ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/220752388

# An Analysis of the Dynamic Behavior of JavaScript Programs

**Conference Paper** *in* ACM SIGPLAN Notices · May 2010

DOI: 10.1145/1806596.1806598 · Source: DBLP

CITATIONS
251

READS
597

**4 authors**, including:

Jan Vitek
Northeastern University
**289** PUBLICATIONS **5,713** CITATIONS

SEE PROFILE

**Some of the authors of this publication are also working on these related projects:**



Data-Centric Concurrency Control (Atomic-Set Serializability) View project



On the Design, Implementation, and Use of Laziness in R View project

All content following this page was uploaded by Jan Vitek on 03 June 2014.

The user has requested enhancement of the downloaded file.

# An Analysis of the Dynamic Behavior of JavaScript Programs

Gregor Richards Sylvain Lebresne Brian Burg Jan Vitek
S3 Lab, Department of Computer Science, Purdue University, West Lafayette, IN
{gkrichar,slebresn,bburg,jv}@cs.purdue.edu

## Abstract

The JavaScript programming language is widely used for web programming and, increasingly, for general purpose computing. As such, improving the correctness, security and performance of JavaScript applications has been the driving force for research in type systems, static analysis and compiler techniques for this language. Many of these techniques aim to reign in some of the most dynamic features of the language, yet little seems to be known about how programmers actually utilize the language or these features. In this paper we perform an empirical study of the dynamic behavior of a corpus of widely-used JavaScript programs, and analyze how and why the dynamic features are used. We report on the degree of dynamism that is exhibited by these JavaScript programs and compare that with assumptions commonly made in the literature and accepted industry benchmark suites.

***Categories and Subject Descriptors*** D.2.8 [*Software Engineering*]: Metrics; D.3.3 [*Programming Languages*]: Language Constructs and Features

***General Terms*** Experimentation, Languages, Measurement

***Keywords*** Dynamic Behavior, Execution Tracing, Dynamic Metrics, Program Analysis, JavaScript

## 1. Introduction

JavaScript[1] is an object-oriented language designed in 1995 by Brendan Eich at Netscape to allow non-programmers to extend web sites with client-side executable code. Unlike more traditional languages such as Java, C# or even Smalltalk, it does not have classes, and does not encourage encapsulation or even structured programming. Instead JavaScript strives to maximize flexibility. JavaScript's success is undeniable. As a data point, it is used by 97 out of the web's 100 most popular sites.[2] The language is also becoming a general purpose computing platform with office applications, browsers and development environments [15] being developed in JavaScript. It has been dubbed the "assembly language" of the Internet and is targeted by code generators from the likes of Java[2,3] and Scheme [20]. In response to this success, JavaScript has started to garner academic attention and respect. Researchers have focused on three main problems: security, correctness and performance. Security is arguably JavaScript's most pressing problem: a number of attacks have been discovered that exploit the language's dynamism (mostly the ability to access and modify shared objects and to inject code via `eval`). Researchers have proposed approaches that marry static analysis and runtime monitoring to prevent a subset of known attacks [6, 12, 21, 27, 26]. Another strand of research has tried to investigate how to provide better tools for developers for catching errors early. Being a weakly typed language with no type declarations and only run-time checking of calls and field accesses, it is natural to try to provide a static type system for JavaScript [2, 1, 3, 24, 13]. Finally, after many years of neglect, modern implementations of JavaScript have started to appear which use state of the art just-in-time compilation techniques [10].

In comparison to other mainstream object-oriented languages, JavaScript stakes a rather extreme position in the spectrum of dynamicity. Everything can be modified, from the fields and methods of an object to its parents. This presents a challenge to static analysis techniques, to type systems, and to compiler optimizations, all of which leverage the parts of a program that are fixed to make guarantees about that program's run-time behavior. If nothing is fixed then there is very little that traditional techniques can do. To cope with this, researchers have considered subsets of JavaScript based on "reasonable" (or sane) assumptions about common behavioral patterns of programs. Similarly, JavaScript implementations are often compared using benchmarks which were ported from other languages and are relatively simple. What if these assumptions were wrong and the benchmarks were not representative of actual workloads? Then it could well be the case that type systems and static analysis techniques developed for JavaScript have little or no applicability to real-world programs, and that compiler writers would be steered towards implementing optimizations that make unrealistic benchmark code run fast but have little effect in the real world.

This paper sets out to characterize JavaScript program behavior by analyzing execution traces recorded from a large corpus of real-world programs. To obtain those traces we have instrumented a popular web browser and interacted with 103 web sites. For each site multiple traces were recorded. These traces were then analyzed to produce behavioral data about the programs. Source code captured when the programs were loaded was analyzed to yield static metrics. In addition to web sites, we analyzed three widely used benchmark suites as well as several applications. We report both on traditional program metrics as well as metrics that are more indicative of the degree of dynamism exhibited by JavaScript programs in the wild.

[1] The language is standardized as ECMAScript [16]. Its various dialects (and their extensions) are referred to by names such as JScript, ActionScript, and JavaScript (officially, Mozilla's implementation of ECMAScript). In this paper, we refer to all implementations collectively as JavaScript.

[2] `http://www.alexa.com`
[2] `http://code.google.com/closure/compiler/`
[3] `http://code.google.com/webtoolkit/`

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

*PLDI'10,* June 5–10, 2010, Toronto, Ontario, Canada.
Copyright © 2010 ACM 978-1-4503-0019/10/06. . . $10.00

## 2. Motivation and Related Work

The original impetus for our interest in JavaScript was to assess the feasibility of a static type system able to successfully and meaningfully type check existing JavaScript programs. Other dynamic languages such as Scheme have had recent success introducing gradual typing systems [25], but they have stopped short of type checking object-oriented extensions (implemented by macros in Scheme). For JavaScript, Anderson *et al.* proposed a type system with *definite* and *potential* types [2, 1, 3], while Heidegger and Thiemann following up on some of their earlier work [24, 18] propose *recency* types in [13], and Furr *et al.* proposed a related system for DRuby [9]. While all of these type systems acknowledge some minor simplifications to the target language, they rely on fairly similar assumptions. For instance, Thiemann writes: "Usually, no further properties are defined after the initialization and the type of the properties rarely changes." This suggests that object types are stable at run-time and can be described using, e.g., traditional row-types. In fact all the proposals take the view that an object's type should be the sum of all possible fields and methods that it could contain, with some of them being undefined; they differ mostly on how to perform strong updates to avoid polluting all properties with undefined values. Interestingly, language implementors make similar assumptions. For instance, Google's V8 JavaScript engine is reported to optimistically associate "classes" to objects on the assumption that their shape will not change too much, though with a fallback case for highly dynamic objects [3]. This design is similar to implementations of one of JavaScript's influences, Self [5], and is expected to work for the same reasons. As the above mentioned hypothesis is crucial for the applicability and usefulness of the results, it deserves careful study. In fact, we have found a number of similar assumptions in the literature which we list below. We first review the salient features of the language to provide sufficient background for readers unfamiliar with JavaScript.

***JavaScript in a Nutshell.*** JavaScript is an imperative, object-oriented language with Java-like syntax, but unlike Java it employs a prototype-based object system. An object is a set of properties, a mutable map from strings to values. A property that evaluates to a closure and is called using the context of its parent object plays the role of a method in Java. Each object has prototype field which refers to another object. Property lookup involves searching the current object, then its parent, and its parent until the property is found. The JavaScript object system is extremely flexible. As a result, it is difficult to constrain the behavior of any given object. For example, it is possible to modify the contents of any prototype at any time or to replace a prototype field altogether. In JavaScript, any function can be a constructor for a "class" of objects, and contains a `prototype` field, initially referencing an empty object. The **`new`** keyword creates an object based on the `prototype` field and using the function as a constructor. The semantics of **`new`** is simple, but unusual: first, an empty object is created, with its parent set to the object referenced by the `prototype` field of the constructor function; second, the constructor is called, with **`this`** bound to the new object. The object referenced by the keyword **`this`** is not determined by lexical scoping, but instead by the caller; finally the return value from the constructor (if any) is discarded, and the **`new`** expression evaluates to **`this`**.

***Common Assumptions about the dynamic behavior of JavaScript.*** We proceed to enumerate the explicit and implicit assumptions that are commonly found in the literature and in implementations.

