.

Lessons.

- The real applications allocate a diverse collection of strings, functions, objects and arrays with strings being the most short-lived and functions being the most long-lived.
- Some real applications have short-lived heaps that are destroyed when one page is unloaded and regenerated when a new page is loaded.
- Live heap contents in the benchmarks do not reflect real applications.
- Using `splay` as a memory-allocation intensive benchmark is likely to mislead implementers in their GC design decisions.

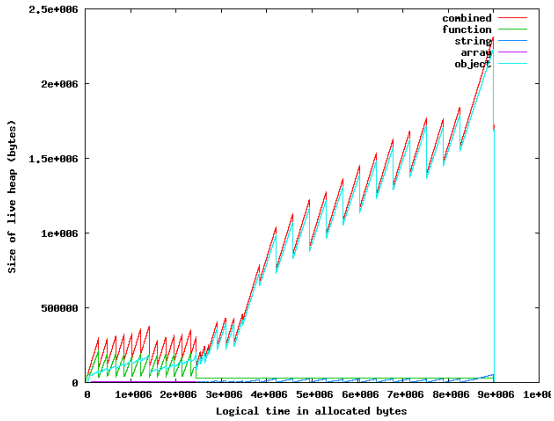
4.2.6 Object Lifetimes

In this section, we consider the lifetime distribution of objects allocated in the different programs. We first consider the overall lifetime distribution curves, and then drill down looking at specific applications.

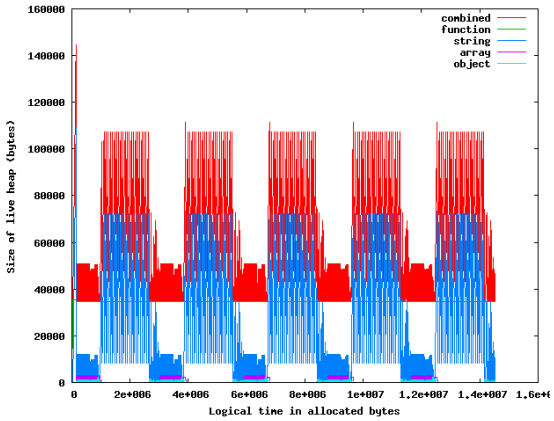
The real web sites. Figure 18 shows the object lifetime distributions in the real applications and in the V8 benchmarks. Results for SunSpider are similar to V8 and included in Appendix D. The figure shows that both the real applications and the benchmarks are bimodal. Both groups of programs have outliers (`gmail` for the real applications and `earley` and `splay` in the V8 benchmarks) in which objects are significantly longer lived the in the other applications. In the real applications, except for `gmail`, about 20% of the objects live from 500 to 1 megabyte of allocation. In the V8 benchmarks, we see that in all the applications except `earley`, objects are extremely short-lived, with 20% living less then 50 kilobytes on average. Both `raytrace` and `regexp`, which allocate the most objects of the benchmarks, have smooth curves but very short object lifetimes, which supports the allocation behavior we observed in the previous section.

Figure 19 shows the object lifetime distribution by type in `gmail`, `facebook`, `bing`, and `earley`. The figure shows how the lifetime distribution varies by object type in the applications. In the figure, the x-axis represents object ages in bytes, and the scale of the axis is chosen so that the total number of bytes allocated by the application occurs at the right end of the axis. Thus, for example, in `facebook`, because the application is split relatively evenly across two pages, most objects live at most half the total bytes allocated by the application.

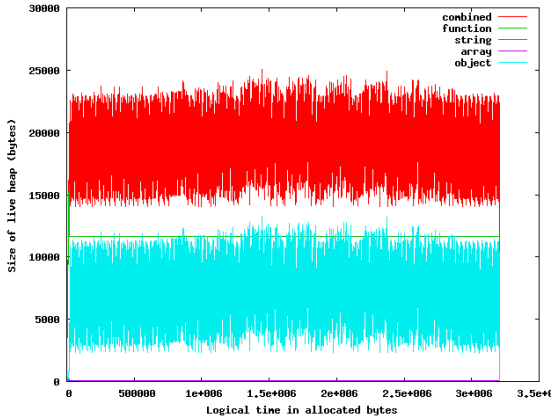
The three real applications, `gmail`, `facebook`, and `bing`, have a number of similarities: 1) strings are the shortest-lived, and functions are the longest-lived; 2) objects and arrays are in between functions and strings, with objects being the longest lived type next to functions. This mirrors our view of the live heap results, where garbage collections had little impact on the live heap contribution of objects. Objects are surprisingly long-lived in both `gmail` and `bing`, where we see that 20-30% of the objects live at least half the duration of the application.



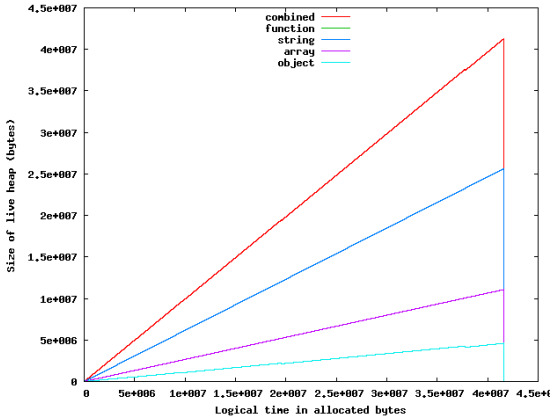
(a) Live heap for V8 `earley`.



(b) Live heap for V8 regexp.

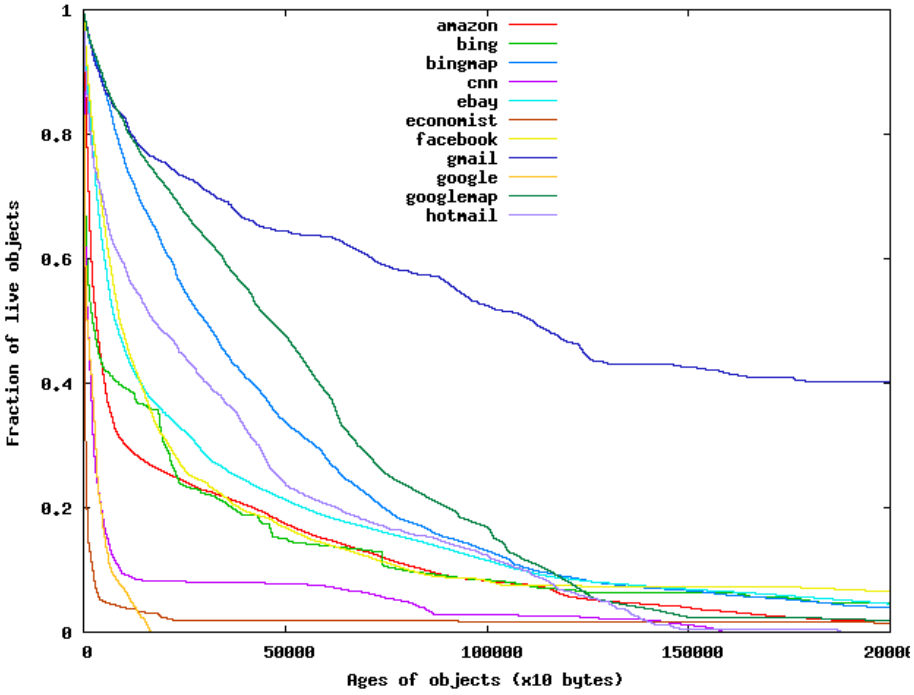


(c) Live heap for V8 raytrace.

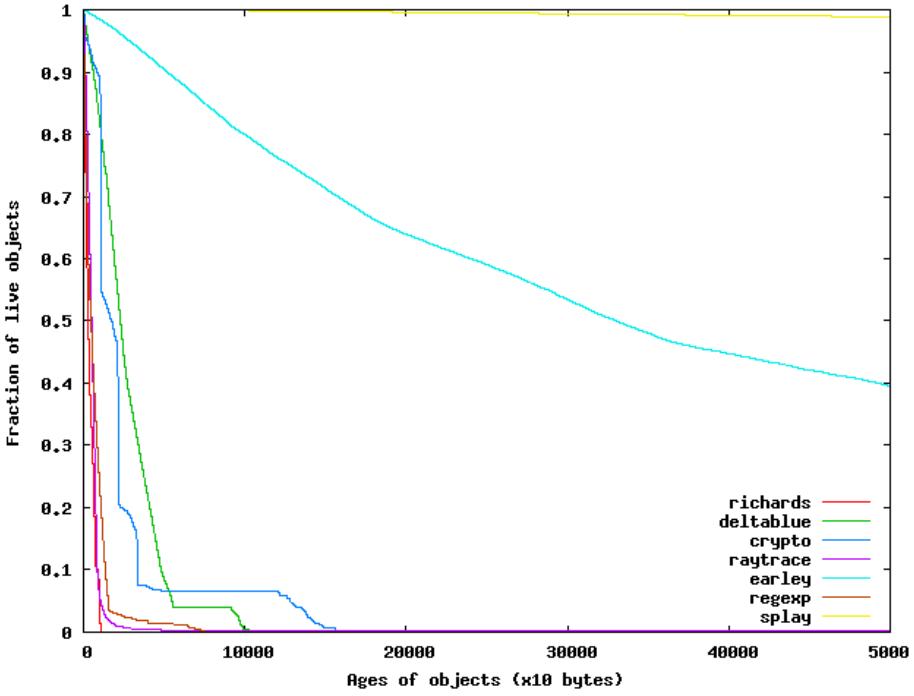


(d) Live heap for V8 splay.

Figure 16: Live Heap contents as a function of time in benchmarks.

