# Performance Issues and Optimizations in JavaScript: An Empirical Study

Marija Selakovic and Michael Pradel

Technical Report TUD-CS-2015-1249

TU Darmstadt, Department of Computer Science

October, 2015

TECHNISCHE
UNIVERSITÄT
DARMSTADT

SOLA
SoftwareLab

# Performance Issues and Optimizations in JavaScript: An Empirical Study

Marija Selakovic
Department of Computer Science
TU Darmstadt, Germany
m.selakovic89@gmail.com

Michael Pradel
Department of Computer Science
TU Darmstadt, Germany
michael@binaervarianz.de

## ABSTRACT

As JavaScript is becoming increasingly popular, the performance of JavaScript programs is crucial to ensure the responsiveness and energy-efficiency of thousands of programs. Yet, little is known about performance issues that developers face in practice and they address these issues. This paper presents an empirical study of 98 fixed performance issues from 16 popular client-side and server-side JavaScript projects. We identify eight root causes of issues and show that inefficient usage of APIs is the most prevalent root cause. Furthermore, we find that most issues are addressed by optimizations that modify only a few lines of code, without significantly affecting the complexity of the source code. By studying the performance impact of optimizations on several versions of the SpiderMonkey and V8 engines, we find that only 42.68% of all optimizations improve performance consistently across all versions of both engines. Finally, we observe that many optimizations are instances of patterns applicable across projects, as evidenced by 139 previously unknown optimization opportunities that we find based on the patterns identified during the study. The results of the study help application developers to avoid common mistakes, researchers to develop performance-related techniques that address relevant problems, and engine developers to address prevalent bottleneck patterns.

## 1. INTRODUCTION

JavaScript has become one of the most popular programming languages. It is widely used not only for client-side web applications, but also for server-side applications, mobile applications, and even desktop applications. The performance of JavaScript code is crucial to ensure that applications respond quickly to requests without consuming unnecessarily high amounts of CPU-time and energy. For example, for server-side code that may respond to thousands of requests every second, even a relatively small performance improvement can have a significant impact on the overall throughput and energy consumption. Likewise, client-side code that performs poorly can cause users to perceive an application as unresponsive [32], which may encourage them to instead use a competitor's web site.

The use of the language has evolved from simple client side scripts to complex programs, such as email clients, word processors, and interactive games. This development has been followed by significant improvements of JavaScript's performance. The main reason for this evolutionary step are the tremendous improvements of JavaScript engines in recent years, for example, due to highly optimized just-in-time (JIT) compilers [7, 19, 12, 5, 1]. Despite the effectiveness of JIT compilation, developers still apply optimizations to address performance issues in their code, and future improvements of JavaScript engines are unlikely to completely erase the need for manual performance optimizations.[1]

For example, a common optimization is to replace a `for-in` loop that iterates over the properties of an object $o$ by code that first computes these properties using the built-in `Object.keys()` function, and then iterates through them with a traditional `for` loop. This optimization often improves performance on the V8 engine because the JIT compiler may refuse to optimize a function that contains a `for-in` loop if the object $o$ is internally represented as a hash map. In contrast, the JIT compiler successfully optimizes the call of `Object.keys()` and the traditional `for` loop.

Despite the importance of JavaScript's performance, little is currently known about performance issues and optimizations in real-world JavaScript projects. This paper addresses this problem and asks the following research questions:

- RQ 1: What are the main root causes of performance issues in JavaScript?
- RQ 2: How complex are the changes that developers apply to optimize their programs?
- RQ 3: What is the performance impact of optimizations?
- RQ 4: Are optimizations valid across JavaScript engines and how does the performance impact of optimizations evolve over time?
- RQ 5: Are there recurring optimization patterns and can they be applied automatically?

Answering these questions helps improve JavaScript's performance by providing at least three kinds of insights. First,

[1]We refer to source code changes that a developer applies to improve performance as *optimizations*, and we refer to automatically applied transformations, e.g., in a JIT compiler, as *compiler optimizations*.

application developers benefit by learning from mistakes done by others. Second, developers of performance-related program analyses and profiling tools benefit from better understanding what kinds of problems exist in practice and how developers address them, e.g., to steer future work towards relevant problems. Third, developers of JavaScript engines benefit from learning about recurring bottlenecks that an engine may want to address and by better understanding how performance issues evolve over time.

To address these questions, we present an empirical study of performance issues and optimizations in real-world JavaScript projects. The study involves 98 fixed issues that developers have documented in bug tracking systems. The issues come from 16 JavaScript projects, including both client-side and server-side code, popular libraries, and widely used application frameworks.

Our main findings are the following:

- The most prevalent root cause of performance issues (52% of all issues) is that JavaScript provides many functionally redundant but performance-wise different APIs, which are often used in a suboptimal way by developers. This finding suggests that developers need guidance in choosing between such APIs, and that future language and API designs may want to reduce the amount of redundancy of APIs.
- Many optimizations affect a small number of source code lines: 28% and 73% of all optimizations affect less than 5 and 20 lines, respectively.
- Many optimizations do not significantly affect the complexity of the source code: 37.11% of all optimizations do not change the number of statements and 47.42% of all optimizations do not change the cyclomatic complexity of the program. This finding challenges the common belief that improving the performance of a program often implies reducing its understandability and maintainability.
- Only 42.68% of all "optimizations" applied by developers provide consistent performance improvements across all studied JavaScript engines. A non-negligible part (15.85%) of changes even degrades performance on some engines. These findings reveal a need for techniques to reliably measure performance and to monitor the performance effect of changes across multiple execution environments.
- Many optimizations are instances of recurring patterns that can be re-applied within the same project and even across projects: 29 of the 98 studied issues are instances of patterns that reoccur within the study. Furthermore, we find 139 previously unreported instances of optimization patterns in the studied projects.
- Most optimizations cannot be easily applied in a fully automatic way, mostly due to the dynamism of JavaScript. We identify five kinds preconditions for safely applying recurring optimization patterns. Statically checking whether these preconditions are met is a challenge. Our results suggest a need for tools that help developers applying recurring optimizations.

In summary, this paper contributes the following:

- The first in-depth study of JavaScript performance issues and the optimizations that developers apply to fix them.
- A documented set of 98 reproduced issues that may serve as a reference point for work on finding and fixing performance bottlenecks. All data and code gathered during the study is available for download.[2]
- Evidence that developers need tools and techniques to improve performance, e.g., to choose between redundant APIs, to apply recurring optimization patterns, and to reliably measure performance improvements.

# 2. METHODOLOGY

This section summarizes the subject projects we use in the empirical study, our criteria for selecting performance issues, and our methodology for evaluating the performance impact of the optimizations applied to address these issues.

## 2.1 Subject Projects

We study performance issues from widely used JavaScript projects that match the following criteria:

- *Project type.* We consider both node.js projects and client-side frameworks and libraries.
- *Open source.* We consider only open source projects to enable us and others to study the source code involved in the performance issues.
- *Popularity.* For node.js projects, we select modules that are the most depended-on modules in the popular npm repository.[3] For client-side projects, we select from the most popular JavaScript projects on GitHub.
- *Number of reported bugs.* We focus on projects with a high number of pull requests ($\geq 100$) to increase the chance to find performance-related issues.

