TECHNISCHE
UNIVERSITÄT
DARMSTADT

ACTIONABLE PROGRAM ANALYSES FOR IMPROVING SOFTWARE PERFORMANCE

Vom Fachbereich Informatik (FB 20)
der Technischen Universität Darmstadt
zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs (Dr.-Ing.)

genehmigte Dissertation von

Marija Selakovic, M.Sc.

geboren in Uzice, Serbien.

Referenten: Prof. Dr. Michael Pradel
Prof. Dr. Frank Tip

Tag der Einreichung: 10.12.2018
Tag der Disputation: 21.01.2019

Selakovic, Marija: Actionable Program Analyses for Improving Software Performance
Thesis written in: Darmstadt, Technische Universität Darmstadt
Year thesis published in TUprints 2019
Date of the viva voce 21.01.2019

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit - abgesehen von den in ihr ausdrücklich genannten Hilfen - selbständig verfasst habe.

Darmstadt, Deutschland, Januar 2019

Marija Selakovic

Academic CV

October 2008 - June 2012
Bachelor of Information Systems and Technology, University of Belgrade

October 2012 - August 2014
Master of Computer Science, University of L'Aquila and VU University Amsterdam

October 2014 - January 2019
PhD candidate in Computer Science, Technische Universität Darmstadt

Abstract

Nowadays, we have greater expectations of software than ever before. This is followed by constant pressure to run the same program on smaller and cheaper machines. To meet this demand, the application's performance has become the essential concern in software development. Unfortunately, many applications still suffer from performance issues: coding or design errors that lead to performance degradation. However, finding performance issues is a challenging task: there is limited knowledge on how performance issues are discovered and fixed in practice, and current performance profilers report only where resources are spent, but not where resources are wasted.

The goal of this dissertation is to investigate actionable performance analyses that help developers optimize their software by applying relatively simple code changes. To understand causes and fixes of performance issues in real-world software, we first present an empirical study of 98 issues in popular JavaScript projects. The study illustrates the prevalence of simple and recurring optimization patterns that lead to significant performance improvements. Then, to help developers optimize their code, we propose two *actionable* performance analyses that suggest optimizations based on reordering opportunities and method inlining. In this work, we focus on optimizations with four key properties. First, the optimizations are *effective*, that is, the changes suggested by the analysis lead to statistically significant performance improvements. Second, the optimizations are *exploitable*, that is, they are easy to understand and apply. Third, the optimizations are *recurring*, that is, they are applicable across multiple projects. Fourth, the optimizations are *out-of-reach for compilers*, that is, compilers can not guarantee that a code transformation preserves the original semantics. To reliably detect optimization opportunities and measure their performance benefits, the code must be executed with sufficient test inputs. The last contribution complements state-of-the-art test generation techniques by proposing a novel automated approach for generating effective tests for higher-order functions.

We implement our techniques in practical tools and evaluate their effectiveness on a set of popular software systems. The empirical evaluation demonstrates the potential of actionable analyses in improving software performance through relatively simple optimization opportunities.

Zusamenfassung

Die heutigen Erwartungen an Software sind größer als je zuvor. Vor allem der Druck bestehende Programme auf günstigerer und weniger leistungsfähiger Hardware auszuführen, lässt Performance zum zentralen Anliegen in der Softwareentwicklung werden. Und doch haben viele Anwendungen Performanceprobleme, also Design- oder Programmierfehler, die zu Leistungseinbußen führen. Eben jene Performanceprobleme zu finden, ist allerdings eine anspruchsvolle Aufgabe: Das Wissen, wie Performanceprobleme in der Praxis gefunden und behoben werden ist begrenzt und heutige Profiler zeigen nur auf, wo Ressourcen ausgegeben, aber nicht wo sie verschwendet werden.

Ziel dieser Arbeit ist es, praktisch umsetzbar Performanceanalysen zu untersuchen, die Entwicklern helfen ihre Software durch relative einfache Änderungen am Quelltext zu optimieren. Um besser zu verstehen wie Performanceproblemen verursacht und behoben werden, stellen wir zunächst eine empirische Studie über 98 Probleme in weit verbreiteten JavaScript-Projekten vor. Die Studie zeigt einfache und wiederkehrende Optimierungsmuster auf, die zu signifikanten Performanceverbesserungen führen. Darauf aufbauend stellen wir zwei praktisch umsetzbar Performanceanalysen vor, die Entwicklern helfen ihre Programme durch Reordering und durch Methoden-Inlining zu optimieren. Dabei fokussieren wir uns auf Optimierungen mit vier zentralen Eigenschaften. Erstens sind die Optimierungen effektiv, das heißt die von der Analyse vorgeschlagenen Änderungen führen zu statistisch signifikanten Leistungssteigerungen. Zweitens sind die Optimierungen nutzbar, das heißt sie sind leicht zu verstehen und anzuwenden. Drittens sind die Optimierungen wiederkehrend, das heißt projektübergreifend anwendbar. Viertens sind die Optimierungen unerreichbar für Compiler, das heißt Compiler können nicht garantieren, dass eine Code-Transformation semantikerhaltend ist. Um die Optimierungsmöglichkeiten zuverlässig zu erkennen und deren Leistungsverbesserung messen zu können, muss der Code schließlich mit ausreichend Testeingaben ausgeführt werden. Der letzte Beitrag in dieser Arbeit ergänzt Testgenerierungstechniken auf dem Stand der Technik durch einen neuen, automatisierten Ansatz zur Generierung effektiver Tests für Funktionen höherer Ordnung.

Wir implementierung die genannten Techniken in praxistauglichen Werkzeugen und bewerten deren Effektivität auf einer Reihe weit verbreiteter Softwaresysteme. Die empirische Auswertung zeigt, dass praktisch umsetzbar Performanceanalysen in der Lage sind die Leistung von Software durch relativ einfache Optimierungsvorschläge zu verbessern.

Acknowledgments

Doing PhD has been a truly life-changing experience for me and it would not have been possible to do without the support and guidance that I received from many people.

Firstly, I would like to say a very big thank you to my advisor Prof. Michael Pradel for his continuous support during my PhD studies and research, patience, encouragement, and immense knowledge. Without his guidance on conducting research projects and constant feedback, this PhD would not have been achievable.

Besides my advisor, I would like to thank the rest of my thesis committee: Prof. Frank Tip, Prof. Mira Mezini, Prof. Felix Wolf and Prof. Christian Reuter, for their insightful comments, interesting questions, and support.

My special thanks also go to great researchers: Michael Barnett, Madan Musuvathi, Todd Mytkowicz, and Andrew Begel, for offering me summer internships in their groups at Microsoft Research and leading me working on diverse and exciting projects. It was an amazing opportunity to work in such a stimulating and productive environment.

I also would like to thank my fellow labmates: Andrew, Jibesh, Christian and Daniel for all interesting, philosophical and valuable discussions and the fun we had in the last four and a half years.

Last but not least, I would like to thank my husband Jovan, my family and friends back home in Serbia and across the world for their support, patience, and love all these years.

Contents

List of Figures

List of Tables

List of Algorithms

Introduction

Regardless of the domain, software performance is one of the most important aspects of software quality: it is important to ensure an application's responsiveness, high throughput, efficient loading, scaling, and user satisfaction. Poorly performing software wastes computational resources, affects perceived quality and increases maintenance cost. Furthermore, a web application that is perceived "slow" can result in an unsatisfied customer who may opt for a competitor's better performing product, resulting in loss of revenue.

To improve software performance, three kinds of approaches have been proposed:

- *Performance profiling.* Developers conduct performance testing in the form of CPU [GKM82] and memory profiling [JSSC15] to identify code locations that use the most resources. However, traditional profiling techniques have at least two limitations: they show where the resources are spent, but not how to optimize the program. Furthermore, they often introduce large overheads, which may affect the software's behavior and reduce the accuracy of the collected information.

- *Compiler optimizations.* Compiler optimizations [ASU86] automatically transform a program into a semantically equivalent, yet more efficient program. However, many powerful optimization opportunities are beyond the capabilities of a typical compiler. The main reason for this is that the compiler cannot ensure that a program transformation preserves the semantics, a problem that is especially relevant for hard-to-analyze languages, such as JavaScript. For example, a just-in-time (JIT) compiler applies speculative optimizations: it uses profiling information to make assumptions about possible input values.

- *Manual tuning.* Finally, developers often rely on manual performance tuning [HNS09] (e.g., manually optimizing code fragments or modifying software and hardware configurations), which can be effective but it is time consuming and often requires expert knowledge.

The need for improving software performance is never-ending. Limitations of existing performance analyses pose several research challenges and motivate the need for techniques that provide advice on how to improve software performance. This dissertation addresses some of those limitations and proposes new approaches to help developers optimize their code with little effort.

```
_.map = function(obj, iterator, context) {
    var results = [];
    if (obj == null) return results;
    _.each(obj, function(value, index, list) {
      results.push(iterator(value, index, list));
    });
    return results;
  };
```

(a) Performance issue.

```
_.map = function(obj, iterator, context) {
    if (obj == null) return [];
    var keys = _.keys(obj);
    var length = keys.length, currentKey;
    var results = Array(length);
    for (var index = 0; index < length; index++) {
      currentKey = keys[index];
      results[index] = iterator(obj[currentKey], currentKey, obj);
    }
    return results;
  };
```

(b) Optimized code.

Figure 1.1: Performance issue from Underscore library (pull request 1708).

1.1 Terminology

In this work, we use the term *actionable analysis* to denote an analysis that demonstrates the impact of implementing suggested optimization opportunities. In particular, an actionable analysis provides evidence of performance improvement (e.g., speedup in execution time or reduced memory consumption) or shows additional compiler optimizations triggered by applying a suggested optimization. Furthermore, the term *optimization* refers to a source code change that a developer applies to improve the performance of a program, and *compiler optimization* refers to an automatically applied transformation by a compiler.

1.2 Challenges and Motivation

Recent research shows that relatively small changes can make a program significantly more efficient [JSS+12]. However, exploring and exploiting such changes is a challenging task. To illustrate the potential of small code transformations on software performance, Figure 1.1 illustrates a performance issue and an associated optimization reported in *Underscore*, one of the most popular JavaScript utility libraries.

Figure 1.1a shows the initial implementation of the *map* method, which produces a new array of values by mapping the value of each property in an object through a transformation function *iterator*. To iterate over object properties, the method

uses an internal *_.each* function. However, a more efficient way is to first compute the object properties using the *keys* function, and then iterate through them with a traditional for loop. The optimized version of the *map* method is shown in Figure 1.1b. This optimization improves performance because JavaScript engines are able to specialize the code in the for loop and execute it faster.

The optimization in Figure 1.1 has four interesting properties. First, the optimization is *effective*, that is, the optimized method is on average 20% faster than the original one. Second, the optimization is *exploitable*, that is, the code transformation affects few lines of code and is easy to apply. Third, the optimization is *recurring*, that is, developers of real-world applications can apply the optimization across multiple projects. Fourth, the optimization is *out-of-reach for compilers*, that is, due to the dynamism of the JavaScript language, a compiler can not guarantee that the code transformation is always semantics preserving.

Detecting such optimization opportunities in a fully automatic way poses at least three challenges:

- *C1: Understanding performance problems and how developers address them.* Despite the overall success of optimizing compilers, developers still apply manual optimizations to address performance issues in their code. The first step in building actionable performance analyses is to understand the common root causes of performance issues and code patterns that developers use to optimize their code. The next step is to identify optimization patterns amenable for actionable performance analyses. Our intuition is that developers are more likely to apply code changes that improve the application's performance but do not sacrifice code readability and maintainability. An example of such an optimization is already given in Figure 1.1.

- *C2: Analysis of program behavior to detect instances of performance issues.* Based on patterns of common performance issues, the next step is to develop techniques to find code locations suffering from those issues and to suggest beneficial optimizations. For actionable analyses, we focus on the optimization opportunities with the four aforementioned characteristics: *effective, exploitable, recurring* and *out-of-reach for compilers*. To identify instances of known code patterns, there have been various approaches based on either static or dynamic program analysis or the combination of both. The key challenge is to develop an analysis that reports as many true optimizations as possible, while keeping the number of false positives and false negatives as low as possible.

- *C3: Exercising code transformations with enough input.* Once the actionable analysis suggests an optimization opportunity, the next step is to ensure the performance benefit of a code transformation by exercising the program with a wide range of inputs. One approach is to use manually written tests to check whether a code change brings a statistically significant improvement. However, manual tests may miss some of the important cases, which can lead to invalid conclusions. An alternative approach is to use automatically generated tests. Despite being effective in detecting various programming errors, test

generation approaches have limited capabilities in generating complex inputs. For example, the *map* method in Figure 1.1 expects three arguments: an object, an iterator function, and an optional context object. Unfortunately, generating effective tests for higher-order functions (e.g., functions that receive callbacks), such as *map*, is a largely unsolved problem.

Thesis Statement This dissertation supports the thesis that *it is possible to create actionable program analyses that help developers significantly improve the performance of their software by applying effective, exploitable, recurring, and out-of-reach for compilers optimization opportunities.*

We propose novel automated approaches to support developers in optimizing their programs. The key idea is to not only pinpoint where and why time is spent, but also to provide actionable advice on how to improve the application's performance.

1.3 Contributions and Outline

In this section, we highlight individual contributions of the proposed approaches and how they correlate with each other. Furthermore, Figure 1.2 illustrates the connection between approaches and research challenges they address.

Study of Performance Issues and Optimizations 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 how they address these issues. Chapter 2 presents the first empirical study on real-world performance issues and optimizations in JavaScript code. 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. Finally, we observe that many optimizations are instances of patterns applicable across multiple projects. Based on these results, we discuss the challenges of applying recurring optimizations in a fully automatic way and advocate for approaches that help developers optimize their code.

Contributions. This work addresses the first challenge (C1) and contributes in understanding real-world JavaScript performance issues and optimizations that developers apply to fix them. Furthermore, it provides a documented set of 98 reproduced issues[1] that may serve as a reference point for work on finding and fixing performance bottlenecks in JavaScript applications. Finally, we show evidence that developers could benefit from tools and techniques to apply recurring optimization patterns and to reliably measure performance improvements.

An Actionable Performance Profiler for Optimizing Orders of Evaluations As presented in Chapter 2, many performance optimizations are simple code

[1] `https://github.com/marijaselakovic/JavaScriptIssuesStudy`

changes that do not affect program complexity but provide significant performance improvements. In Chapter 3, we present *DecisionProf*, the first dynamic analysis that finds optimization opportunities in inefficient orders of evaluations. *DecisionProf* computes all possible orders in logical expressions and switch statements, finds the optimal order, and suggests a reordering opportunity to the developer only if it yields a statistically significant performance improvement. The approach provides *actionable* advice: it suggests a code transformation to exploit an optimization opportunity.

Contributions. DecisionProf addresses the second challenge (C2), and to the best of our knowledge, it is the first profiler that detects inefficiently ordered subexpressions in logical expressions and switch statements. Furthermore, it suggests simple refactorings to optimize the code. We implement the approach in a practical tool for JavaScript and show that applying suggested optimizations leads to performance improvements in widely used JavaScript projects and popular benchmarks.

Cross-Language Optimizations in Big Data Systems Building scalable big data programs currently requires programmers to combine relational code (SQL) with non-relational code (Java, C# or Scala). This programming model greatly simplifies the distribution and fault-tolerance of big data processing. However, the presence of cross-language interaction poses additional challenges for profiling and optimizing big data jobs. In Chapter 4, we present a profiling infrastructure to understand key performance bottlenecks in SCOPE, a modern data-processing system developed at Microsoft. We find that programs with non-relational code take between 45-70% of the data center's CPU time. To reduce cross-language interaction, we propose a static analysis to find optimization opportunities based on *method inlining*. Method inlining is a simple code transformation that replaces the function call in a predicate with a function body. By doing this, the logic of the function becomes visible to the compiler, resulting in more compiler optimizations. The output of the static analysis is *actionable*: it suggests only inlining opportunities that trigger additional compiler optimizations.

Contributions. This work addresses the second challenge (C2), and it is the first approach that demonstrates the potential of cross-language optimizations in big-data jobs. We present a profiling infrastructure that enables analyzing millions of jobs without introducing any additional overhead. Furthermore, we show the effectiveness of a relatively simple code transformation, based on *method inlining*, in improving the performance of a big-data system.

Test Generation for Higher-Order Functions To reliably measure the performance impact of applied optimizations, the optimized program must be executed with sufficient test inputs. Unfortunately, current test generation techniques are challenged by higher-order functions in dynamic languages, such as JavaScript functions that receive callbacks. In particular, existing test generators suffer from the unavailability of statically known type signatures, do not provide functions or provide only trivial functions as inputs, and ignore callbacks triggered by the code under test. Chapter 5 presents *LambdaTester*, a test generation technique for higher-order functions in dynamic languages. The approach automatically infers

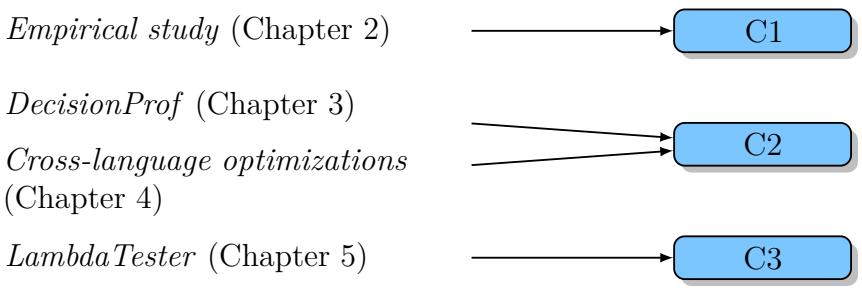


Figure 1.2: Connections between individual contributions and research challenges.

at what argument position a method under test expects a callback, generates and iteratively improves callback functions given as input to this method, and uses novel test oracles that check whether and how callback functions are invoked. We illustrate the effectiveness of the approach by finding correctness problems in many polyfill implementations.

Contributions. LambdaTester addresses the third challenge (C3) and contributes to state-of-the-art test generation approaches in at least two ways: it generates effective tests for higher-order functions written in dynamic languages and improves the generation of callbacks that modify program state in non-trivial ways.

The remaining chapters of this dissertation present related work (Chapter 6) and discusses future work and conclusions (Chapter 7).

1.4 List of Publications

This dissertation is based on several peer-reviewed publications as listed below:

- [SP16] Marija Selakovic and Michael Pradel. "Performance issues and optimizations in JavaScript: An empirical study." *In International Conference on Software Engineering (ICSE)*. 2016.
- [SGP17] Marija Selakovic, Thomas Glaser and Michael Pradel. "An actionable performance profiler for optimizing the order of evaluations." *In International Symposium on Software Testing and Analysis (ISSTA)*. 2017
- [SBMM18] Marija Selakovic, Michael Barnett, Madan Musuvathi, Todd Mytkowicz. "Cross-language optimizations in Big data systems: A case study of SCOPE."*In International Conference on Software Engineering (ICSE-SEIP)*. 2018.
- [MS18] Marija Selakovic, Michael Pradel, Rezwana Karim, Frank Tipp. "Test generation for higher-order functions in dynamic languages." *In Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 2018.
- [AGM+17] Esben Andreasen, Liang Gong, Anders Møller, Michael Pradel, Marija Selakovic, Koushik Sen, and Cristian-Alexandru Staicu. "A survey of dynamic analysis and test generation for JavaScript." *ACM Computing Surveys*. 2017.