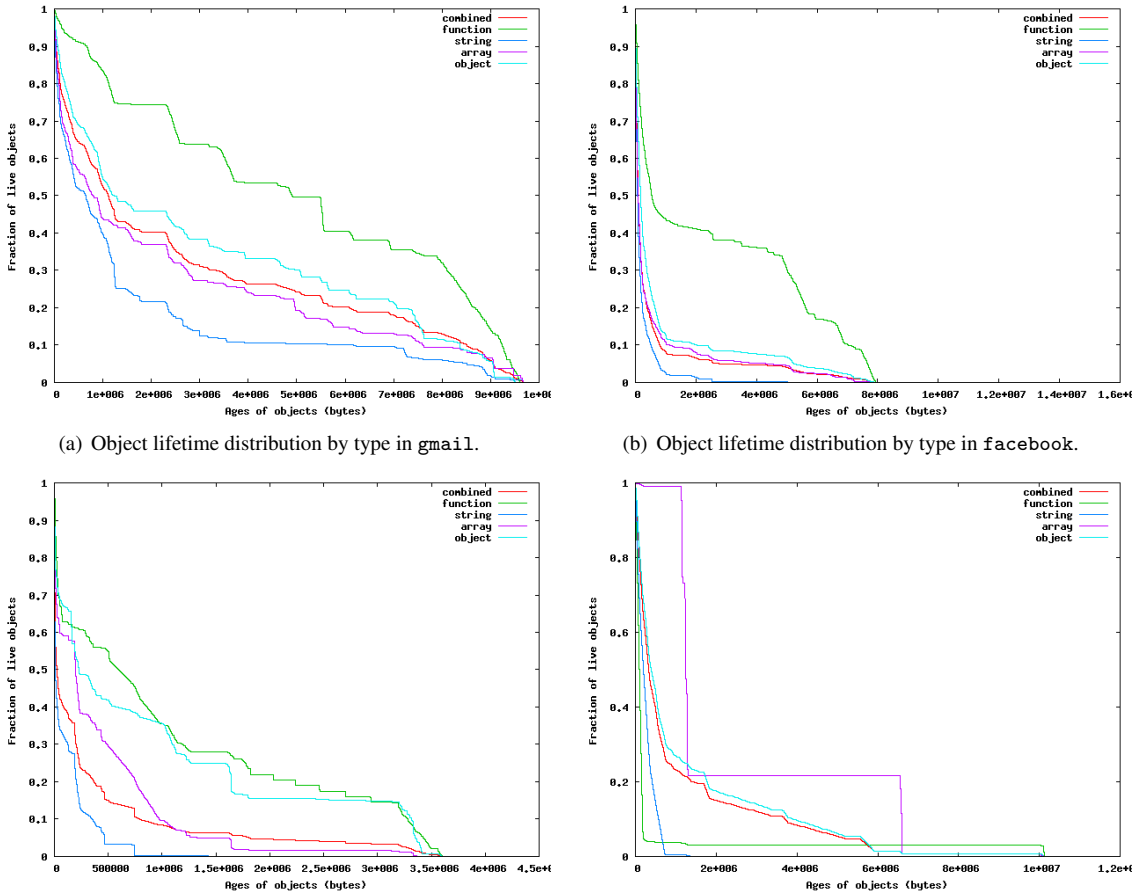


(a) Object lifetime distribution in real applications.



(b) Object lifetime distribution in V8 benchmarks.

Figure 18: Overall object lifetime distributions in real applications and V8.



(a) Object lifetime distribution by type in `gmail`.

(b) Object lifetime distribution by type in `facebook`.

(c) Object lifetime distribution by type in `bing`.

(d) Object lifetime distribution by type in `earley`.

Figure 19: Object lifetime distribution by type in selected applications.

The earley application, one of the most complex benchmarks, the other hand, has relatively simple behavior. earley allocates many short-lived function objects in its first phase, and then accumulates many longer-lived objects in the second phase. The result is that the overall lifetime distribution is quite skewed to older objects. As with the real applications, strings in earley are very short-lived.

4.2.7 Object Allocation Discussion

We have considered the object allocation behavior of the real applications and the benchmarks, and found significant differences.

- First, we observed that the mix of types allocated by the real applications is much different than most of the benchmarks, containing a large quantity of script functions and strings. Objects are less frequently allocated in the real applications and the lifetime of objects is considerably longer then that of strings in many cases. The fact that arrays and objects have relatively long lifespans in the real applications may reflect an attempt by the application developer to avoid allocating objects to reduce the garbage collection overhead in the application. In any case, we fail to observe this type distribution in most of the V8 and SunSpider benchmarks.

- Second, our analysis of the contents of the live heaps suggests that current web applications fall into two categories: those with page transitions that clear the JavaScript heap, and those that do not. In applications that do not have many page transitions, such as gmail, we observe that arrays and objects are relatively long-lived compared to strings. Of applications with many page transitions, such as amazon, by definition almost all objects are short-lived. Such sites do not require sophisticated memory management and would benefit most from a very fast and simple allocator. Being able to predict what class a site falls into and using an appropriate allocator might have performance benefits.

- Finally, in considering object lifetimes, we see that strings are by far the shortest lived types in JavaScript and that functions are commonly long-lived. Except for earley and splay, object lifetimes in the V8 and SunSpider benchmarks are extremely short-lived, suggesting that performance results of these benchmarks will not reliably reflect the effectiveness of the JavaScript engine's memory management implementatation. Even in earley, object lifetimes are significantly shorter than is observed in many of the real web applications, while in splay objects are almost never freed.

4.3 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.

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 in other circumstances. For example, handlers can be run when a timer timeouts, when a page loads, or called when an asynchronous XMLHttpRequest is completed. To fully understand the constraints on writing handlers, it is also important to know that in current browsers, such as Internet Exlorer and Firefox, 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. To achieve the goal of being a compelling interactive application, current web applications structure user interaction by attaching short-running handlers to DOM elements (as we document below). A handler that takes a significant amount of time to execute will make the web application appear sluggish and non-responsive.

Web applications also invoke JavaScript code in contexts that are not handlers. For example, when JavaScript source is processed as part of parsing the web page, it is also executed.

Figure 20 presents measures of the event handling behavior in the real applications and the V8 benchmarks[2]. We see that the number of events handled by the real applications is quite significant (typically thousands of events) where the benchmarks handle only a small number. Furthermore, we see that a substantial fraction of all bytecodes executed by the real applications occur in handler functions. As mentioned, handlers are called in many different contexts. The unique events column indicates the number of unique contexts in which handlers were invoked. These typically number in the hundreds. 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

[2] 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

(a) Event handler characteristics in real applications.

	# of events	unique events	executed instructions handler	executed instructions 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

(b) Event handler characteristics in the V8 benchmarks.

Figure 20: Summary of event handler characteristics.

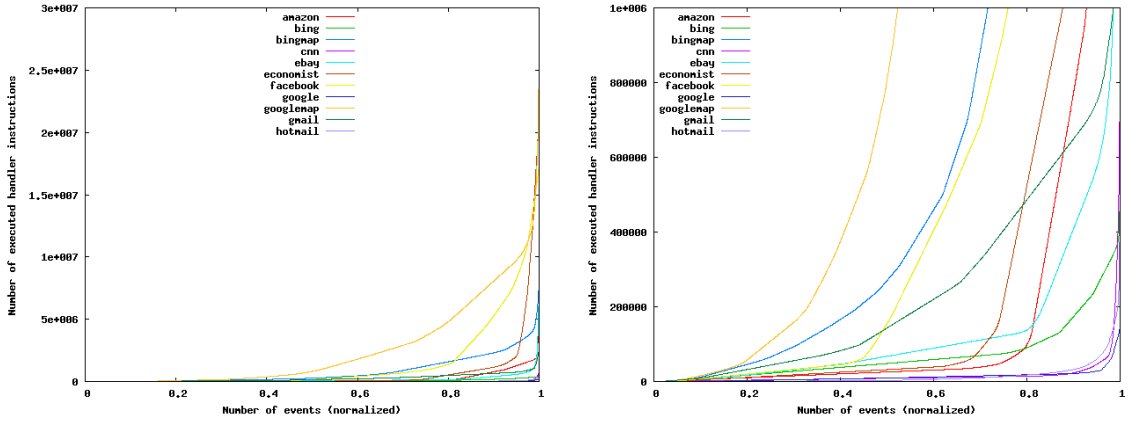
it has the highest number of events. Such short-duration handlers probably are invoked, test a single value, and then return.

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. In the `economist`, in which we observed a high average function call overhead, we see that many of the handlers are very fast (30 bytecode median) while many must also be quite slow (raising the average to 9,000 bytecodes per handler).

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.

We now consider the distribution of handler durations in more detail. Figure 21 shows the cumulative distribution of handler durations. The x-axis depicts the instances of handler durations, sorted by smallest first and normalized to one. The y-axis depicts the total cumulative bytecodes executed by handlers with duration equal or less than the x-axis value. For example in the first figure, we see that for `googlemap`, the shortest 80% of handlers account for a total of about 5 million instructions and the remaining 20% account for another 20 million more. The (a) figure shows the entire y-axis range, and the (b) figure zooms in on the y-axis to illustrate the sharpness of the knee of the curve better. The important conclusion from these figures is that there are many instances of short handlers executing in these applications, but that the majority of total instructions are executed in the longer running handlers. `googlemap`, `bingmap`, and `facebook` appear to have the least skew in their handler distributions, in some instances because they have taken care to reduce the execution time of the longest running handlers (e.g., in `facebook` and `bingmap`) and in other instances because the amount of work done during many handler events is substantial (e.g., in `googlemap`).

Figure 22 illustrates the distribution of handler durations for each of the applications. As with the previous figure, we show the handler instances sorted by duration on the x-axis and on the y-axis, we plot the actual duration (not the cumulative duration). As we saw previously from the medians, most 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 instructions to complete.



(a) Event handler characteristics in the real applications.

(b) Event Handler Characteristics (Y-axis zoomed)

Figure 21: Distribution of cumulative handler durations.

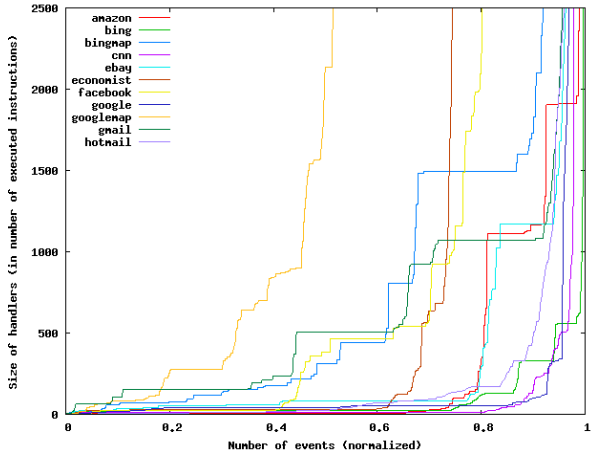


Figure 22: Distribution of handler durations.

5 Discussion

We have reviewed the behavior of both real JavaScript applications and benchmarks in a number of different ways, and determined that the benchmarks behave quite differently than the real applications. In this section, we review design decisions made by JavaScript engine designers (JEDs) and discuss what impact our results might have on the design process. We also speculate about current trends toward increasing JavaScript application complexity and how such trends might impact JavaScript engine design choices.

5.1 Trends

Application Complexity: While we have focused our current efforts on some of the most popular and complex web applications currently deployed, we believe that important web applications of the future are likely to be increasingly large and complex. For instance, Office 2010 Web Apps are likely to be some of the most complex deployed web applications to date [6]. This trend will almost certainly increase the disparity between the benchmarks and real applications that we have documented in this paper, further reducing the value of the benchmark results in guiding design decisions.

The Mobile Web: We also believe that the Mobile Web, the part of the web that is accessed from mobile devices such as smartphones like the iPhone, is likely to have a significant impact on the design of future web applications. Mobile devices are more compute- and memory-constrained than existing desktop and laptop computers, and power consumption is a significant constraint for these devices. Web application developers will need to tailor their web content to perform well on mobile devices and in many cases develop alternate content for them. For example, the Facebook application accessed via the iPhone browser is very different than Facebook accessed from a desktop computer. Improvements in JavaScript engine implementations on mobile devices will enable user experiences closer to the desktop experience. While we have not currently studied the JavaScript behavior of mobile web applications, we leave such as study for future work. Studies like ours that document what real mobile web applications look like would be a valuable guide for JEDs designing virtual machines for mobile devices.