Table 1 lists the studied projects, which kind of platform each project targets, and each project's number of lines of JavaScript code. Overall, we consider 16 open source projects with a total of 63,951 lines of code.

## 2.2 Selection of Performance Issues

We select performance issues from the bug trackers as follows:

1. *Keyword-based search or explicit labels.* One of the studied projects, Angular.js, explicitly labels performance issues, so we focus on them. For all other projects, we search the title, description, and comments associated with all issues for performance-related keywords, such as: "performance", "optimization", "responsive", "fast", and "slow".
2. *Random selection or inspection of all issues.* For the project with explicit performance labels, we inspect all such issues. For all other projects, we randomly sample at least 15 issues that match the keyword-based search, or we inspect all issues if there are less than 15 matching issues.
3. *Confirmed and accepted optimizations.* We consider an optimization only if it has been accepted by the developers of the project and if it has been integrated into the code repository.

[2] `https://github.com/marijaselakovic/JavaScriptIssuesStudy`
[3] `https://www.npmjs.com/browse/depended`

Table 1: Projects used for the study.

| Project | Description | Kind of platform | Total LOC | Number of issues |
|---|---|---|---|---|
| Angular.js | MVC framework | client | 7,608 | 27 |
| jQuery | Feature-rich library | client | 6,348 | 9 |
| Ember.js | MVC framework | client | 21,108 | 11 |
| React | A declarative library | client | 10,552 | 5 |
| Underscore | Utility library | client and server | 1,110 | 12 |
| Underscore.string | String manipulation helper | client and server | 901 | 3 |
| Backbone | MVC framework | client and server | 1,131 | 5 |
| EJS | Embedded templates | client and server | 354 | 3 |
| Moment | Date manipulation library | client and server | 2,359 | 3 |
| NodeLruCache | Caching support library | client and server | 221 | 1 |
| Q | Library for asynchronous promises | client and server | 1,223 | 1 |
| Cheerio | jQuery implementation for server | server | 1,268 | 9 |
| Chalk | Terminal string styling library | server | 78 | 3 |
| Mocha | Test framework | server | 7,843 | 2 |
| Request | HTTP request client | server | 1,144 | 2 |
| Socket.io | Realtime application framework | server | 703 | 2 |
| Total | | | 63,951 | 98 |

4. *Reproducibility.* We study a performance issue only if we succeed in executing a test case that exercises the code location $l$ reported to suffer from the performance problem. We use of the following kinds of tests.
   - A test provided in the issue report that reproduces the performance problem.
   - A unit test published in the project's repository that exercises $l$.
   - A newly created unit test that calls a public API function that triggers $l$.
   - A microbenchmark that contains the code at $l$, possibly prefixed by setup code required to exercise the location.
5. *Split changes into individual optimizations.* Some issues reported in bug tracking systems, such as complaints about the inefficiency of a particular function, are fixed by applying multiple independent optimizations. Because our study is about individual performance optimizations, we consider such issues as multiple issues, one for each independent optimization.
6. *Statistically significant improvement.* We apply the test that triggers the performance-critical code location to the version of the project before and after applying the optimization. We measure the execution times and keep only issues where the optimization leads to a statistically significant performance improvement.

The rationale for focusing on unit tests and microbenchmarks in step 4 is twofold. First, JavaScript developers extensively use microbenchmarks when deciding between different ways to implement some functionality.[4] Second, most projects we study are libraries or frameworks, and any measurement of application-level performance would be strongly influenced by our choice of the application that uses the library or framework. Instead, focusing on unit tests and microbenchmarks allows us to assess the performance impact of the changed code while minimizing other confounding factors.

In total, we select and study 98 performance issues, as listed in the last column of Table 1.

[4]For example, `jsperf.com` is a popular microbenchmarking web site.

## 2.3 Performance Measurement

Reliably measuring the performance of JavaScript code is a challenge, e.g., due to the influence of JIT compilation, garbage collection, and the underlying operating system. To evaluate to what extent an applied optimization affects the program's performance we adopt a methodology that was previously proposed for Java programs [9]. In essence, we repeatedly execute each test in $N_{VM}$ newly launched VM instances. At first, we perform $N_{warmUp}$ test executions in each VM instance to warm up the JIT compiler. Then, we repeat the test $N_{measure}$ more times and measure its execution times. To determine whether there is a statistically significant difference in execution time between the original and the optimized program we compare the sets of measurements $M_{before}$ and $M_{after}$ from before and after the optimization. If and only if the confidence intervals of $M_{before}$ and $M_{after}$ do not overlap, we consider the difference to be statistically significant. Based on preliminary experiments we use $N_{warmUp} = 5$, $N_{measure} = 10$, and $N_{VM} = 5$, because these parameters repeat measurements sufficiently often to provide stable performance results. We set the confidence level to 95%. Because very short execution times cannot be measured accurately, we wrap each test in a loop that makes sure that it executes for at least 5ms. All experiments are performed on an Intel Core i7-4600U CPU (2.10GHz) machine with 16GB of memory running Ubuntu 14.04 (64-bit).

## 2.4 JavaScript Engines

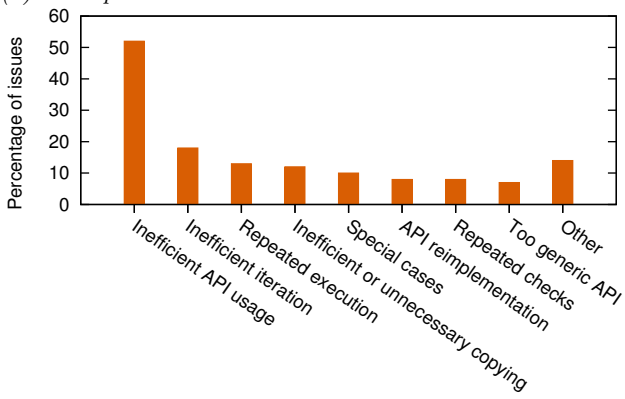
JavaScript engines evolve quickly, e.g., by adding novel JIT optimizations [7, 19, 12, 5, 1] or by adapting to trends in JavaScript development[5]. To understand how the performance impact of an optimization evolves over time, we measure the performance of tests on multiple engines and versions of engines. Table 2 lists the engines we consider. We focus on the two most popular engines: V8, which is used, e.g., in the Chrome browser and the node.js platform, and SpiderMonkey, which is used, e.g., in the Firefox browser and the GNOME desktop environment. For each engine, we use at least three different versions, taking into account only versions that are published after introducing JIT compilation, and including the most recent published version. All considered versions are published in different years and we

[5]`http://asmjs.org`

Table 2: JavaScript engines used to test performance impact of optimizations (SM = SpiderMonkey).

| | Engine version | Platform | Project type |
|---|---|---|---|
| SM | 24 | Firefox | client |
| | 31 | Firefox | client |
| | 39 | Firefox | client |
| V8 | 3.14 | Node.js | server |
| | 3.19 | Chrome | client |
| | 3.6 | Chrome and node.js | client and server |
| | 4.2 | Chrome and io.js | client and server |