Performance Issues and Optimizations in JavaScript

The first step in developing actionable performance analyses is to understand real-world performance issues that developers face in practice and how they address those issues. In this chapter, we introduce an empirical study on performance issues and optimizations in real-world JavaScript projects. We choose JavaScript because it has become one of the most popular programming languages. It is used not only for client-side web applications, but also for server-side applications, mobile applications, and even desktop applications. The development of the language has been enabled by significant improvements of JavaScript engines in recent years, e.g., due to highly optimizing just-in-time (JIT) compilers [GES+09, LV10, HG12, CASP13, ACS+14]. Despite the effectiveness of JIT compilation, developers still manually 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.

However, little is currently known about performance issues and optimizations in real-world JavaScript projects. This chapter 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 such 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?

Answers to these questions help improve JavaScript's performance by providing at least three kinds of insights. First, application developers benefit by learning from mistakes made 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. 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.

We address these question by studying 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%) is that JavaScript provides APIs that are functionally equivalent but have different performance, and that developers often use these APIs in a suboptimal way. This finding suggests that developers need guidance in choosing among such APIs, and that future language and API designs may want to reduce the amount of redundancy in 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 [McC76] of the program. This finding challenges the common belief that improving the performance of a program often implies reducing its understandability and maintainability [Knu74, DRSS01].

- Only 42.68% of all "optimizations" 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, primarily due to the dynamism of JavaScript. We identify five kinds of 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 apply recurring optimizations.

2.1 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.

Table 2.1: Projects used for the study and the number of reproduced issues per project.

Project	Description	Kind of platform	LoC	# Issues
Angular.js	MVC framework	Client	7,608	27
jQuery	Client-side library	Client	6,348	9
Ember.js	MVC framework	Client	21,108	11
React	Library for reactive user interfaces	Client	10,552	5
Underscore	Utility library	Client and server	1,110	12
Underscore.string	String manipulation	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-side	Server	1,268	9
Chalk	Terminal string styling library	Server	78	3
Mocha	Testing framework	Server	7,843	2
Request	HTTP request client	Server	1,144	2
Socket.io	Real-time application framework	Server	703	2
Total			63,951	98

2.1.1 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 npm repository.[1] 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 (≥ 100) to increase the chance to find performance-related issues.

Table 1 lists the studied projects, their target platforms, and the number of lines of JavaScript code. Overall, we consider 16 projects with a total of 63,951 lines of code.

2.1.2 Selection of Performance Issues

We select performance issues from bug trackers as follows:

[1] `https://www.npmjs.com/browse/depended`

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 of 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.

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 an API function that triggers l.
   - A newly created 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, 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 versions 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.

We create a new unit test or microbenchmark for the code location l only if the test is not provided or published in the project's repository. The rationale for focusing on unit tests and microbenchmarks is twofold. First, JavaScript developers extensively use microbenchmarks when deciding between different ways to implement some functionality.[2] 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

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

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 2.1.

2.1.3 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 [GBE07]. 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 so 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).

2.1.4 JavaScript Engines

JavaScript engines evolve quickly, e.g., by adding novel JIT optimizations [GES+09, LV10, HG12, CASP13, ACS+14] or by adapting to trends in JavaScript development[3]. 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 2.2 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 take engines for which their version number indicates that the engine potentially introduces significant changes compared to the previous selected version. Table 2.2 lists for each engine which types of projects it supports. 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 2.1.

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

Table 2.2: 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

2.2 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 cause. Figure 2.1 summarizes our findings, which we detail in the following.

2.2.1 API-related Root Causes

The root cause of 65 performance issues is related to whether and how the program uses 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:[4]

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

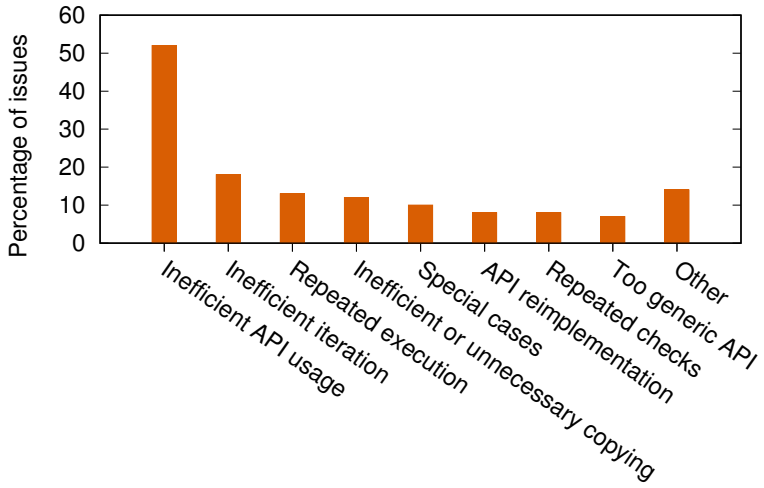
The optimization is to use a more efficient API. For the example, the developers modify the code as follows:

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

Inefficient API usage is the most prevalent root cause, with a total of 50 issues. Figure 2.1b 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.

[4]Issue 39 of Underscore.js.

(a) Most prevalent root causes.



(b) APIs that are used inefficiently.

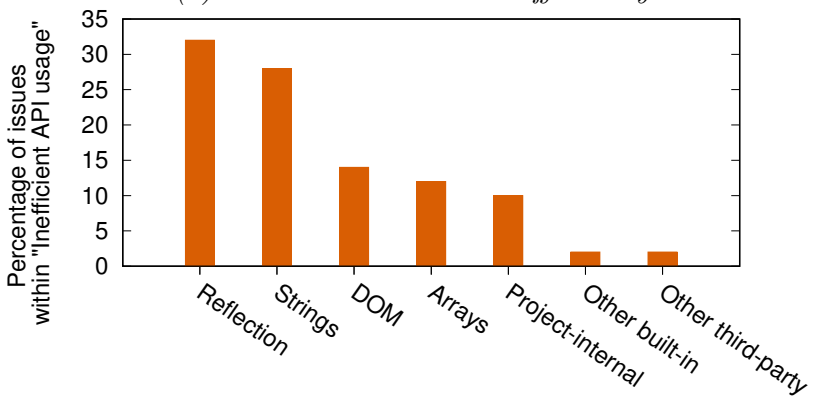


Figure 2.1: Root causes of performance issues.

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.[5]

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`:[6]

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

[5]Issue 9067 of Angular.js.
[6]Issue 102 of jQuery.

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]
```

2.2.2 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:[7]

```
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:[8]

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

To create a regular expression, the developer may use the `RegExp` constructor or a regular expression literal, like in this example. The code is inefficient because creating the regular expression is an expensive operation that is repeatedly executed.

[7] Issue 11338 of Ember.js.
[8] Issue 1097 of Backbone.

The optimization applied to avoid such performance issues is to store the results of the computation for later reuse, e.g., through memoization [TPG15]. 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. 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.[9] The developers optimized 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.

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.[10] The developers optimized 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.[11]

```
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. Similar expressions

[9] Issue 9942 of Angular.js.
[10] Issue 7501 of Angular.js.
[11] Issue 928 of Underscore.js.

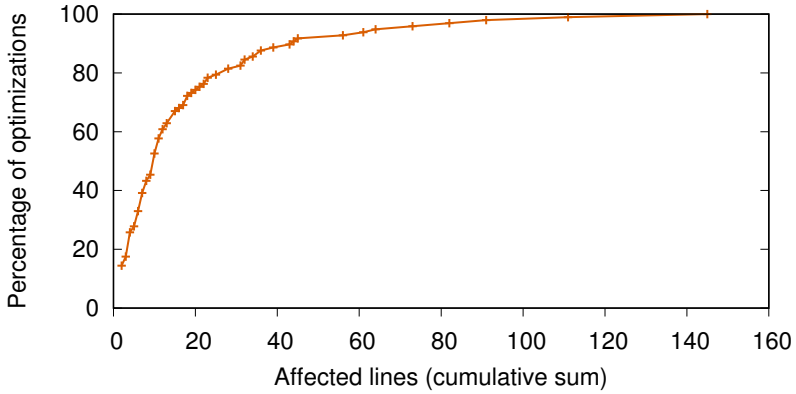


Figure 2.2: Number of source code lines that are affected by optimizations.

would be addressed by the *common-subexpression elimination* optimization that is performed in many compilers. However, non-local changes that require moving an expression outside the function call are outside the scope for existing compilers.

Our analysis 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 for automatically finding performance problems [GPS15, TPG15, XMA+10]. However, our results suggest that there is a need for additional techniques to help developers find and fix instances of other common performance issues.

2.3 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 2.3.1) and to what degree applying these changes affects the complexity of the program's source code (Section 2.3.2).

2.3.1 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.

We find that optimizations affect between 2 and 145 lines of JavaScript source code, with a median value of 10. Figure 2.2 shows the cumulative sum of the number of affected lines per change, i.e., how many optimizations are achieved with less than a particular number of affected lines. 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

(a) Effect on the number of statements. *(b) Effect on cyclomatic complexity.*

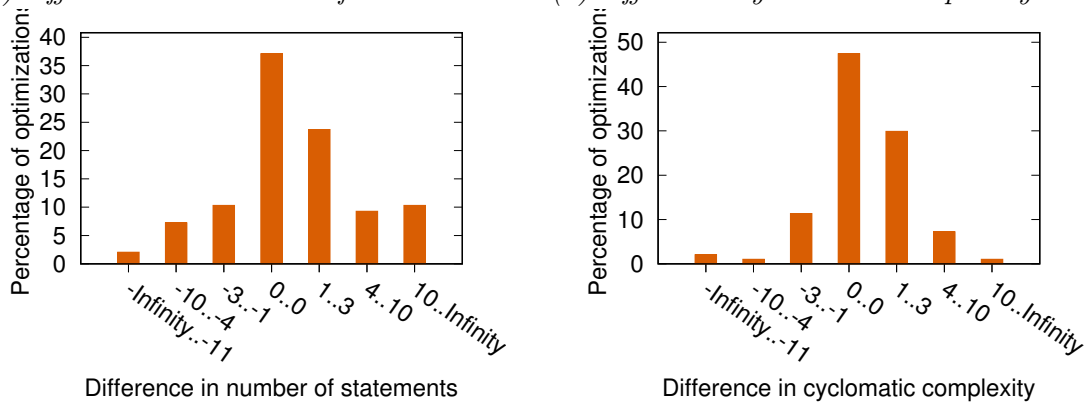


Figure 2.3: Effect of applying an optimization on the cyclomatic complexity.

optimizations are possible with relatively simple changes, which empirically confirms an assumption made by prior research on performance bug detection [JSS+12, NSML13, NCRL15].

2.3.2 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 [McC76] 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 2.3a and 2.3b 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 2.3a shows that 24% of all optimizations add between one and three statements to the 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 [Knu74, DRSS01]. We conclude from these results that many optimizations are possible without increasing the complexity of the optimized program.

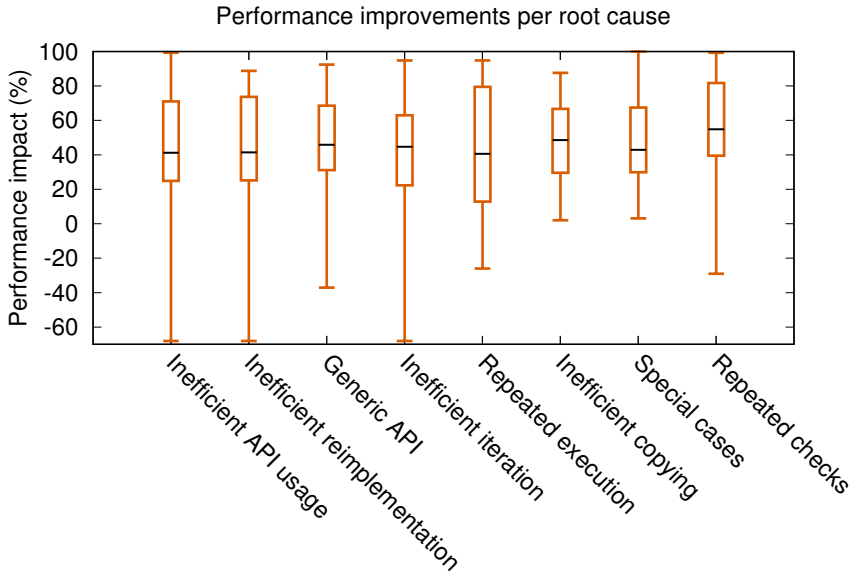


Figure 2.4: Performance improvements obtained by optimizations per root cause. (The bottom and the top of the box indicate the first 25% and 75% of the data, and the middle line indicates the median value. Vertical lines extend the bottom and the top of the box to indicate the minimum and the maximum values.)

2.4 Performance Impact of Optimizations

The following addresses the question of how performance is affected by applying JavaScript optimizations. To address this question, we execute the tests of all 98 optimizations on all considered JavaScript engines where the respective code can be executed (Section 2.1). In total, we obtain 568 performance improvement results.

Figure 2.4 shows the performance results obtained by optimizations for each root cause. The figure illustrates that optimizations lead to a wide range of improvements, with the majority of optimizations saving between 25% and 70% of the execution time. Perhaps surprisingly, the figure shows that some optimizations cause a performance degradation. We further analyze these cases in Section 2.5. In general, the performance results depend on the tests we use, and we can not generalize the results for a specific root cause. For example, the optimizations tested with microbenchmarks usually yield large improvements, because they run small snippets of code.

Given these results and the results from Section 2.3, 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 2.5 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 reasonable, e.g., when the modified code is in a heavily used

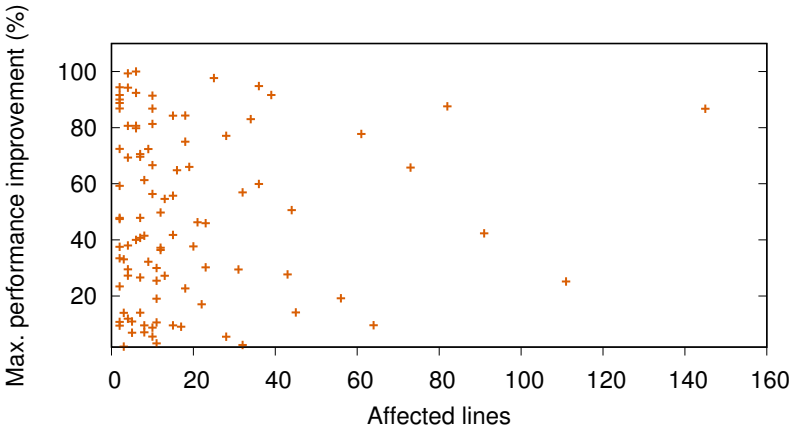


Figure 2.5: Relation between the number of lines affected by a change and the achieved performance improvement.

library function, but may also be a sign for "premature optimizations" [Knu74]. 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.

2.5 Consistency Across Engines and Versions

Since different JavaScript engines apply different optimizations, changing code to improve performance in one engine risks degrading it in another engine. Furthermore, since engines evolve quickly, an optimization applied to speed up the program in one version of an engine may have the opposite effect in 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 in 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 question.

2.5.1 Consistency Across Engines

Table 2.3 compares the performance impact of changes in 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

Table 2.3: 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.

		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	–	–

performance impact in all versions. For each combination of these five cases, the table shows the percentage of changes that fall into the respective category. Because the study includes only issues that provide an improvement in at least one engine (Section 2.1.2), the four cases 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 in 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 in 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 do not affect performance in SpiderMonkey.

2.5.2 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 2.6 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 in 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 decide 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

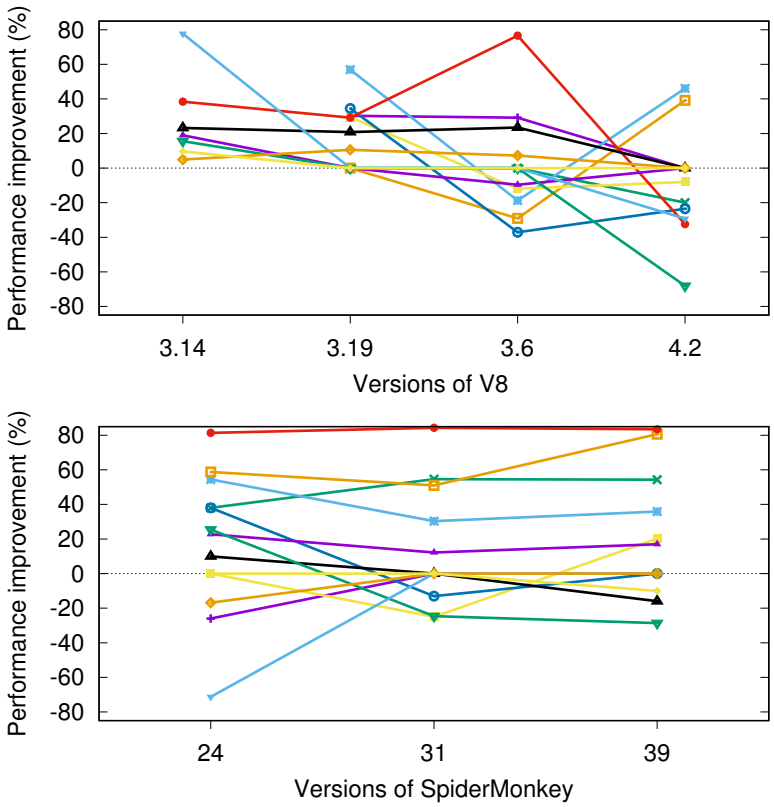


Figure 2.6: Performance improvement of changes in different versions of engines. Each line represents one performance optimization (same line style in both figures means the same optimization).

that turns out to be counterproductive in a new engine, and to specialize JavaScript code for particular engines.

2.6 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 2.6.1). To address the second part of the question, we identify pre-conditions for applying the recurring optimization patterns in a fully automated way (Section 2.6.2).

2.6.1 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 2.4, 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 2.4, Patterns 6–10). For each optimization pattern in Table 2.4, we provide a short description, as well as an example of code before and after the optimization.

In Table 2.5, 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 2.4. Each such analysis performs an AST traversal of a JavaScript program to find matches of the patterns. Due to the well-known difficulties of statically analyzing JavaScript in a sound way (Section 2.6.2), 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 project. In total, the analyses suggest 142 optimizations, of which we manually prune 3 changes because they break the semantics of the program. In Table 2.5, the numbers after the vertical lines show how many previously unreported optimization opportunities the analyses find for each project. We omit four projects for which we do not find any match.

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 have been reporting optimization opportunities to the developers and some of them have already been applied incorporated into the projects.[12]

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 [MKM11, MKM13, BGH07].

[12]See `https://github.com/marijaselakovic/JavaScriptIssuesStudy` for an updated list of reported issues.

Table 2.4: Recurring optimization patterns.

Id	Description	Before	After

1	Prefer `Object.keys()` over computing the properties of an object with a `for-in` loop.	`for (var k in obj) {`
		` if (obj.hasOwnProperty(k))`
		` { ... }`
		`}`
			`var keys = Object.keys(obj);`
			`for (var i=0; l=keys.length;`
			`i<l; i++)){`
			`var key = keys[i];`
			`...`
			`}`

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) !== '$' &&` `key.charAt(1) !== '$'`
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)`
		`=== "[object Error]")`
			`if (err instanceof Error ||`
			`toString.call(err)`
			`=== "[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.	`arr.reduce(`
		`function (str, name) {`
		`return ...;`
		`}, str);`
			`for (var i=0; l=arr.length;`
			`i<l; i++){`
			`var name = arr[i];`
			`str = ...;`
			`}`
			`return str;`

10	When joining an array of strings, handle single-element arrays efficiently.	`[].slice.call(arguments)`
		`.join(' ');`
			`arguments.length === 1 ?`
			`arguments[0] + '' :`
			`[].slice.call(arguments)`
			`.join(' ');`

Table 2.5: 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	Σ
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
Σ	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

2.6.2 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 particular properties of the prototype are not overridden. For example, for Pattern 2, the `substr` property of `String.prototype` must not be overridden. The code in Figure 2.7 illustrates the situation where the Pattern 2 can not be automatically applied because the overridden `substr` method does not extract the characters in the same way as the native `substr` method does.