Frameworks: There are many JavaScript libraries and frameworks such as jQuery and `prototype.js` already being used to build web applications. We did not make an effort to categorize the web sites we investigated to identify which frameworks were used to construct them. It would be an interesting to understand the impact of

particular frameworks on the metrics we consider. Some would certainly have more events, or uses of messages, etc. Based on the diversity of sources of JavaScript content that we document from the real sites (often more than 100 different contexts in which JavaScript is introduced), it is likely that now and in the future many interesting web applications will draw source content from numerous frameworks. It remains to be seen whether over time, as web application development matures, some of these frameworks will dominate and benefit from possible JavaScript engine support.

Similarly, many web applications, both in our sample and otherwise, are produced by automatically generating JavaScript code from other languages using a toolkit such as GWT [17] or Volta [30]. We suspect that such generated code might have properties different from hand-written code, however, we leave the study of aspects of machine-generated JavaScript programs for future work.

5.2 Object Model

Designers of any virtual machine have to think about how to represent objects, pointers, etc., to achieve compact representations and high performance. JEDs have to also consider the complex interaction between JavaScript objects, the DOM, and native browser objects, adding considerable complexity to the overall design. Further, JEDs have the additional complexity of efficiently implementing an object model in the presence of prototype-based language. Unfortunately, our current results do not speak directly to either issues related to the JavaScript/DOM interaction, or the efficient implementation of JavaScript in the absence of declared classes. Our results do show that a large fraction of all objects managed by the JavaScript runtime are code objects and that many of the remaining objects are strings.

As a result, an efficient implementation of string creation and concatenation are likely to provide a significant performance benefit, as documented elsewhere [7]. Likewise, source code, functions, and bytecodes account for a substantial fraction of overall allocation. Streamlining the implementation of these objects for reduced memory footprint in the common case of cold code would be beneficial. Similarly, code sharing with a copy-on-write scheme might prove helpful in a browser that has multiple concurrently open tabs.

We observed that web sites like Amazon and Ebay tear down and reconstruct significant heaps on every page visit, resulting in significant wasted effort. This real web behavior is a relatively recent phenomenon, and as a result we are not aware of any virtual machines optimized for this case. We believe that JavaScript virtual machines that are designed for speculatively retaining part of their heap contents between page visits might improve overall performance in cases where the user visits multiple pages within the same web application.

5.3 Code Generation

Recent web postings have placed emphasis on the execution performance of JavaScript code in recent browser releases, and significant improvements in benchmark performance have been achieved [32]. A key element of this performance improvement is the use of Just-In-Time compiler technology to generate efficient code for the tight loops present in some of the benchmarks. Many research approaches to JIT compilation have been proposed and evaluated in the context of Java implementations, most notably IBM's Jikes RVM [1] and the Java JIT implemented by Suganama et al. at the IBM Tokyo Research Lab. [36].

JavaScript implementors have demonstrated that these same techniques are also effective in the context of JavaScript runtimes, with the current implementations of Google Chrome's V8 browser, Apple's Safari browser and Mozilla's Firefox browser. What has not been demonstrated to date is the degree to which the ideas that have proven effective in Java JIT implementations are also effective in JavaScript. Notably, Java implementations for the most part are executed in a server environment, with potentially long-running applications.

One of the key insights from our work is that the amount of JavaScript typically handled by a real web application is one to two orders of magnitude larger than any of the benchmarks currently being used. As a result, the benchmarks may fail to identify a number of performance issues with current implementations. These issues include the time to JIT code and the amount of memory required to store JITed code. A very simple way to improve current benchmarks in this regard would be to include a megabyte of extra unused source code with each benchmark as a way to understand how the compiler handles larger quantities of little-used code. We also have identified real web application behaviors, such as `amazon`, where the time spent on a particular page does not necessarily warrant the effort to generate high quality code. On the other end of the spectrum, the benchmarks also fail to demonstrate the effectiveness of the existing JITs in cases where a user remains using a web application for longer periods of time. Issues such as code cache management become important in such scenarios because the amount of code used will grow monotonically over time.

Our results also suggest that the value of optimizing tight loops over integer data is over-emphasized in the benchmark programs. We suspect that most real web applications have few long-running loops. With opcodes per call averaging 20-50 instructions, JavaScript applications will likely see benefit from effective inlining approaches,

which have also been demonstrated in Java (e.g., see [37]).

5.4 Memory Management

Much research has been done on improving the overhead and pause times of garbage collection algorithms in many languages, including Java. Key concerns of GC designers include the distribution of GC-related pauses and their impact on application responsiveness, application memory footprint, GC execution overhead, and data reference locality. To date, memory management implementations in JavaScript engines have not been highly optimized and, because of issues related to pointers from the JavaScript heap to the DOM, often prone to memory leaks [2]. Some versions of browsers, for purposes of compatibility with the DOM, have implemented JavaScript object allocation as calls to the underlying C runtime allocator, preventing the performance benefits that bump-pointer allocation provides [18].

We have shown that the JavaScript benchmarks have highly uncharacteristic memory allocation behavior, when they allocate much memory at all. As a result, we believe that the current benchmarks underplay the importance of the JavaScript memory management implementation and if they continue to be used for performance comparisons, will incorrectly skew JavaScript implementations away from sophisticated memory management. We note in passing that the performance measures reported by the SunSpider and V8 benchmarks do not in any way reflect the memory size of the browser process, thus essentially eliminating any need to optimize memory usage for good benchmark scores. Given the status quo, the best strategy for getting high scores on these benchmarks is to turn off garbage collection completely as they are executing, as any time spent in GC will penalize the resulting execution time.

Our measurements indicate that strings are far more important in JavaScript than in previous managed languages such as C# and Java. Furthermore, we observe that the benchmarks also do not accurately reflect the degree to which managing function objects has an impact on memory usage and collector efficiency. We have also observed that in single-page web applications, such as `gmail`, many objects and arrays have relatively long lifetimes compared to strings. We believe that further study is required to understand the allocation behavior of long-lived web applications. One interesting difference between object behavior in Java and JavaScript is that a significant part of the structured data in JavaScript is explicitly held in the DOM elements, shifting the balance of the JavaScript allocation to interfacing with the DOM and external web sites with strings.

5.5 Tools

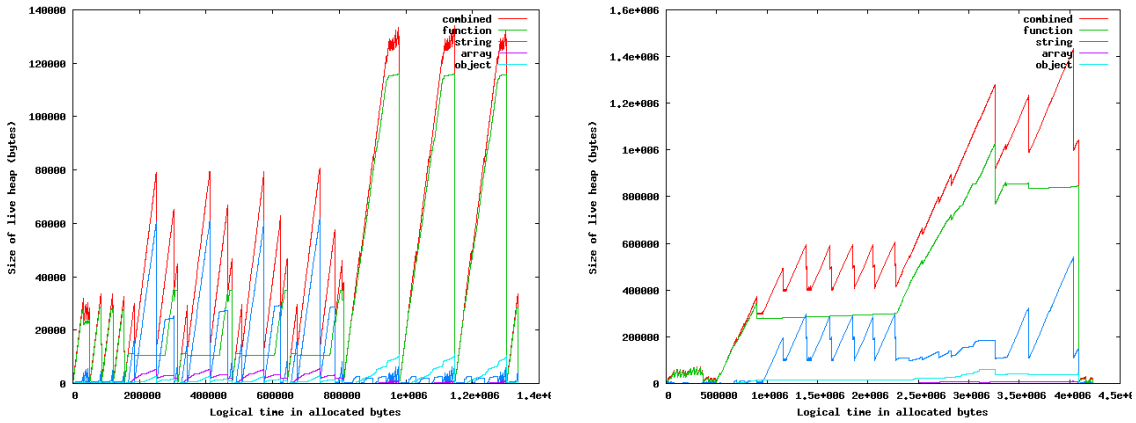
Our measurements have highlighted a number of behaviors in both the benchmarks and the real web sites that are unrepresentative and likely inefficient. For example, we observed a long-running loop in the `economist` web site and we observed that the V8 benchmark `splay` fails to free any of the large amount of memory it allocates. It is likely that such inefficiencies exist in such important benchmarks and applications because the tools to expose these inefficiencies are unavailable to many developers. We believe that if the kinds of information we report in this paper were easily available to many web application developers, the efficiency of JavaScript in web applications overall would likely increase. We believe that information about event handler frequencies, durations, and timing would be of considerable interest, especially with respect to efforts to increase application responsiveness.

While tools such as AjaxScope [24] use JavaScript source code rewriting to dynamically profile web pages, our measures, taken by directly instrumenting the JavaScript engine itself, provide a finer-grain view of JavaScript execution. Our approach was also facilitated by our ability to modify the JavaScript interpreter loop, giving us a natural virtual unit of execution (the bytecode) to report measures of execution time. We recommend that future JavaScript engines be constructed in such a way that hooks providing information similar to what we report be provided in the implementation.

5.6 What is a Web Application?

The live heap graphs presented in Section 4.2.5 also provide insight about design choices made by web site developers and illustrate that the relationship between a "web application" and a set of pages depends on the implementation. For example, Figure 23 shows the live heap graphs for `google` and `bing`. These two web sites offer very similar functionality and we performed the same sequence of operations on them during our visit.

We see from our measurements of the JavaScript heap, however, that the implementations of the two applications is 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` approach is highlighted in this case by looking at the right hand side of each subfigure. In `google`, we see that the contents of the JavaScript heap, including all the functions, is recreated repeatedly during our visit, whereas in the `bing` heap, the functions are allocated only once. We also note that the heap size of the `google` heap is significantly smaller than the `bing` heap. These significant differences might either be a result of Google making a conscious decision that their approach has efficiencies or, given that the `bing`



(a) Live heap for `google`.

(b) Live heap for `bing`.

Figure 23: Live heap contents as a function of time for two search applications.

search application was implemented in recent years, and therefore is likely to be more influence by the AJAX programming model, it may be a historical artifact.

6 Related Work

There have been many efforts to measure the behavior of programs over the years, most recently focused on Java. Understanding program behavior, including memory allocation, is necessary to design better compilers, runtimes, and hardware to support executing programs written in a language. Important algorithmic concepts, such as branch prediction, instruction and data caching, and generational garbage collection, all result directly from observing the behavior of real programs.

6.1 JavaScript and Dynamic Languages

There are surprisingly few papers measuring specific aspects of JavaScript behavior, despite how widely used it is in practice. A very recent paper by Lebresneet al. measures aspects of the use of dynamic aspects of JavaScript programs in actual use [27]. While their goals are very different from ours (their purpose is to develop a type system for JavaScript), some of their conclusions are similar. Specifically, they look closely at how objects use the prototype chain in real applications. Like us, they consider V8 benchmarks as well as real web sites and find differences between the benchmarks and real sites. Unlike us, they do not provide a comprehensive analysis of JavaScript behavior along the axes we consider (code, data, and events).

One closely related paper focuses on the behavior of interpreted languages. Romer et al. [34] consider the runtime behavior of several interpreted languages, including Tcl, Perl, and Java, and show that architectural characteristics, such as cache locality, is a function of the interpret itself and not the program that it is interpreting. While the paper does provide some insights into the mix of abstract instructions being executed, it does not specifically consider object allocation or event handling as we do.