*(a) Most prevalent root causes.*




*(b) APIs that are used inefficiently.*

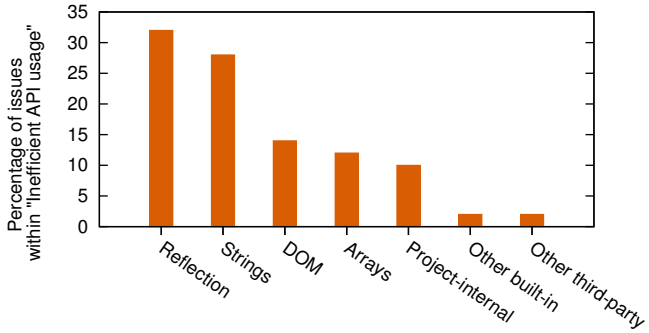



Figure 1: Root causes of performance issues.

take engines for which their version number indicates that the engine potentially introduces significant changes compared to the previous selected version. Table 2 lists for each engine which types of projects it support. We execute the tests of a project on all engines that match the kind of platform (client or server), as listed in column three of Table 1.

# 3. RQ 1: ROOT CAUSES OF PERFORMANCE ISSUES

This section addresses the question which root causes real-world performance issues have. To address this question, we identify eight root causes that are common among the 98 studied issues, and we assign each issue to one or more root causes. Figure 1 summarizes our findings, which we detail in the following.

## 3.1 API-related Root Causes

The root cause of 65 performance issues is related to how the program uses or does not use an API. We identify three specific root causes related to API usage.

### *Inefficient API usage.*

The most common root cause (52% of all issues), is that an API provides multiple functionally equivalent ways to achieve the same goal, but the API client does not use the most efficient way to achieve its goal. For example, the following code aims at replacing all quotes in a string with escaped quotes, by first splitting the string into an array of substrings and then joining the array elements with the quote character:[6]

```
.. =  str.split("'").join("\\'");
```

The optimization applied to avoid such performance issues is to use the API in a more efficient way. For the example, the developers modify the code as follows:

```
.. = str.replace(/'/g, "\\'");
```

Inefficient API usage is the most prevalent root cause, with a total of 50 issues. Figure 1b further classifies these issues by the API that is used inefficiently. The most commonly misused APIs are reflection APIs, such as runtime type checks, invocations of function objects, and checks whether an object has a particular property. The second most common root cause is inefficient use of string operations, such as the above example.

### *Inefficient reimplementation.*

The root cause of 8% of all issues is that the program implements some functionality that is already implemented in a more efficient way, e.g., as part of the built-in API. The optimization applied to avoid such performance issues is to use the existing, more efficient implementation. For example, Angular.js had implemented a `map()` function that applies a given function to each element of an array. Later, the developers optimize the code by using the built-in `Array.prototype.map()` function, which implements the same functionality.[7]

### *Generic API is inefficient.*

Another recurring root cause (7% of all issues) is to use an existing API that provides a more generic, and therefore less efficient, functionality than required by the program. For example, given a negative number $n$, the following code accesses the $|n|$th-to-last element of the array `arr`:[8]

```
arr.slice(n)[0]
```

The code is correct but inefficient because `slice()` copies parts of the array into another array, of which only the first element is used. The optimization applied to avoid such performance issues is to implement the required functionality without using the existing API. For the example, the developers improve performance by directly accessing the required element:

```
arr[arr.length + n]
```

[6]Issue 39 of Underscore.js.
[7]Issue 9067 of Angular.js.
[8]Issue 102 of jQuery.

## 3.2 Other Root Causes

Besides API-related problems, we identify six other common causes of performance issues.

*Inefficient iteration.*

JavaScript provides various ways to iterate over data collections, such as traditional `for` loops, `for-in` loops, and the `Array.prototype.forEach()` method. A common root cause of poor performance (18% of all issues) is that a program iterates over some data in an inefficient way. The optimization applied to avoid such performance issues is to iterate in a more efficient way. For example, the following code iterates through all properties of `arg` using a `for in`-loop:[9]

```
for (var prop in arg) {
  if (arg.hasOwnProperty(prop)) {
    // use prop
  }
}
```

This iteration is inefficient because it requires to check whether the property is indeed defined in `arg` and not inherited from `arg`'s prototype. To avoid checking each property, the developers optimize the code by using `Object.keys()`, which excludes inherited properties:

```
var updates = Object.keys(arg);
for (var i = 0, l = updates.length; i < l; i++) {
  var prop = updates[i];
  // use prop
}
```

*Repeated execution of the same operations.*

13% of all issues are caused by a program that repeatedly performs the same operations, e.g., during different calls of the same function. For example, the following code repeatedly creates a regular expression and uses it to split a string:[10]

```
function on(events, ...) {
  events = events.split(/\s+/);
  ...
}
```

The code is inefficient because creating the regular expression is an expensive operation that is repeatedly executed. The optimization applied to avoid such performance issues is to store the results of the computation for later reuse, e.g., through memoization [39].

For the above example, the developers compute the regular expression once and store it into a variable:

```
var eventSplitter = /\s+/;
function on(events, ...) {
  events = events.split(eventSplitter);
  ...
}
```

*Unnecessary or inefficient copying of data.*

Another recurrent root cause (12% of all issues) is to copy data from one data structure into another in an inefficient or redundant way [40]. The optimization applied to avoid such performance issues is to avoid the copying or to implement it more efficiently. For example, a function in Angular.js used to copy an array by explicitly iterating through it and by appending each element to a new array.[11] The developers optimize this code by using the built-in `Array.prototype.slice()` method, which is a more efficient way to obtain a shallow copy of an array.

[9]Issue 11338 of Ember.js.
[10]Issue 1097 of Backbone.
[11]Issue 9942 of Angular.js.

*A computation can be simplified or avoided in special cases.*

10% of all issues are due to code that performs a computation that is unnecessarily complex in some special case. The optimization applied to avoid such performance issues is to check for the special case and to avoid or to simplify the computation. For example, the developers of Angular.js used `JSON.stringify(value)` to obtain a string representation of a value. However, the value often is a number and calling `stringify()` is unnecessarily complex in this case.[12] The developers optimize the code by checking the runtime type of the value and by using the much cheaper implicit conversion into a string, `""+value`, when the value is a number.

*Repeated checks of the same condition.*

Several issues (8%) are because the program repeatedly checks the same condition, even though some of the checks could be avoided. For example, the following code repeatedly checks whether a given object is a function, which is inefficient because the object cannot change between the checks.[13]

```
function invoke(obj, method) {
  _.map(obj, function(value) {
    isFunc = _.isFunction(method)
    ...
  });
}
```

The optimization applied to avoid such performance issues is to refactor the control flow in such a way that the check is performed only once. For the above example, the developers hoist the `isFunction()` check out of the `map()` call.

Our analysis of root causes shows that various performance issues can be mapped to a relatively small number of recurring root causes. Some but not all of these root causes have been addressed by existing approaches on automatically finding performance problems [10, 39, 40]. Our results suggest that there is a need for additional techniques, in particular, to help developers choose among multiple functionally equivalent ways to use an API.