```
String.prototype.substr = function(a,b){
    return this[a] + this[b];
}
str.substr(i,1)
```

Figure 2.7: Example of prototype overriding

- *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 6, the `split` variable must refer to the `String.prototype.split()` method.

Table 2.6 shows which conditions need to be satisfied to apply each of optimization pattern in fully automatic way.

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.

Table 2.6: Pre-conditions for applying recurring optimization patterns.

Id	Type check	Native function	Prototype	Third-party function	Expression
1	●	●	●	○	○
2	●	○	●	○	○
3	○	●	○	●	○
4	●	○	○	●	○
5	●	●	●	○	○
6	●	●	●	○	○
7	○	●	●	○	●
8	●	○	○	○	○
9	●	●	●	○	○
10	●	○	●	○	○

2.7 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, as we consider projects written in JavaScript, our conclusions are valid for this language only.

Performance Tests. We measure the performance impact of optimizations with unit tests and microbenchmarks; the application-level 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 measure performance with several versions of two JavaScript engines that 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.

2.8 Summary

This chapter presented 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 based on optimization patterns applicable across projects, we show a great potential for techniques to further optimize existing programs. Our results and observations provide an excellent starting point on developing actionable analyses that help developers improve the application's performance by applying relatively simple code optimizations.

Performance Profiling for Optimizing Orders of Evaluation

Chapter 2 discussed the most common performance problems and optimizations in JavaScript projects. It shows that many optimizations are instances of relatively simple, recurring patterns that significantly improve the performance of a program without increasing code complexity. However, automatically detecting and applying such optimizations opportunities is challenging due to dynamic features of the JavaScript language.

In this chapter, we focus on a recurring and easy to exploit optimization opportunity called *reordering opportunity*. A reordering opportunity optimizes the orders of subexpressions that are part of a decision made by the program. As an example, Figure 3.1 shows two instances of reported reordering optimizations in two popular JavaScript projects. The code in Figure 3.1a checks three conditions: whether a regular expression matches a given string, whether the value stored in `match[3]` is defined and whether the value of `arg` is greater than or equal to zero. This code can be optimized by swapping the first two checks (Figure 3.1b) because checking the first condition is more expensive than checking the second condition. After this change, when `match[3]` evaluates to `false`, the overall execution time of evaluating the logical expression is reduced by the time needed to perform the regular expression matching. The second example, Figures 3.1c, shows a performance issue found in Socket.io, a real-time application framework. The routine encodes a packet and checks its metadata. The order of checks in the original code did not reflect the likelihood of the cases to be true, leading to suboptimal performance. The developers refactored the code into Figure 3.1d, where the most common case is checked first, which avoids executing unnecessary comparisons.

A commonality of these examples is that the program takes a decision and that the decision process can be optimized by changing the order of evaluating subexpressions. Once detected, such opportunities are easy to exploit by reordering the checks so that the decision is taken with the least possible cost. At the same time, such a change often does not sacrifice readability or maintainability of the code. Beyond the examples in Figure 3.1, we found various other reordering optimizations in real-world code[1], including several reported by us to the respective developers.[2]

[1] E.g., see jQuery pull request #1560.

[2] E.g., see Underscore pull request #2496 and Moment pull request #3112.

```
arg = (/[def]/.test(match[8]) && match[3] && arg >= 0 ? '+'+ arg : arg);
```

(a) Optimization opportunity.

```
arg = (match[3] && /[def]/.test(match[8]) && arg >= 0 ? '+'+ arg : arg);
```

(b) Optimized code.

```
switch (packet.type) {
    case 'error':
        ...
        break;
    case 'message':
        ...
        break;
    case 'event':
        ...
        break;
    case 'connect':
        ...
        break;
    case 'ack':
        ...
        break;
}
```

(c) Optimization opportunity.

```
switch (packet.type) {
    case 'message':
        ...
        break;
    case 'event':
        ...
        break;
    case 'ack':
        ...
        break;
    case 'connect':
        ...
        break;
    case 'error':
        ...
        break;
}
```

(d) Optimized code.

Figure 3.1: Performance issues from Underscore.string (pull request 471) and Socket.io (pull request 573).

To support developers in detecting and exploiting such optimization opportunities we propose *DecisionProf*, a performance profiler for finding optimal orders of evaluations. The key idea is to compare the computational costs of all possible orders of checks in logical expressions and switch statements, to find the optimal order of these checks, and to suggest a refactoring to the developer that reduces the overall execution time. *DecisionProf* dynamically analyzes the cost of each check and the value it evaluates to. An order of evaluations is optimal for the profiled executions if it minimizes the overall cost of making a decision by first evaluating those checks that determine the overall result most of the time.

DecisionProf has four important properties:

- *Automatically suggested code transformations.* Compared to existing profilers, *DecisionProf* significantly increases the level of automation involved in optimizing a program. The approach fully automatically detects optimization opportunities and suggests code transformations to the developer, which distinguishes our approach from traditional profilers that focus on bottlenecks instead of optimization opportunities.

- *Guaranteed performance improvement.* Before suggesting an optimization, *DecisionProf* applies it and measures whether the modified code improves the

execution time. Only if a change provides a statistically significant performance improvement, the suggestion is reported to the developer.

- *Soundness of proposed modifications.* Because *DecisionProf* does not guarantee that a suggested code transformation preserves the semantics, it may, in principle, report misleading suggestions. However, our evaluation shows that all suggested optimizations are semantics-preserving.

- *Input-sensitivity.* As every profiler, *DecisionProf* is input-sensitive, i.e., it relies on inputs that trigger representative executions. In this work, we assume that such inputs are available, which often is the case in practice, as evidenced by the wide use of traditional profilers.

While the basic idea of our approach is simple, there are several interesting challenges. One major challenge for finding the optimal order of evaluations is to evaluate every subexpression that is relevant for a decision during profiling. A naive approach that always executes all subexpressions is likely to change the semantics of the program because of the side effects of these evaluations. We address this challenge with a novel technique for side effect-free evaluation of expressions. During such an evaluation, the approach tracks writes to variables and object properties. Afterwards, the approach restores the values of all variables and properties to the values they had before the evaluation started.

We evaluate *DecisionProf* by applying it to 43 real-world JavaScript projects, including popular libraries and benchmarks. Across these projects, the profiler finds 52 optimization opportunities that result in statistically significant speedups. Optimizing the order of evaluations does not change the behavior of the program, and reduces the execution time of individual functions between 2.5% and 59% (median: 19%). Even though the optimizations are simple, they yield application-level performance improvements, which range between 2.5% and 6.5%.

3.1 Problem Statement

This section characterizes the problem of inefficiently ordered evaluations and describes challenges for identifying and optimizing them.

3.1.1 Terminology

Real-world programs compute various boolean values, e.g., to make a control flow decision. Often, such *decisions* are the result of evaluating multiple boolean expressions that are combined in some way. We call each such an expression a *check*. For example, Figure 3.1a shows a decision that consists of three checks that are combined into a conjunction. Figure 3.1c shows a decision that depends on a sequence of checks, each of which is a comparison operation. A program specifies the *order of evaluation* of the checks that contribute to a decision. This order may be commutative, i.e., changing the order of checks does not change the semantics of the program. For example, the refactorings in Figure 3.1 are possible because the checks are commutative.

For decisions where the order of evaluation is commutative, different orders may have different performance. These differences exist because the semantics of most programming languages does not require to evaluate all checks, e.g., due to short-circuit evaluation of boolean expressions. The most efficient order depends on the probability of the checks to yield `true` and on the cost of evaluating the checks. For example, suppose that both checks in `a() && b()` have the same computational costs. If in 80% of all executions the first check evaluates to `true` and the second check evaluates to `false`, then the evaluation of `a()` is wasted 80% of the time. To avoid such unnecessary computation, one can reorder the checks, which yields a logically equivalent but more efficient decision. In logical expressions, each check is composed of one or more *leaf expressions*, i.e., subexpressions that do not contain another logical expressions. In this work we consider logical expressions where *leaf expressions* are combined by both disjunctive and conjunctive operators.

3.1.2 Reordering Opportunities

The problem addressed in this chapter is finding inefficiently ordered evaluations, called *reordering opportunities*, and suggesting a more efficient order to the developer. We consider arbitrarily complex decisions, e.g., decisions that involve nested binary logical expressions. The goal of *DecisionProf* is to find a optimal order of the checks involved in a decision. Optimal here means that the total cost of making the decision is minimal for the profiled executions. This total cost is the sum of the costs of those individual checks that need to be evaluated according to the semantics of the programming language. The two examples in Figure 3.1 are reordering opportunities.

3.1.3 Challenges for Detecting Reordering Opportunities

Even though the basic idea of reordering checks is simple, detecting reordering opportunities in real-world programs turns out to be non-trivial. We identify three challenges.

- *Measuring the cost and likelihood of checks.* To identify reorderings of checks that reduce the overall cost of a decision, we must assess the cost of evaluating individual checks and the likelihood that a check evaluates to `true`. The most realistic way to assess computational cost is to measure the actual execution time. However, short execution times cannot be measured accurately. To compute the optimal evaluation order, we require an effective measure of computational cost, which should be a good predictor of actual execution time while being measurable with reasonable overhead.

- *Analyze all checks involved in a decision.* To reason about all possible ways to reorder the checks of a decision, we must gather cost and likelihood information for all checks involved in the decision. However, dynamically analyzing all checks involved in a decision may not be necessary in a normal execution. For example, consider that the first check in Figure 3.1a evaluates to `false`. In this case, the overall value of the expression is determined as `false`, without executing the other two checks.

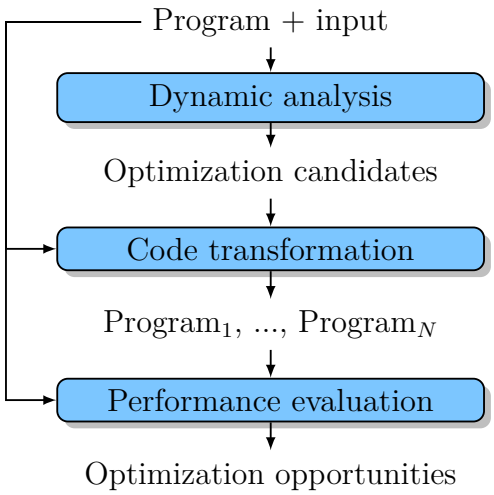


Figure 3.2: Overview of *DecisionProf*.

- *Side effect-free evaluation of check.* Evaluating checks may have side effects, such as modifying a global variable or an object property. Therefore, naively evaluating all checks, even though they would not be evaluated in the normal program execution, may change the program's semantics. To address this issue, we need a technique for evaluating individual expressions without permanently affecting the state of the program.

3.2 Analysis for Detecting Reordering Opportunities

In this section, we describe *DecisionProf*, a profiling approach that automatically finds reordering opportunities at runtime and proposes them to the developer. Figure 5.4 gives an overview of the approach. The input to *DecisionProf* is an executable program. First, the profiler executes the program while applying a dynamic analysis that identifies optimization candidates. Second, for each candidate, the approach applies the optimization via source-to-source transformation. Third, for the modified version of the program, *DecisionProf* checks whether the optimization reduces the execution time of a program. If and only if the changes lead to a performance improvement, the approach suggests them as reordering opportunities to the developer. The remainder of this section details each component of the approach.

3.2.1 Gathering Runtime Data

The first step of *DecisionProf* is to analyze the execution of the program to identify candidates for reordering opportunities. We gather two pieces of information about every dynamic occurrence of a check involved in a decision: The computational *cost* of evaluating the check and the *value* of the check, i.e., whether the boolean expression evaluates to `true` or `false`. *DecisionProf* gathers these runtime data in

```
startDecision;
startCheck; /[def]/.test(match[8])); endCheck;
startCheck; match[3]; endCheck;
startCheck; arg >= 0; endCheck;
undoSideEffects; endDecision;

arg = (/[def]/.test(match[8]) && match[3] && arg >= 0 ? '+'+ arg : arg);
```

Figure 3.3: Preprocessed logical expression from Figure 3.1a.