1. **The prototype hierarchy is invariant.** The assumption that the prototype hierarchy does not change after an object is created is so central to the type system work that [2, 3] chose to not even model prototypes. Research on static analysis typically does not mention prototype updates [6, 12, 24, 17]. Yet, any modification to the prototype hierarchy can potentially impact the control flow graph of the application just as well as the types of affected objects.
2. **Properties are added at object initialization.** Folklore holds that there is something akin to an "initialization phase" in dynamic languages where most of the dynamic activity occurs and after which the application is mostly static [14]. For JavaScript this is embodied by the assumption that most changes to the fields and methods of objects occur at initialization, and thus that it is reasonable to assign an almost complete type to objects at creation, leaving a small number of properties as *potential* [3, 24, 13, 2].
3. **Properties are rarely deleted.** Removal of methods or fields is difficult to accommodate in a type system as it permits non-monotonic evolution of types that breaks subtyping guarantees usually enforced in modern typed languages. If deletion is an exceptional occurrence (and one that can be predicted), one could use potential types for properties that may be deleted in the future. But, this would reduce the benefits of having a type system in the first place, which is probably why related work chooses to forbid it [3, 24, 18, 2]. Static analysis approaches are usually a bit more tolerant to imprecision and can handle deletes, but we have not found any explanation of its handling in existing data flow analysis techniques ([12, 6, 17]).
4. **The use of `eval` is infrequent and does not affect semantics.** The use of `eval` on arbitrary strings has the potential of invalidating any results obtained by static analysis or static type checking. Thus many works simply ignore it [3, 17, 24, 2], while other assume that uses are either trivial or related to de-serialization using the JSON protocol [12, 18].
5. **Declared function signatures are indicative of types.** Type systems for JavaScript typically assume that the declared arity of a function is representative of the way it will be invoked [3, 24, 2]. This is not necessarily the case because JavaScript allows calls with different arities.
6. **Program size is modest.** Some papers justify very expensive analyses with the explicit assumption that handwritten JavaScript programs are small [18], and others implicitly rely on this as they present analyses which would not scale to large systems [17, 12].
7. **Call-site dynamism is low.** Some JavaScript implementations such as Google V8 rely on well-known implementation techniques to optimize JavaScript programs such as creating classes (in the Java sense) for objects and inline caches. These techniques will lead to good performance only if the behavior of JavaScript is broadly similar to that of other object-oriented languages.
8. **Execution time is dominated by hot loops.** Trace-based Just-in-time compilers such as TraceMonkey [10] rely on the traditional assumption that execution time is dominated by small loops.
9. **Industry benchmarks are representative of JavaScript workloads.** Standard benchmark suites such as SunSpider, Dromaeo and V8, are used to tune and compare JavaScript implementations and to evaluate the accuracy of static analysis techniques [18]. But conclusions obtained from use of those benchmarks are only meaningful if they accurately represent the range of JavaScript workloads in the wild.

[3] As reported in a presentation by Kevin Millikin at Google Developer Day 2008.

The goal of this paper is to provide supporting evidence to either confirm or invalidate these assumptions. We are not disputing the validity of previous research, as even if a couple of the above assumptions proved to be unfounded, previous work can still serve as a useful starting point for handling full JavaScript. But we do want to highlight limitations to widespread adoption of existing techniques and point to challenges that should be addressed in future research.

***Related Work.*** Until now, to the best of our knowledge, there has been no study of the dynamic behavior of JavaScript programs of comparable depth or breadth. Ratanaworabhan *et al.* have performed a similar study concurrently to our own, and its results are similar to ours [22]. There have been studies of JavaScript's dynamic behavior as it applies to security [28] [8], but the behaviors studied were restricted to those particularly relevant to security. We conducted a small scale study of JavaScript and reported preliminary results in [19], and those results are consistent with the new results presented here. Holkner and Harland [14] have conducted a study of the use of dynamic features (addition and deletion of fields and methods) in the Python programming language. Their study focused on a smaller set of programs and concluded that there is a clear phase distinction. In their corpus dynamic features occur mostly in the initialization phase of programs and less so during the main computation. Our results suggest that JavaScript is more dynamic than Python in practice. There are many studies of the runtime use of selected features of object-oriented languages. For example, Garret *et al.* reported on the dynamism of message sends in Self [11], Calder *et al.* characterized the difference of between C and C++ programs in [4], and Temporo *et al.* studied the usage of inheritance in Java in [23]. These previous papers study in great detail one particular aspect of each language. In this particular work, we strive for an overview of JavaScript, and leave detailed analysis for future work. Finally, we were inspired by the work of Dufour *et al.* [7] and their rigorous framework for discussing runtime metrics for Java.

## 3. Tracing and Analysis Infrastructure

The tracing infrastructure developed for this paper is based on an instrumented version of the WebKit [4] web browser engine integrated into Apple's Safari browser. While there are standalone interpreters available, they would not be able to deal with the mixture of DOM and AJAX that is commonplace in most JavaScript-enabled sites. For flexibility, analysis is performed offline. Our instrumented browser records a trace containing most operations performed by the interpreter (reads, writes, deletes, calls, defines, etc.) as well as events for garbage collection and source file loads. Invocations to `eval` trigger an event similar to the one for source file loads, and the evaluated string is saved and traced like any other part of the program's execution. Complete traces are compressed and stored to disk. While it does have some performance overhead, our instrumentation does not cause a noticeable slowdown in interactive applications, and none of our users complained about performance. Traces are analyzed offline and the results are stored in a database which is then mined for data. The offline trace analysis component is essentially an abstract interpreter for the event stream. It is able to replay any trace creating an abstract representation of the heap state of the corresponding JavaScript program. The trace analyzer maintains rich and customizable historical information about the program's behavior, such as access histories of each object, call sites and allocation sites, and so on. Finally, several static analyses (`eval` classification, code size metrics) are performed on the recovered source files using the parsing framework from the Rhino JavaScript compiler.[5]

As WebKit does not hide its identity to JavaScript code, it is possible for code to exhibit behavior peculiar to WebKit. Techniques like this are often used to work around bugs in JavaScript implementations or browsers. For instance, the Prototype JavaScript library includes the following check for WebKit.

```
WebKit: ua.indexOf('AppleWebKit/') > -1,
```

It then uses that check to create different implementations of `setOpacity`, `getRootElement`, `shouldUseXPath` and other functions which may exhibit browser-dependent behavior. Although this does introduce a possible bias which is very difficult to detect, all other JavaScript implementations are equally detectable and so create comparable bias. We would be interested in comparable studies using other engines, to determine whether the results differ in significant ways.

## 4. Corpus and Methodology

We have selected 100 web sites based on the Alexa list of most popular sites on the Internet, along with a number of sites of particular interest (including 280slides, Lively Kernel, and a medley of different web sites visited in a single session). Moreover we also recorded traces for the three main industry benchmark suites (SunSpider, Dromaeo, and V8). For each of these sites we asked several of our colleagues to interact with the site in a "meaningful" manner. Each interaction with a different web site was saved in a different trace. Multiple traces for the same site are averaged in our metrics.

In the remainder of this paper we focus on the results of 17 sites that we believe to be representative of the full range of behaviors and usage of popular libraries. The list of sites we have retained is shown in Figure 1. Data for all the web sites, as well as our tracing and analysis framework, database, and graphs are available on the project web site[6]. For each site, we also list publicly-available JavaScript libraries utilized by the site, if any. Sites that use the same libraries tend to have similar coding styles and program structure. It is instructive to see whether similarities also exist in the dynamic behavior of these programs, regardless of different application logic and use cases.

| Alias | Library | URL |
|---|---|---|
| 280S | Objective-J[1] | `280slides.com` |
| BING | | `bing.com` |
| BLOG | | `blogger.com` |
| DIGG | jQuery[2] | `digg.com` |
| EBAY | | `ebay.com` |
| FBOK | | `facebook.com` |
| FLKR | | `flickr.com` |
| GMAP | Closure[3] | `maps.google.com` |
| GMIL | Closure | `gmail.com` |
| GOGL | Closure | `google.com` |
| ISHK | Prototype[4] | `imageshack.us` |
| LIVE | | `research.sun.com/projects/lively` |
| MECM | SproutCore[5] | `me.com` |
| TWIT | jQuery | `twitter.com` |
| WIKI | | `wikipedia.com` |
| WORD | jQuery | `wordpress.com` |
| YTUB | | `youtube.com` |
| ALL | | Average over 103 sites |

**Figure 1.** Selected JavaScript-enabled web sites.
[1] `cappuccino.org` [2] `jquery.com` [3] `code.google.com/closure`
[4] `prototypejs.org` [5] `sproutcore.com`

[4] `webkit.org`.

[5] `www.mozilla.org/rhino`.

[6] `http://www.cs.purdue.edu/homes/gkrichar/js`

## 5. General Program Metrics

We start with general program metrics that can easily be related to less-dynamic languages.

### 5.1 Corpus Size

Figure 2 gives the size of the programs loaded by the JavaScript interpreter (including strings passed to `eval`) in bytes. They range from small, with 280S at only 116 KB, to quite large with FBOK at 14 MB, and include all libraries required by the applications. These numbers include the results of multiple page loads of the same code, as from the interpreter's stand point there is no guarantee that requesting the same file twice will return the same result. The second column gives the unique lines of code. Notice that code size remains large, topping at 1.7MB for GMIL. The third column gives the length of traces in events. This is not directly correlated to the computation time: since most sites using JavaScript are interactive in nature, there is unfortunately no meaningful notion of wall-clock time that we can use to gauge computational effort. The number of recorded events vary from thousands to millions. Not surprisingly, LIVE, which is a programming environment written in JavaScript, generates the biggest traces, followed by MECM and GMAP, two highly-interactive and data-intensive web sites. The fourth column of the figure gives the number of functions statically occurring in the loaded code including functions added by `eval` expressions.