6.2 Java and C#

Dieckmann and Holzle consider the memory allocation behavior of the SPECJVM Java benchmarks [10]. They consider object type, live heap composition, and object lifetime as we do. One of our goals was to mirror Dieckmann's and Holzle's work for the JavaScript language, since JavaScript and Java are very different languages. In terms of differences, they do not examine other aspects of Java behavior such as code execution or exception handling. Finally, the do not examine differences between Java benchmarks and real Java applications. While Dieckmann and Holzle consider SPECJVM to represent real-world applications, a strong motivation for the creation of the DaCapo Java benchmark suite [3] was the understanding that SPECJVM was not representative of more complex Java applications. We have the fortune of being able to measure real JavaScript applications directly and report on their behavior.

A number of papers have examined the memory reference characteristics of Java programs [10, 25, 33, 35, 38] specifically to understand how hardware tailored for Java execution might improve performance. The complex interaction between reference locality, high allocation rates, garbage collection, and concurrency creates a fruitful domain for research and innovation. Also, because both Java and JavaScript are garbage collected, studies of Java memory behavior can to some degree also inform hardware design to support JavaScript. However, we have also

observed that there are significant differences between JavaScript memory allocation and Java as well, for example, in the way in that strings play a major role. It is likely that more detailed studies of JavaScript execution are warranted to understand how effective hardware or runtime support can improve performance over existing methods.

Doufour et al. present a framework for categorizing the runtime behavior of programs using precise and concise metrics [11]. They classify behavior in terms of five general categories of measurement: size, data structures, polymorphism, memory use, and concurrency. They go on to report measurements of a number of Java applications and benchmarks and use their result to classify the programs into more precise categories. Our measurements correspond to some of the metrics mentioned by Doufour et al., although we do not do as systematic a job of categorization as they do. We consider some dimensions of execution that they do not, such as event handler metrics, and compare benchmark behavior with real application behavior.

Compared to the abundance of papers measuring different aspects of Java behavior, there are relatively few papers documenting C# program behavior. Kassim et al. [22] focus on micro-benchmarks performance in C#. Krintz maintains links to a collection of C# benchmarks, some of which are transcribed versions of Java benchmarks [26].

6.3 C and C++

Our work also follows from previous work comparing the code behavior and object allocation of C and C++ programs. Calder et al. [4] measure a large range of metrics in a collection of C and C++ programs and conclude that C++ programs, especially when written to take advantage of the object-oriented features of C++, behave differently than C programs.

Specifically, C++ programs have many more procedure calls, execute more loads and stores, and allocate more dynamic memory. We examine a number of similar metrics in JavaScript programs, including metrics such as event handling, which were not considered in previous work. Because JavaScript often executes in the context of a browser, our work also identifies web application properties that have previously not been presented.

6.4 Benchmarking

Several papers have attempted to improve the quality of Java benchmarks. The Java Grande benchmarks are a collection of Java programs with a focus on numerical computations that are used in evaluating the effectiveness of a particular Java implementation for high performance computing [29]. The most notable general purpose benchmark suite in the category, the DaCapo benchmarks [3], have largely superceded the SPECJVM benchmarks as the benchmarks of choice for evaluating research on Java implementations. The authors of this suite make the compelling case that doing effective research on the Java platform requires benchmarks that are both real and more complex than benchmark suites used to evaluate languages like C, C++, and Fortran. One of the goals of our work is to establish a baseline of understanding that can inform the creation of JavaScript benchmarks that parallel the DaCapo benchmarks for the Java language.

A number of papers have benchmarked browsers, which include the JavaScript subsystem. Nielson et al. measure the performance of four popular browsers in dimensions that include JavaScript, rendering, and what they call AJAX tests, which include GET and POST performance and the performance of the DOM APIs [31]. To measure JavaScript performance, they rely on the SunSpider benchmarks, "...well recognized as a reliable measure of a browser's JavaScript performance".

7 Conclusions

We have presented the first detailed measurements of the behavior of JavaScript applications, including widely-used and commercially important web applications and sites such as GMail and Facebook, as well as the JavaScript benchmark suites, SunSpider and V8, which are widely used to report the performance of JavaScript engines. Because web applications reveal all their client JavaScript code to the browser, we have the unprecedented opportunity to provide detailed behavior measurements of live commercial applications. We have measured three specific areas of JavaScript runtime behavior:

1. Functions and code;
2. Heap-allocated object and data;
3. Event and handlers.

Our results show that JavaScript web applications are large, complex, and highly interactive programs. Furthermore, while the functionality they implement varies significantly (search, mail, e-commerce), we also observe that the real applications have much in common with each other as well.

In contrast, the JavaScript benchmarks are fleetingly 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,

suggesting that previously unexplored optimization strategies may be productive in practice.

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 gratefully acknowledge the efforts of Corneliu Barsan, who helped us with the instrumentation of the IE8 JavaScript engine. We also thank Allen Wirfs-Brock for early discussions regarding our strategy for measurement. We also thank Trishul Chilimbi, David Detlefs, Leo Meyerovich, Karthik Pattabiraman, and Herman Venter for their support and feedback during the course of this research.

A Heap Statistics

In this appendix, we present our measurements of the live heap over time and object lifetime distribution for all real applications and benchmarks (see Figures 24 to 35). While we highlighted certain aspects of behavior in the body of the paper, here the reader can investigate the full range of behaviors exhibited. These figures serve to further highlight to contrasting complexity of the real applications and the benchmarks.

B Trace Statistics

The results presented in this paper were extracted from execution traces that our instrumentation system provides. The binary-encoded traces contain much more information than we have presented here. In this section, we show the full trace sizes to provide an indication how much data was collected.

In the case of instruction execution, our traces capture information about the bytecodes executed and the order of execution. In the case of object allocation, our traces identify the point at which objects are allocated and freed when the garbage collector determines that they are no longer usable.

C Object Allocation Rates

Figure 37 presents the total number of objects allocated by type in the real applications and benchmarks.

	Instruction trace size	Data trace size
amazon	114,446,160	21,512,000
bing	17,106,480	4,342,912
bingmap	197,601,120	25,622,896
cnn	71,806,320	8,759,248
ebay	136,659,600	14,399,584
economist	83,924,640	57,689,424
gmail	303,004,080	8,396,720
google	7,319,520	1,387,504
googlemap	807,679,440	31,056,496
hotmail	11,141,280	1,938,192
facebook	151,426,800	20,036,480
min	7,319,520	1,387,504
max	807,679,440	57,689,424

(a) Real Application Summary

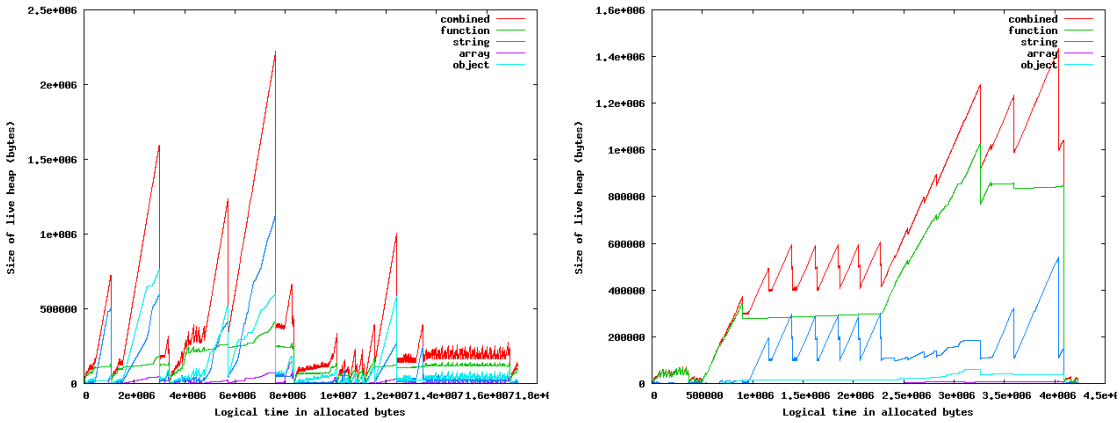
	Instruction trace size	Data trace size
richards	58,326,480	1,629,952
deltablue	81,558,720	578,512
crypto	71,714,160	10,888,288
raytrace	154,787,760	7,997,664
earley	585,851,760	19,635,536
regexp	69,120	13,583,840
splay	488,460,240	140,354,000
min	69,120	578,512
max	585,851,760	140,354,000

(b) V8 Application Summary

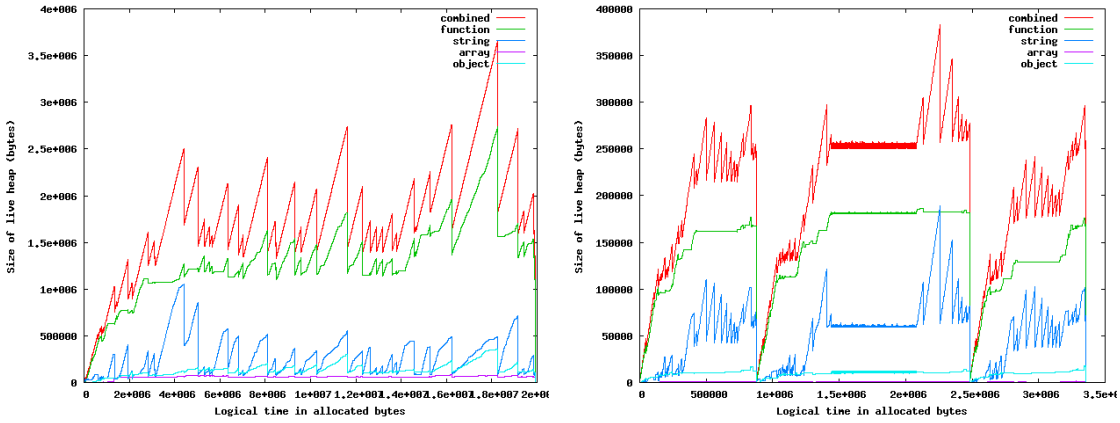
	Instruction trace size	Data trace size
3d-raytrace	40,774,320	1,022,416
access-nbody	3,285,360	5,404,880
bitops-nsieve	3,600	672
controlflow	176,754,240	560
crypto-aes	7,236,720	960,352
date-xparb	25,948,800	6,345,136
math-cordic	54,011,520	992
regexp-dna	2,160	12,336
string-tagcloud	45,989,280	3,131,344
min	2,160	560
max	176,754,240	6,345,136

(c) SunSpider Application Summary

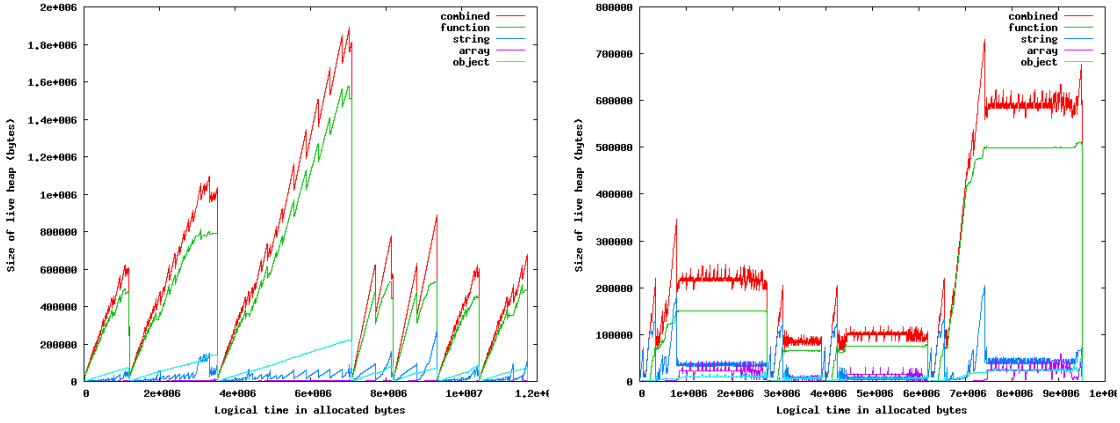
Figure 36: Trace file sizes.