```
startDecision;
switch (packet.type) {
    startCheck; packet.type==='error'; endCheck;
    case 'error':
        ...
    startCheck; packet.type==='message'; endCheck;
    case 'message':
       ...
    startCheck; packet.type==='event'; endCheck;
    case 'event':
       ...
    startCheck; packet.type==='connect'; endCheck;
    case 'connect':
       ...
    startCheck; packet.type==='ack'; endCheck;
    case 'ack':
       ...
}
endDecision;
```

Figure 3.4: Preprocessed switch statement from Figure 3.1c.

two steps. At first, it statically pre-processes the source code of the analyzed program. Then, it dynamically analyzes the pre-processed program to collect runtime data.

Pre-processing

DecisionProf pre-processes the program to ensure that each check involved in a decision gets executed, even if it would not be executed in the normal execution, and to introduce helper statements for measuring the cost of each check.

Logical expressions. To ensure that each checks gets evaluated even if it would not be evaluated in the normal program execution, the pre-processor copies each leaf expression of a logical expression in front of the statement that contains the logical expression. Furthermore, the pre-processor annotates the beginning and end of a decision, and the beginning and end of each leaf expression with helper statements. Because always evaluating all checks may change the program's semantics, the pre-processor also annotates the end of a decision with a helper statement `undoSideEffects`, which we explain in Section 3.3. Figure 3.3 shows the pre-processed code for the logical expression of Figure 3.1a. The underlined helper statements are interpreted by the dynamic analysis.

Switch statements. The pre-processor annotates for each switch statement the beginning and end of the decision, and the beginning and end of each check. Because evaluating a check in a switch statement is a comparison without any side effects, there is no need to undo any side effects. Figure 3.4 shows the pre-processed code for the switch statement of Figure 3.1c.

Dynamic Analysis

DecisionProf executes the pre-processed program and dynamically collects the values and the computational costs of each check involved in a decision. To this end, the approach associates with each decision a cost-value history:

Definition 1 (Cost-value histories)**.** The cost-value history h of a check involved in a decision is a sequence of tuples (c, v), where v denotes the value of the check and c represents the cost of evaluating the check. The cost-value histories of all checks involved in a decision are summarized in a history map $\mathcal{H}$ that assigns a history to each check.

To gather cost-value histories, the analysis reacts to particular runtime events:

- When the analysis observes a statement `startDecision`, it pushes the upcoming decision onto a stack *decisions* of currently evaluated decisions.
- When the analysis observes a statement `startCheck`, it pushes the check that is going to be evaluated onto a stack *checks* of currently evaluated checks. Furthermore, the analysis initializes the cost c of the upcoming evaluation to one.
- When reaching a branching point, the analysis increments the cost counter c of each check in *checks*. We use the number of executed branching points as a proxy measure for wallclock execution time, avoiding the challenges of reliably measuring short-running code.[3]
- When the analysis observes `endCheck`, it pops the corresponding check from *checks*. Furthermore, the analysis appends (c, v) to h, where h is the cost-value history of the check as stored in the history map $\mathcal{H}$ of $top(decisions)$, c is the cost of the current check evaluation, and v is the boolean outcome of evaluating the check.
- When reaching `endDecision`, the analysis pops the corresponding decision from *decisions*.
- When the analysis observes `undoSideEffects`, it restores the state of the program to the state before the corresponding `startDecision` statement (Section 3.3).

[3]Section 3.5.4 evaluates the accuracy of the proxy metric.

Table 3.1: Cost-value histories from executions of Figure 3.1a.

Check	Execution		
	1st	2nd	3rd
`/[def]/.test(match[8])`	(3, true)	(3, true)	(3, false)
`match[3]`	(1, true)	(1, false)	(1, false)
`arg`	(1, true)	(1, true)	(1, true)

The reason for using stacks to represent the currently evaluated decisions and checks is that they may be nested. For example, consider a logical expression `a() || b ()`, where the implementation of `a` contains another complex logical expression.

Our implementation refines the described analysis in two ways. First, the analysis monitors runtime exceptions that might occur during the evaluation of the decision. If an exception is thrown, the analysis catches the error, restores the program state, and excludes the decision from further analysis. Such exceptions typically occur because the evaluation of one check depends on the evaluation of another check. Second, the analysis considers switch statements with case blocks that are not terminated with a `break` or `return` statement. For such case blocks, the analysis merges the checks corresponding to the cases that are evaluated together into a single check.

Table 3.1 shows a cost-value history gathered from three executions of the logical expression in Figure 3.1a. For example, when the logical expression was executed for the first time, the check `/[def]/.test(match[8])` was evaluated to `true` and obtaining this value imposed a runtime cost of 3.

3.2.2 Finding Optimization Candidates

Based on cost-value histories obtained through dynamic analysis, *DecisionProf* computes a optimal order of checks for each executed decision in the program. The computed order is optimal in the sense that it minimizes the overall cost of the analyzed executions. *DecisionProf* uses two specialized algorithms for computing the optimal orders of evaluations in logical expressions and switch statements, respectively.

Optimally Ordered Logical Expressions

To find an optimal order of checks in logical expressions, we present a recursive algorithm that optimizes all subexpressions of a given expression in a bottom-up manner. Algorithm 1 summarizes the main steps. The algorithm uses the history map $\mathcal{H}$ to keep track of the checks that have already been optimized. Initially, the map contains histories for all leaf expressions, i.e., expressions that do not contain any logical operator. For each such leaf expression, the history map contains a cost-value history h gathered during the dynamic analysis.

Given a logical expression e, the algorithm checks whether its left and right subexpressions have already been optimized by checking whether they have an entry

Algorithm 1 Algorithm to find optimal order of logical expression.

Input: Logical expression e and history map $\mathcal{H}$
Output: Optimized expression e'

function *optimize*(e)
  if $e.left$ is not in $\mathcal{H}$ **then**
    $e.left \leftarrow optimize(e.left)$
  if $e.right$ is not in $\mathcal{H}$ **then**
    $e.right \leftarrow optimize(e.right)$
  $e' \leftarrow findOptimalOrder(e)$
  return e'

function *findOptimalOrder*(e)
  $c_{orig} \leftarrow computeCost(e.left, e.right, e.operator)$
  $c_{swap} \leftarrow computeCost(e.right, e.left, e.operator)$
  if $c_{orig} \leq c_{swap}$ **then**
    $e' \leftarrow e$
  else
    $e' \leftarrow e$ with left and right swapped
  $h_{e'} \leftarrow$ cost-value history of optimized expression e'
  Add $e \mapsto h_{e'}$ to $\mathcal{H}$
  return e'

function *computeCost*(e_{left}, e_{right}, op)
  $h_{left} \rightarrow \mathcal{H}(e_{left})$
  $h_{right} \rightarrow \mathcal{H}(e_{right})$
  $c \leftarrow 1$
  foreach i in 0 to $length(h_{left})$ **do**
    $cv_{left} \leftarrow h_{left}[i]$
    $cv_{right} \leftarrow h_{right}[i]$
    if op is `&&` **then**
      if $cv_{left}.value$ is `true` **then**
        $c \leftarrow c + cv_{left}.cost + cv_{right}.cost$
      else
        $c \leftarrow c + cv_{left}.cost$
    else if op is `||` **then**
      if $cv_{left}.value$ is `false` **then**
        $c \leftarrow c + cv_{left}.cost + cv_{right}.cost$
      else
        $c \leftarrow c + cv_{left}.cost$
  return c

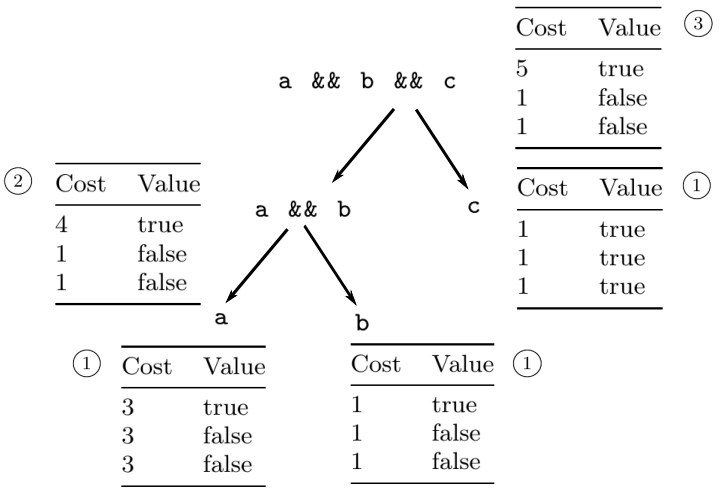


Figure 3.5: Example of finding the optimal order of checks.

in $\mathcal{H}$. If no such entry exists, the algorithm recursively calls *optimize* to optimize these subexpressions before deciding on their optimal order. Once both the left and the right subexpression are optimized, the algorithm calls *findOptimalOrder*. This function computes the cost of both possible orders of the left and right subexpression and swaps them if the cost of the swapped order is smaller than of the original order. Afterwards, the function updates the history map $\mathcal{H}$ by computing a sequence of cost-value entries for the optimized expression. Each cost-value entry of the optimized expression is derived based on the cost-value histories of subexpressions by applying the short-circuit rules.

Function *computeCost* summarizes how the algorithm computes the cost of a particular order of two checks. The basic idea is to iterate through the value-cost history and to apply the short-circuit rules of the programming language. For example, if two checks are combined with the logical `&&`-operator, then the cost includes the cost of the second check only if the first check is `true`.

For example, consider applying the algorithm to the logical expression in Figure 3.1a and the cost-value history in Table 3.1. Figure 3.5 illustrates the tree of subexpressions for the example and the value-cost history associated with each check. For space reasons, we abbreviate the logical expressions as `a && b && c`. The leaf expressions each have an associated history (marked with 1). Based on these histories, the algorithm computes the optimal cost of the first, innermost logical expression, `a && b`. The costs in the three executions with the original order are 4, 4, and 3. In contrast, the costs when swapping the checks are 4, 1, and 1. That is, swapping the subexpressions reduces the overall cost. Therefore, *findOptimalOrder* sets $e' =$ `b && a` and computes the history of this optimized expression (marked with 2). The cost-value history of `b && a` is derived from the cost-value histories of `b` and `a`. Next, the algorithm moves up in the expression tree and optimizes the order of e_{left}=`b && a` and e_{right} =`c`. Comparing their costs shows that swapping these subexpressions is not beneficial, so the algorithm computes the history of the subexpression (marked with 3), and finally it returns `b && a && c` as the optimized expression.

Optimally Ordered Switch Statements

To optimize the order of checks in switch statements, *DecisionProf* sorts checks by their likelihood to evaluate to `true`, starting with the most likely check. To achieve this, *DecisionProf* instantiates a new order o with an empty set and sorts entries in cv by their frequency of being evaluated to `true`. Then, for each element $cases$ in cv, the approach checks whether there is an element el in o, such that their intersection is non-empty. In this case, el is replaced by the union of el and $cases$. Otherwise, it adds $cases$ from cv to o. The rationale is to preserve the order of cases that can be executed together. For example, it is possible to have several cases that share the same code block or blocks that do not contain `break` statements. After iterating over all checks in cv, the approach returns the optimal order o of checks. If some checks are not evaluated during the execution of a decision, they are added to the end of o in their original order.

For example, reconsider the switch statement in Figure 3.1c. Suppose that the switch statement is executed 10 times as follows: 5 times the "message" case, 2 times the "event" and "ack" cases and once the "connect" case. Starting from $o = []$, *DecisionProf* sorts the cost-value history of the switch statement and computes $o = [message, event, ack, connect, error]$. The case "error" is not evaluated in the analyzed execution and therefore is added to the end of o.

Based on the results from the profiler, *DecisionProf* identifies as optimization candidates all those decisions where the order of evaluations in the given program is not optimal. The following two subsections describe how *DecisionProf* assesses whether exploiting these candidates yields a statistically significant performance improvement.

Transformation

Based on the list of optimization candidates found in the previous phase of *DecisionProf*, the approach generates a variant of the program that adapts the order of checks to the optimal order. To this end, *DecisionProf* traverses the AST of the program and identifies the source code location that implements the decision. Then, the approach rewrites the AST subtree that corresponds to the decision into the optimal order of checks, and generates an optimized variant of the program source code. For our running examples, *DecisionProf* automatically applies the two optimizations illustrated in Figure 3.1.

Performance Evaluation

The final step of *DecisionProf* is to assess whether applying a reordering increases the performance of the analyzed program. The rationale is to suggest optimizations to the developer only if there is evidence that the change is beneficial. To measure the performance impact of a change, *DecisionProf* runs both the original and the optimized program in a several fresh instances of the JavaScript engine while collecting the measurements of the actual execution time for each program (details in Section 3.5.1). To determine whether there is a statistically significant difference, the approach applies the t-test on the collected measurements with a confidence

```
160 var a = 0;
161 function foo(){
162     a++;
163     var b=1;
164     if (a===1) return true;
165     else return false;
166 }
167 startDecision;
168     startCheck:  foo();
169     startCheck:  someExpression;
170 undoSideEffects;
171 if (foo() && someExpression) ...
```

Figure 3.6: Changes in program behavior due to side effects.

level of 90%. Finally, *DecisionProf* suggests a change as a reordering opportunity if the change improves the execution time of a program.

3.3 Safe Check Evaluation

The profiling part of *DecisionProf* (Section 3.2.1) evaluates all checks of a decision, even though they may not be evaluated during the normal program execution. If evaluating a normally not evaluated check has side effects, our profiling might cause the program to behave differently than it would without profiling. To avoid such a divergence of behavior, the profiling part of *DecisionProf* evaluates checks in such way that side effects are undone after the evaluation, which we call *safe check evaluation*.

As a motivating example, consider Figure 3.6. The profiler evaluates `foo()` before executing the `if` statement (line 168), which causes a write to the global variable `a`. Using a naive approach that continues the execution after this side effect, the `if` statement at line 171 would evaluate to `false`, because `a` gets incremented again at line 162.

To avoid changing the program behavior during profiling, we use a dynamic analysis that tracks writes to object properties and variables, and undoes these side effects when reaching an `undoSideEffects` statement.

3.3.1 Tracking Side Effects

To track side effects, the analysis records writes into the following two data structures:

Definition 2 (Log of property writes)**.** A log of property writes *propLog* is a sequence of tuples $(obj, prop, value)$, where *obj* is an object, *prop* is a property name, and *value* is the value that *prop* holds before the evaluation.

Definition 3 (Log of variable writes)**.** A log of variable writes *varLog* is a sequence of pairs $(var, value)$, where *var* is a variable name and *value* is the value *var* holds before the evaluation.

While evaluating checks for profiling, the analysis records all property writes and variable writes into these logs. For variable writes, the analysis only considers variables that may affect code executed after the check. For this purpose, the analysis checks whether the variable is defined in the same or a parent scope of the currently evaluated decision, and only in this case records the write into the log.

To find the scope of a variable, *DecisionProf* computes the scope for every variable declaration and function definition in a program as follows (inspired by [SKBG13]):

- When the execution of a program starts, *DecisionProf* creates an empty object that represents the global scope and an array *stack*, representing the stack of scopes. Initially, *stack* contains only one element, the global scope.
- Before the execution of a function body starts, the analysis creates a new scope and pushes it onto *stack*. Before the execution of a function body finishes, the analysis pops the top element from *stack*. The current scope of a program's execution is always the top element of *stack*.
- When the analysis identifies a function definition or a variable declaration in a program, it expands the current scope by adding new properties with the name of the variable or the declared function.

For Figure 3.6, consider the evaluation of `foo()` at line 168. The analysis records the write to variable `a`, along with its original value 0. In contrast, since the write to variable `b` is local to `foo()`, the analysis does not record it.

3.3.2 Undoing Side Effects

When the evaluation of a decision finishes, the `undoSideEffects` statement triggers the analysis to undo all recorded side effects. The analysis undoes variable writes by dynamically creating and executing code that writes the original values into each variable. Specifically, for every entry in *varLog*, the analysis creates a variable assignment statement where the left-hand side of the assignment is the variable name and the right-hand side is the original value. Then, the analysis passes the sequence of assignments to the `eval()` function, which evaluates the string as code. For the example in Figure 3.6, the analysis creates and executes the following code: `var a = 0;` To undo property writes, the analysis iterates over all entries in *propLog* and restores the original value of each modified property.

3.4 Implementation

We implement *DecisionProf* into a profiling tool for JavaScript programs. Static pre-processing and applying optimizations are implemented as AST-based transformations built on top of an existing parser[4] and code generator[5]. The dynamic analysis builds on top of the dynamic analysis framework Jalangi [SKBG13]. We believe that our approach is applicable to other languages than JavaScript, e.g.,

[4]http://esprima.org/
[5]https://github.com/estools/escodegen

based on existing dynamic analysis tools, such as Valgrind or PIN for x86 programs, and ASM or Soot for Java.

3.5 Evaluation

We evaluate the effectiveness and efficiency of *DecisionProf* by applying it to 43 JavaScript projects: 9 widely used libraries and 34 benchmark programs from the JetStream suite, which is commonly used to assess JavaScript performance. In summary, our experiments show the following:

- *Does DecisionProf find reordering opportunities?* The approach identifies 52 previously unknown reordering opportunities (Section 3.5.2)
- *What is the performance benefit of optimizing the order of evaluations?* Applying the optimizations suggested by *DecisionProf* yields performance improvements between 2.5% and 59%. (Section 3.5.2)
- *How efficient is the approach?* Compared to normal program execution, *DecisionProf* imposes a profiling overhead between 3x and 1,210x (median: 116). (Section 3.5.3)
- *Is counting the number of branching points an accurate approximation of execution time?* We find that the measure *DecisionProf* uses to approximate execution time is strongly correlated with actual execution time. (Section 3.5.4)
- *How effective would DecisionProf be if it conservatively pruned all non-commutative checks via static analysis?* For 28 of 52 optimizations it is non-trivial to statically show that they are semantics preserving, i.e., a conservative variant of *DecisionProf* would miss many optimizations. (Section 3.5.5)

3.5.1 Experimental Setup

Subject Programs and Inputs We use 9 JavaScript libraries, listed in the upper part of Table 3.2. They are widely used both in client-side web applications and in Node.js applications. In addition to the libraries, we also use 34 programs from the JetStream benchmark suite. We choose JetStream because it is, to the best of our knowledge, the most comprehensive performance benchmark suite for JavaScript. It includes Octane, Sunspider, benchmarks from LLVM compiled to JavaScript, and a hand-translated benchmark based on the Apache Harmony project. The lower part of Table 3.2 lists the subset of benchmark programs where *DecisionProf* detects beneficial reordering opportunities.

To execute the libraries, we use their test suites, which consist mostly of unit-level tests. We assume for the evaluation that these inputs are representative for the profiled code base. The general problem of finding representative inputs to profile a given program [GFX12, BJS09, DR16] is beyond the scope of this dissertation.

Table 3.2: Projects used for the evaluation (LogExp = number of logical expressions, Switch = number of switch statements, StaPru = statically pruned logical expressions, Ev = number of evaluated logical expressions and switch statements, DynPru = dynamically pruned logical expressions, Cand = reordering candidates, t_{base} = execution time without profiling (seconds), t_{prof} = execution time with profiling (seconds), Overh = profiling overhead, Impr (%) = performance improvement.)

					Pruning				Overhead			Opportunities			Impr(%)
Project	Tests	LoC	LogExp	Switch	StaPru	Ev	DynPru	Cand	t_{base}	t_{prof}	Overh	LogExp	Switch	Total	
Libraries:															
Underscore	161	1,110	65	3	4	48	1	23	0.17	11.08	65x	2	0	2	3.7 - 14
Underscore.string	56	905	25	2	6	16	0	6	0.70	2.42	3x	1	0	1	3 - 5.8
Moment	441	2,689	116	2	23	88	9	31	1.38	17.20	12x	1	0	1	3 - 14.6
Minimist	50	201	17	0	4	13	0	7	0.05	0.78	15x	1	0	1	4.2 - 6.5
Semver	28	863	33	3	3	30	1	15	0.18	4.45	25x	4	1	5	3.5 - 10.6
Marked	51	928	26	1	3	13	0	4	0.08	1.17	15x	0	1	1	3 - 4.4
EJS	72	549	14	3	4	9	0	7	0.08	1.19	15x	2	0	2	5.6 - 6.7
Cheerio	567	1,396	74	3	12	58	2	22	1.18	15.08	13x	9	0	9	6.2 - 40
Validator	90	1,657	119	2	0	112	1	83	0.1	2.58	26x	3	0	3	3 - 10.9
Total	1,516	10,928	489	21	59	387	14	198				21	2	23	
Benchmarks:															
float-m		3972	119	5	11	23	0	21	1.21	841.2	691x	1	2	3	2.5
crypto-aes		295	3	0	0	3	0	3	0.01	0.84	70x	3	0	3	5.2
deltablue		483	8	1	0	8	0	8	0.02	1.7	90x	2	0	2	6.5
gbemu		9481	142	20	4	67	0	61	0.25	33.73	132x	13	5	18	5.8
Total		14231	272	26	15	101	0	93				19	7	26	

Performance Measurements Reliably measuring the performance of executions is non-trivial [MDHS09]. *DecisionProf* follows the methodology by Georges et al. [GBE07], both as part of the approach (Section 3.2.2) and for the evaluation. In essence, *DecisionProf* repeatedly starts a fresh JavaScript engine (N_{VM} times), repeats the execution N_{warmUp} times to warm up the JIT compiler, measures the execution time of $N_{measure}$ further repetitions, and applies statistical significance tests to decide whether there is a speedup. We use $N_{VM} = 5$, $N_{warmUp} = 5$, $N_{measure} = 10$. Since very short execution times cannot be measured accurately, we wrap inputs that are unit tests into a loop that makes sure to execute for at least 5ms. To measure the performance of benchmarks, we apply the statistical test on measurements collected from 50 executions of the original and the modified benchmark. All experiments are done on a 48-core machine with a 2.2GHz Intel Xeon CPU and an eight-core machine with a 3.60GHz Intel Core i7-4790 CPU, all running 64-bit Ubuntu Linux 14.04 LTS. We use Node.js 4.4 and provide it with the default of 1GB of memory for running the unit tests and 4GB of memory for running the benchmarks.

Code Transformations When checking whether an optimization candidate improves the performance, *DecisionProf* can either apply one candidate at a time or all candidates at once. For the unit-level tests of libraries, we configure *DecisionProf* to consider each candidate individually because we were interested in whether a single change may cause a speedup. For the benchmarks programs, we configure *DecisionProf* to apply all candidates at once. The rationale is that achieving application-level speedups is more likely when applying multiple optimizations than with a single optimization. We bound the number of applied optimizations per program to at most 20 optimized logical expressions and 20 optimized switch statements, enabling us to manually inspect all optimizations in reasonable time.

3.5.2 Detected Reordering Opportunities

In total, *DecisionProf* detects 52 reordering opportunities. The column "Opportunities" in Table 3.2 shows how many optimizations the approach suggests in each project, and how many of them are in logical expressions and switch statements, respectively. To the best of our knowledge, none of the optimizations detected by *DecisionProf* have been previously reported.

Examples. Table 3.3 illustrates seven representative examples of reordering opportunities. The columns "Original" and "Optimized" show for each opportunity the original code and the optimized code, as suggested by *DecisionProf*. The "Performance improvement" columns show for how many tests the optimization improves and degrades performance, along with the respective improvements. For example, the first opportunity provides performance improvements for three tests, without degrading the performance of other tests. The logical expression checks whether a given input is `NaN` (not a number). In most analyzed executions, the first check is wasted because most inputs are numbers that are not `NaN`. We reported this opportunity to the developers and they have accepted the optimization suggested

Table 3.3: Examples of reordering opportunities found by *DecisionProf* (+ = number of positively affected tests, - = number of negatively affected tests, % = percentages of speedups or slowdowns).

Id	Project	Original	Optimized	Performance changes in tests			
				+	%	-	%
Libraries:							
1	Underscore	`_.isNumber(obj) && isNaN(obj)`	`isNaN(obj) && _.isNumber(obj)`	3	19.04, 3.76, 5.02	0	
2	Moment	`hour === 12 && !isPm`	`!isPm && hour === 12`	3	4.00, 4.62, 19.62	1	4.37
3	Cheerio	`isTag(elem) && elems.indexOf (elem) === -1`	`elems.indexOf(elem) === -1 && isTag(elem)`	2	26.84, 34.93	0	
4	Validator	`(0, _isHexadecimal2.default) (str) && str.length === 24`	`str.length === 24 && (0, _isHexadecimal2.default)(str)`	1	10.91	0	
5	Minimist	`!flags.strings[key] && isNumber(val)`	`isNumber(val) && !flags.strings[key]`	2	4.23, 6.54	0	
Benchmarks:							
6	Gbemu	`numberType != 'float32'&& GameBoyWindow.opera && this.checkForOperaMathBug()`	`GameBoyWindow.opera && numberType != 'float32' && this.checkForOperaMathBug()`	Application-level: 5.8%			
7	Deltablue	`i.mark == mark || i.stay || i.determinedBy == null`	`i.stay || i.mark == mark || i.determinedBy == null`	Application-level: 6.5%			

by *DecisionProf*.[6] The second example illustrates a case where the optimization degrades the performance of one test. Because it also improves the performance for three other tests, *DecisionProf* reports it as a reordering opportunity.

Performance improvements. Overall, *DecisionProf* uncovers reordering opportunities that yield speedups between 2.5% and 59% (median: 19%). The last column of Table 3.2 summarizes the speedups. The improvement include both speedups of library code, measured by unit tests, and application-level speedups, measured by the benchmarks. For the libraries, the approach reports an opportunity if the number of positively affected tests exceeds the number of negatively affected tests. In total over all detected opportunities, 67% of all tests with a performance difference are positively affected. For 11 of 23 opportunities, all tests are positively affected. For opportunities with both positively and negatively affected tests, we expect the developer to decide which test cases are more representative, and whether the optimization should be applied.

Effect of pruning. To better understand the impact of pruning likely non-commutative checks, Table 3.2 shows the number of statically pruned decisions ("StaPru"), how many of the remaining decisions are executed by the test suites ("Ev"), and how many of the executed decisions are pruned dynamically ("DynPru"). The "Cand" column shows how many reordering candidates pass the testing-based validation. Measuring the performance impact of these potential optimizations prunes most candidates. This result shows the importance of the last phase of *DecisionProf*, which avoids suggesting code changes that do not improve performance.

False positives. We manually inspect all changes suggested by DecisionProf to evaluate whether any of them may change the semantics of the program. We find that all suggested optimization are semantics-preserving, i.e., the approach has no false positives in our evaluation.

Reported optimizations. To validate our hypothesis that developers are interested in optimizations related to the order of checks, we reported a small subset of all detected reordering opportunities. Three out of seven reported optimizations got confirmed and fixed within a very short time, confirming our hypothesis.

3.5.3 Profiling Overhead

The overall execution time of *DecisionProf* is dominated by the time to dynamically analyze the program or the test executions, and by the time to measure the performance impact of potential optimizations. To assess the overhead of the dynamic analysis, the "Overhead" columns of Table 3.2 illustrate the execution time of the test suites and benchmarks with and without profiling. The overhead for test suites ranges between 3x and 65x, which is comparable to other profilers [XAM+09, NSML13, TPG15, GPS15]. However, due to the complexity of some benchmarks, the overhead for these programs ranges between 16x and 1,210x. The time spent to measure the performance impact of optimizations ranges between 1 minute and several hours, depending on the number of affected tests and program's execution time. Since running *DecisionProf* does not any require manual interven-

[6] Pull request #2496 of Underscore.

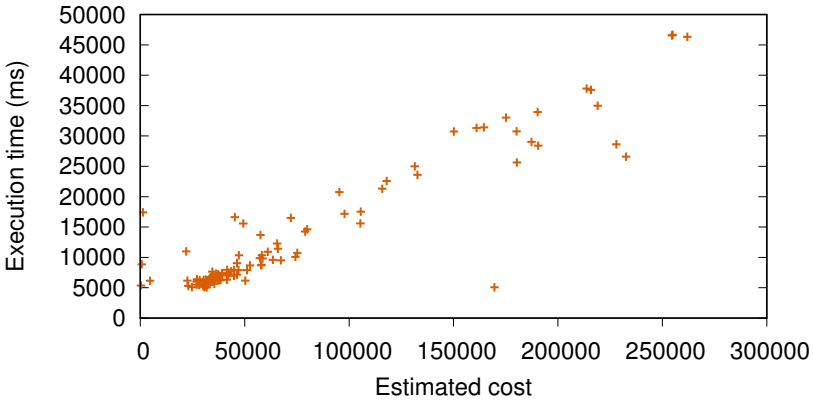


Figure 3.7: Correlation between estimated vs. actual cost.

tion and reports actionable suggestions, we consider the computational effort to be acceptable for developers.

3.5.4 Estimated vs. Actual Cost

To assess whether our proxy measure for execution time, the number of executed branching points, is an accurate estimate, we measure the correlation between both values for benchmarks and individual tests. To measure the execution time of benchmarks, we run each program ten times and keep the average value of all executions. To estimate the actual execution time of individual tests, we run each test ten times and keep all tests where the average execution time is above a minimum measurable time (5ms), resulting in 120 tests.

Figure 3.7 illustrates the correlation between execution time in ms and the estimated cost for individual tests. The correlation coefficient for unit tests and benchmarks is 0.98 and 0.92, respectively, which indicates a strong positive linear relationship between the number of evaluated branching points and execution time. We conclude that our proxy metric is an accurate approximation of execution time.

3.5.5 Guaranteeing That Optimizations are Semantics-Preserving

A conservative variant of our approach could report an optimization only if it can statically show the optimization to be semantics-preserving. To assess how effective this approach would be, we manually analyze whether the detected opportunities are amenable to a sound static analysis. We identify three critical challenges for such an analysis, which are at best partly solved in state of the art analyses for JavaScript. First, several opportunities involve function calls, which are not easy to resolve in JavaScript due to its dynamic features, such as dynamically overriding the methods of objects. Second, several opportunities involve calls of native functions, which a static analysis would have to model, including any variants of their implementation that may exist across the various JavaScript engines. Third, several opportunities involve property reads and writes, which might trigger arbitrary code via getter and

setter functions. A sound static analysis would have to show that properties are not implemented with getters and setters, or analyze the effects of them. Out of the 52 beneficial reordering opportunities, 28 involve at least one of these challenges, i.e., a conservative variant of our approach would likely miss these opportunities. These results confirm our design decision in favor of unsoundness, which turns out to be a non-issue in practice [LSS+15, SPL18, AMN17].

3.6 Summary

In this chapter, we present *DecisionProf*, a profiler that identifies optimization opportunities related to the order of evaluating subexpressions involved in a complex decision. The core idea is to profile the computational cost and the value of each check and to compute the optimal order of evaluating checks. We apply the approach to 9 real-world JavaScript projects and 34 benchmarks where it finds 23 previously unreported reordering opportunities that reduce the execution time in statistically significant ways.

The opportunities reported by *DecisionProf* are effective and actionable. They are effective because the approach assesses the performance impact of every optimization before reporting it, instead of requiring a developer to manually experiment with code changes. They are actionable because a developer must only decide whether to use a suggested optimization, instead of manually identifying a bottleneck and finding an optimization for it. As a result, our approach further increases the level of automation in optimizing the performance of a program compared to state of the art profilers.

Cross Language Optimizations in Big Data Systems

Chapters 2 and 3 illustrate how relatively small code changes can significantly improve the execution time of JavaScript applications. While this is true for JavaScript-based web applications, frameworks and libraries, the question is whether similar findings hold for complex, distributed applications that run simultaneously on multiple machines.

In this chapter, we demonstrate the potential of the *method inlining* code optimization in a large-scale data processing system. Method inlining is a simple program transformation that replaces a function call with the body of the function. We search for method inlining opportunities in programs written in SCOPE [CJL+08], a language for big-data processing queries that combines SQL-like declarative language with C# expressions.

To demonstrate the effectiveness of method inlining, Figure 4.1 illustrates two semantically equivalent SCOPE programs that interleave relational logic with C# expressions. Figure 4.1a shows the situation where the user implements the predicate in the `WHERE` clause as a separate C# method. Unfortunately, the presence of non-relational code blocks the powerful relational optimizations in the SCOPE compiler. As a result, the predicate is executed in a C# virtual machine. On the other hand, Figure 4.1b shows a slight variation where the user *inlines* the method body in the `WHERE` clause. Now, the predicate is amenable to two potential optimizations:

1. The optimizer may choose to *promote* one (or both) of the conjuncts to an earlier part of the script, especially if either `A` or `B` are columns used for partitioning the data. This can dramatically reduce the amount of data needed to be transferred across the network.

2. The SCOPE compiler has a set of methods that it considers to be *intrinsics*. An intrinsic is a .NET method for which the SCOPE runtime has a semantically equivalent native function, i.e., implemented in C++. For instance, the method `String.isNullOrEmpty` checks whether its argument is either null or else the empty string. The corresponding native method is able to execute on the native data encoding, which does not involve creating any .NET objects or instantiating the *CLR*, i.e., the .NET virtual machine.

The resulting optimizations improve the throughput of the SCOPE program by as much as 90% percent.