A few dynamic behaviors are expected to be similar in most languages. For example, the 90/10 rule holds, i.e. 90% of execution time is spent in 10% of functions or less. The column labeled Hot gives the number of functions accounting for 90% of execution. This metric is obtained by counting the number of trace events recorded in each function, sorting the functions by size and counting the functions that comprise 90% of the events. ISHK and WORD spend 90% of their time in 1% or less of the program's functions. All other programs range between 6% and 15% hot with a median of 8%. The numbers are noticeably higher than the hot percentages for Java programs reported in [7] where the average is 3%. From the point of view of an optimizing compiler, smaller numbers are better. We also report the proportion of live functions, functions that have been executed at least once. This number ranges between 13% (FLKR) and 61% (GMAP). Due in part to the popularity of large framework-like JavaScript libraries, in each site no more than 61% of the defined functions are live, with a median of 35% live. The proportion of live to loaded code is slightly higher than that observed for Java programs [7], but this is not surprising when one considers the typical size of JavaScript versus Java libraries and programs.

| Site | Source size | Unique source size | Trace size | Func. count | Hot | Live |
|---|---|---|---|---|---|---|
| 280S | 116 KB | 81KB | 11,931 K | 4,293 | 6.8% | 44% |
| BING | 815 KB | 186KB | 1,199 K | 2,457 | 6.4% | 46% |
| BLOG | 1,347 KB | 775KB | 91 K | 5,087 | 11.5% | 16% |
| DIGG | 1,106 KB | 759KB | 1,734 K | 2,957 | 8.7% | 39% |
| EBAY | 3,156 KB | 1,034KB | 2,239 K | 10,791 | 11.7% | 31% |
| FBOK | 14,904 KB | 1,604KB | 5,309 K | 43,469 | 5.8% | 19% |
| FLKR | 8,862 KB | 246KB | 490 K | 19,149 | 14.0% | 13% |
| GMAP | 1,736 KB | 833KB | 13,125 K | 5,146 | 7.8% | 61% |
| GMIL | 2,084 KB | 1,719KB | 6,047 K | 10,761 | 7.6% | 38% |
| GOGL | 2,376 KB | 839KB | 1,815 K | 10,250 | 15.0% | 28% |
| ISHK | 915 KB | 420KB | 5,376 K | 2,862 | 0.6% | 35% |
| LIVE | 1,081 KB | 938KB | 48,324 K | 2,936 | 7.4% | 49% |
| MECM | 4,615 KB | 646KB | 14,084 K | 14,401 | 6.6% | 24% |
| TWIT | 837 KB | 160KB | 2,252 K | 2,967 | 9.2% | 45% |
| WIKI | 1,009 KB | 115KB | 53 K | 1,226 | 14.6% | 24% |
| WORD | 1,386 KB | 235KB | 6,403 K | 3,118 | 1.0% | 42% |
| YTUB | 2,897 KB | 562KB | 541 K | 11,321 | 13.0% | 22% |
| ALL | 2,544 KB | 790KB | 4,151 K | 10,625 | 2.2% | 26% |

**Figure 2. Program sizes.** "Source size" is the total amount of source seen by the interpreter, including source loaded more than once and `eval`s. "Unique source size" excludes multiple loads of the same source, but still includes `eval`.

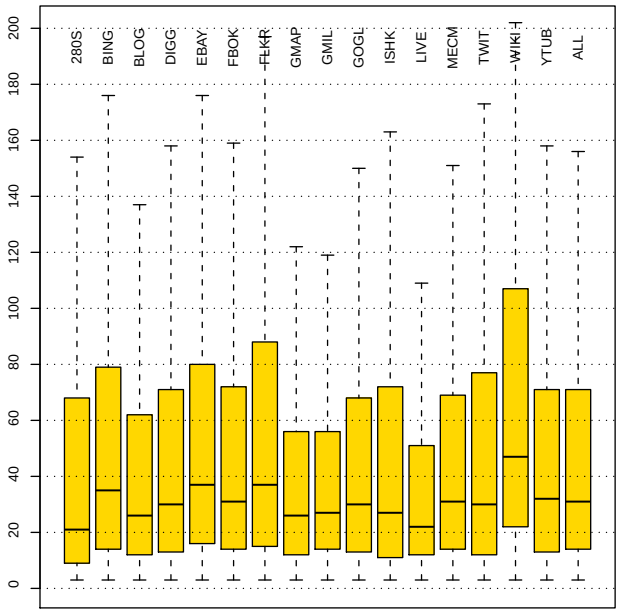


**Figure 3. Static function size.** The per-site quartiles and median static function size, measured by the number of AST nodes generated from parsing the function.

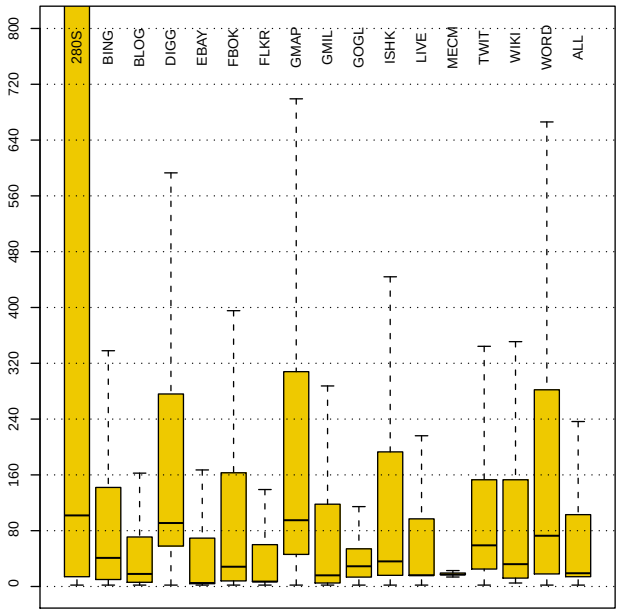


**Figure 4. Dynamic function size.** The per-site quartiles and median function size, measured in the number of trace events.

Figure 3 gives the average size of the code functions occurring in the JavaScript program source. These seem fairly consistent across sites. More interestingly, Figure 4 shows the number of events per function, which roughly corresponds to the number of bytecodes evaluated by the interpreter (note that some low-level bytecodes such as branches and arithmetic are not recorded in the trace). It is interesting to note that the median is fairly high, around 20 events. This suggests that, in contrast to Java, there are fewer short methods (e.g. accessors) in JavaScript and thus possibly fewer opportunities to benefit from inlining optimizations.

### 5.2 Instruction Mix

The instruction mix of JavaScript program is also fairly traditional: more read operations are expected than write operations. As shown in Figure 5, reads are far more common than writes: over all traces the proportion of reads to writes is 6 to 1. Deletes comprise only .1% of all events. That graph further breaks reads, writes and deletes into various specific types; prop refers to accesses

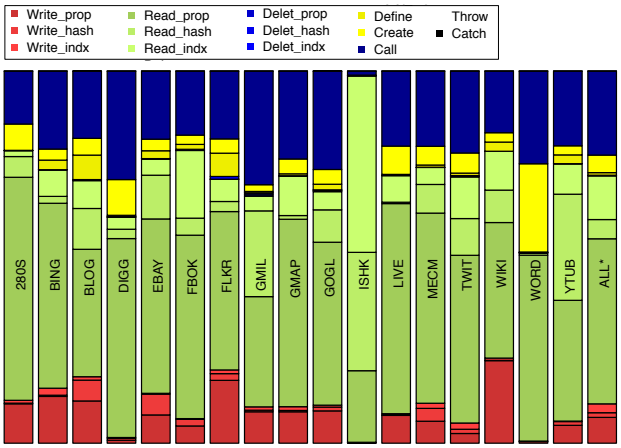


**Figure 5. Instruction mix.** The per-site proportion of read, write, delete, call instructions (averaged over multiple traces).