# 4. RQ 2: COMPLEXITY OF OPTIMIZATIONS

This section addresses the question how complex the source code changes are that developers apply to optimize their programs. To address this question, we analyze the project's code before and after each optimization. We study both the complexity of the changes themselves (Section 4.1) and to what degree applying these changes affects the complexity of the program's source code (Section 4.2).

## 4.1 Complexity of Changes

To assess the complexity of changes applied as optimizations, we measure for each change the number of affected lines of source code, i.e., the sum of the number of removed lines and the number of added lines. To avoid biasing these measurements towards particular code formatting styles, we apply them on a normalized representation of the source code. We obtain this representation by parsing the code and pretty-printing it in a normalized format that does not include comments.

[12]Issue 7501 of Angular.js.
[13]Issue 928 of Underscore.js.

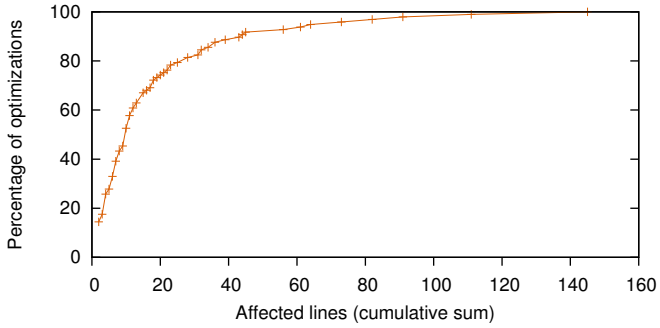



Figure 2: Number of source code lines that are affected by optimizations.

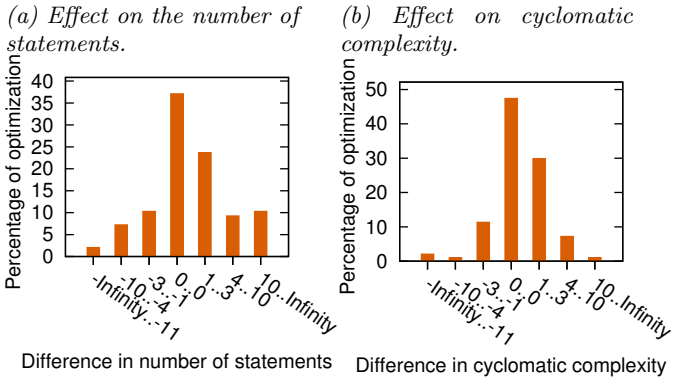



Figure 3: Effect of applying an optimization on the cyclomatic complexity.

We find that optimizations affect between 2 and 145 lines of JavaScript source code, with a median value of 10. Figure 2 shows the cumulative sum of the number of affected lines per change. The graphs shows that 73% of all optimizations affect less than 20 lines of code, and that 28% of all optimizations affect even less than 5 lines of code. We conclude from these results that a significant portion of optimizations are possible with relatively simple changes, which empirically confirms an assumption made by prior research on performance bug detection [15, 29, 28].

## 4.2 Change in Complexity of Program

To understand to what degree optimizations influence the complexity of the source code of the optimized program, we measure the number of statements in the program and the cyclomatic complexity [23] of the program before and after each change. These metrics approximate the understandability and maintainability of the code. For each change, we obtain the metric before and after the change, $n_{before}$ and $n_{after}$, and we summarize the effect of the change as $n_{after} - n_{before}$. A positive number indicates that the change increases the complexity of the program because the changed program contains additional statements or increases the cyclomatic complexity, whereas a negative number indicates that the program becomes less complex due to the change.

Figures 3a and 3b summarize our results. The graphs show what percentage of optimizations affect the number of statements and the cyclomatic complexity in a particular range. For example, Figure 3a shows that 24% of all optimizations add between one and three statements to the

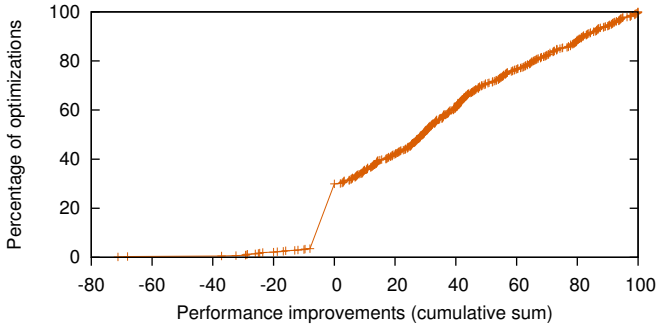



Figure 4: Performance improvements obtained by optimizations (cumulative sum).

program. We find that a large portion of all optimizations do not affect the number of statements and the cyclomatic complexity at all: 37.11% do not modify the number of statements, and 47.42% do not modify the cyclomatic complexity. A manual inspection of these optimizations shows that they modify the code in minor ways, e.g., by moving a statement out of a loop, by adding an additional subexpression, or by replacing one function call with another. It is also interesting to note that a non-negligible percentage of optimizations decreases the number of statements (19.59%) and the cyclomatic complexity (14.43%). These results challenge the common belief that optimizations come at the cost of reduced code understandability and maintainability. We conclude from these results that many optimizations are possible without increasing the complexity of the optimized program.

# 5. RQ 3: PERFORMANCE IMPACT OF OPTIMIZATIONS

The following addresses the question which performance impact developers achieve by optimizing their programs. To address this question, we execute the test of all 98 optimizations on all considered JavaScript engines where the respective code can be executed (Section 2). In total, we obtain 568 performance improvement results.

Figure 4 shows the improvements obtained by optimizations as the cumulative sum over all 568 performance results. Perhaps surprisingly, the figure shows that 3.52% of all "optimizations" cause a performance degradation and that 26.41% provide no statistically significant improvement on some JavaScript engines. We further analyze these cases in Section 6. For the remaining performance results, the figure shows that optimizations lead to a wide range of improvements: For example, 5.63% of the optimizations save up to 10% of the execution time, and 5.81% of all optimizations save over 90% of the execution time.

Given these results and the results from Section 4, one may wonder whether there is any correlation between the "pain" and the "gain" of optimizations, i.e., between the number of lines affected by a change and the performance improvement that the change yields. To address this question, Figure 5 shows the relation between these two metrics for all issues. The figure does not show any correlation (Pearson's correlation coefficient: 5.85%).

We draw three conclusions from our results. First, developers apply some optimizations even though the achieved performance impact is relatively small. This strategy seems

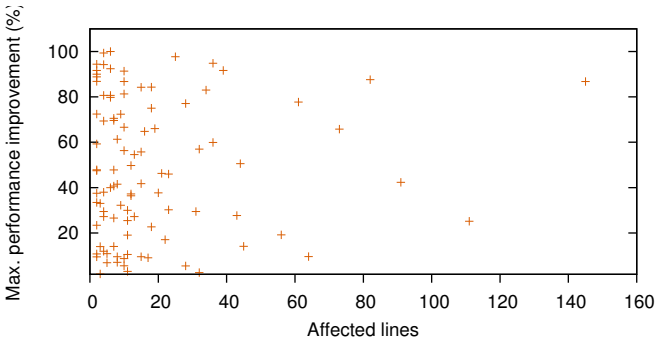


Figure 5: Relation between the number of lines affected by a change and the achieved performance improvement.

reasonable, e.g., when the modified code is in a heavily used library function, but may also be a sign for "premature optimizations" [16]. Second, developers apply some optimizations even though these optimizations cause a performance degradation on some JavaScript engines, either consciously or without knowing what impact a change has. Third, some optimizations lead to significant savings in execution time, and future work on profiling should pinpoint such optimization opportunities to developers.