```
data = SELECT *
  FROM inputStream
  WHERE M(A, B);

#CS
bool M(string x, string y) {
  return !String.IsNullOrEmpty(x) && y == "Key1";
}
#ENDCS
```

(a) Predicate invisible to optimizer.

```
data = SELECT *
  FROM inputStream
  WHERE !String.IsNullOrEmpty(A) AND B == "Key1";
```

(b) Predicate visible to optimizer.

Figure 4.1: Examples of SCOPE programs.

The cross-language interaction in SCOPE, i.e., between the native and C# runtimes, poses a significant cost in the overall system. The goal of this chapter is to study and better understand the key performance bottlenecks in this modern data-processing system and demonstrate the potential for cross-language optimization based on method inlining. While this work is primarily about SCOPE, we believe our results and optimizations generalize to other data-processing systems.

To motivate a need for reducing cross-runtime interactions, we first present a new profiling infrastructure based on a combination of offline static analysis of the executed code in addition to low-overhead online measurements captured by SCOPE's runtime. Then, we describe the results of our profiling of over 3 million SCOPE programs across five data centers within Microsoft. We find that programs with non-relational code take between 45-70% of data center CPU time. Furthermore, we propose a novel static analysis to find inlining opportunities. By inlining a method call, the compiler/optimizer becomes aware of the logic contained in the body of the method and can trigger additional optimizations. Finally, we discuss the effectiveness of such optimizations in six case studies by optimizing jobs from 5 different teams at Microsoft.

4.1 Background

SCOPE [CJL+08] is a big data query language and it combines a familiar SQL-like declarative language with the extensibility and programmability provided by C# types and the C# expression language. In addition to C# expressions, SCOPE allows user-defined functions (*UDFs*) and user-defined operators (*UDOs*).

4.1.1 Execution of a Script

A SCOPE program is a script implemented as a directed acyclic graph (DAG) where each vertex is a set of operators executed on the same physical (or virtual) machine. We use the term *node* for the physical or virtual machine that a vertex is executed on. The edges of the DAG are communication channels that use a high-speed communication network between nodes. The operators within a vertex are the end product of a very sophisticated optimizer; expressions written within a certain construct in the script may end up being executed in vertices that do not correspond to the construct in a simple manner. For instance, a sub-expression from a `WHERE` clause, *filter*, may be *promoted* into a vertex that extracts an input table from a data source, whereas the rest of the filter may be in a vertex that is many edges distant from the input layer. An execution of a script is called a *job*.

4.1.2 Intrinsics

Much like Hadoop streaming [DQRJ+10], SCOPE jobs consist of multiple runtimes and languages and while the details of this chapter are about SCOPE, the general problem is shared among many big data systems. The SCOPE compiler attempts to generate both C++ and C# operators for the same source-level construct. Each operator, however, must execute either entirely in C# or C++: mixed code is not provided for. Thus, when possible, the C++ operator is preferred because the data layout in stored data uses C++ data structures. For example, a simple projection of a subset of the columns can be done entirely without using the CLR. But when a script contains a C# expression that cannot be converted to a C++ function, such as in Figure 4.1a, the CLR must be started and each row in the input table must be converted to a C# representation, i.e., a C# object representing the row must be created, before the C# expression can be evaluated in the CLR.

Because this can be inefficient, the SCOPE runtime contains C++ functions that are semantically equivalent to a subset of the .NET Framework methods that are frequently used; these are called *intrinsics*. The SCOPE compiler then emits calls to the (C++) intrinsics in the C++ generated operator, which is then used at runtime in preference to the C# generated operator. (As opposed to using *interop* to execute native code from within the CLR.)

4.1.3 Compiler/Optimizer Communication

In general, the C# code is compiled as a black box: no analysis or optimization is peformed at this level. One consequence is that any calls to a UDF within a SCOPE expression (filter predicate, projection function) require the operator containing the call to be implemented in C#.

4.2 Profiling Infrastructure for Data Centers

SCOPE jobs run on a distributed computing platform, called Cosmos, designed for storing and analyzing massive data sets. Cosmos runs on five clusters consisting of

thousands of commodity servers [CJL+08]. Cosmos is highly scalable and performant: it stores exabytes of data across hundreds of thousands of physical machines. Cosmos runs millions of big-data jobs every week and almost half million jobs every day.

Finding optimization opportunities that are applicable to such a large number of diverse jobs is a challenging problem. We can hope to find interesting conclusions only if our profiling infrastructure is scalable. To achieve this, the important aspect to consider is *what type of information we should analyze*. In the following sections, we describe our major decisions when building infrastructure for profiling big data jobs.

4.2.1 Job Artifacts

After execution of a SCOPE job, the runtime produces several artifacts that contain code and runtime information for every job stage. Job artifacts are indefinitely stored within Cosmos itself in a *job repository*. This provides two benefits: we can derive data for a relatively large number of jobs since we do not require re-running them, and we can also answer more complex, but interesting questions, such as which job stages run as C++ vs. C#. We provide an overview of the subset of artifacts which we use to profile the data center.

Job Algebra The job algebra is a graph representation of the job execution plan. Job vertices are presented as outer-most nodes in a graph. Each job vertex contains all operators that run inside that vertex and an operator can be either user-defined or native. Optionally, if all operators are native, the vertex can be marked with the *nativeOnly* flag, indicating that the entire vertex runs as native (C++). However, it does not distinguish between native and user-defined operators.

Runtime Statistics The runtime statistics provides information on execution time for every job vertex and every operator inside the vertex. Among other statistics it includes CPU times, which we use as the primary metric of performance. Big data systems process significant amounts of data but often are CPU-bound[ORR+15] due to the large overheads behind serialization and de-serialization so we measure (in addition to bytes read and written) CPU time.

Generated Code The SCOPE compiler generates both C# and C++ code for every job. An artifact containing the C++ code has for every vertex a code region containing a C++ implementation of the vertex and another code region that provides class names for every operator that runs as C#. An operator in a vertex is translated to either C++ code or to C# code depending on whether an operator contains sources of C# code that are not intrinsics. An artifact containing the C# code includes implementations of non-native operators and user-written classes and functions defined inside the script. Both source and binary are available for the generated code.

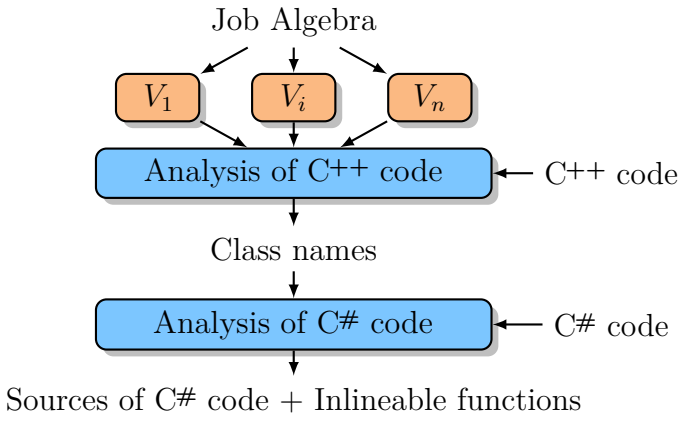


Figure 4.2: Overview of the static analysis.

4.2.2 Static Analysis

After collecting the artifacts described in Section 4.2.1, we perform static analysis to detect different sources of C# code in every vertex of a job. This is important to understand the opportunities for optimizing job vertices through C# to C++ translation. For instance, an operator can run as managed code due to a call to a method from .NET framework, or because of more complex C# code.

Figure 4.2 gives an overview of our analysis. It has two main components: *Analysis of C++ code* and *Analysis of C# code*. Each analysis performs at the granularity of a job vertex. The goal is to look for opportunities to run an entire vertex as C++ code, which would remove all steps of data serialization between user and native operators within the vertex.

The first step of the analysis is to extract the names of each job vertex, which serves as a unique identifier for the vertex. Then for each vertex, the analysis parses the generated C++ to find the class containing the vertex implementation. As discussed in Section 4.2.1, for each vertex, the C++ implementation contains two code regions: one that indicates which part of a vertex runs as C++ code and another region listing class names of operators that run as C# code. If the list of C# operators is empty, we conclude that the entire vertex runs as C++ code. Otherwise, the analysis outputs class names that contain C# operators. Then, it parses C# code to find definition and implementation for every class name. For a managed operator there are two possible sources of C# code: generated code, which we whitelist and skip in our analysis and the user-written code. After analyzing user code, the final sources of C# code are:

- .NET framework calls
- User written functions
- User written operators

We are particularly interested in the first two categories. It is not unusual for an operator to run as C# just because of a single call to a framework method. The idea is to find what are the most important framework methods because we can optimize large number of vertices by providing native implementations for those methods.

Finding Sources of Managed Code To find .NET framework calls, it is enough to check whether a method definition comes from one of a small set of the core binaries in the .NET runtime. The analysis finds user-written functions by looking for their definition inside the script or in third-party binaries. Because the job repository keeps only binaries of third-party projects, we further analyze only user functions for which the source code is available. It is easier to optimize these functions through inlining (as described in Section 4.2.2), because we can manually confirm the correctness of inlined code. Finally, in SCOPE, users can easily implement their own operators: extractors (for parsing and constructing rows from a file), processors and reducers (for row processing), and combiners (for processing rows from two input tables). The analysis finds user operators by checking the interface the class implements. We do not consider user operators for C++ translation: they generate quite complex code which would be non-trivial to translate into C++.

Analysis of User-Written Code Inlining of a user-written function refers, per the standard definition, to replacing the call to the function in the script with the body of the function. We define *inlineable* methods as follows.

Definition 4 (Inlineable method)**.** Method m is *inlineable* if it has the following properties:

- It contains only .NET framework calls
- It does not contain loops and try-catch blocks
- It does not contain any assignment statements.
- It does not contain any references to the fields of an object.
- For all calls inside the method, arguments are passed by value (i.e., no *out* parameters or call-by-reference parameters).

Furthermore, we distinguish among inlineable methods those that allow for C++ generation, because all called .NET framework methods are *intrinsics*. However, the analysis of other inlineable functions is important, because it provides the intuition on how many vertices are potentially optimizable in this way.

4.3 Evaluation

We analyze over 3,000,000 SCOPE jobs over a period of six days that run on five data centers at Microsoft. In summary, our experiments answer the following questions:

- *What is the proportion of time spent in native vs. non-native job vertices?* Between 43.70 % and 73.32 % of data center time is spent in job vertices that run managed code (Section 4.3.2).

Table 4.1: Analyzed jobs and their CPU time.

Data center	Number of jobs	CPU time (in hours)
cosmos8	375,974	28,559,063
cosmos9	171,203	40,714,052
cosmos11	851,222	23,312,271
cosmos14	474,911	21,299,039
cosmos15	1,200,026	31,324,407
Total:	3,073,336	145,208,834

- *What proportion of time can be optimized by inlining UDFs using the current list of intrinsics?* Given the set of UDFs we found in our survey and the current list of intrinsics we can optimize up to 0.16 % of data center time (Section 4.3.3). Considering the size of data centers at Microsoft, this percentage translates to almost 100,000 hours spent in big data jobs.

- *What proportion of time can be optimized by extending the list of intrinsics? Which methods should be the most important for C++ implementation?* By increasing the list of intrinsics and optimizing all inlineable methods we can optimize up to 6.76 % of data center time. Furthermore, we conclude that *String* methods are the most important .NET framework methods amenable for C++ implementations (Section 4.3.4).

4.3.1 Experimental Setup

To understand performance bottlenecks in SCOPE jobs we analyze over 3,000,000 jobs across 5 data centers. Table 4.1 lists, for each data center, the number of analyzed jobs along with their CPU time measured in hours. We observe that number of jobs and CPU time significantly vary between data centers. This is expected because different data centers are usually tailored for different types of jobs. For example, data center *cosmos9* runs big data machine learning jobs, which are among the most expensive jobs. However, data center *cosmos15* runs the highest proportions of jobs we analyze because it is mostly used for running relatively simple, short running jobs.

The table shows that the jobs take a significant amount of resources. While we do not have access to the actual cost of these jobs, a very conservative estimate is to use 0.6 cents an hour, the cost of the cheapest Amazon EC2 instance [1] at the time of writing. Given that most of these jobs run recurrently every day (some every hour), extrapolating these costs amounts to over 650 million dollars per year. Thus, even a 1% performance improvement on these jobs will result in significant performance savings.

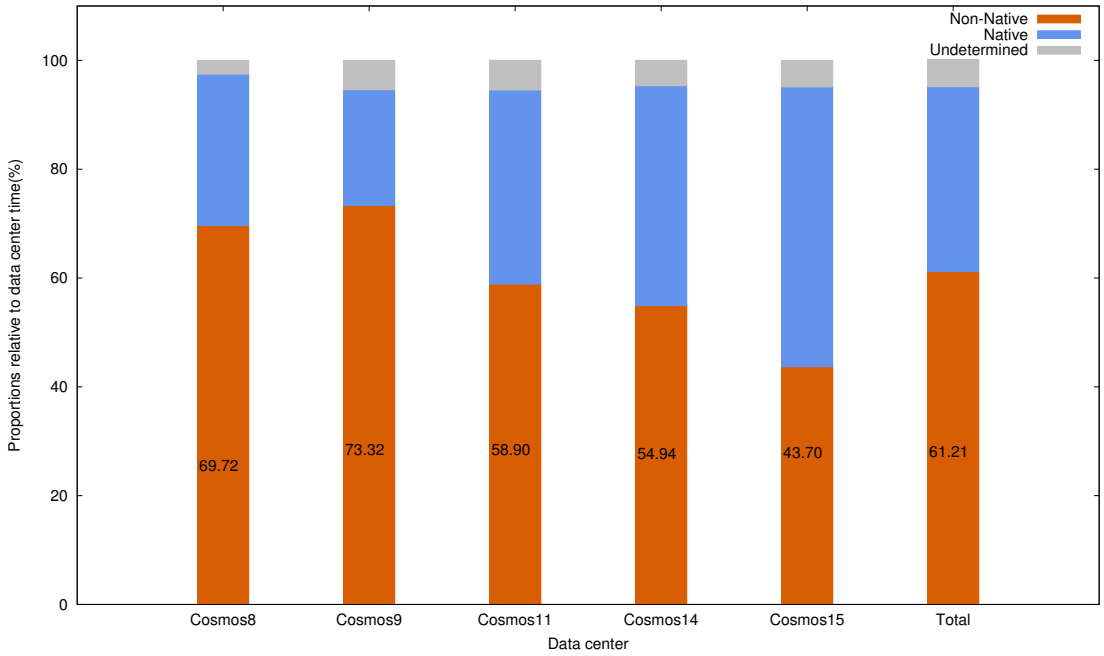


Figure 4.3: Time spent in native vs. non-native vertices.

4.3.2 Native vs. Non-Native Time

The first goal of our analysis is to determine the amount of time spent between native vertices and vertices containing non-native code in SCOPE jobs. As mentioned in Section 4.1.2, SCOPE runs all relational logic efficiently in native code, while user-defined non-relational code is run in the CLR. Apart from avoiding the inherent overheads of running in non-native mode, the relational logic has the additional advantage of using all of the traditional optimizations modern databases typically perform. From prior analyses, it was known that around 80% of the SCOPE jobs use only relational constructs and thus run purely natively. Thus, at the outset, it was not obvious that non-relational optimizations would provide overall datacenter performance improvements.

Figure 4.3 shows for every data center the time spent executing native vertices versus vertices with non-native code. For this analysis, we combined the time taken by every job vertex from *Runtime Statistics* with the analysis of C++ code that determines whether each operator within a vertex is run in native or non-native mode. Our analysis of the C++ code is conservative and reports an operator as running in non-native mode only if the analysis is able to detect the source of the C# code. Due to this conservative analysis, we tag some vertices as undetermined if the job metadata claims to include non-native operators but we are unable to detect the source. Modulo bugs in the SCOPE job metadata, these undetermined vertices are likely to be non-native vertices. But without improving our analysis we are unable to confirm this.

Figure 4.3 shows that the time spent in vertices with non-native operators represents a large fraction of data center time, ranging from 43.7% for cosmos15 to

[1]as of August 2017

73.3% for cosmos9. We can derive many conclusions from these results. First, these results could reflect the fact that the decades of work in optimizing relational code has borne fruit — purely relational components account for a smaller percentage of datacenter runtime. Second, it could very well be the case that jobs with inherently expensive computations require logic that does not fit within the relational subset of SCOPE, and thus requires the use of non-native code. Finally, these results could reflect the inherent overheads of running non-native code in the context of big-data processing.

We did preliminary experimentation on a small subset of the jobs locally to study the performance bottlenecks of SCOPE jobs with managed operators. Our profiles show that the presence of non-relational components reduce the throughput of a job by a factor of 10× or more, with the performance bottleneck being the serialization/deserialization overhead of converting data into and from C# objects. Note that we are unable to run most of the jobs locally as accesses to the data they process is severely restricted due to privacy concerns. Thus, it is quite possible that the results from our preliminary experimentation might not be representative of the jobs that run on the datacenters. Nevertheless, conversations with the SCOPE team validated these experiments, and Figure 4.3 shows the potential performance improvements possible by optimizing the interaction between native and non-native parts of SCOPE.

4.3.3 Optimizable Job Vertices