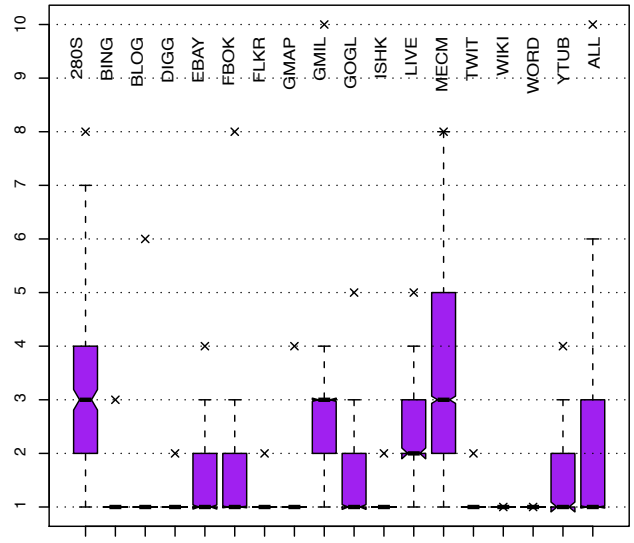


**Figure 6. Prototype chain length.** The per-site quartile and maximum prototype chain lengths.

using dot notation (e.g. `x.f`), hash refers to access using indexing notation (e.g. `x[s]`), indx refers to accesses using indexing notation with a numeric argument. The overall number of calls is high, 20%, as the interpreter does not perform any inlining. Exception handling is rather infrequent with a grand total of 1,328 throws over 478 million trace events. There are some outliers such as ISHK, WORD and DIGG where updates are a much smaller proportion of operations (and influenced by the sheer number of objects in these sites), but otherwise the traces are consistent.

### 5.3 Prototype Chains

One higher-level metric is the length of an object's prototype chain, which is the number of prototype objects that may potentially be traversed in order to find an object's inherited property. This is roughly comparable to metrics of the depth of class hierarchies in class-based languages, such as the Depth of Inheritance (DIT) metric discussed in [23]. Studies of C++ programs mention a maximum DIT of 8 and a median of 1, whereas Smalltalk has a median of 3 and maximum of 10. Figure 6 shows that in all but four sites, the median prototype chain length is 1. Note that we start our graph at chain length 1, the minimum. All objects except `Object.prototype` have at least one prototype, which if unspecified, defaults to the `Object.prototype`. The maximum observed prototype chain length is 10. The majority of sites do not seem to use prototypes for code reuse, but this is possibly explained by the existence of other ways to achieve code reuse in JavaScript (i.e., the ability to assign closures directly into a field of an object). The programs that do utilize prototypes have similar inheritance properties to Java [23].

### 5.4 Object Kinds

Figure 7 breaks down the kinds of objects allocated at run-time into a number of categories. There are a number of frequently used built-in data types: dates (`Date`), regular expressions (`RegExp`), document and layout objects (DOM), arrays (`Array`) and runtime errors. The remaining objects are separated into four groups: anonymous objects, instances, functions, and prototypes. Anonymous objects are constructed with an object literal using the `{...}` notation, while instances are constructed by calls of the form `new C(...)`. A function object is created for every `function` expression evaluated by the interpreter and a prototype object is automatically added to every function in case it is used as a constructor. Over all sites and traces, arrays account for 31% of objects allocated. `Dates` and DOM objects come next with 12% and 14%, respectively. Functions, prototypes, and instances each account for 10% of the allocated objects, and finally anonymous objects account for

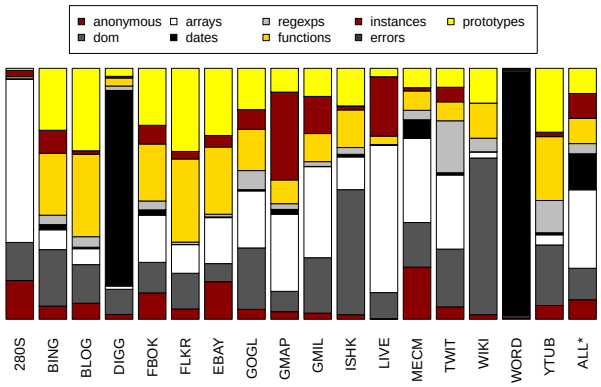


**Figure 7. Kinds of allocated objects.** The per-site proportion of runtime object kinds (averaged over multiple traces).

7%. It is interesting to observe that some sites are outliers: for instance, the proportion of objects allocated by WORD is dominated by 150K `Date` objects, and similarly for DIGG[7]. Other sites, LIVE and 280S notably, are more array-intensive. It should be noted that in JavaScript, any object can be treated as an array (using the index notation `x[3]`) but our analysis has shown that in practice, only objects created by the built-in `Array` constructor are routinely accessed by this syntax with numeric indices.

## 6. Measuring Program Dynamism

Different dimensions of dynamism are captured in the execution traces; we discuss them next.

### 6.1 Call Site Dynamism

Dynamic binding is a central feature of object-oriented programming. Many authors have looked at the degree of polymorphism of individual call sites in the program source as a reflection of how "object-oriented" a given program is. More pragmatically, when a call site is known to be monomorphic, i.e. it always invokes the same method, then the dispatching code can be optimized and the call is a candidate for inlining. It is not unusual to be able to identify that over 90% of call sites are monomorphic in Java. To estimate polymorphism in JavaScript, one must first overcome a complication. A common programming idiom in JavaScript is to create objects inline. So the following code fragment

```
for (...) { ...  = { f : function (x) { return x; } }; }
```

will create many objects, that all have a method `f()`, but each has a different function object bound to `f`. Naively, one could count calls, `x.f()`, with different receivers as being polymorphic. We argue that for our purposes it is preferable to count a call site as polymorphic only if it dispatches to a function with a different body, i.e. calls to clones should be considered monomorphic. Figure 8 shows the frequency of clones across all traces. While 150,422 functions objects have a distinct body, we found 16 bodies that are shared by tens of thousands of function objects, with 1 function body shared by 41,244 objects. GMAP, LIVE and MECM each had function bodies with over 10,000 associated function objects.

Figure 9 demonstrates that only 81% of call sites in JavaScript are actually monomorphic; this is an upper bound for what a compiler or static analysis can hope to identify. In practice, it is likely that there are fewer opportunities for devirtualization and inlining in JavaScript programs than in Java programs. It is noteworthy that every program has at least one megamorphic call site, with a maximum of one call site having 1,437 different targets in 280S (which is otherwise perfectly predictable with 99.99% of the call sites being monomorphic!). BING, FBOK, FLKR, GMIL, GMAP and GOGL each had at least one call site with more than 200 targets. FBOK is another outlier with 3.5% of the call sites having 5 or more targets.

### 6.2 Function Variadicity

The declared arity of a function in JavaScript does not have to be respected by callers. If too few arguments are supplied, the value of the remaining arguments will be set to `undefined`. If more arguments are supplied than expected, the additional arguments are accessible in the `arguments` variable, which is an array-like object containing all arguments and a reference to the caller and callee. Furthermore, any function can be called with an arbitrary number of arguments and an arbitrary context by using the built-in `call` method[8]. As such, functions may be variadic without being declared as variadic, and may have any degree of variadicity.

Many built-in functions in JavaScript are variadic: some prominent examples include `call`, Array methods like `push`, `pop`, `slice`, and even the `Array` constructor itself (which initializes an array with any number of provided arguments). Libraries such as Prototype and jQuery use `call` and `apply` frequently to control the execu-

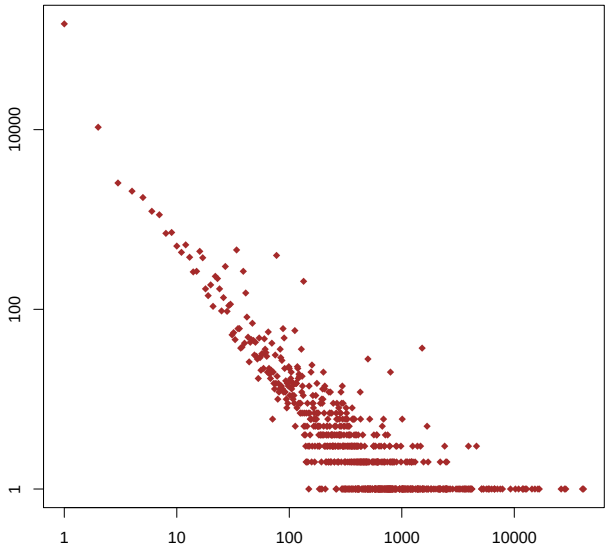


**Figure 8. Clones.** Plots the number of function objects per function body (x-axis) and the sum of such function bodies (y-axis) over all traces, in log-scale. For example, the second from left point represents the roughly 10,000 function bodies that each have 2 corresponding function objects (and thus, 1 clone).