# 6. RQ 4: CONSISTENCY ACROSS ENGINES AND VERSIONS

Since different JavaScript engines apply different optimization strategies, developers that change some code to improve performance on one engine risk to degrade it on another engine. Furthermore, since engines evolve quickly, an optimization applied to speed up the program on one version of an engine may have the opposite effect on another version of the same engine. To assess to what degree developers struggle with these risks, this section addresses the question how consistent performance improvements are across different JavaScript engines and across multiple versions of the same engine.

Similar to RQ 3, we address this question by measuring the performance impact of the performance issues on all considered JavaScript engines. Since we want to compare performance impacts across engines, we include only issues that we can execute in both V8 and SpiderMonkey, i.e., we exclude non-browser optimizations. In total, we consider 82 issues for this research questions.

## 6.1 Consistency Across Engines

Table 3 compares the performance impact of changes on the V8 and SpiderMonkey engines. For each engine, the table distinguishes five cases: + means that a change improves performance in all versions of the engine, +0 means that the change improves performance or does not affect performance, +- means that the change improves performance in some version but degrades it in another version, 0- means no performance change or a performance degradation, and finally, - means a negative performance impact in all versions. For each combination of these five cases, the table shows the percentages of changes that fall into the respective category. Because the study includes only issues that provide an improvement in at least one engine (Section 2.2),

| | | SpiderMonkey | | | | |
|---|---|---|---|---|---|---|
| | | + | +0 | +- | 0- | - |
| V8 | + | 42.68 | 8.5 | 0 | 0 | 0 |
| | +0 | 13.4 | 19.5 | 1.2 | 3.7 | 0 |
| | +- | 4.9 | 0 | 3.7 | 1.2 | 0 |
| | 0- | 1.2 | 0 | 0 | – | – |
| | - | 0 | 0 | 0 | – | – |

Table 3: Percentage of optimizations that result in positive (+), positive or no (+0), positive or negative (+-), no or negative (0-), and negative (-) speedup in V8 and SpiderMonkey.

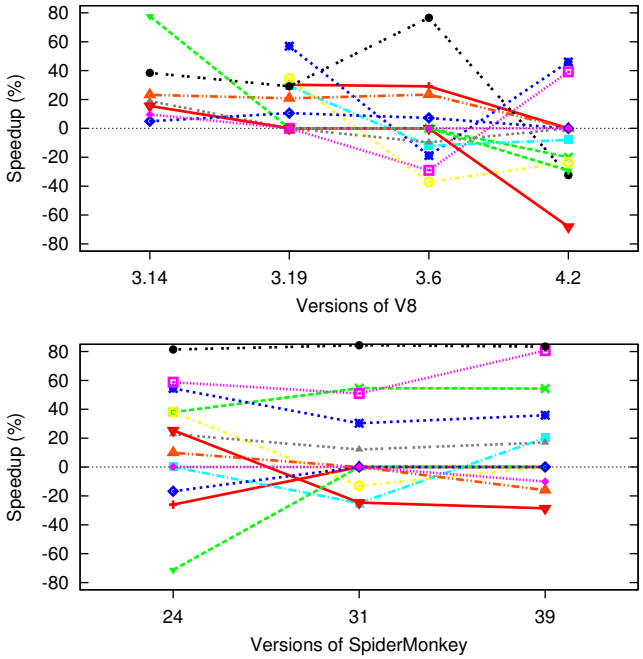


Figure 6: Speedup of changes in different versions of engines. Each line represents one performance optimization (same line style in both figures means the same optimization).

the four table cells in the bottom right corner of the table cannot occur.

We find that only 42.68% of all changes speed up the program in all versions of both engines, which is what developers hope for when applying an optimization. Even worse, 15.85% of all changes degrade the performance on at least one engine, i.e., a change supposed to speed up the program may have the opposite effect. Interestingly, a non-negligible percentage of changes speed up the program in one engine but cause slowdown in another. For example, 4.9% of all changes that increase performance in all versions of SpiderMonkey cause a slowdown on at least one version of V8. Some changes even degrade performance in some versions of both engines. For example, 3.7% of all changes may have a positive effect in V8 but will either decrease or not affect performance in SpiderMonkey.

## 6.2 Consistency Across Versions of an Engine

To better understand how the performance impact of a change evolves across different versions of a particular engine, Figure 6 shows the speedups of individual changes in

the V8 (top) and SpiderMonkey (bottom) engines. For readability, the figure includes only those 15.85% of all changes that cause a slowdown on at least one version of some engine. The graphs show that performance can differ significantly across different versions. For example, a change that provides an almost 80%-speedup in version 3.6 of the V8 engine causes a non-negligible slowdown in version 4.2 of the same engine. The graphs also show that the performance impact of a change sometimes evolves in a non-monotonic way. That is, a beneficial optimization may turn into a performance degradation and then again into a beneficial optimization.

In summary, our results show that performance is a moving target. This finding motivates future work that supports developers in achieving satisfactory performance despite the heterogeneity of JavaScript engines, such as techniques to support developers in deciding when to apply an optimization, to reliably and automatically measure the performance effect of a change, to track the effect of an optimization over time, to undo an optimization that turns out to be counterproductive in a new engine, and to specialize JavaScript code for particular engines.

Table 5: Instances of recurring optimization patterns (a|b, a = number of pattern instances in the 98 studied issues, b = number of previously unreported pattern instances found with static analysis).