We say a job vertex is *optimizable* if its only source of managed code comes from inlineable methods that in turn have only calls to existing intrinsics. An example of such a job is shown in Figure 4.1. This is an extremely conservative definition, but it allows us to quantify how much data center time we can optimize given the current list of intrinsics. Moreover, by inlining method calls, we expect an entire vertex to run as native code, which should significantly improve the vertex execution time.

Figure 4.4 shows the proportion of CPU time of *optimizable* vertices relative to data center time and to time spent in vertices with non-native code. We observe that with the current list of intrinsics we can optimize a relatively small proportion of data center time. For example, in cosmos9 that runs the most expensive jobs, we can optimize at most 0.01% of data center time. The situation is slightly better in cosmos14 and cosmos15, but in these data centers, the proportion of non-native time is relatively lower compared to cosmos9.

The crucial observation is that given results illustrate only the time in data centers that can be affected by inlining method calls in optimizable vertices. To measure the actual performance improvement it is necessary to rerun every optimized job. Further details on performance improvements for several jobs we optimize are given in Section 4.4.

4.3.4 Potentially Optimizable Job Vertices

To motivate the importance of providing the C++ implementation for more framework methods, we measure how much time is spent in the following type of vertices:

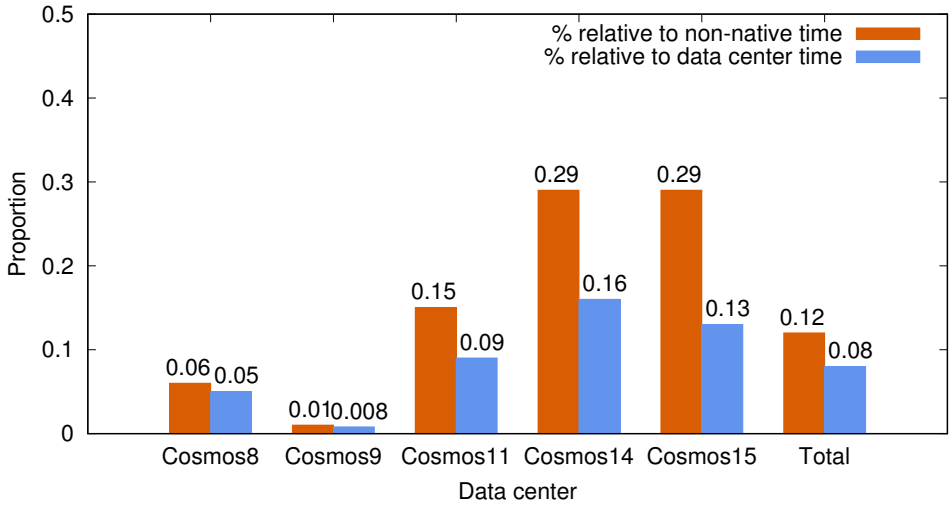


Figure 4.4: Optimizable job vertices.

- Vertices with .NET framework calls as the only source of managed code
- Vertices with .NET framework calls or calls to inlineable methods as the only source of managed code

We call these vertices *potentially optimizable*, because they can run as native by increasing the list of intrinsics.

Figure 4.5 shows the proportion of time spent in potentially optimizable vertices relative to data center time. We measure the proportions by assuming that all .NET framework methods have C++ implementation. Results illustrate that we can optimize between 1.01 and 6.76 % of data center time by just increasing the list of intrinsics. Even though 1.01 % of time spent in cosmos9 looks relatively low, it counts for almost 407,140 CPU hours for a period of several days. Knowing this type of impact motivates the future work on enabling more C++ translation of framework methods.

Most Relevant .NET Framework Methods Assuming that all .NET framework methods have C++ implementation is unrealistic. To provide more insights on the framework methods that actually matter, we perform two types of study: the study of the most relevant methods considering the execution time of a vertex and the study of the most important method types.

For the first study, we take all .NET framework methods called in potentially optimizable vertices and rank them based on the vertex execution time. Table 4.2 shows for every data center ten most important framework methods. The last row further illustrates how much data center time can be optimized if all methods in the list become intrinsics. We further highlight methods that appear to be relevant across many data centers. For example, *System.String.ToLower* and *System.String.Concat* are among the most relevant methods across all data centers. Furthermore, if the native implementation is provided for the first ten methods in cosmos14, it would be enough to optimize more than 5% of data center time.

Table 4.2: Most relevant .NET Framework methods per data center. All methods are within the *System* namespace. Methods in bold are those that appear in the top 10 in at least 3 of the five data centers.

Cosmos8	Cosmos9	Cosmos11	Cosmos14	Cosmos15
Convert.ToInt64	**String.Equals**	String.Replace	**DateTime.ToString**	**String.ToLower**
Int32.Parse	**String.ToLower**	**String.ToLower**	String.IndexOf	String.LastIndexOf
String.ToLower	Int32.Parse	String.ToUpper	DateTime.ToLocalTime	**DateTime.ToString**
String.Concat	String.Replace	**String.Concat**	**String.ToLower**	**String.Concat**
String.Replace	Convert.ToDateTime	String.Trim	String.ToUpper	Convert.ToUInt64
Double.Parse	Regex.isMatch	Math.Max	Regex.IsMatch	Enumerable.SelectMany
Math.Round	DateTime.ToUniversalTime	**String.Equals**	**String.Equals**	Enumerable.Distinct
Char.NewArr	**String.Concat**	TimeSpan.Days	**String.Concat**	String.Format
String.ToUpper	TimeSpan.Days	**DateTime.ToString**	String.Trim	**String.Equals**
String.Upper	DateTime.Subtract	String.ToCharArray	String.Split	String.IndexOf
1.27%	0.63%	1.61%	5.15%	1.8%

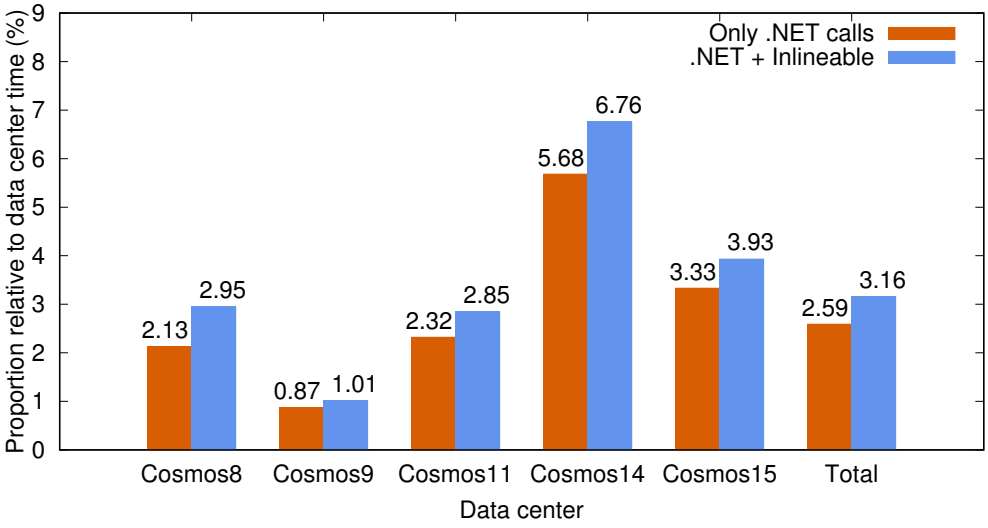


Figure 4.5: Potentially optimizable job vertices

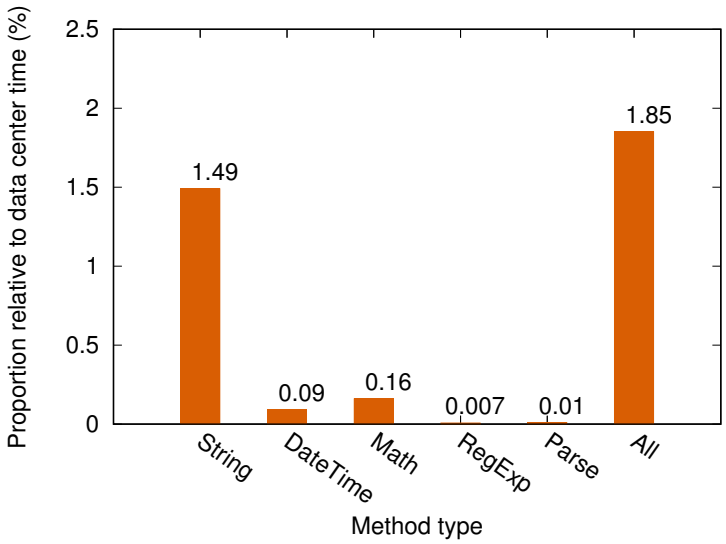


Figure 4.6: Relevance of .NET framework method types (cosmos11).

Figure 4.6 illustrates the most important method types for data center cosmos11. The results are comparable for other data centers. *String* methods dominate and they count as the only source of non-native code in 1.49% of the time spent in potentially optimizable vertices. Other method types are significantly less relevant, but when combined they influence 1.85% of the data center time. These studies show the potential for improving data center performance by providing more intrinsics and thus enabling more C++ translation.

4.4 Case Studies

In order to quantify the effects of optimizing the SCOPE scripts through method inlining, we performed several case studies. We reported jobs that have *optimizable*

Table 4.3: Summary of case studies. The reported changes are percent improvements in CPU time and throughput.

Job Name	*C++ translation*	*Job Cost*	*CPU time*		*Throughput*
			Vertex Change	*Job Change*	
A	yes	medium	59.63%	23.00%	30%
B	yes	medium	no change	no change	no change
C	yes	low	41.98%	25.00%	38%
D	no	-	-	-	-
E	yes	high	7.22%	4.79%	5%
F	yes	low	no change	no change	115%

vertices, meaning that the job owner can optimize the script by inlining a method that calls only intrinsics.

Because the input data for each job is not available, we had to contact the job owners and ask them to re-run the job with a manually-inlined version of their script. We were able to have 6 jobs re-run by their owners, categorized by their total CPU time: short, medium and long.

4.4.1 Optimizations with Effects on Job Algebra

As illustrated by Figure 4.1, the optimizer may choose to modify the job algebra given the new information available to it. For example, predicates might be pushed deeper into the DAG which can result in dramatic data reduction. However, none of the case studies ended up causing this kind of optimization.

4.4.2 Optimizations without Effects on Job Algebra

Even if the physical plan does not change, the resulting program might be more efficient if it avoids the native to managed transition. For SCOPE, the set of intrinsics means that by lifting more non-relational code into the parts of the program where such things are visible to the optimizer, more code can be executed in C++ instead of in C#.

In total, we looked at 6 re-run jobs, summarized in Table 4.3. For one job (D), the optimization did not trigger C++ translation of an inlined operator because the operator called to a non-intrinsicable method that we mistakenly thought was an intrinsic. After detecting this problem, we fix the set of intrinsics and use the new set to obtain data presented in Section 4.3.

For jobs A and B, we were able to perfom the historical study over a period of 18 days. Both jobs are medium-expensive jobs, run daily and contain exactly one optimizable vertex due to a user-written functions. In both cases, inlining the function resulted in the entire vertex being executed in C++. Figure 4.7 shows the improvements in CPU time and throughput for an optimized vertex in Job A over an 18 day period, the last 5 of which were with the inlined method. The values are normalized by the average of the unoptimized execution times; the optimized version saves approximately 60% of the execution time. However, the normalized

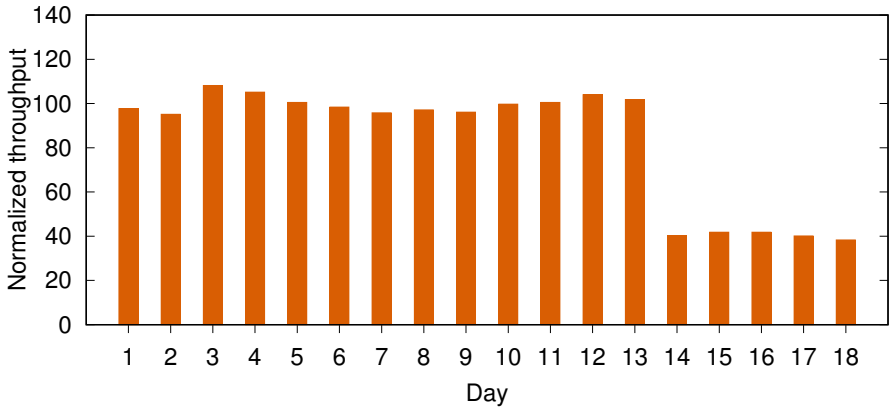


Figure 4.7: Case Study A .

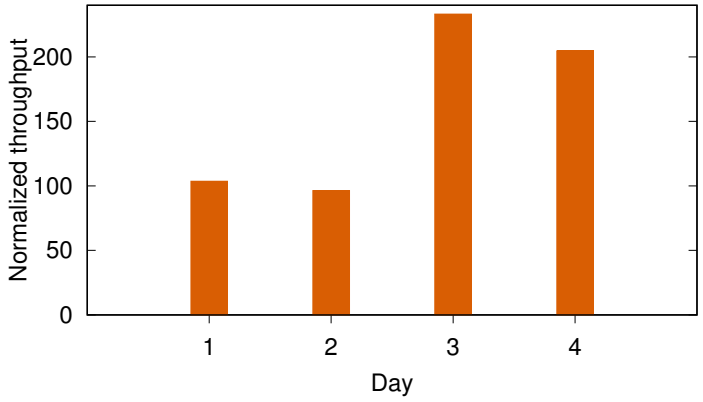


Figure 4.8: Case Study F .

vertex CPU time in Job B does not show any consistent improvement for the last five jobs. Closer analysis of the vertex shows that the operator which had been in C# accounted for a very tiny percentage of the execution time for the vertex. This is consistent with our results for Job A, where the operator had essentially been 100% of the execution time of the vertex.

We also optimized Job F, a very low cost job. It only runs a few times a month, so we were able to obtain timing information for only a few executions. The vertex containing the optimized operator accounted for over 99% of the overall CPU time for the entire job. We found the CPU time to be highly variable; perhaps this is because the job runs so quickly so it is more sensitive to the batch environment in which it runs. However, we found the throughput measurements to be consistent: the optimized version provided twice the throughput for the entire job (again, compared to the average of the unoptimized version).

Finally, for jobs C and E we were not able to perform the same kind of historical study: instead we have just one execution of the optimized scripts. For this execution we found improvements in both vertex and job CPU times.

4.5 Threats to Validity

Underapproximation of Performance Impact The amount of time that can be optimized by either increasing the list of intrinsics or method inlining is an underapproximation of the total optimizable time. We do not consider effects of inlining on another compiler optimizations. For example, if a method is inlined on a column used to partition data, the inlining would not only trigger more C++ generation, it would also enable filter promotion [Dar78]. Understanding the impact of inlining on other compiler optimizations is left for future work.

Assumptions for Static Analysis Our static analysis detects sources of C# code based on several assumptions. For example, we use naming conventions when pruning generated methods in C# implementation of non-native operators. A user can potentially call some of these methods in the script, meaning that we would skip a valuable source of user-written C# code. However, in practice, such methods are not used in the context of big-data jobs and our manual exploration of many SCOPE scripts illustrates that our assumptions hold.

Challenges for Implementing More Intrinsics We discuss the relevance of providing C++ implementation for more .NET Framework methods. However, providing C++ translation for some of these methods poses several challenges. For example, memory management in C# is very different because it has a garbage collector, while C++ does not. Another challenge is related to different string encodings in C# and C++ runtimes, and for some corner cases, there is no clear one-to-one mapping. However, increasing the list of intrinsics would certainly bring significant performance benefits in SCOPE jobs, and there is a clear motivation for future work to address this problem.

4.6 Summary

We believe having an expressive general-purpose language like C# or Java integrated into a big data query language is a good thing: programmers should be able to re-use existing components in languages that they are already comfortable with. However, such multi-language paradigms break the barriers that current program analysis and optimization tools are based on.

In this chapter, we propose a new profiling infrastructure for a large-scale data processing system and an approach to find optimization opportunities based on method inlining. The idea is to statically analyze job artifacts and to propose source code optimizations only if they enable more compiler optimizations by removing unnecessary cross-runtime interactions. By empirically studying over 3 million

SCOPE programs across five data centers within Microsoft, we find that up to 0.3% of data center time can be optimized by inlining method calls. Furthermore, we present six case studies showing that triggering more generation of native code in SCOPE programs yields significant performance improvement: inlining just one method resulted in as much as 25% improvement for an entire program.

Test Generation of Higher-Order Functions in Dynamic Languages

In Chapter 3 we proposed *DecisionProf*, a dynamic analysis for optimizing inefficient orders of evaluations. To find reordering opportunities, *DecisionProf* relies on inputs provided by test suites. Similarly, other dynamic analyses are applied with manually written tests or by manually exploring the program. However, such inputs are often not sufficient to cover all possible program paths or to trigger behavior that is of interest to the dynamic analysis.

To address the problem of insufficient test inputs, a possible solution is to use test generation in combination with dynamic analysis. Automatically generated tests can either extend manual tests or serve as the sole driver to execute applications during dynamic analysis. Existing test generation uses a wide range of techniques, including feedback-directed random testing [PE07, PLB08a], symbolic execution [Kin76, CDE08], concolic execution [GKS05a, SMA05], bounded exhaustive testing [BKM02], evolutionary test generation [FA11], UI-level test generation [Mem07, MvD09, SGP17], and concurrency testing [PG12, SR14].

For dynamic analysis to be precise, test generation must provide high quality test cases. This means that generated tests should exercise as many execution paths as possible and achieve good code coverage. However, despite their effectiveness in identifying programming errors, current test generation approaches have limited capabilities in generating structurally complex inputs [ZZK16]. In particular, they do not consider higher-order functions that are common in functional-style programming, e.g., the popular `map` or `reduce` APIs, and in dynamic languages, e.g., methods that compose behavior via synchronous or asynchronous callbacks.

Testing a higher-order function requires the construction of tests that invoke the function with values that include callback functions. To be effective, these callback functions must interact with the tested code, e.g., by manipulating the program's state. Existing test generators do not address the problem of higher-order functions at all or pass very simple callback functions that do not implement any behavior or return random values [CH11].

The problem of generating higher-order functions is further compounded for dynamically typed languages, such as JavaScript, Python, and Ruby. For these languages, in addition to the problem of creating an effective callback function, a test generator faces the challenge of determining where to pass a function as an

argument. Addressing this challenge is non-trivial in the absence of static type signatures.

This chapter tackles the problem of automatically testing higher-order functions in dynamic languages by presenting a novel test generation framework called *LambdaTester*. In this framework, test generation proceeds in two phases. The *discovery phase* is concerned with discovering, for a given method[1] under test m, at which argument position(s) the method expects a callback function. To this end, the framework generates tests that invoke m with callback functions that report whether or not they are invoked. Then, the *test generation phase* creates tests that consist of a sequence of calls that invoke m with randomly selected values, including function values at argument positions where the previous phase discovered that functions are expected. Both phases take as input setup code that creates a set of initial values, which are used as receivers and arguments in subsequently generated calls.

We present several instantiations of the *LambdaTester* framework that differ in the way in which callback functions are constructed during the test generation phase. These instantiations include the use of: (i) empty functions, (ii) functions that return random values [CH11], (iii) callbacks mined from a corpus of existing code, and (iv) a novel feedback-directed technique that generates callbacks using guidance from a dynamic analysis. Technique (iv) observes memory locations that are read during the execution of previously generated tests and generates function bodies that write to those locations.

We implement our ideas in a test generation tool for JavaScript. In an empirical evaluation, we use *LambdaTester* to generate tests for 43 higher-order functions in 13 popular JavaScript libraries. These libraries provide so-called *polyfills*, i.e., JavaScript implementations of APIs that may not be provided natively by all execution environments of the JavaScript language. We apply *LambdaTester* to polyfills for array APIs, including the *es5-shim*, *mozilla*, and *polyfill.io* libraries, and to polyfills of the promise APIs, including *bluebird*, *Q*, and *when*. To evaluate the effectiveness of the generated tests, we execute the tests both with the library implementation and the corresponding native implementation of the tested API, and detect situations where their behaviors differ. Here, behavioral differences are detected using an automated test oracle that compares execution behavior, e.g., the values being returned by the methods under test, the invocations of callback functions passed as arguments, the output written by the tested code, and whether the methods under test terminate.

Our experimental evaluation shows that *LambdaTester* reveals various behavioral differences between polyfills and their corresponding native implementations, including previously unknown bugs in popular polyfills. Overall, the approach detects differences in 12 of 13 libraries. Comparing the different techniques for creating callback functions shows that callbacks that modify program state in non-obvious ways are more effective than simpler approaches. The most effective technique for creating callbacks is our novel feedback-directed technique, exposing differences missed by all other techniques.

[1]We use the terms "function" and "method" interchangeably in this work because our approach tests methods while the term "higher-order function" is well established.