| | Callsites with N function bodies | | | | | |
|---|---|---|---|---|---|---|
| **Site** | **1** | **2** | **3** | **4** | **>5** | **Max** |
| 280S | 99.9% | 0.0% | 0.0% | 0.0% | 0.0% | 1,437 |
| BING | 93.6% | 4.8% | 1.0% | 0.3% | 0.3% | 274 |
| BLOG | 95.4% | 3.4% | 0.5% | 0.2% | 0.5% | 95 |
| DIGG | 95.4% | 3.2% | 0.4% | 0.3% | 0.7% | 44 |
| EBAY | 91.5% | 7.1% | 0.5% | 0.5% | 0.5% | 143 |
| FBOK | 76.3% | 14.8% | 3.7% | 1.7% | 3.5% | 982 |
| FLKR | 81.9% | 13.2% | 3.6% | 0.5% | 0.8% | 244 |
| GMAP | 98.2% | 0.8% | 0.4% | 0.2% | 0.4% | 345 |
| GMIL | 98.4% | 1.2% | 0.2% | 0.1% | 0.2% | 800 |
| GOGL | 93.1% | 5.5% | 0.6% | 0.3% | 0.6% | 1,042 |
| ISHK | 90.2% | 8.1% | 1.0% | 0.0% | 0.8% | 42 |
| LIVE | 97.0% | 1.7% | 0.5% | 0.3% | 0.5% | 115 |
| MECM | 94.2% | 4.1% | 1.2% | 0.2% | 0.4% | 106 |
| TWIT | 89.5% | 7.2% | 1.7% | 0.3% | 1.3% | 60 |
| WIKI | 87.9% | 6.7% | 1.9% | 0.2% | 3.2% | 32 |
| WORD | 86.8% | 7.9% | 2.7% | 1.9% | 0.6% | 106 |
| YTUB | 83.6% | 10.6% | 5.4% | 0.1% | 0.4% | 183 |
| All | 81.2% | 12.1% | 3.0% | 1.2% | 2.5% | 1,437 |

**Figure 9. Call site polymorphism.** Number of different function bodies invoked from a particular callsite (averaged over multiple traces).

[7] Upon further investigation, both sites make extensive use of certain UI/animation libraries that allocate `Date` objects for use with callback timers.

[8] The built-in function `apply` is identical to `call` in utility, but avoids a variadic design by expecting a context and an array argument instead of a context and a variable number of arguments.

| | Functions with N distinct arities | | | | | |
|---|---|---|---|---|---|---|
| **Site** | **1** | **2** | **3** | **4** | **>5** | **Max** |
| 280S | 99.3% | 0.6% | 0.0% | 0.1% | 0.1% | 9 |
| BING | 94.2% | 4.9% | 0.7% | 0.2% | 0.0% | 4 |
| BLOG | 97.1% | 2.3% | 0.4% | 0.2% | 0.0% | 4 |
| DIGG | 92.5% | 6.3% | 0.9% | 0.3% | 0.1% | 5 |
| EBAY | 95.9% | 3.6% | 0.3% | 0.0% | 0.3% | 9 |
| FBOK | 93.9% | 4.8% | 0.6% | 0.6% | 0.1% | 6 |
| FLKR | 94.2% | 4.6% | 0.9% | 0.3% | 0.0% | 4 |
| GMAP | 93.4% | 5.5% | 0.6% | 0.3% | 0.2% | 6 |
| GMIL | 95.3% | 3.8% | 0.6% | 0.2% | 0.2% | 30 |
| GOGL | 94.6% | 4.3% | 0.7% | 0.2% | 0.2% | 9 |
| ISHK | 97.6% | 2.3% | 0.1% | 0.0% | 0.0% | 3 |
| LIVE | 92.7% | 6.1% | 0.8% | 0.3% | 0.1% | 7 |
| MECM | 91.9% | 6.5% | 0.6% | 0.5% | 0.5% | 7 |
| TWIT | 90.9% | 7.4% | 1.3% | 0.5% | 0.0% | 4 |
| WIKI | 96.7% | 3.3% | 0.0% | 0.0% | 0.0% | 2 |
| WORD | 92.6% | 6.6% | 0.6% | 0.2% | 0.0% | 4 |
| YTUB | 98.5% | 1.4% | 0.1% | 0.0% | 0.0% | 4 |
| All | 93.5% | 4.8% | 0.7% | 0.4% | 0.6% | 30 |

**Figure 10. Function variadicity.** Proportion of functions used variadically.

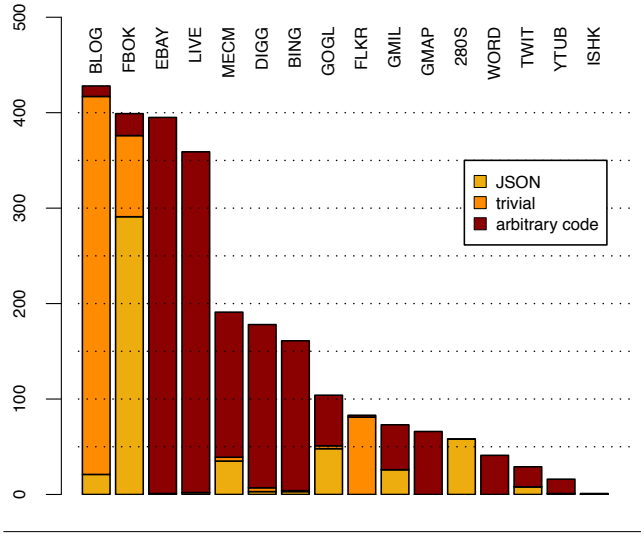


**Figure 11. Uses of `eval`.** Count of the invocations of `eval` (averaged over multiple traces). Sites sorted by total number of invocations, descending.

tion context when invoking callback closures. These two libraries (and many other applications) also use arrays for their internal representation, which leads to many uses of variadic `Array`-related functions. Depending on the coding style, functions with optional arguments can either declare these optional arguments (leading to some calls of arity less than the declared arity), or test for the presence of optional (unnamed) arguments in the `arguments` object. Both coding styles are seen in real-world JavaScript programs, so both calls of arity less than and calls of arity greater than that declared are often observed.

In practice, variadic functions are common and were observed in every website analyzed. Figure 10 indicates the portion of function bodies which are called with differing arities (that is, which are used variadically). In all sites, at least 90% of functions are non-variadic. However, highly-variadic functions, with as many as 30 distinct arities, also occur, and few sites have less than 5% variadic functions.

### 6.3 Uses of `eval`

One of the most dynamic features of JavaScript, and of most dynamic languages, is `eval` which runs arbitrary code provided as a string. A key question for any static approach is how that feature is used. Figure 11 shows that all sites use it, from a handful of times in ISHK to 428 times for BLOG. Clearly this means that `eval` can't be discounted. The next question is thus: what do programs do within `eval` strings? Figure 11 further breaks down `eval`s in three sub-categories. The first one (JSON) is a simple form of deserialization. JavaScript Object Notation (JSON) is a standard format for exchanging data in the form of strings, the syntax of which coincides with that for JavaScript object literals. Therefore, `eval`ing a JSON string results in the construction of a new object. This is a relatively innocuous use of `eval` and in sites such as FBOK it accounts for the vast majority of calls. The second category appearing in Figure 11 (trivial) is that of "trivial" uses of `eval`. This is another surprisingly common use in which the argument is an identifier. Trivial uses of `eval` are essentially a very powerful form of reflection. Consider

```
if   (eval("flash"+i+"Installed")==true) ...
```

from FLKR, which is called multiple times to perform a different action depending on what version of Flash is installed, referencing the field `i` in the current scope. The remaining invocations of `eval` (arbitrary code) are by a wide margin the most common, and are often complicated code, involving variable assignment, changing functions, etc.

It is important to stress here the sheer variety of arbitrary code. The following is a *random* selection of `eval`'d arbitrary code strings throughout the traces, edited only for readability and privacy:

- `window.dc_AdLinkBold = false`
- ```
  playlist[204] = new function() {
    this.album_id = 204; this.album_name = "[elided]";
    this.album_rating_avg = 3.9; this.OA = 1255051920;
    this.album_rating_user = -0.1;
  };
  ```
- ```
  this.load = function() {
    var a = arguments, len = a.length, s = "";
    for (var i=0; i<len; i++) s += ",a[" + i + "]";
    return eval('this._processEvent("load"' + s + ')');
  }
  ```
- `typeof(l[i].parent.onAfterLoad) == 'function'`
- `objRef.onHandleInteraction=new Function()`

It is clear that `eval` is significant to the logic of many JavaScript programs, and furthermore that its behavior is neither constrained nor consistent in real programs. Some of these behaviors would be quite difficult to emulate without `eval`, as they perform arbitrary changes to the environment of the JavaScript program. Every possible change that `eval`'d code could make would need to be predicted ahead of time and implemented in support functions.

JavaScript and the DOM provide means other than `eval` to inject code at runtime, such as the `document.write` of a script tag, or `document.createElement("script");`, but these methods are entirely reliant on the browser's document object model. Since our tracing infrastructure instrumented only the JavaScript interpreter

and not the rest of the browser, we were unable to detect these other mechanisms.