| Id | Projects | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Ember | React | Less | Mocha | Angular | Underscore.string | EJS | Socket.io | Moment | Cheerio | Underscore | Request | Chalk | Total |
| 1 | 1\|20 | 0\|21 | 0\|12 | 1\|0 | 2\|2 | 0\|0 | 0\|0 | 0\|1 | 0\|0 | 0\|0 | 4\|0 | 0\|0 | 0\|0 | 8\|56 |
| 2 | 0\|0 | 0\|0 | 0\|1 | 0\|0 | 0\|1 | 0\|0 | 6\|1 | 0\|0 | 0\|0 | 0\|0 | 4\|0 | 0\|0 | 0\|0 | 6\|3 |
| 3 | 0\|6 | 0\|1 | 0\|3 | 0\|6 | 0\|2 | 2\|7 | 0\|7 | 0\|1 | 0\|0 | 0\|1 | 0\|0 | 0\|0 | 0\|1 | 2\|35 |
| 4 | 0\|0 | 0\|0 | 0\|0 | 0\|0 | 6\|0 | 0\|0 | 0\|0 | 0\|0 | 0\|0 | 0\|0 | 0\|0 | 0\|0 | 0\|0 | 6\|0 |
| 5 | 0\|2 | 0\|0 | 0\|0 | 0\|0 | 2\|0 | 0\|0 | 0\|0 | 0\|0 | 0\|0 | 0\|0 | 0\|0 | 0\|0 | 0\|0 | 2\|2 |
| 6 | 0\|0 | 0\|0 | 0\|0 | 0\|0 | 0\|0 | 0\|0 | 0\|0 | 0\|0 | 0\|0 | 0\|0 | 1\|0 | 0\|0 | 0\|0 | 1\|0 |
| 7 | 0\|9 | 0\|0 | 0\|1 | 1\|4 | 0\|0 | 0\|0 | 0\|0 | 0\|0 | 0\|3 | 0\|2 | 0\|2 | 0\|0 | 0\|0 | 1\|21 |
| 8 | 0\|2 | 0\|0 | 0\|0 | 0\|0 | 1\|1 | 0\|0 | 0\|0 | 0\|0 | 0\|0 | 0\|0 | 0\|0 | 0\|0 | 0\|0 | 1\|3 |
| 9 | 0\|0 | 0\|0 | 0\|1 | 1\|0 | 0\|0 | 0\|0 | 0\|0 | 0\|0 | 0\|0 | 0\|2 | 0\|0 | 0\|2 | 0\|0 | 1\|5 |
| 10 | 0\|1 | 0\|0 | 0\|2 | 0\|0 | 0\|3 | 0\|1 | 0\|0 | 0\|5 | 0\|2 | 0\|0 | 0\|0 | 0\|0 | 1\|0 | 1\|14 |
| Total | 1\|40 | 0\|22 | 0\|20 | 3\|10 | 11\|9 | 2\|8 | 6\|8 | 0\|7 | 0\|5 | 2\|5 | 5\|2 | 0\|2 | 1\|1 | 29\|139 |

# 7. RQ 5: RECURRING OPTIMIZATION PATTERNS

The following addresses the question whether there are recurring optimization patterns and whether they can be applied automatically. To address the first part of the question, we manually identify a set of optimizations that can be re-applied across several projects and semi-automatically search for instances of these *optimization patterns* beyond the 98 studied issues (Section 7.1). To address the second part of the question, we identify pre-conditions for applying the recurring optimization patterns in a fully automated way (Section 7.2).

## 7.1 Prevalence of Recurring Optimization Patterns

To identify performance optimizations that may apply in more than a single situation we inspect all 98 issues. First, we identify optimizations that occur repeatedly within the study (Table 4, Patterns 1–5). Second, since the 98 studied issues may not expose multiple instances of a pattern that would occur repeatedly in a larger set of issues, we also identify patterns that may occur repeatedly (Table 4, Patterns 6–10). For each optimization pattern in Table 4, we provide a short description, as well as an example of code before and after the optimization.

In Table 5, the numbers before the vertical lines show how often each optimization pattern occurs within the 98 issues considered in the study. For instance, there are two instances of Pattern 1 (avoid `for-in` loops) in the Angular.js project. The last column shows that the studied issues contain eight optimizations that match Pattern 1. In total, 29 of the 98 studied optimizations match one of the 10 optimization patterns.

To study whether there are occurrences of the optimization patterns beyond the 98 studied optimizations, we develop a simple, AST-based, static analysis for each pattern in Table 4. Each such analysis performs an AST traversal of a JavaScript program to find matches of the pattern's AST in the AST of the program. Due to the well-known difficulties of statically analyzing JavaScript in a sound way (Section 7.2 will provide more details), the analyses cannot guarantee that the optimization patterns can indeed be applied at the identified code locations without changing the program's semantics. To enable us to manually check whether a match is a valid optimization opportunity, the analyses also rewrite the program by applying the respective optimization pattern. We then manually inspect the rewritten program and prune changes that would modify the program's semantics.

We apply the ten analyses to the current version of each of the subject projects. In total, the analyses suggest 142 optimizations, of which we manually prune 3 changes because they would break the semantics of the program. In Table 5, the numbers after the vertical lines show how many previously unreported optimization opportunities the analyses finds for each project. We omit four projects for which we do not find any match of any pattern.

In total, we find 139 instances of recurring optimization patterns. The most common patterns are Pattern 1 (avoid `for-in` loop), Pattern 3 (use implicit string conversion), and Pattern 7 (use `instanceof`). For two patterns (4 and 6), the analyses do not find any previously unreported instances. The results show that patterns are recurring within a project. For example, Pattern 1 has been applied only once in the Ember project by the developers, but we find 21 additional code locations that offer the same optimization opportunity. Furthermore, the results show that recurring patterns can be applied across projects. For example, the only instance of Pattern 7 within the 98 studied issues is in the Mocha project, but there are 17 additional instances in other projects.

We conclude from these results that there is a need for tools and techniques that help developers apply an already performed optimization at other code locations, both within the same project and across projects, possibly along the lines of existing work [24, 25, 4].

Table 4: Recurring optimization patterns and pre-conditions for applying them (T = Type check, NF = Native function is not overridden, P = Prototype is not overridden, TF = Function from third-party library is not overridden, V = Check on value of an expression).

| Id | Description | Example | | Preconditions | | | | |
|---|---|---|---|---|---|---|---|---|
| | | Before | After | T | NF | P | TF | V |
| 1 | Prefer `Object.keys()` over computing the properties of an object with a `for-in` loop. | `for (var key in obj) {`<br>`  if (obj.hasOwnProperty(key))`<br>`  { ... }`<br>`}` | `var keys = Object.keys(obj);`<br>`for (var i=0, l=keys.length;`<br>`  i<l; i++) {`<br>`  var key=keys[i];`<br>`  ...`<br>`}` | ● | ● | ● | ○ | ○ |
| 2 | To extract a substring of length one, access the character directly instead of calling `substr()`. | `str.substr(i, 1)` | `str[i]` | ● | ○ | ● | ○ | ○ |
| 3 | To convert a value into a string, use implicit type conversion instead of `String()`. | `starts = String(starts);` | `starts = '' + starts;` | ○ | ● | ○ | ● | ○ |
| 4 | Use jQuery's `empty()` instead of `html('')`. | `body.html('');` | `body.empty();` | ● | ○ | ○ | ● | ○ |
| 5 | Use two calls of `charAt()` instead of `substr()`. | `key.substr(0, 2) !== '$$'` | `key.charAt(0) !== '$' &&`<br>`  key.charAt(1) !== '$'` | ● | ● | ● | ○ | ○ |
| 6 | To replace parts of a string with another string, use `replace()` instead of `split()` and `join()`. | `str.split("'").join("\\'")` | `str.replace(/'/g, "\\'")` | ● | ● | ● | ○ | ○ |
| 7 | Instead of checking an object's type with `toString()`, prefer the `instanceof` operator. | `if (toString.call(err)`<br>`  === "[object Error]") ...` | `if (err instanceof Error ||`<br>`  toString.call(err)`<br>`  === "[object Error]") ...` | ○ | ● | ● | ○ | ● |
| 8 | For even/odd checks of a number use `&1` instead of `%2`. | `index % 2 == 0` | `index & 1 == 0` | ● | ○ | ○ | ○ | ○ |
| 9 | Prefer `for` loops over functional-style processing of arrays. | `styles.reduce(`<br>`  function (str, name) {`<br>`  return ...;`<br>`}, str);` | `for (var i=0; i< styles.length;`<br>`  i++) {`<br>`  var name=styles[i];`<br>`  str = ...; }`<br>`return str;` | ● | ● | ● | ○ | ○ |
| 10 | When joining an array of strings, handle single-element arrays efficiently. | `[].slice.call(arguments)`<br>`  .join(' ');` | `arguments.length === 1 ?`<br>`  arguments[0] + '' :`<br>`  [].slice.call(arguments)`<br>`  .join(' ');` | ● | ○ | ● | ○ | ○ |