```
198 Array.prototype.map = function map(callback) {
199   if (this === undefined || this === null) {
200     throw new TypeError(this + ' is not an object');
201   }
202   if (typeof callback !== 'function') {
203     throw new TypeError(callback + ' is not a function');
204   }
205
206   var object = Object(this), scope = arguments[1],
207       arraylike = object instanceof String ? object.split('') : object,
208       length = Math.max(Math.min(arraylike.length, 9007199254740991), 0) || 0,
209       index = -1, result = [];
210
211   while (++index < length) {
212     if (index in arraylike) {
213       result[index] = callback.call(scope, arraylike[index], index, object);
214     }
215   }
216
217   return result;
218 };
```

Figure 5.1: Implementation of `Array.prototype.map` from polyfill.io.

5.1 Challenges and Motivating Examples

This section presents and illustrates challenges associated with generating effective tests for programs with higher-order functions in the context of dynamically typed languages. Given one or more methods under test m that expect a callback function cb as an argument, we identify five challenges for testing m:

- *C1: Determining where m expects a callback function as an argument.*
- *C2: Generating callback functions cb that modify memory locations in such a way that it influences the behavior of m.*
- *C3: Generating callback functions cb that return values that influence the behavior of m, or that modify properties of objects passed into m as the receiver or as arguments.*
- *C4: Generating tests that chain multiple calls to higher-order functions.*
- *C5: Detecting callback-related behavioral differences during the execution of m.*

The above challenges are relevant for any code that uses higher-order functions in a dynamically typed language. We now illustrate these challenges using two examples. Both examples are concerned with generating tests that expose bugs in *polyfills* for JavaScript, i.e., code that implements a feature that is unavailable in cases where a user is running an application using an outdated version of a browser or JavaScript engine.

```
219 p = ["a", "b", "c"];
220 q1 = p.map(function(v){ return v+v; });  //["aa","bb","cc"]
221 q2 = p.map(function(v){ p.length = false; return v+v; });  //["aa"]
```

Figure 5.2: Examples of map method.

5.1.1 Array.prototype.map

Figure 5.1 shows the implementation of `Array.prototype.map` from *polyfill.io* version 3.25.1. This code provides an implementation of the `map` method for use on platforms that predate JavaScript 1.6, where the method was introduced. A brief review of the code reveals that lines 199–201 check that the receiver is an object and throw a `TypeError` otherwise, and that lines 202–204 check that the callback is a function and throw a `TypeError` otherwise. On lines 206–209, several variables are initialized. If the function is invoked on an array, the variables `arraylike` and `length` contain the array and the array's length, respectively. If the receiver object is a string value, then variable `arraylike` is initialized to an array of which the elements contain the string's characters. Variable `result` is initialized to an empty array. The loop on lines 211–215 visits each index in the original array, looks up the value at that index, computes a new value by invoking the callback function, and stores the result at the corresponding index in the `result` array.

The code in Figure 5.1 computes the expected results on most but not all inputs. For example, consider the call to `map` on line 220 in Figure 5.2. The function in Figure 5.1 assigns to `q1` an array `["aa", "bb", "cc"]` as expected. However, the result of the second `map` call on line 221 is equal to `["aa"]`, whereas the native implementation of `Array.prototype.map` assigns to `q2` an array `["aa", undefined, undefined]`.

Detecting this behavioral difference requires a test that passes a callback function as the first argument (see C1) and this callback function should manipulate the `length` property of the array `p` on which the `map` method is invoked (see C2). While existing test generators for JavaScript are able to generate simple callback functions, we are not aware of a previous technique that generates callback functions that modify specific properties of objects passed in as arguments, which is necessary to expose the bug in the `map` method in Figure 5.1. In this work, we explore a technique that identifies object properties, such as `arraylike.length`, that are read in the method under test, and that generates callbacks that deliberately manipulate these properties.

5.1.2 Promises

Promises are a mechanism for asynchronous programming that was introduced in the ECMAScript 6 specification. A promise represents the value of an asynchronous computation, and it is in one of three states: pending, fulfilled, or rejected. Initially, a promise is in the pending state, and it transitions to the fulfilled or rejected state when functions `resolve` or `reject` are invoked, passing a value as an argument. To enable programmers to associate *reactions* with a promise, promises define

```
222 var p0 = new Promise(function(resolve,reject){ resolve(undefined) });
223 var p1 = p0.then(function(v){ return p1; });
224 var p2 = p1.then(function(v){ console.log("Value: " + v); });
225 var p3 = p2.catch(function(e){ console.log("Error: " + e); });
```

(a) Example of the circular promise chain.

```
226 var p0 = new Promise(function(resolve,reject){ resolve(7) });
227 var p1 = p0.then(undefined);
228 var p2 = p1.then(function(v){
229   console.log(v);
230   return v+1;
231 });
```

(b) Repeated calls to `then`.

```
232 var p = Promise.reject(17);
233 p.catch(function (){ console.log("hello"); },null,false);
```

(c) Call to `catch` with multiple arguments.

Figure 5.3: Examples of promise calls.

higher-order functions `then` and `catch`, which receive callback functions that execute asynchronously when that promise is resolved or rejected. These operations enable programmers to create a chain of asynchronous computations and propagate errors from one asynchronously executed function to the next.

At the time of writing this chapter, a popular web site[2] lists 76 polyfill implementations of JavaScript promises that aim at conforming to the Promises/A+ specification[3] upon which the ECMAScript 6 specification is based. Testing these implementations is a challenging task for several reasons:

- `then` and `catch` can be invoked with arguments that are functions but also with non-function values,
- `then` and `catch` return another promise, thus enabling programmers to create a chain of asynchronous computations, and
- `then` can be invoked with one argument to define a fulfill reaction, or with two arguments, to define both a fulfill reaction and a reject reaction, and
- the behavior of reactions defined using `then` and `catch` depends on the fact whether or not the returned value is a promise.

As we discuss in Section 5.4, our test generation technique finds numerous test cases that expose situations where polyfill implementations behave differently from the native implementation. For example, consider the example in Figure 5.3a. When this test is executed using Node.js 8.5.0, i.e., the native implementation, then it prints:

[2] `https://promisesaplus.com/implementations`
[3] `http://wiki.commonjs.org/wiki/Promises/A`

```
Error: TypeError: Chaining cycle detected for promise #<Promise>
```

The *bluebird* promise polyfill[4] prints a slightly different message:

```
Error: TypeError: circular promise resolution chain
```

These messages allude to the fact that the value being returned by the callback function on line 223 is the same value that is being returned by the call to `then` on the same line. However, the *Q* promise polyfill[5] fails to perform a circularity check and goes into an infinite loop that never terminates, which is clearly undesired behavior. Exposing this bug in *Q* requires generating a function that returns the same promise `p1` that is returned by the first call to `then` at line 223 (see C3). We are unaware of previous test generation techniques that are capable of generating such tests.

The example in Figure 5.3b illustrates some of the other complexities that arise in generating effective tests in the presence of higher-order functions. Here, a chain of promises is constructed using repeated calls to `then`. Note that, on line 227, the value `undefined` is passed to `then` instead of a function value. According to the specification, this should be equivalent to passing the identity function `function(v){ return v; }`, and executing the code using the native promises implementation prints "`7`". However, the polyfills provided by *Q* and *When*[6] print "`[Function]`" instead. Note that, to expose these errors, it was necessary to generate a test that contains a chain of function calls (see C4), and that it requires the test generator to generate calls to `then` with arguments that are both functions and non-function values.

As a final example, consider the test case in Figure 5.3c. In this example, the native implementation of promises executes without any errors and prints "`hello`". However, *bluebird* throws an uncaught exception

```
Unhandled rejection TypeError: Cannot read property 'apply' of null
```

without printing any output. Further investigation reveals that, in this case, the callback is not invoked by *bluebird* (see C5).

5.2 Framework for Testing Higher-Order Functions

This section presents our *LambdaTester* framework for testing higher-order functions. Given a set of methods under test and, optionally, some setup code required to test these methods, the framework generates tests that invoke the methods under test. The key novelty of *LambdaTester* is to effectively test methods that receive other functions, i.e., callbacks, as arguments. To support testing of such higher-order functions, the framework consists of two phases. The first phase, called *discovery phase*, infers for each method under test at what argument positions the

[4] `https://github.com/petkaantonov/bluebird`.
[5] See `https://github.com/kriskowal/q`.
[6] See `https://github.com/cujojs/when`.

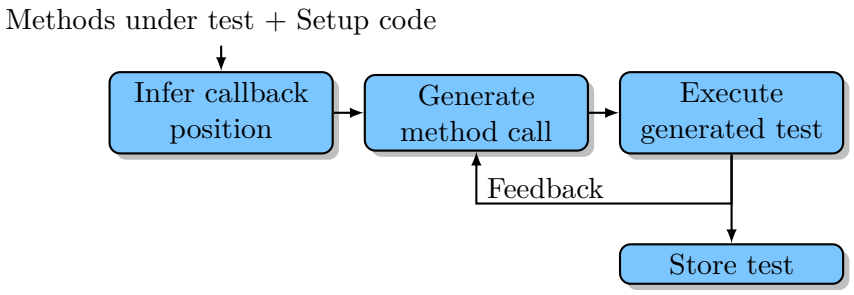


Figure 5.4: Overview of *LambdaTester*.

method expects a callback argument (Section 5.2.1). The second phase, called *test generation phase*, creates tests that pass callback functions and other argument values to the methods under test (Section 5.2.2). The test generation phase uses a form of feedback-directed, random testing [PE07] to incrementally extend and execute tests. We augment feedback-directed, random testing with four techniques to create callback arguments.

Before presenting the details of *LambdaTester*, we define our terminology. Each tests begins with a piece of user-provided setup code:

Definition 5 (Setup code)**.** Setup code *setup* is a sequence of pairs (*var*, *exp*), where *var* is a variable name and *exp* is the expression assigned to *var*.

The purpose of the setup code is to create a set of values to be used as receivers or arguments of method calls. For example, to test methods on promises, the user needs to provide setup code that creates some initial promise objects. For the test in Figure 5.3a, the first line shows the setup code of the test.

The basic ingredient of generated tests are method calls:

Definition 6 (Method call)**.** A method call c is a tuple (m, var_{rec}, $var_{arg1} \cdots var_{argn}$, var_{return}), where m is a method name, var_{rec} is the name of the variable used as the receiver object of the call, $var_{arg1}, ..., var_{argk}$ are the names of variables used as arguments and var_{return} is the name of the variable to which the call's return value is assigned.

Finally, the overall goal of the approach is to generate tests:

Definition 7 (Test)**.** A test *test* is a sequence ($setup, c_i, ..., c_n$) where *setup* is the setup code and $c_i, ..., c_n$ are generated method calls.

Figure 5.4 illustrates the process of test generation. For each method under test, the approach attempts to infer the positions of callback arguments. Afterwards, the approach repeatedly generates new method calls and executes the growing test. During each test execution, the approach collects feedback that guides the generation of the next method call. Finally, the approach stores the generated tests, which can then be used for bug finding (Section 5.3).

Algorithm 2 Algorithm to infer callback position

Input: Set M of names of methods under test, setup code *setup*
Output: Map C that maps each method name to a set of P_{cb} of callback positions