Somewhat related to trivial uses of `eval` is JavaScript's hashmap syntax for accessing fields. The following two expressions are equivalent: `l.map` and `l["map"]`. However, the latter may be used with any string constructed at runtime, even strings that are not syntactically valid identifier names. Thus it can obscure from the compiler or verifier which field will be accessed. Code review suggests that this occurs when (a) objects are in fact hash maps, (b) libraries, such as Prototype, construct "classes" based on dynamic descriptions, (c) when the code needs to update an unknown field. Traces do not indicate any consistency in the behavior and use of hashmap notation. Figure 12 shows, for all objects which are ever accessed with hashmap notation, for traces using the Prototype library (above) and jQuery (below), which style of access is used over time (Direct access is `x.f` and Hashmap access is `x["f"]`). If there was a clear separation between hashmaps and normal objects one would expect to see only hashmap accesses in the figure. Instead, the fact that the majority of object accesses are still direct accesses suggests that objects are accessed with a mixture of direct and hash notation. The Prototype library uses hashmap notation to build class-like prototypes from abstract descriptions. As expected, sites using this library (e.g., ISHK) have far more hashmap activity during construction than post-construction (the dashed line in the graph indicates the end of the object's initialization). Other sites, however, are less consistent.

### 6.4 Object Protocol Dynamism

It is often asserted that although properties can be added to objects at any point, they are generally added only during initial construction, and are later only read and updated. The measure of the number of fields/methods that are added or deleted after initialization is thus likely to be a good metric of the dynamism of JavaScript programs. For so-called instance objects, i.e. objects that are created by invoking a constructor function, there is a clear sense of what object initialization means: it is the time spent in the constructor. For anonymous objects, things are much less clear. Figure 13 shows the accesses performed on constructed objects over the lifespan of the object for two sites: a site showing the expected and desired behavior (TWIT), and a poorly-behaved site (GOGL). Time is measured in events that have the object as a target for a read, add, delete and update operation. Object lifespans are normalized to construction time, such that the vertical line in the graph separates the constructor from post-constructor accesses. Although it is clear that field additions are far more common during the constructor, most sites add fields to objects well into their lifespan. Furthermore, many objects have a "hump" of field adds immediately after the constructor ends, suggesting that a heuristic to determine when an object is fully constructed, and for that matter when its type is well-defined, would be quite difficult to find. This hump is created by factory methods, inheritance emulation and other patterns which consistently add fields to an object immediately after it is instantiated. Field deletions are uncommon, but occur in FBOK, GMIL, GOGL and YTUB. Another view of the same information is provided in Figure 14. Although most sites have far greater protocol-changing accesses (field additions and deletions) during construction time, post-construction protocol changes are as many as 10% of accesses

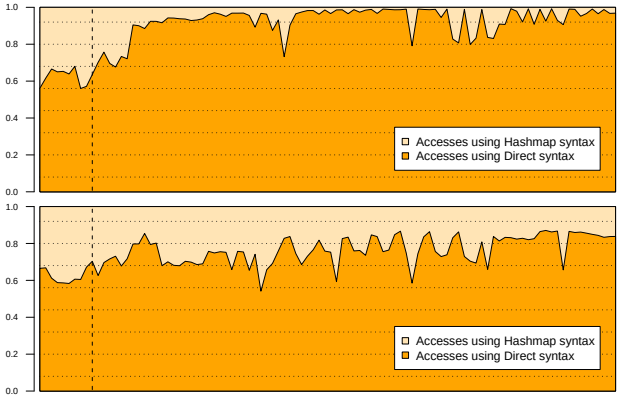


**Figure 12. Hashmap access.** How objects accessed at least once via hashmap syntax are accessed over time. Above, the average of all sites using the Prototype library, and below, the average of all sites using the jQuery library. The dashed line represents the end of object construction.

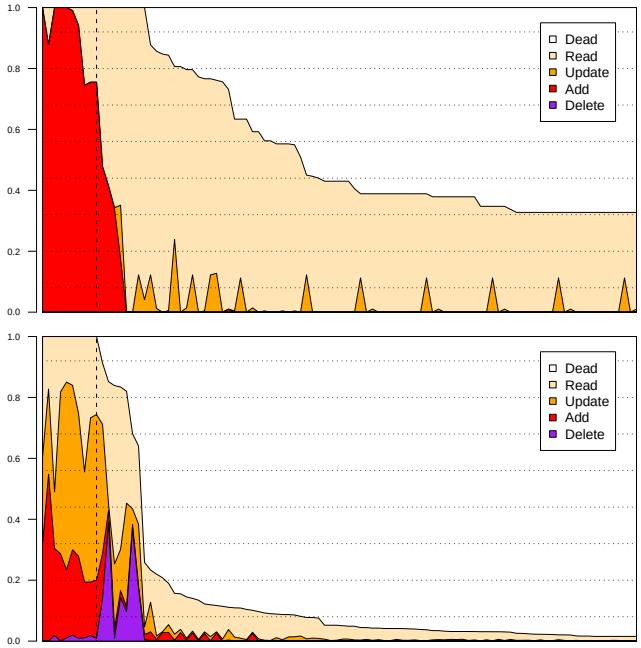


**Figure 13. Object timelines.** Above, TWIT. Below, GOGL. The dashed line indicates the end of object construction.

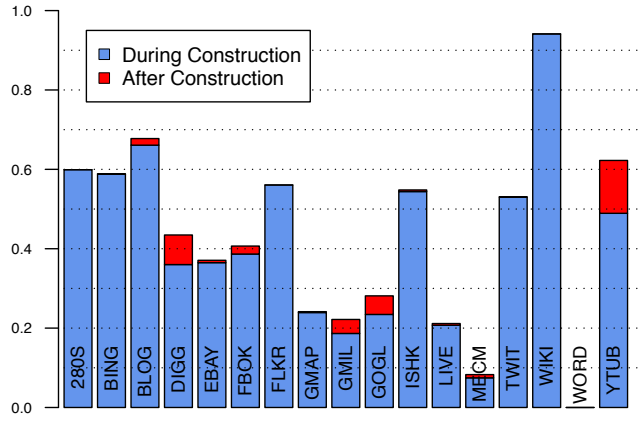


**Figure 14. Object protocol changes.** Average ratio of object protocol changes (field additions and deletions) to all activity, both during and after construction.

on some sites. This is in spite of the fact that Figure 14 extends our heuristic for the termination of initialization to after the post-construction hump mentioned above.

### 6.5 Constructor Polymorphism

JavaScript's mechanism for constructing objects is more dynamic than that of class-based languages because a constructor is simply a function that initializes fields programmatically. Contrast the following constructor function to the declarative style imposed by a class-based language:

```
function C(b) {  if(b) this.y = 0; else this.x = 0; }
```

The objects returned by `new C(true)` and `new C(false)` will have different (in this case, disjoint) sets of properties. If one can envision as the set of properties returned by a constructor as a "type", then it is natural to wonder how many constructors return different types at different times during the execution of a program. Figure 15 gives an overview of the polymorphism of constructors over all traces. The majority of constructors always return the same set of properties. But, 573 constructors return two different sets of properties, and one outlier that returns 246 different "types" was observed in GMIL.

This polymorphism can arise for a number of reasons, but a common one is that the dynamism of JavaScript allows libraries to abstract away the details of implementing object hierarchies. Often, these abstractions end up causing all object construction to use a single static constructor function, which is called in different contexts to create different objects, such as the following constructor function from the Prototype library.

```
function klass() {
    this.initialize.apply(this, arguments);
}
```

All user objects inherit this constructor, but have distinct initialize methods. As a result, this constructor is polymorphic in the objects it creates.

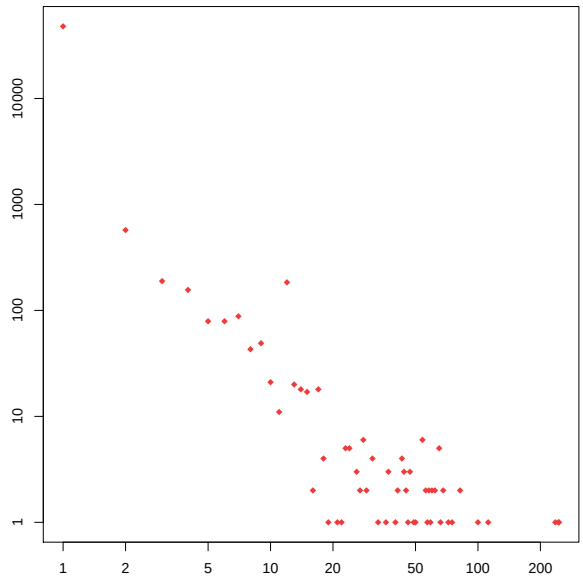


**Figure 15. Constructor polymorphism.** Plots the number of distinct sets of properties (x-axis) against the number of constructor functions observed to create objects with that many distinct sets of properties (y-axis). (Log scale)

### 6.6 Constructor Prototype Modification

The `prototype` field of a constructor defines which properties an object created by this constructor will inherit. However, the value of the prototype field can be changed, which means that two objects created by the same constructor function may have different prototypes, and so different APIs. Changing the prototype field is generally done before any objects are created from that prototype, and is often done by helper functions such as the following from the Prototype library to mimic subclassing.

```
  function subclass() {};
...
    if (parent) {
      subclass.prototype = parent.prototype;
      klass.prototype = new subclass;
      parent.subclasses.push(klass);
    }
```

We did not record the number of occurrences this pattern at runtime, but clearly the possibility that the above code will be executed can not be discounted.