## 7.2 Preconditions For Automatic Transformation

The second part of the RQ 5 is whether recurring optimization patterns can be applied automatically. Answering this question is an important first step for developing techniques that help developers find optimization opportunities based on recurring patterns, and for developers of JIT engines who may want to address some of these patterns in the engine. To address the question, we identify for each optimization pattern the preconditions that must be satisfied to safely apply the optimization in an automatic way. We find the following kinds of preconditions:

- **Type check**. Check the type of an identifier or expression. For example, the `obj` identifier in Pattern 1 must have type `Object`.
- **Native function is not overridden**. Check that a built-in JavaScript function is not overridden. For example, in Pattern 1, both `hasOwnProperty()` and `keys()` must be the built-in JavaScript functions.
- **Prototype is not overridden**. Check that certain properties of the prototype are not overridden. For example, for Pattern 1, the `hasOwnProperty` property of `Object.prototype` must not be overridden.
- **Function from third-party library is not overridden**. Check that a function from a third-party library is not overridden. For example, the `html()` and `empty()` functions in Pattern 4 must be functions from the jQuery library.
- **Check the value of an expression**. Check whether an expression has a particular value. For example, to apply Pattern 7, the `toString` variable must refer to the `Object.prototype.toString()` method.

The last columns of Table 4 show which kinds of preconditions need to be satisfied to automatically and safely apply the optimization patterns. Due to the dynamic features of the JavaScript language, it is challenging to statically analyze whether these preconditions are met. Possible solutions include a more sophisticated static analyses or dynamic checks that ensure that the conditions are met at runtime. We conclude from the fact that each pattern is subject to several kinds of preconditions that applying optimizations in JavaScript in a fully automatic way is not trivial, and that

finding techniques that address this challenge is subject to future work.

# 8. THREATS TO VALIDITY

*Subject Projects.*

Our study focuses on 16 open source projects and the results may not be representative for closed source projects or other open source projects. Furthermore, we consider projects written in JavaScript only, and our conclusions are valid for this language only.

*Performance Tests.*

We measure the performance impacts of optimizations with unit tests and microbenchmarks, and the application-level performance impact of the optimizations may differ. We believe that our measurements are worthwhile because JavaScript developers heavily use unit tests and microbenchmarks to make performance-related decisions.

*JavaScript Engines and Platforms.*

We evaluate performance impacts on several versions of two popular JavaScript engines, which implement JIT compilation. Our results may not generalize to other JIT engines, such as Chakra, which is used in Internet Explorer, or interpreter-based engines. Since the most popular server-side platform, node.js, and popular browsers build upon the V8 or SpiderMonkey engines, we expect that our measurement are relevant for developers.

*Underapproximation of Recurring Optimization Patterns.*

Our methodology for findings instances of recurring optimization patterns may miss instances, e.g., because the static analyses rely on naming heuristics. As a result, the number of previously unreported instances of optimization pattern is an underapproximation. Since our goal is not to precisely quantify the prevalence of recurring patterns but to answer the question whether such patterns exist at all, this limitation does not invalidate our conclusions.

# 9. RELATED WORK

## 9.1 Studies of Performance Issues

Studies show that performance bugs occur frequently [15] and that they account for a non-negligible amount of developer time [41]. Jin et al. report that many performance problems in C and C++ can be fixed by following efficiency rules [15]. Liu et al. [18] study performance bugs in smartphone applications and propose specialized program analyses that detect two common kinds of problems. Linares-Vasquez et al. [17] study how API usages on Android influence energy consumption, which is closely related to performance. Our study differs from the existing studies by studying the root causes of issues, the complexity of optimizations, the performance impact of the applied optimizations, and the evolution of the performance impact over time. Furthermore, we are the first to study performance issues in JavaScript, which differs from C, C++, and Java both on the language and the language implementation level.

## 9.2 Studies of JavaScript

Ocariza et al. [30] study the root causes and the impact of correctness bugs in JavaScript. Their earlier work [31] analyzes console logs to study the characteristics of failures. A recent study by Gallaba et al. [8] focuses on the usage of JavaScript callbacks. Other studies consider the dynamic behavior of JavaScript applications [36], the prevalence of the `eval()` function [35], JavaScript inclusions [27], and the harmfulness and prevalence of type coercions [34]. In contrast to all these studies, this paper focuses on performance issues.

## 9.3 Efficiency of JavaScript Engines

The need for efficiently executing JavaScript code is addressed by optimizing just-in-time compilers that produce type-specialized code [7, 19, 12], that efficiently represent objects [1], and that specialize functions based on previously observed parameters [5]. Our work shows that engines employ different optimization strategies and that some optimizations provide very different performance across the engines. We envision that the findings of our study help engine developers to focus on performance bottlenecks that they may currently not be aware of.

## 9.4 JavaScript Analysis

Program analyses for JavaScript include dynamic analyses to find type inconsistencies [33] and violations of code quality rules [11], dynamic determinacy analysis [37], dynamic information flow analysis [13], change impact analysis [2], combined dynamic-static analysis to check if an interface description matches a library implementation [6], static analysis of library clients [20], UI-level test generation [26, 3, 38, 32], and type analysis for JavaScript [14]. Our study motivates future research on program analyses that help developers improve the performance of JavaScript code, and provides evidence to steer such work toward relevant problems.

## 9.5 Detecting Performance Bottlenecks

There are various analyses and profiling approaches to detect performance bottleneck, such as memory bloat [40], repeated operations in loops [29], JIT-unfriendly code [10], memoization opportunities [39], repeated patterns of method calls [22], and inefficient use of collections [21]. Our work highlights common performance issues in JavaScript and provides empirical evidence to steer future approaches for detecting performance issues.

# 10. CONCLUSION

Optimizations of JavaScript code deserve attention because thousands of JavaScript applications rely on responsive and energy-efficient computations. Despite the fact that JavaScript engines keep getting faster, developers still deal with performance issues by applying optimizations to their code. This paper presents the first systematic study of real-world JavaScript performance issues. We collect, reproduce, and make available 98 issues and optimizations collected from 16 popular JavaScript projects. Our results provide insights about the most prevalent root causes of performance issues, the complexity of changes that developers apply to optimize programs, the performance impact of optimizations and its evolution over time. Furthermore, our work provides evidence that many optimizations are instances of recurring optimization patterns. By finding 139 previously unreported

optimization opportunities using patterns applicable across projects, we show a great potential for techniques to further optimize existing programs. Our results and observations motivate future work on developing techniques and tools that help developers solve performance issues, and that provide directions to steer such work toward problems that are relevant in practice.