(a) `amazon` (b) `bing`

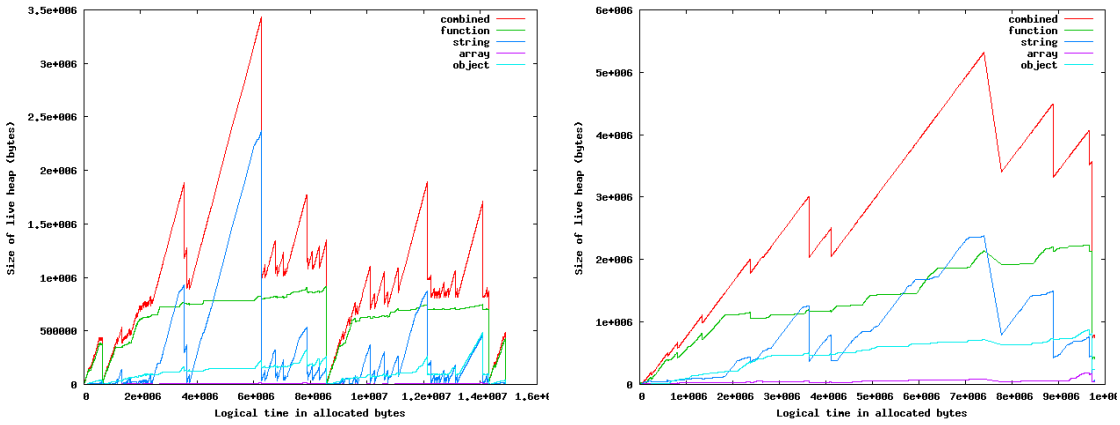


(c) `bingmaps` (d) `cnn`

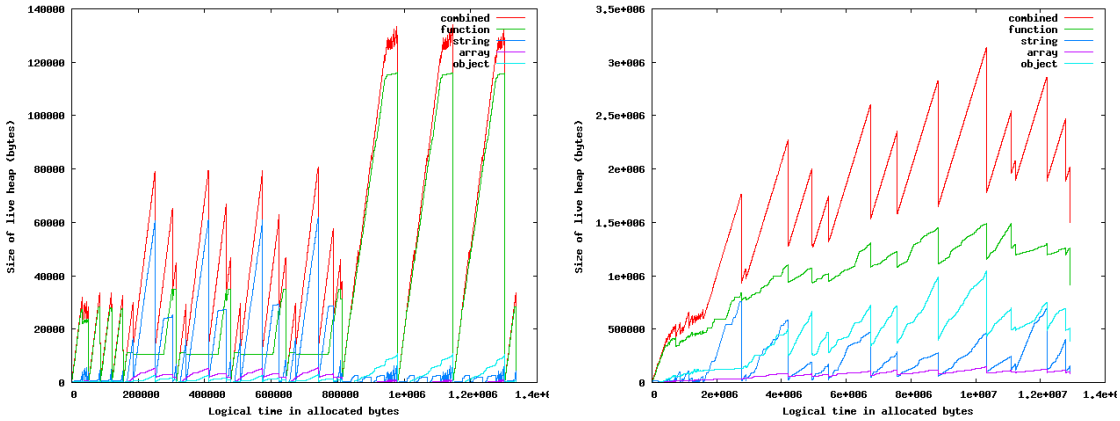


(e) `ebay` (f) `economist`

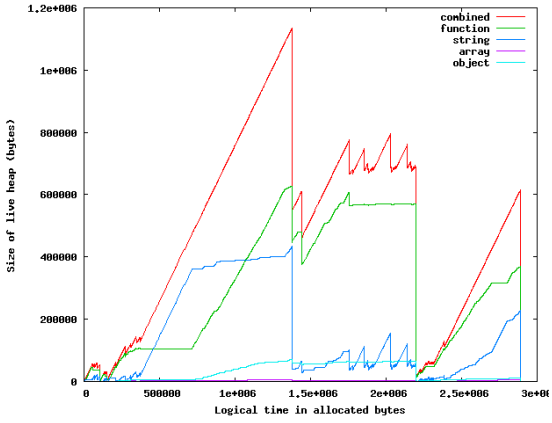
Figure 24: Live heap object composition over time (real sites part 1).



(a) `facebook` (b) `gmail`

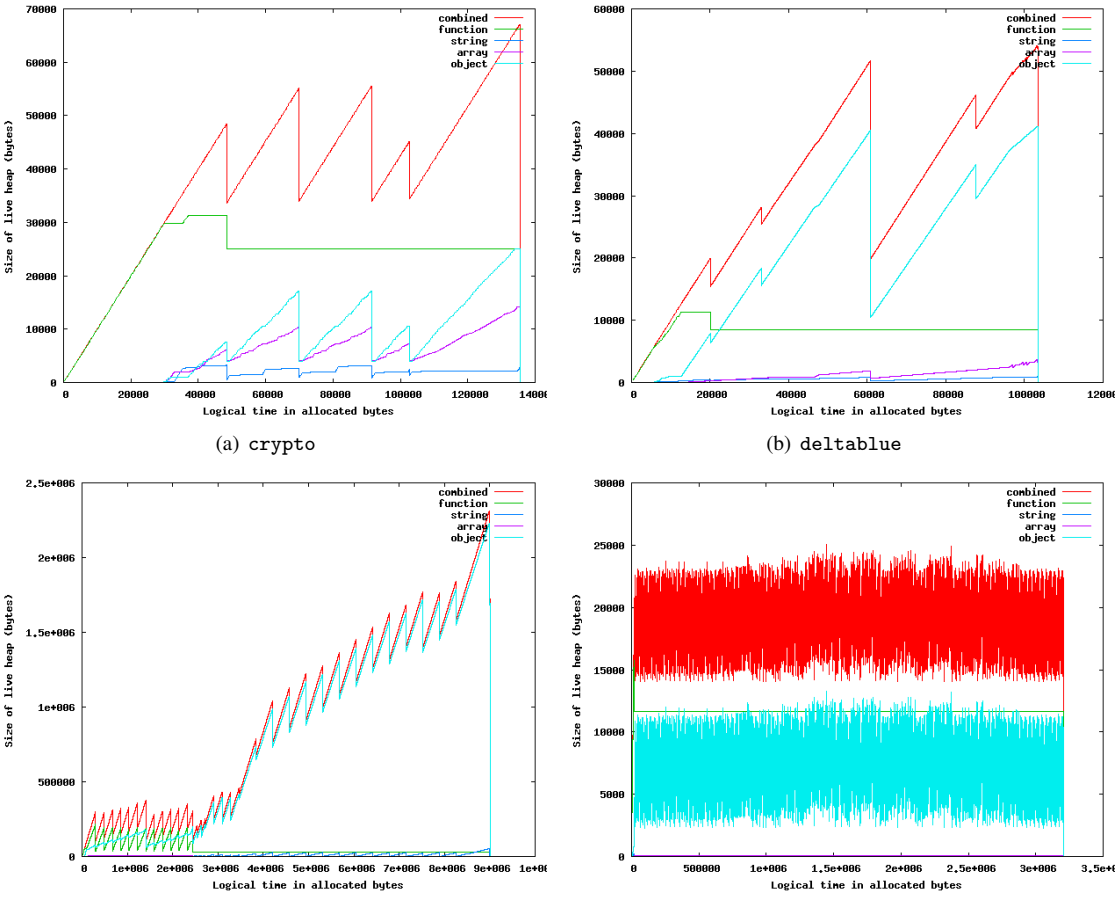


(c) `google` (d) `googlemap`



(e) `hotmail`

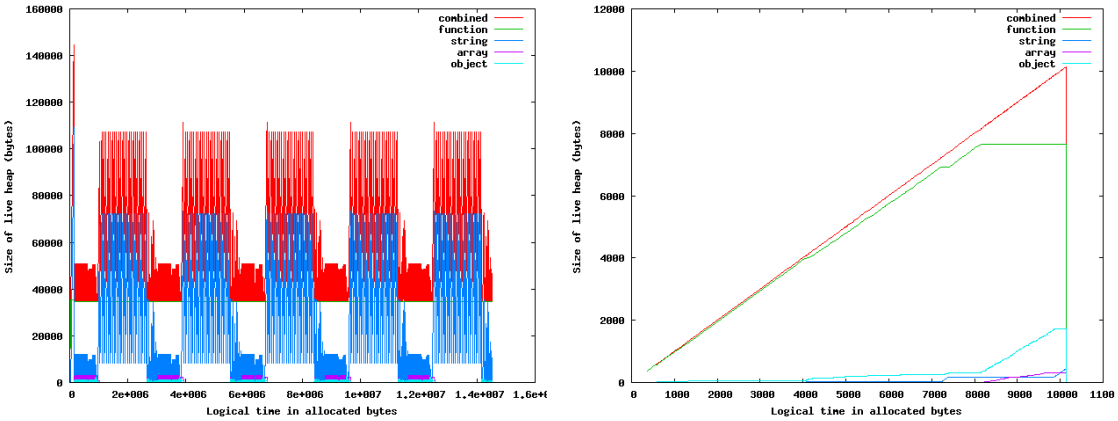
Figure 25: Live heap object composition over time (real sites part 2).



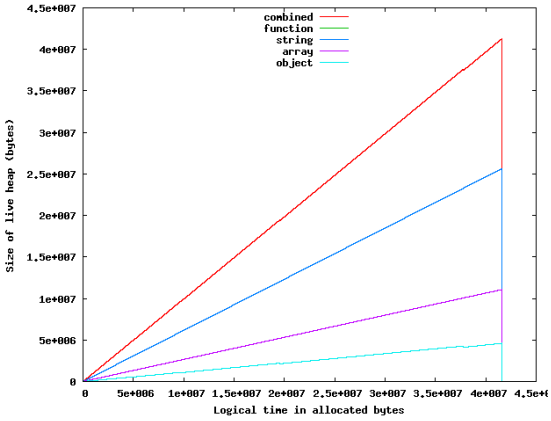
(a) `crypto` (b) `deltablue`

(c) `earley` (d) `raytrace`

Figure 26: Live heap object composition over time (V8 part 1).