### 6.7 Changes to the Prototype Chain

An object's protocol can change over time by adding or deleting fields from any of its prototypes. Although we found this behavior to be uncommon for user-created types, it is very common for libraries to extend the builtin types of JavaScript, in particular `Object` and `Array`. For instance, the Prototype library includes a number of collection-like classes, but also extends `String.prototype` and `Array.prototype` such that they can be used as collections, by adding e.g. the `toArray`, `truncate` and `strip` methods to them, as well as extending `Array` to include all of the definitions from Prototype's `Enumerable` type:

```
Object.extend(Array.prototype, Enumerable);
```

Some code uses this ability to change prototypes as a form of modularity. Since prototypes can be modified at any time, features can be implemented in separate parts of the code even if they affect the same type. Again, we do not report runtime occurrences, but observe that this is something that must be accounted for by tools and static type disciplines.

### 6.8 Object Lifetimes

As in many languages, most objects in JavaScript are generally very short-lived. Figure 16 shows the percentiles of object lifetimes seen across all traces, in terms of events performed on those objects (we do not record wall clock time in traces). 25% of all objects are never used, and even the 90th percentile of objects are alive for only 7 events. This does not include any integers or strings which the runtime never boxes into an object (which is to say, numbers and strings that never have fields accessed). The conclusion is clearly that much of data is manipulated very infrequently and thus suggest that lazy initialization may be a winning optimization.

| | Percentile | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 25 | 50 | 75 | 85 | 90 | 95 | 97 | 98 | 99 | 100 |
| **Events** | 0 | 1 | 3 | 6 | 9 | 14 | 25 | 37 | 74 | 1,074,322 |

**Figure 16. Object lifetimes.** The longevity of objects in terms of the number of events performed on them.

### 6.9 The Effects of JavaScript Libraries

Many contemporary sites utilize JavaScript libraries; in our corpus, 44 sites used a publicly-available library, and 7 sites used more than one library simultaneously. The most popular were jQuery and

Prototype, appearing on 21 and 9 sites, respectively. Such libraries provide simplified and well-tested coding patterns for problems including UI widgets, animation, DOM manipulation, serialization, asynchronous page loading, and class-based workarounds built on top of JavaScript's prototype-based system. In general, the presence of a particular library does not imply a major change in the program's dynamic behavior. This is in part due to the large feature sets of most libraries. Prototype offers all of the functionality mentioned above (besides UI widgets and animation), and jQuery similarly offers all of the above (except an implementation of "classes"). Because there are many use cases for each library, there are few characteristic runtime behaviors exhibited. Exceptions to this tend to be artifacts of implementation techniques specific to a library (such as Prototype's dynamic construction of prototype objects, or the disproportionate allocation of `Date` objects by animation libraries).

## 7. Measuring the Behavior of Benchmarks

There are several popular benchmark suites used to determine the quality and speed of JavaScript implementations. However, using these benchmarks as metrics assumes that they are representative of JavaScript programs at large. We looked at three suites in particular to determine their relevance: *SunSpider:* (SUNS) A wide range of compute-intensive benchmarks. Includes deserialization, a raytracer, and many other primarily mathematical tasks. *V8:* (V8BM) The benchmarks associated with Google's Chrome browser. Again they include computationally-intensive benchmarks., such as cryptography and another raytracer. *Dromaeo:* (DROM) Unlike the other suites, these benchmarks are intended to test the browser's DOM, as opposed to the JavaScript engine itself. In several ways, these benchmarks have proven to be inconsistent with the real-world JavaScript code we tested. We discuss our main observations:

### 7.1 Object Kinds

Benchmarks tend to heavily stress a few types of objects, which have little similarity to the object types used by real-world sites. Figure 17 shows the benchmarks' disproportionate number of instances and arrays. Comparing the benchmarks to the All Sites bar, one can clearly observe that constructed objects (instances) are overrepresented in V8BM and SUNS, whereas DROM is almost exclusively preoccupied with arrays.

The extensive use of constructed objects in benchmarks is notable. In SUNS, 39% of objects are instances, and in V8BM, 63% are. In the real-world sites, only GMAP and LIVE produced more than 10% instance objects (with GMAP and LIVE producing 35% and 24%, respectively). It seems likely therefore that a JavaScript implementation that favored other object types would be poorly represented by SUNS and V8BM.

### 7.2 Uses of `eval`

While SUNS has benchmarks which use `eval`, performing 2785 `eval`s in our trace with only 33 deserializing JSON data, V8BM performs no `eval`s. DROM performed 32 `eval`s, with only 1 deserializing JSON data. This suggests that SUNS is more representative

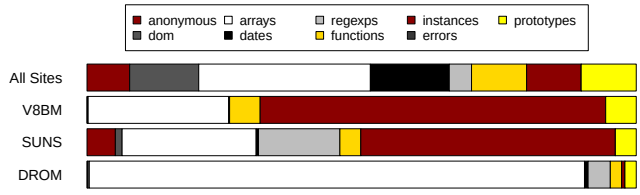


**Figure 17. Kinds of allocated objects.**

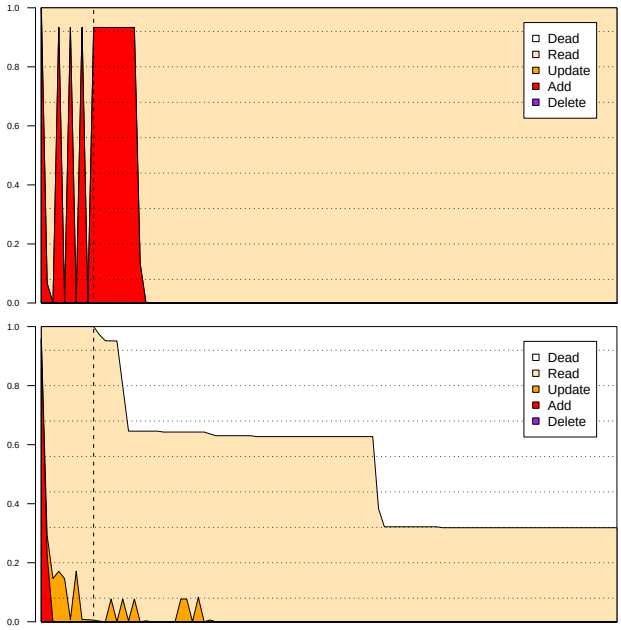


**Figure 18. Object timelines.** SUNS (above) and V8BM (below). The dashed line indicates the end of object construction.

of real-world workloads, the others less so. The latter is not surprising given the nature of the benchmarks (there is a lot of mathematical computation which is not typical of most JavaScript programs in the wild).

### 7.3 Object Protocol Dynamism

Although many sites have relatively sane and stable use of objects, with object initialization occurring mostly during construction, several do not. Figure 18 shows the object timelines of SUNS and V8BM. The behavior of most sites at construction time is modeled by SUNS, with a post-construction hump as seen in several real-world sites. However, the lifetime of objects in SUNS is atypical, with most objects fairly long-lived. V8BM's object dynamism is completely dissimilar to any real-world site, to the benefit of Google's V8 JavaScript engine. The lifetimes of objects in V8BM is similar to object lifetimes of real-world JavaScript, with the exception that objects have fairly constant lifetimes, as shown by the steep dropoffs in living objects in Figure 18. This peculiarity was not seen in any real-world sites. DROM uses no constructed objects, as its intention is primarily to test the implementation of the DOM API, and is thus not very useful as general purpose JavaScript benchmark.

### 7.4 Function Variadicity and Polymorphism

Variadicity in the benchmarks was not dissimilar to real-world programs. DROM and SUNS each had about 5% of functions used variadically (close to the 6% seen accross all sites), and V8BM had about 2% variadic. Polymorphism was rarer in the benchmarks, with 3%, 2% and 1% of call sites being polymorphic in DROM, SUNS and V8BM (respectively). As 19% of call sites were polymorphic across all sites, implementations which do not handle polymorphic call sites well will perform better with benchmarks than real-world JavaScript.

