International Research Journal of Engineering and Technology (IRJET) e-ISSN: 2395-0056
Volume: 04 Issue: 12 | Dec-2017 www.irjet.net p-ISSN: 2395-0072

Runtime Behaviour of JavaScript Programs

Vipin Kumar Dhiman[1], Raveendra Kumar Bharati[2] , Varun Bansal[3]

[1]M.Tech Research Scholar, Department of Computer Science Engineering, Shobhit University, Gangoh, Saharanpur, Uttar Pradesh
[2,3]Assistant professor, Department of Computer Science Engineering, Shobhit University, Gangoh, Saharanpur, Uttar Pradesh

---

Abstract*: Javascript is a dynamic computer programming language. It is lightweight and most commonly used as a part of web pages, whose implementations allow client-side script to interact with the user and make dynamic pages. It is an interpreted programming language with object-oriented capabilities. 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. 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.*

Key words: *Dynamic Behaviour, Execution Tracing, Dynamic Metrics, Program Analysis, JavaScript.*

INTRODUCTION:

JavaScript 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 tradition all 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. The language is also becoming a general purpose computing platform with office applications, browsers and development environments 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 and Scheme. 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. 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. 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. 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.

Common Assumptions about the dynamic behaviour 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 chose to not even model prototypes. Research on static analysis typically does not mention prototype updates .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. 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. Properties are rarely deleted. Removal of methods or fields is difficult to accommodate in a type system as it permits nonmonotonic 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. 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.

4. 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 . This is not necessarily the case because JavaScript allows calls with different arities.

5. Program size is modest. Some papers justify very expensive analyses with the explicit assumption that handwritten JavaScript programs are small, and others implicitly rely on this as they present analyses which would not scale to large systems.

6. 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 Justin- time compilers such as TraceMonkey 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. But conclusions obtained from use of those benchmarks are only meaningful if they accurately represent the range of JavaScript workloads in the wild.

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. There have been studies of JavaScript's dynamic behavior as it applies to security, 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, and those results are consistent with the new results presented here. Holkner and Harland 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, Calder et al. characterized the difference of between C and C++ programs in, and Temporo et al. studied the usage of inheritance in Java. These previous papers study in great detail one particular aspect of each language.

Measuring Program Dynamism:

Different dimensions of dynamism are captured:

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. 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.

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. 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 execution 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.

	Callsites with N function bodies					
Site	**1**	**2**	**3**	**4**	**>5**	**Max**
280s	99.9%	0.0%	0.0%	0.0%	0.0%	1437
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%	1042
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%	1437

Figure 1. Call site polymorphism. Number of different function bodies invoked from a particular callsite (averaged over multipletraces).

3. 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.

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.

4. 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 cannot be discounted.

5. 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 b e used as collections, by adding e.g. the to Array, 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. Object Lifetimes:

As in many languages, most objects in JavaScript are generally very short-lived. Figure 2 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.

Figure 2. Object lifetimes. The longevity of objects in terms of the number of events performed on them.

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.

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

1. The protype hierarchy is invariant

2. Properties are added at object initialization.

3. Properties are rarely deleted.

4. The use of eval is infrequent and does not affect semantics.

5. Declared function signatures are indicative oftypes.

6. Program size is modest.

7. Call-site dynamism is low.

8. Execution time is dominated by hot loops.

Acknowledgments:

I am indebted to express my deep gratitude to Mr. Varun Bansal, Head in Charge, Department of Computer Science Engineering, Shobhit University, Gangoh for their valuable guidance during each and every phase of this work.

I take this opportunity to express a deep sense of gratitude towards my guide Mr. Raveendra Kumar Bharati, for providing excellent guidance, encouragement and inspiration throughout the project work. Mr. Raveendra Kumar Bharati showed immense understanding and patience for my thesis during my difficult times, especially when I was pivoting around topics. Without his invaluable guidance, this work would never have been a successful one. I would also like to thank all my faculty members for their valuable suggestions and helpful discussions.

References:

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

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

[3] www.tutorialspoint.com

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

[5] P. Ratanaworabhan, B. Livshits, D. Simmons, and B. Zorn. JSMeter: Characterizing

[6] www.javaTpoint.com

[7] www.W3schools.com

[8] Thinking in JavaScript-Aravind Shenoy

[9] JQuery Cookbook-O'Reilly

[10] 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.