# 11. REFERENCES


[1] W. Ahn, J. Choi, T. Shull, M. J. Garzarán, and J. Torrellas. Improving JavaScript performance by deconstructing the type system. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 496–507, 2014.

[2] S. Alimadadi, A. M. 0001, and K. Pattabiraman. Hybrid dom-sensitive change impact analysis for javascript. In *ECOOP*, pages 321–345, 2015.

[3] S. Artzi, J. Dolby, S. H. Jensen, A. Møller, and F. Tip. A framework for automated testing of JavaScript web applications. In *ICSE*, pages 571–580, 2011.

[4] M. Boshernitsan, S. L. Graham, and M. A. Hearst. Aligning development tools with the way programmers think about code changes. In *CHI*, pages 567–576, 2007.

[5] I. Costa, P. Alves, H. N. Santos, and F. M. Q. Pereira. Just-in-time value specialization. In *CGO*, pages 1–11, 2013.

[6] A. Feldthaus and A. Møller. Checking correctness of typescript interfaces for javascript libraries. In *Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 1–16. ACM, 2014.

[7] 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 *PLDI*, pages 465–478, 2009.

[8] K. Gallaba, A. Mesbah, and I. Beschastnikh. Don't call us, we'll call you: Characterizing callbacks in JavaScript. In *Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, page 10 pages. IEEE Computer Society, 2015.

[9] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous Java performance evaluation. In *Conference on Object-Oriented Programming, Systems, Languages, and Application (OOPSLA)*, pages 57–76. ACM, 2007.

[10] L. Gong, M. Pradel, and K. Sen. JITProf: Pinpointing JIT-unfriendly JavaScript code. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2015.

[11] L. Gong, M. Pradel, M. Sridharan, and K. Sen. DLint: Dynamically checking bad coding practices in JavaScript. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2015.

[12] B. Hackett and S. Guo. Fast and precise hybrid type inference for JavaScript. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 239–250. ACM, 2012.

[13] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld. JSFlow: Tracking information flow in JavaScript and its APIs. In *SAC*, pages 1663–1671, 2014.

[14] S. H. Jensen, A. Møller, and P. Thiemann. Type analysis for JavaScript. In *Proc. 16th International Static Analysis Symposium (SAS)*, volume 5673 of *LNCS*. Springer-Verlag, August 2009.

[15] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu. Understanding and detecting real-world performance bugs. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 77–88. ACM, 2012.

[16] D. E. Knuth. Computer programming as an art. *Commun. ACM*, 17(12):667–673, Dec. 1974.

[17] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk. Mining energy-greedy api usage patterns in android apps: an empirical study. In *MSR*, pages 2–11, 2014.

[18] Y. Liu, C. Xu, and S. Cheung. Characterizing and detecting performance bugs for smartphone applications. In *ICSE*, pages 1013–1024, 2014.

[19] F. Logozzo and H. Venter. RATA: Rapid atomic type analysis by abstract interpretation—application to JavaScript optimization. In *CC*, pages 66–83, 2010.

[20] M. Madsen, B. Livshits, and M. Fanning. Practical static analysis of JavaScript applications in the presence of frameworks and libraries. In *ESEC/SIGSOFT FSE*, pages 499–509, 2013.

[21] I. Manotas, L. Pollock, and J. Clause. Seeds: a software engineer's energy-optimization decision support framework. In *ICSE*, pages 503–514, 2014.

[22] D. Maplesden, E. D. Tempero, J. G. Hosking, and J. C. Grundy. Subsuming methods: Finding new optimisation opportunities in object-oriented software. In *ICPE*, pages 175–186, 2015.

[23] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, Dec. 1976.

[24] N. Meng, M. Kim, and K. S. McKinley. Systematic editing: generating program transformations from an example. In *PLDI*, pages 329–342, 2011.

[25] N. Meng, M. Kim, and K. S. McKinley. Lase: locating and applying systematic edits by learning from examples. In *ICSE*, pages 502–511, 2013.

[26] A. Mesbah, E. Bozdag, and A. van Deursen. Crawling Ajax by inferring user interface state changes. In *International Conference on Web Engineering (ICWE)*, pages 122–134, 2008.

[27] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. V. Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. You are what you include: large-scale evaluation of remote JavaScript inclusions. In *CCS*, pages 736–747, 2012.

[28] A. Nistor, P.-C. Chang, C. Radoi, and S. Lu. Caramel: Detecting and fixing performance problems that have non-intrusive fixes. In *ICSE*, 2015.

[29] A. Nistor, L. Song, D. Marinov, and S. Lu. Toddler: Detecting performance problems via similar memory-access patterns. In *International Conference on Software Engineering (ICSE)*, pages 562–571, 2013.

[30] F. S. Ocariza Jr., K. Bajaj, K. Pattabiraman, and A. Mesbah. An empirical study of client-side

JavaScript bugs. In *Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 55–64, 2013.

[31] F. S. Ocariza Jr., K. Pattabiraman, and B. Zorn. Javascript errors in the wild: An empirical study. In *Proceedings of the 2011 IEEE 22Nd International Symposium on Software Reliability Engineering*, ISSRE '11, pages 100–109, Washington, DC, USA, 2011. IEEE Computer Society.

[32] M. Pradel, P. Schuh, G. Necula, and K. Sen. EventBreak: Analyzing the responsiveness of user interfaces through performance-guided test generation. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2014.

[33] M. Pradel, P. Schuh, and K. Sen. TypeDevil: Dynamic type inconsistency analysis for JavaScript. In *International Conference on Software Engineering (ICSE)*, 2015.

[34] M. Pradel and K. Sen. The good, the bad, and the ugly: An empirical study of implicit type conversions in JavaScript. In *European Conference on Object-Oriented Programming (ECOOP)*, 2015.

[35] G. Richards, C. Hammer, B. Burg, and J. Vitek. The eval that men do - a large-scale study of the use of eval in JavaScript applications. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 52–78, 2011.

[36] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of JavaScript programs. In *Conference on Programming Language Design and Implementation, PLDI*, pages 1–12, 2010.

[37] M. Schäfer, M. Sridharan, J. Dolby, and F. Tip. Dynamic determinacy analysis. In *PLDI*, pages 165–174, 2013.

[38] S. Thummalapenta, K. V. Lakshmi, S. Sinha, N. Sinha, and S. Chandra. Guided test generation for web applications. In *International Conference on Software Engineering (ICSE)*, pages 162–171. IEEE, 2013.

[39] L. D. Toffola, M. Pradel, and T. R. Gross. Performance problems you can fix: A dynamic analysis of memoization opportunities. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2015.

[40] G. H. Xu, N. Mitchell, M. Arnold, A. Rountev, E. Schonberg, and G. Sevitsky. Finding low-utility data structures. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 174–186, 2010.

[41] S. Zaman, B. Adams, and A. E. Hassan. A qualitative study on performance bugs. In *Working Conference on Mining Software Repositories (MSR)*, pages 199–208. IEEE, 2012.