## 8. Conclusion

This paper has provided the first large-scale study of the runtime behavior of JavaScript programs. We have identified a set of representative real-world programs ranging in size from hundreds of kilobytes to megabytes, using an instrumented interpreter we have recorded multiple traces per site, and then with an offline analysis tool we have extracted behavioral information from the traces. We use this information to evaluate a list of nine commonly made assumptions about JavaScript programs. Each assumption has been addressed, and most are in fact false for at least some real-world code. To summarize, we found that:

1. **The prototype hierarchy is invariant.** Libraries often change JavaScript's builtin prototypes in order to add behavior to types which would be fixed in a less-flexible language, such as `Object` and `Array`. Although changes to user-created types are more rare, they do occur as a means of modularity.
2. **Properties are added at object initialization.** This assertion is only true for a subset of sites, with particularly poorly-behaved sites adding and deleting fields late in the lifespan of very long-lived objects. This presents a challenge to any attempt at imposing a type upon objects at runtime, as even the very simple notion of type being a list of fields is subject to changes over objects' lifespans.
3. **Properties are rarely deleted.** Deletions were found to be in fact quite common on some sites. Most JavaScript programs do not, however, use field deletion at all.
4. **The use of `eval` is infrequent and does not affect semantics.** Our data shows that `eval`s are not infrequent, and on most sites they have arbitrary and unpredictable behavior. The secondary assumption, that `eval` is used primarily for deserialization, also turns out to be false for most sites. Furthermore, nearly every site that uses JavaScript also uses `eval`.
5. **Declared function signatures are indicative of types.** In most languages, variadic functions are rare. Our data indicates that nearly 10% of functions are variadic in JavaScript.
6. **Program size is modest.** When analyzing the amount of code the interpreter must parse and run, many sites are (with repeated code) running megabytes of code. Even ignoring identical reloaded code, most sites load hundreds of kilobytes. Full-program analysis is likely infeasible with this amount of code, even if it was all known statically (i.e. not created by `eval`)
7. **Call-site dynamism is low.** Across all traces only 81% of call-sites are monomorphic. Even if determining which sites are monomorphic statically was achievable, the number of polymorphic call sites and degree of polymorphism is high enough that they would provide a significant hurdle to statically analyzing code paths.
8. **Execution time is dominated by hot loops.** This assumption is in fact true, but less so than in Java. Our data shows that 10% of the functions ever called are hot, and only 50% of functions are ever called.

Figure 19 gives a subjective opinion on which sites violated which of these assumptions. Although the opinion on whether the assumption is violated is subjective, they rely on the same underlying data as the rest of this paper.

[9] Only egregious violators are noted here, as only one site had no arbitrary evals at all

[10] These are sites for which the 90-10 rule (that 90% of execution time is spent in 10% of functions) does not hold

| | **1** | **2** | **3** | **4**[9] | **5** | **6** | **7** | **8**[10] |
|---|---|---|---|---|---|---|---|---|
| 280S | X | | | | | | | |
| BING | | | | X | X | | | |
| BLOG | | X | | | | X | | X |
| DIGG | X | X | | X | X | X | | |
| EBAY | | | | X | | X | | X |
| FBOK | X | X | X | | X | X | X | |
| FLKR | | | | | X | | X | X |
| GMAP | X | | | X | X | X | | |
| GMIL | X | X | X | X | | X | | |
| GOGL | X | X | X | X | X | X | | X |
| ISHK | X | X | | | | | X | |
| LIVE | X | X | | X | X | X | | |
| MECM | X | X | | X | X | X | | |
| TWIT | X | | | | X | | X | |
| WIKI | | | | | | | X | X |
| WORD | X | | | X | X | | X | |
| YTUB | | X | X | | | X | X | X |

**Figure 19. Violations.** For each assumption (above), a subjective opinion of which sites (left) violate that assumption.

Given how thoroughly these common assumptions are violated, it seems that JavaScript is indeed a harsh terrain for static analysis. However, no sites violate all of the assumptions, so optimizations based on them could work in many cases. Optimizations requiring whole-program analysis are unlikely to be successful, as many sites use a large amount of JavaScript, and `eval` is frequent and unpredictable. Rigidly static type systems are unlikely to be usable with JavaScript; any applicable type system must be open to the very real possibility of object protocol changes. Any typing framework which depends on the typing of function parameters will struggle with JavaScript's high degree of variadicity.

## Acknowledgments

The authors thank Tobias Wrigstad and Johan Östlund for their enthusiasm and work on the early stages of this project; as well as the anonymous reviewers and Manuel Serrano for their comments. This work is supported in part by NSF grants CCF 0938232 and CNS 0716659 as well as ONR award N000140910754.

## References

[1] Christopher Anderson. *Type Inference for JavaScript*. PhD thesis, Department of Computing, Imperial College London, March 2006.

[2] Christopher Anderson and Sophia Drossopoulou. BabyJ: From object based to class based programming via types. *Electr. Notes Theor. Comput. Sci.*, 82(7), 2003.

[3] Christopher Anderson and Paola Giannini. Type checking for JavaScript. *Electr. Notes Theor. Comput. Sci.*, 138(2), 2005.

[4] Brad Calder, Dirk Grunwald, and Benjamin Zorn. Quantifying behavioral differences between c and c++ programs. *Journal of Programming Languages*, (4), 1994.

[5] Craig Chambers, Dave Ungar, and Erin Lee. An efficient implementation of SELF a dynamically-typed object-oriented language based on prototypes. *SIGPLAN Not.*, 24(10):49–70, 1989.

[6] Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. Staged information flow for JavaScript. In *Programming Language Design and Implementation, (PLDI)*, 2009.

[7] Bruno Dufour, Karel Driesen, Laurie J. Hendren, and Clark Verbrugge. Dynamic metrics for java. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, 2003.

[8] Ben Feinstein and Daniel Peck. Caffeinemonkey: Automated collection, detection and analysis of malicious JavaScript. In *Black Hat USA 2007*, Las Vegas, NV, USA, 2007.

[9] Michael Furr, Jong hoon An, Jeffrey Foster, and Michael Hicks. Static type inference for ruby. In *Symposium on Applied Computing (SAC)*, 2009.

[10] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. Trace-based just-in-time type specialization for dynamic languages. In *Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[11] C.D. Garret, Jeff Dean, David Grove, and Craig Chambers. *Measurement and application of dynamic receiver class distributions*. Univ of Washington, 1994.

[12] Arjun Guha, Shriram Krishnamurthi, and Trevor Jim. Using static analysis for ajax intrusion detection. In *International Conference on World Wide Web (WWW)*, 2009.

[13] Phillip Heidegger and Peter Thiemann. Recency types for dynamically-typed, object-based languages. In *Foundations of Object Oriented Languages (FOOL)*, 2009.

[14] Alex Holkner and James Harland. Evaluating the dynamic behaviour of Python applications. In *Australasian Computer Science Conference (ACSC)*, 2009.

[15] Daniel Ingalls, Krzysztof Palacz, Stephen Uhler, Antero Taivalsaari, and Tommi Mikkonen. The lively kernel a self-supporting system on a web page. In *Self-Sustaining Systems*, 2008.

[16] ECMA International. *ECMA-262: ECMAScript Language Specification*. ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, third edition, December 1999.

[17] Dongseok Jang and Kwang-Moo Choe. Points-to analysis for JavaScript. In *Symposium on Applied Computing (SAC)*, 2009.

[18] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for JavaScript. In *Static Analysis Symposium (SAS)*, 2009.

[19] Sylvain Lebresne, Gregor Richards, Johan Östlund, Tobias Wrigstad, and Jan Vitek. Understanding the dynamics of JavaScript. In *Workshop on Script to Program Evolution (STOP)*, 2009.

[20] Florian Loitsch and Manuel Serrano. Hop client-side compilation. In *Symposium on Trends on Functional Languages*, 2007.

[21] Sergio Maffeis, John C. Mitchell, and Ankur Taly. Isolating JavaScript with filters, rewriting, and wrappers. In *European Symposium on Research in Computer Security (ESORICS)*, 2009.

[22] Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin Zorn. JSMeter: Comparing the behavior of JavaScript benchmarks with real web applications. In *USENIX Conference on Web Application Development (WebApps)*, June 2010.

[23] Ewan D. Tempero, James Noble, and Hayden Melton. How do java programs use inheritance? an empirical study of inheritance in java software. In *European Conference on Object-Oriented Programming (ECOOP)*, 2008.

[24] Peter Thiemann. Towards a type system for analyzing JavaScript programs. In *European Symposium on Programming (ESOP)*, 2005.

[25] Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of Typed Scheme. In *POPL*, pages 395–406, New York, NY, USA, 2008. ACM.

[26] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Krügel, and Giovanni Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *Network and Distributed System Security Symposium (NDSS)*, 2007.

[27] Dachuan Yu, Ajay Chander, Nayeem Islam, and Igor Serikov. JavaScript instrumentation for browser security. In *Symposium on Principles of Programming Languages (POPL)*, 2007.

[28] Chuan Yue and Haining Wang. Characterizing insecure JavaScript practices on the web. In *18th International World Wide Web Conference*, pages 961–961, April 2009.

View publication stats