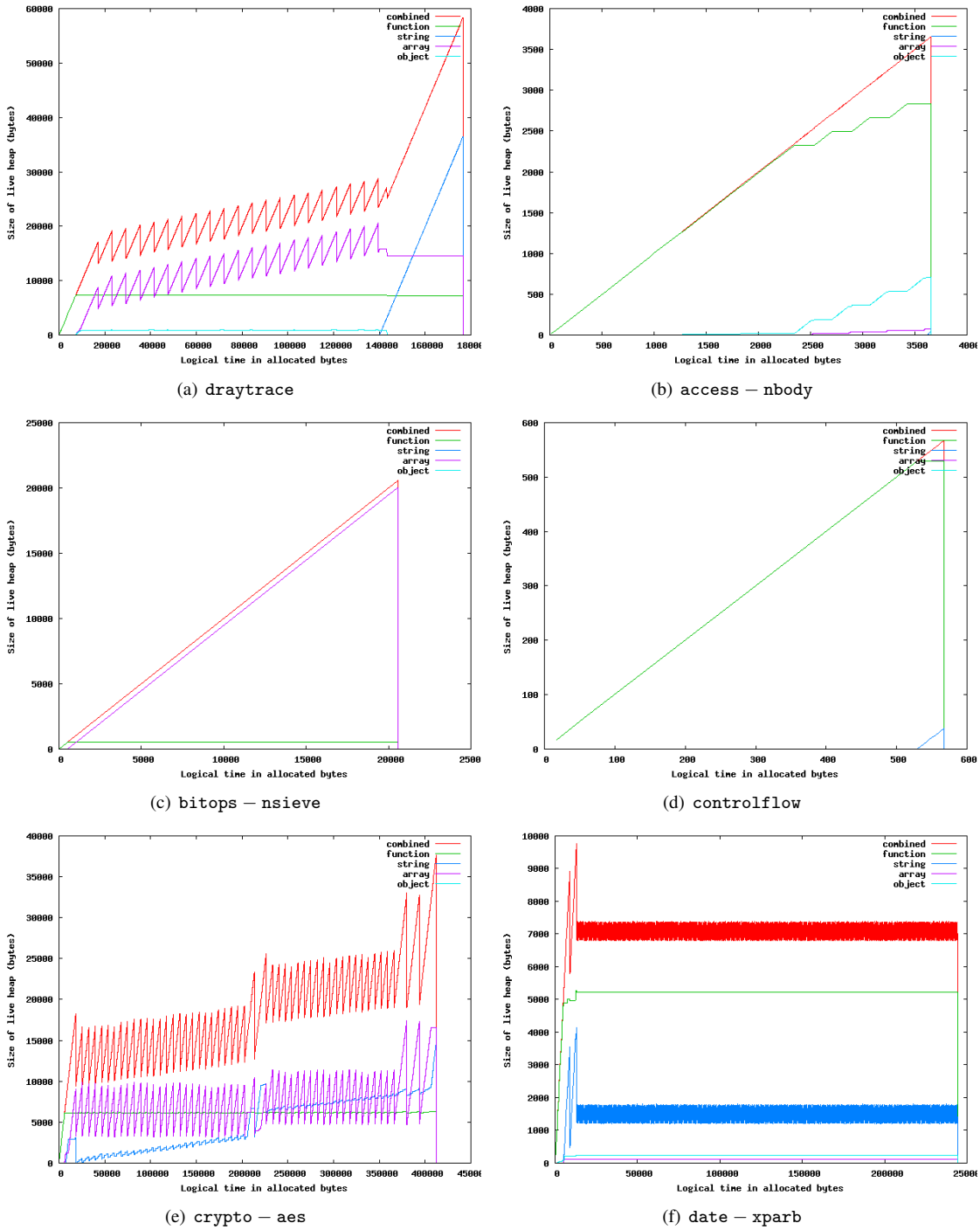


(a) `regexp` (b) `richards`



(c) `splay`

Figure 27: Live heap object composition over time (V8 part 2).

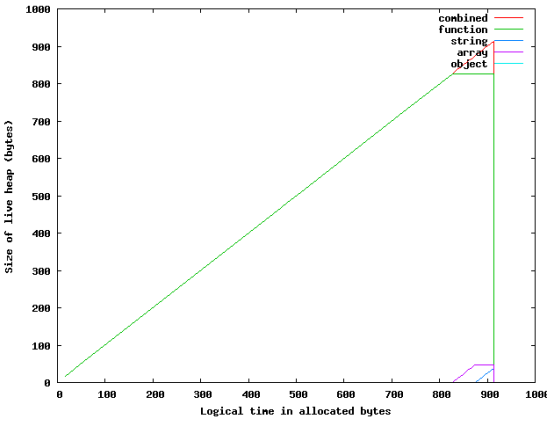


(a) `draytrace` (b) `access — nbody`

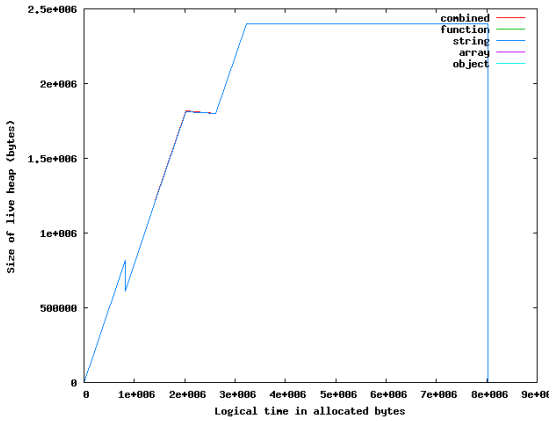
(c) `bitops — nsieve` (d) `controlflow`

(e) `crypto — aes` (f) `date — xparb`

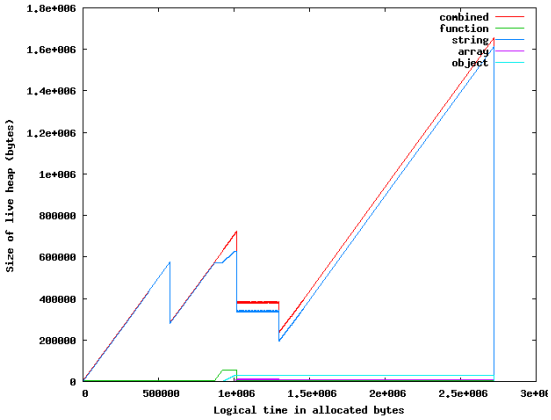
Figure 28: Live heap object composition over time (SunSpider part 1).



(a) `math` — `cordic`

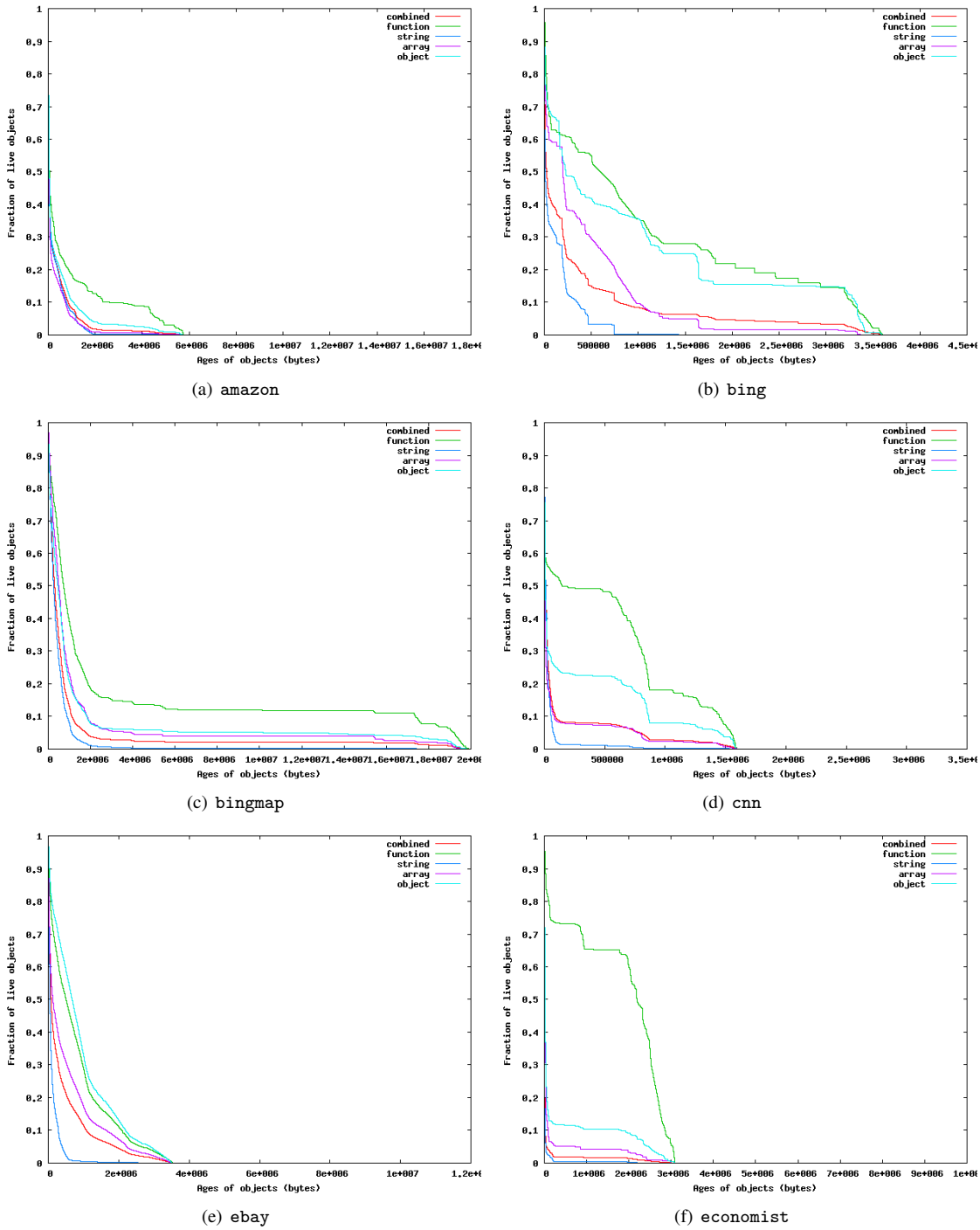


(b) `regexp` — `dna`



(c) `string` — `tagcloud`

Figure 29: Live heap object composition over time (SunSpider part 2).

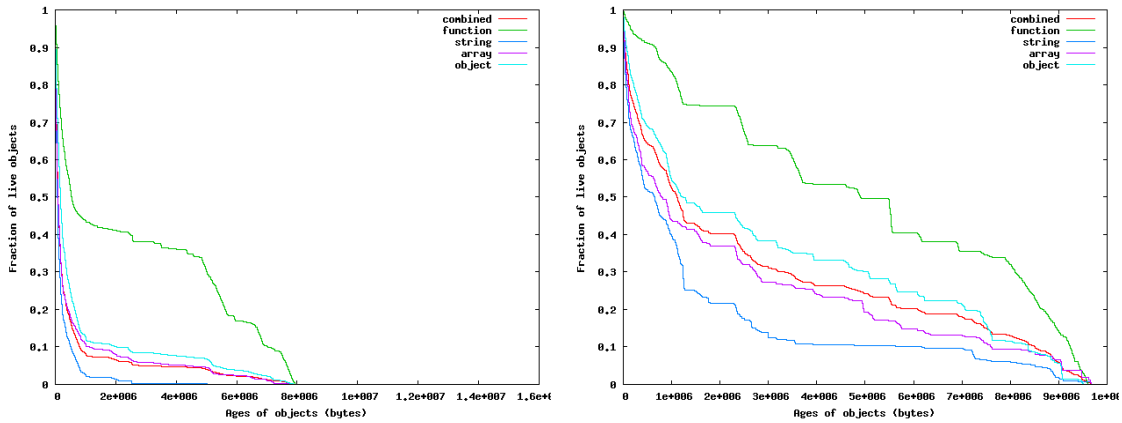


(a) amazon (b) bing

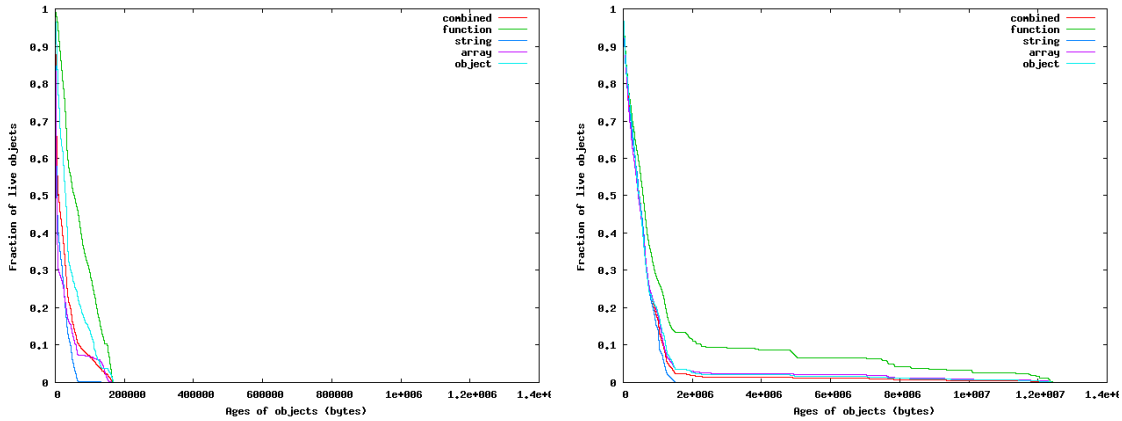
(c) bingmap (d) cnn

(e) ebay (f) economist

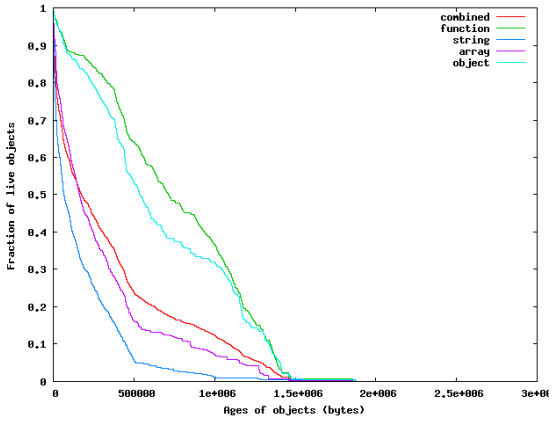
Figure 30: Heap object age (real sites 1).



(a) `facebook` (b) `gmail`

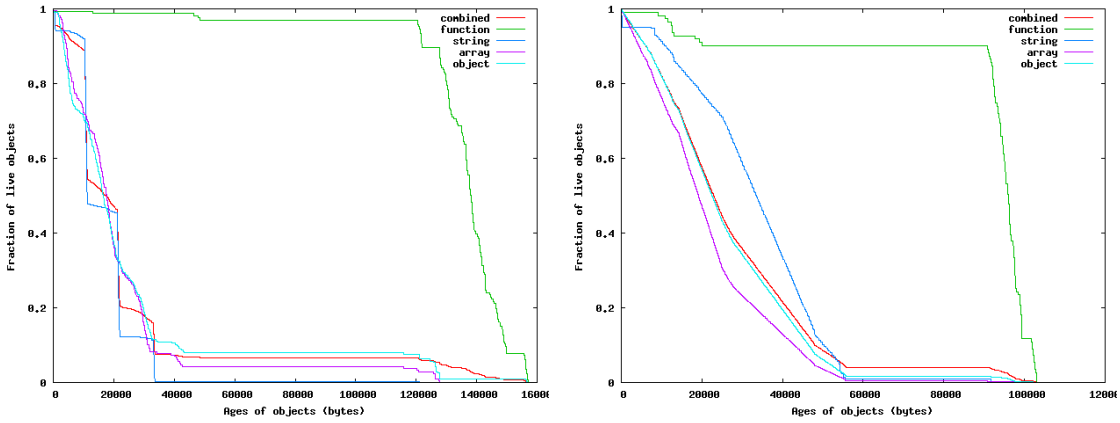


(c) `google` (d) `googlemap`



(e) `hotmail`

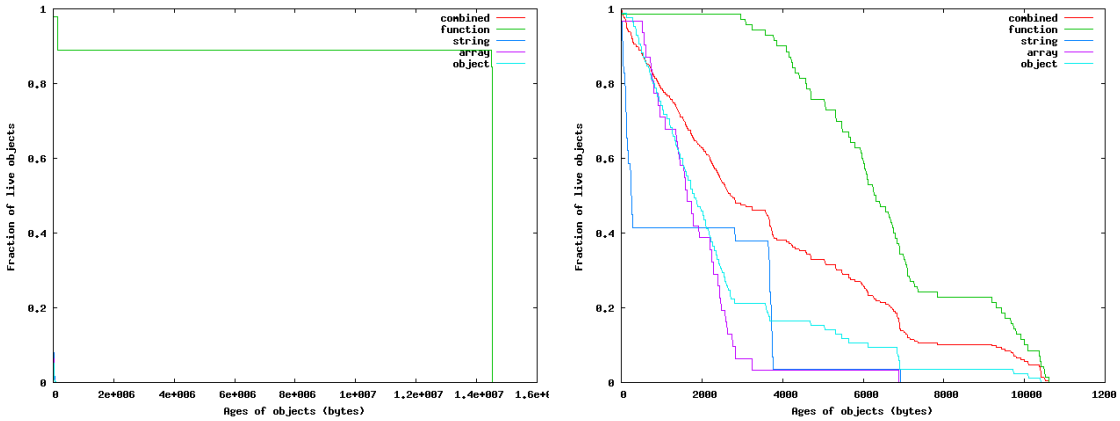
Figure 31: Heap object age (real sites 2).



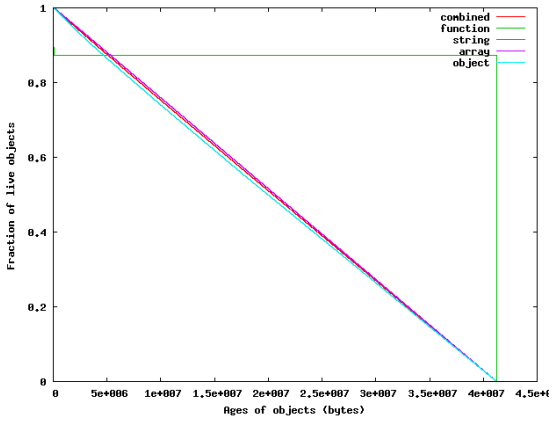
(a) `crypto` (b) `deltablue`

(c) `earley` (d) `raytrace`

Figure 32: Heap object age (V8 part 1).

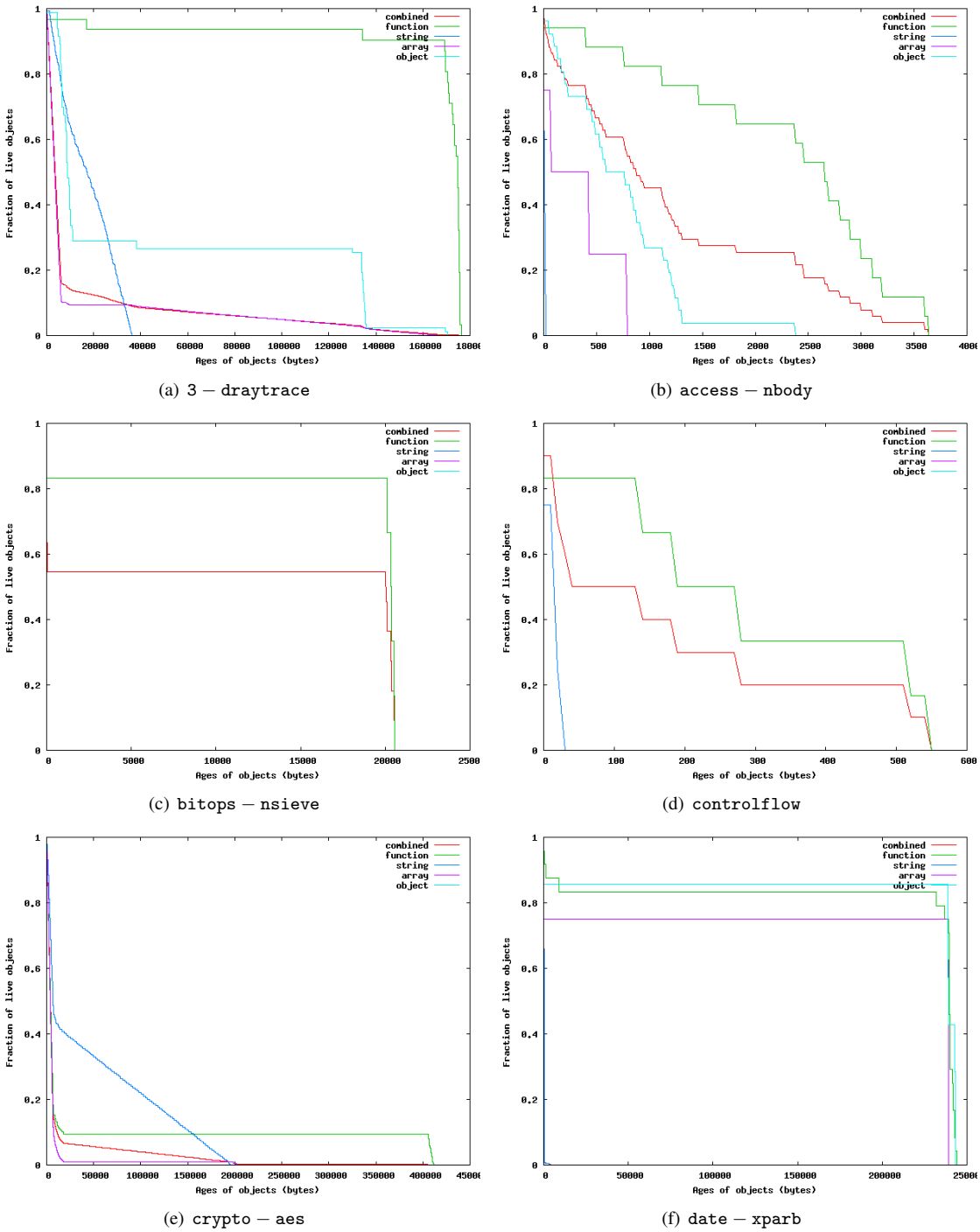


(a) `regexp` (b) `richards`



(c) `splay`

Figure 33: Heap object age (V8 part 2).



(a) 3 — `draytrace`

(b) `access` — `nbody`

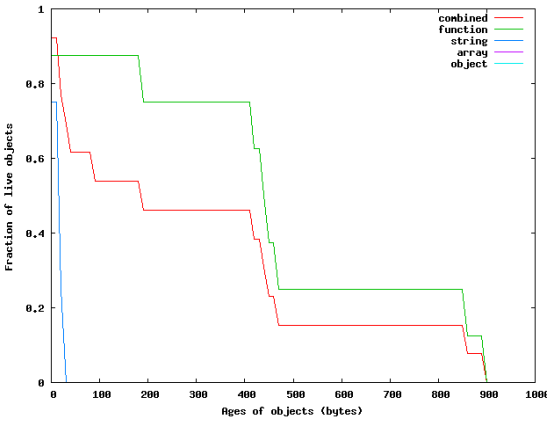
(c) `bitops` — `nsieve`

(d) `controlflow`

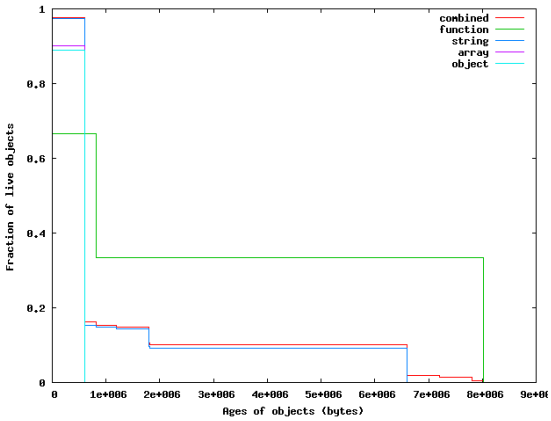
(e) `crypto` — `aes`

(f) `date` — `xparb`

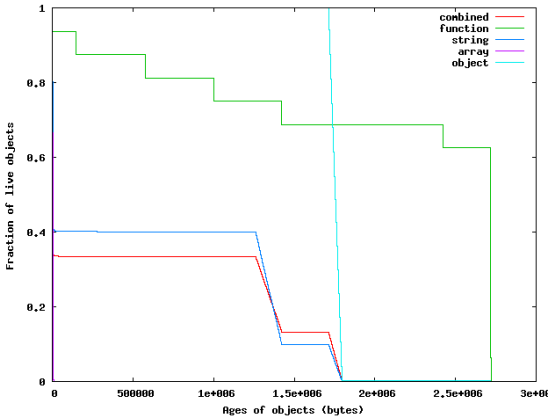
Figure 34: Heap object age (SunSpider part 1).



(a) `math` — `cordic`



(b) `regexp` — `dna`



(c) `string` — `tagcloud`

Figure 35: Heap object age (SunSpider part 2).

	Script Func	Arrays	String	Native Func	Date	Objects	Others	Total
amazon	24,156	34,620	209,876	464	1,570	38,326	571	309,583
bing	11,384	1,876	46,519	150	1,575	2,432	45	63,981
bingmap	46,767	19,659	330,987	89	1,268	39,898	188	438,856
cnn	6,895	2,366	47,239	437	145	2,729	415	60,226
ebay	38,984	20,987	157,359	525	364	46,801	468	265,488
economist	5,811	6,317	349,855	238	148	6,581	1,791	370,741
facebook	25,381	24,047	212,081	472	334	57,777	2,298	322,390
gmail	21,532	16,601	37,180	140	1,067	22,965	79	99,564
google	4,499	3,205	16,841	256	54	1,875	184	26,914
googlemap	37,620	54,619	250,566	78	5,839	148,029	481	497,232
hotmail	9,089	2,166	24,079	353	120	5,101	128	41,036

(a) Real Application Summary

	Script Func	Arrays	String	Native Func	Date	Objects	Others	Total
richards	70	22	29	13	3	64	2	203
deltablue	111	838	342	15	2	2,345	2	3,655
crypto	166	181	1,784	20	9	264	2	2,426
raytrace	110	9	34	20	2	66,667	2	66,844
earley	8,548	312	36,871	31	5	276,023	2	321,792
regexp	45	37,279	139,383	18	2	36,006	2	212,735
splay	47	273,925	821,789	11	8	566,658	2	1,662,440

(b) V8 Application Summary

	Script Func	Arrays	String	Native Func	Date	Objects	Others	Total
3d-raytrace	31	10,665	910	9	3	83	1	11,702
access-nbody	17	4	4	4	2	26	1	58
access-nsieve	6	3	4	3	2	0	0	18
bitops-nsieve	6	1	4	3	2	0	0	16
controlflow	6	0	4	2	2	0	0	14
crypto-aes	190	21,352	3,588	16	3	1	1	25,151
date-xparb	24	4	67,342	17	3	20,008	0	87,398
math-cordic	8	1	4	3	4	0	0	20
regexp-dna	3	10	195	7	2	10	0	227
string-tagcloud	16	10,002	25,293	20	2	2,509	1	37,843

(c) Sunspider Application Summary

Figure 37: Allocated object summary.

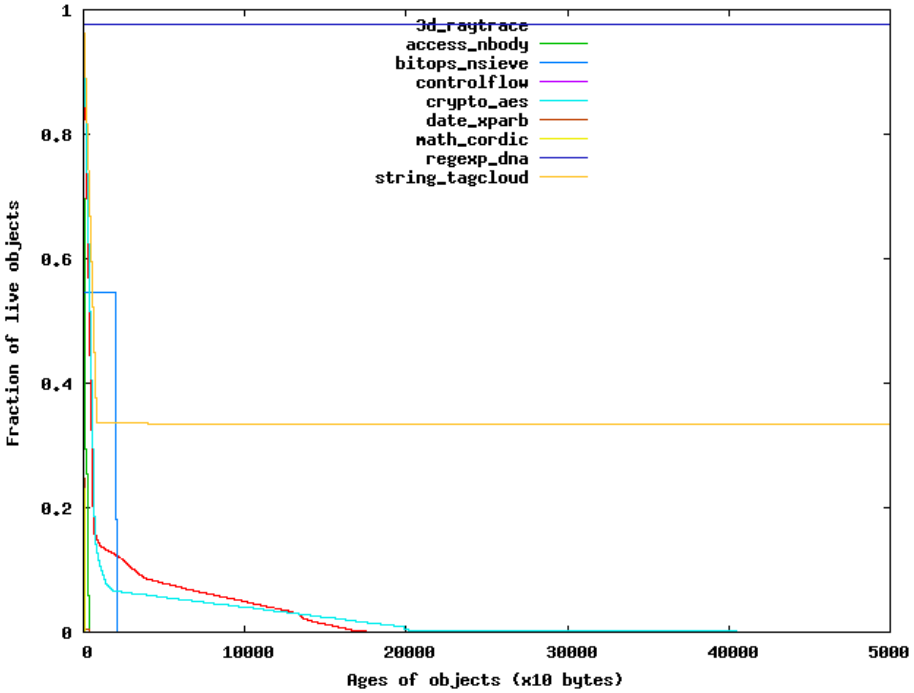


Figure 38: Overall object lifetime distributions in SunSpider.

D SunSpider Object Lifetimes

Figure 38 shows the object lifetime distributions for the SunSpider benchmarks.

References

[1] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, 2000.

[2] A. Bhattacharya and K. S. Sundar. Memory leak patterns in javascript. `http://www.ibm.com/developerworks/web/library/wa-memleak/`, Apr. 2007.

[3] S. M. Blackburn et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceeding of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 169–190, Oct. 2006.

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

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

[6] M. Corporation. Office Web Apps: Store, edit, and share documents online. `http://www.microsoft.com/office/2010/en/office-web-apps/default.aspx`, 2009.

[7] Crisp. String performance in Internet Explorer. `http://therealcrisp.xs4all.nl/blog/2006/12/09/string-performance-in-internet-explorer/`, Dec. 2006.

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

[9] 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.

[10] 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.

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

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

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

[14] 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 *Pro-*

ceedings of the Conference on Programming Language Design and Implementation, pages 465–478, 2009.

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

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

[17] Google Web toolkit. `http://code.google.com/webtoolkit`.

[18] M. Hertz and E. D. Berger. Quantifying the performance of garbage collection vs. explicit memory management. In *Proceedings of European Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 313–326, 2005.

[19] M. Hertz, S. M. Blackburn, J. E. B. Moss, K. McKinley, and D. Stefanović. Generating object lifetime traces with Merlin. *ACM Transactions on Programming Languages and Systems*, 28(3):476–516, May 2006.

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

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

[22] R. Kassim, N. B. Abu, and K. H. Awang. Application performance benchmark: An experimental analysis on C# programs. *Information Technology, 2008. ITSim 2008*, 1:1–5, Aug. 2008.

[23] 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.

[24] E. Kiciman and B. Livshits. AjaxScope: a platform for remotely monitoring the client-side behavior of Web 2.0 applications. In *Proceedings of Symposium on Operating Systems Principles*, pages 17–30, 2007.

[25] 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. ACM, 2000.

[26] C. Krintz. A collection of Phoenix-compatible C# benchmarks. `http://www.cs.ucsb.edu/~ckrintz/racelab/PhxCSBenchmarks/`.

[27] S. Lebresne, G. Richards, J. Östlund, T. Wrigstad, and J. Vitek. Understanding the dynamics of JavaScript. In *Proceedings for the Workshop on Script to Program Evolution*, pages 30–33, 2009.

[28] B. Livshits and E. Kiciman. Doloto: code splitting for network-bound web 2.0 applications. In M. J. Harrold and G. C. Murphy, editors, *Proceedings of the International Symposium on Foundations of Software Engineering*, pages 350–360. ACM, 2008.

[29] J. A. Mathew, P. D. Coddington, and K. A. Hawick. Analysis and development of Java Grande benchmarks. In *Proceedings of the ACM Conference on Java Grande*, pages 72–80, 1999.

[30] Microsoft Corporation. Microsoft Live Labs Volta. `http://labs.live.com/volta/`, 2007.

[31] J. Nielson, C. Williamson, and M. Arlitt. Benchmarking modern browsers. In *Proceedings IEEE Workshop on Hot Topics in Web Systems and Technologies*. IEEE, 2008.

[32] R. Paul. Firefox to get massive JavaScript performance boost. `http://arstechnica.com/open-source/news/2008/08/firefox-to-get-massive-javascript-\performance-boost.ars`, Aug. 2008.

[33] 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.

[34] 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.

[35] 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.

[36] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the IBM Java Just-in-Time Compiler. *IBM Systems Journal*, 39(1):175–193, 2000.

[37] T. Suganuma, T. Yasue, and T. Nakatani. An empirical study of method in-lining for a java just-in-time compiler. In *Proceedings of the Java Virtual Machine Research and Technology Symposium*, pages 91–104. USENIX, 2002.

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

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

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

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