```
1: Initialize C[m] with an empty set for each m ∈ M
2: V ← V_setup ∪ V_rand
3: foreach each m ∈ M do
4:     while testing budget not exceeded do
5:         test ← new test starting with setup
6:         foreach each pos_cb < max_params do
7:             var_rec ← selectReceiver(V, m)
8:             args ← empty sequence
9:             foreach each pos < max_params do
10:                if pos = pos_cb then
11:                    var_cb ← callback function that logs calls to it
12:                    Append var_cb to args
13:                else
14:                    var_arg ← randomChoice(V)
15:                    Append var_arg to args
16:            Append (m, var_rec, args, _) to test
17:            feedback ← execute(test)
18:            if feedback has non-empty log then
19:                Add pos_cb to C[m]
20: return C
```

5.2.1 Discovery Phase: Inferring Callback Positions

In dynamic languages, the expected number and types of method parameters are generally unknown. In particular, our test generator cannot rely on static type signatures to decide where to pass a callback argument. To find out at which argument positions a method under test expects a callback, the *discovery phase* of our approach explores all possible callback positions. To this end, the approach creates tests that pass callbacks at each argument position, while leaving the number and types of the other arguments unconstrained. The approach then collects feedback from executing these tests to determine which callbacks are executed, allowing the approach to infer the argument positions where callbacks are expected.

Algorithm 2 illustrates our technique for finding callback positions for a given set M of methods and setup code *setup*. The output of the algorithm is a map C that maps each method name to a set of possible callback positions. As receiver objects and arguments of method calls the algorithm considers two sets of variables. First, we use a set V_{rand} of variables that store randomly generated values. To initialize this set, *LambdaTester* randomly generates values for primitive types, such as *strings*, *booleans*, and *numbers*, as well as common object types, such as *arrays* and *objects*. Moreover, we add *null* and *undefined* to the V_{rand} set. Second, we use the set V_{setup} of variables assigned to in the setup code. To obtain this set, the approach statically analyzes the setup code and extracts all declared variables. For example, after parsing the setup code in the first line of Figure 5.3a, V_{setup} contains the variable `p0`.

The main loop of the algorithm repeatedly invokes each method under test until exceeding the testing budget, e.g., a fixed number of method invocations. For each method call, the algorithm passes *max_params* arguments such that callback functions are passed at different argument positions. In our experiments, we set *max_params* to five because the higher-order functions we analyze do not expect more than five arguments. In many dynamic languages, such as JavaScript, if a method is called with more arguments than expected, the redundant arguments are simply ignored. For every argument position pos_{cb}, the algorithm creates a new method call that passes a callback function as the argument at position pos_{cb} and randomly selected values from V at all other argument positions.

When creating a call to a method m, the algorithm selects the receiver object from those elements in V that have a property named m. This selection, indicated by *selectReceiver* in the algorithm, is based on feedback from executing the setup code and the code that initializes the values in V_{rand}. During this initial execution, the approach gathers type information about all variables in V, including which properties the values stored in these variables provide.

For example, to generate a call to the `catch` method of promises based on the setup code in the first line of Figure 5.3a, the algorithm may select `p0` as the receiver variable because it provides a method named `catch`. Likewise, to generate a call to the `reduce` method, the algorithm may select a receiver from a randomly generated array in V_{rand} because arrays provide a `reduce` property.

After preparing all variables involved in a method call, the algorithm creates and then executes a test that contains the setup code followed by the call. During the execution of the test, the algorithm gathers feedback on its execution. Specifically, the algorithm keeps track of whether the callback function passed at position pos_{cb} is invoked. If the callback function is invoked, the algorithm infers that a function argument is expected at this position and updates the map C accordingly.

Finally, after calling the methods under test with various different arguments, the algorithm returns the inferred callback positions.

5.2.2 Test Generation Phase

After inferring at what argument positions the methods under test expect a callback argument, the second phase of *LambdaTester* creates tests that pass different kinds of callbacks to the methods under test. The approach combines feedback-directed, random test generation with different techniques for creating callback functions. In the following, we first present how *LambdaTester* creates callback functions (Section 5.2.2) and then describe the overall test generation algorithm (Section 5.2.2).

Generation of Callback Functions

Effectively testing higher-order functions requires callback functions to be passed as arguments to the methods under test. The *LambdaTester* framework currently supports four techniques for generating callback functions, which we present below.

```
function callback() {
};
```

(a) Empty callback ("Cb-Empty").

```
function callback() {
  return 17;
};
```

(b) Callback generated by QuickCheck ("Cb-QuickCheck").

```
function callback() {
  return Math.floor(10.8) +
         Math.floor(20.4) +
         Math.min(3, 5);
};
```

(c) Callback mined from existing code ("Cb-Mined").

```
function callback(a,b) {
  receiver.foo = "abc"
  b = null;
  return {x: 23};
};
```

(d) Callback generated based on dynamically analyzing the method under test ("Cb-Writes").

Figure 5.5: Examples of generated callbacks.

Empty Callbacks The most simple approach for creating callbacks is to simply create an empty function that does not perform any computation and that does not explicitly return any value. Figure 5.5a gives an example of an empty callback. We consider this approach as a baseline for comparison with the following approaches.

Callbacks by QuickCheck QuickCheck [CH11] is a state-of-the-art test generator originally designed for functional languages. To test higher-order functions, QuickCheck is capable of generating functions that return random values, but the functions that it generates do not perform additional computations and do not modify the program state. There are several re-implementations of QuickCheck for dynamic languages. We integrated an implementation for JavaScript[7] into *Lambda-Tester*. Integrating other existing testing tools that generate callbacks into our framework would be straightforward.

Figure 5.5b gives an example of a callback generated by QuickCheck.

Existing Callbacks Given the huge amount of existing code written in popular languages, another way to obtain callback functions is to extract them from already written code. To find existing callbacks for a method m, the approach statically analyzes method calls in a corpus of code and extracts function expressions passed to methods with a name equal to m. For example, to test the `map` function of arrays in JavaScript, we search for callback functions given to `map`. The rationale for extracting callbacks specifically for a each method m is that callbacks for a specific API method may follow common usage patterns, which may be valuable for testing these API methods.

To extract existing callbacks, we consider JavaScript code provided by a popular code corpus [RBVK16]. We analyze this code with an AST-based analysis that extracts all function expressions that are passed as an argument to a function. For

[7]The supported implementation of QuickCheck is available at `https://quickcheckjs.readme.io/`

each extracted function, the analysis stores the name of the called function along with the callback function. During test generation, *LambdaTester* then randomly selects from those extracted callbacks that match the current method under test. Figure 5.5c gives an example of an existing callback.

Callbacks Generation Based on Dynamic Analysis The final and most sophisticated technique to create callbacks uses a dynamic analysis of the method under test to guide the construction of a suitable callback function. The technique is based on the observation that callbacks are more likely to be effective for testing when they interact with the tested code. To illustrate this observation, consider the following method under test:

```
function testMe(callbackFn, bar) {
  // code before calling the callback

  // calling the callback
  var ret = callbackFn();

  // code after calling the callback
  if (this.foo) { ... }
  if (bar) { ... }
  if (ret) { ... }
}
```

To effectively test this method, the callback function should interact with the code executed after invoking the callback. Specifically, the callback function should modify the values stored in `this.foo`, `ret`, and `bar`. The challenge is how to determine the memory locations that the callback should modify.

We address this challenge through a dynamic analysis of memory locations that the method under test reads after invoking the callback. We apply the analysis when executing tests, and feed the resulting set of memory locations back to the test generator to direct the generation of future callbacks. The basic idea behind the dynamic analysis is to collect all memory locations that (i) are read after the first invocation of the callback function and that (ii) are reachable from the callback body. The reachable memory locations include memory reachable from the receiver object and the arguments of the call to the method under test, the return value of the callback, and any globally reachable state. To gather the relevant memory locations, the dynamic analysis performs the following actions during the execution of the method under test:

- *Store arguments and receiver object at method entry.* When the execution of the method under test starts, the analysis stores the method arguments and the receiver object.
- *Track calls to callback function.* The analysis observes when the callback function is invoked and then starts to track memory reads.
- *Track callback-reachable variable reads.* When the analysis observes a read to a variable, it checks whether the variable is transitively reachable from the

receiver object or the arguments of the call of the method under test, from the return value of the callback, or from the global object. If the value is reachable from one of these starting points, the analysis stores its access path to retrieve the value, i.e., a sequence of property accesses applied to the receiver or argument objects. For example, consider the reads of `ret` and `bar` in the above example. Because both happen after the callback invocation and are memory locations reachable from the callback body, the analysis reports the access paths `ret` and `arg2`, where `arg2` refers to the second argument passed to the method under test.

- *Track callback-reachable property reads.* Similar to variable reads, the analysis checks for each property read whether the read value is reachable from the callback body. For example, consider the read of `this.foo` in the above example. As `this` refers to the receiver object of the call to `testMe`, the value can be reached via the *foo* property of the receiver object, i.e., the stored access path is `receiver.foo`. The callback function could modify this value before the read by writing to `receiver.foo`, where `receiver` is the variable in the test that refers to the receiver object.

For the above example, the set of dynamically detected memory locations is: { `receiver.foo`, `arg2`, `ret` }.

Based on the dynamically detected memory locations, *LambdaTester* generates a callback body that interacts with the function under test. To this end, the approach infers how many arguments a callback function receives by first executing the method under test with a callback that inspects *arguments.length*. Then, *LambdaTester* generates callback functions that write to the locations read by the method under test and that are reachable from the callback body. The approach randomly selects a subset of the received arguments and of the detected memory locations, and assigns a random value to each element in the subset.

Figure 5.5d shows a callback function generated for the above example, based on the assumption that the callback function receives two arguments. As illustrated by the example, the feedback from the dynamic analysis allows *LambdaTester* to generate callbacks that interact with the tested code by writing to memory locations that are relevant for the method under test.

Feedback-Directed Test Generation

The generation of callback functions according to one of the four techniques presented in Section 5.2.2 are the core of *LambdaTester*. We now describe how the framework uses these callbacks and other values to generate tests in a feedback-directed, random manner. For methods under test that expect function arguments according to the discovery phase of *LambdaTester*, the approach generates sequences of method calls that probabilistically pass callback arguments. For methods that do not expect callbacks, the approach generates sequences of calls with an unconstrained list of arguments.

Algorithm 3 illustrates our test generation approach. For a given set M of methods, the approach generates tests that contain sequences of calls to methods in

Algorithm 3 Test generation algorithm

Input: Set M of names of methods under test, setup code *setup*, map C
Output: Generated tests T

```
1: T ← ∅
2: R ← ∅
3: V ← V_setup ∪ V_rand
4: while testing budget not exceeded do
5:     test ← New test starting with setup
6:     while max_calls not reached do
7:         m ← randomChoice(M)
8:         pos_cb ← randomChoice(C[m])
9:         var_rec ← selectReceiver(V, m)
10:        n ← randomChoiceInRange(pos_cb, max_args)
11:        args ← empty sequence
12:        foreach pos ≤ n do
13:            if pos == pos_cb and random() ≤ use_callback_prob then
14:                var_arg ← generateCallback(R, V)
15:                Append var_arg to args
16:            else
17:                var_arg ← randomChoice(V)
18:                Append var_arg to args
19:            Create fresh variable var_ret and add it to V
20:        Append (m, var_rec, args, var_ret) to test
21:        feedback ← execute(test)
22:        if feedback indicates a crash then
23:            Add test to T
24:            break
25:        Update R with feedback
26:    Add test to T
27: return T
```

M. The inputs to the algorithm are: (i) the set of methods M, (ii) user-provided setup code *setup*, and (iii) the map C, which maps each method name to a set of callback positions. The output of the algorithm is the set T of generated tests.

During test generation, the algorithm maintains two sets of values. First, it maintains the set V of variables, which – as for Algorithm 2 – comprises variables initialized in the setup code and in the generated tests, as well as randomly initialized variables. Second, the algorithm maintains a set R of memory locations, which are the output of the dynamic analysis of memory reads in the method under test. These locations serve as feedback that helps the test generator create effective callbacks (Section 5.2.2).

The algorithm incrementally generates method calls until a maximum number of calls *max_calls* per test is reached. To generate a method call, the algorithm randomly picks a name from M and a callback position *index* from C. The approach selects the receiver object from the variables V by randomly choosing only from elements in V that provide a method called m. For the callback argument, the

algorithm invokes *generateCallback*, which implements one of the four techniques discussed in Section 5.2.2. Values of other non-function arguments are selected from the pool V. The algorithm adds the variable var_{ret}, which stores the return value a newly added calls, to the set of variables V. That is, the test generator considers return values as a potential receiver objects or arguments in future calls.

After creating a method call, the algorithm adds the call to the current test, executes the test and collects feedback from the test's execution. The feedback consists of two kinds of information. First, the algorithm observes whether the generated test crashes by throwing an exception. In this case, further extending this test is not useful and the algorithm breaks out of the inner loop that appends further calls. Second, the algorithm receives feedback from the dynamic analysis of the memory locations read during the test execution and updates the set R by adding these locations to the set.

The main loop of the algorithm continues to create tests until the given testing budget has been exceeded.

5.3 Test Oracle: Differential Testing of Polyfills

The primary goal of test generation techniques is to detect bugs. To assess the effectiveness of our testing framework in finding bugs we generate tests for polyfills that accept callback arguments.

In JavaScript parlance, a *polyfill* is a user-defined implementation of an API that provides a method's functionality in older execution environments that do not natively support it. For example, before ECMAScript 6 added native support for promises, the `Promise` object had been available in JavaScript through third-party libraries such as *bluebird* and *Q*. However, polyfills are non-trivial to implement because approximating all possible behaviors supported by the native implementation is sometimes very challenging. To find bugs in polyfills we use differential testing [McK98] and consider the output of the native implementation as the ground truth.

When testing higher-order functions, it is often insufficient to determine whether the native implementation and the polyfill produce the same output state given the same input state. For example, two implementations can produce the same output for some inputs but invoke callback functions a different number of times, which clearly indicates that these implementations are not equivalent. In this section, we define test oracles relevant for testing higher-order functions, including novel callback-related oracles.

To compare test executions of native and polyfill implementations and find behavioral differences we define the notion of an execution summary.

Definition 8 (Execution summary)**.** An *execution summary* captures the result of a test execution. It contains the following information:

- The state of receiver objects and return values of calls.
- The state of arguments passed to callback functions.

- The number of invocations of callback functions.
- Output written to the standard output stream.
- Output written to the standard error stream.

To record the state of an object in the execution summary, *LambdaTester* serializes the object. *LambdaTester* also serializes the arguments passed to callback functions for every callback invocation. For primitive types, serialization is straightforward: we simply store the values. When serializing objects not created by a constructor, e.g., arrays, we rely on the *JSON* serialization API for converting an object to its string representation. In contrast, if an object is created by a constructor (e.g., promises) the internal object representation depends on the constructor implementation, and this representation may vary across polyfills. In this case, we do not serialize the object, but record only the constructor name. Hence, if two promise libraries have the same behavior but use different representations for promise objects, then we consider them equivalent.

To compare two implementations, *LambdaTester* also considers output written to the standard output and standard error streams. Standard output contains the output produced by the test's execution, and standard error contains error messages thrown during the test's execution. A challenge when comparing output written to these streams is that different implementations of a polyfill tend to produce different warning messages and error messages. For example, *bluebird* and the native implementation produce different error messages when an attempt is made to construct a circular promise chain, as we discussed in Section 5.1.2. To avoid false positives due to different warning messages, we consider two implementations as different only if one implementation produces an empty message while the other produces a non-empty message.

Based on the definition of execution summary, *LambdaTester* considers the following eight oracles:

- Standard error - two implementations write different output to the standard error stream. Here we distinguish two sub-categories:
  - Error messages - indicates a situation when one implementation reports an error message and the other does not.
  - Warnings - similar to the previous sub-category but this compares warning messages.
- Non-termination - a situation where one implementation terminates and the other does not.
- Standard output - two implementations produce different standard output.
- State of receiver objects - differences in the state of receiver objects.
- State of return values - differences in the state of return values.
- Callback arguments - differences in the state of callback arguments.

- Callback invocations - differences exposed by the number of callback invocations.

The last two oracles are callback-related oracles. Our experimental evaluation (Section 5.4) shows that many behavioral differences would be missed if callback-related oracles were not considered.

5.4 Evaluation

We evaluate *LambdaTester* on 43 higher-order functions taken from 13 popular libraries. This section reports on experiments that aim to answer the following research questions:

- RQ1: *How effective are the different variants of LambdaTester in finding behavioral differences? LambdaTester* finds behavior differences in 12 out of 13 libraries. When comparing the different techniques for creating callbacks, we find that our novel *Cb-Writes* approach is the most effective (Section 5.4.2).
- RQ2: *What kinds of behavioral differences are detected by LambdaTester? LambdaTester* detects a diverse set of differences, including clearly undesired behavior, such as non-termination bugs and crashes in polyfills, as well as differences in the number of times that a callback is invoked (Section 5.4.3).
- RQ3: *How effective are different variants of LambdaTester in increasing code coverage?* The *Cb-Writes* is the most effective approach for increasing the statement coverage of the array and promise polyfills.
- RQ4: *How efficient is LambdaTester?* The time required by *LambdaTester* to generate a single test ranges between 0.4 and 12 seconds, making it a practical tool for automatically testing higher-order functions.

5.4.1 Experimental Setup

Benchmarks We evaluate our approach on higher-order functions taken from 13 popular JavaScript libraries listed in Table 5.1. Polyfill.io, mozilla, and es5-shim implement polyfills for eight array methods indicated in the table. The other ten libraries implement JavaScript promises. For the promise libraries, we select the most popular[8] implementations of promises that aim to be compatible with the ECMAScript 6 standard. To test promise polyfills, we consider two methods that expect callbacks as arguments: `then` and `catch`.

Test Generation Approaches We compare the effectiveness of several variants of our testing approach as summarized in Table 5.2. The *Base* approach generates tests that call a single method with randomly selected arguments. It is unaware of callback arguments and never generates callbacks as arguments. The following

[8] According to the star rating on `github.com`.

Table 5.1: Benchmarks used for the evaluation.

Name	Version	LoC	API methods
Polyfill.io	3.25.1	189	filter, find, every, some, forEach, map, reduce, reduceRight
Mozilla polyfills	-	199	filter, find, every, some, forEach, map, reduce, reduceRight
es5-shim	4.5.10	2098	filter, find, every, some, forEach, map, reduce, reduceRight
Q	1.5.1	1235	then, catch
bluebird	3.5.1	5188	then, catch
when	3.7.8	1844	then, catch
then/promise	8.0.1	567	then, catch
rsvp.js	4.8.2	963	then, catch
native-promise-only	0.8.1	292	then, catch
lie	3.3.0	309	then, catch
pacta	0.9.0	403	then, catch
es6-promises	1.0.10	274	then, catch
bloodhound-promises	1.4.14	652	then, catch

Table 5.2: Test generation approaches used for the evaluation.

Approach	Description	Feedback-directed	Infer callback positions	Callback functions
Base	Random test generator that is unaware of callbacks. It never passes any callback function as an argument.	No	No	—
Cb-Empty	A callback-aware test generator that infers arguments that expect callbacks, and that passes empty callback functions as arguments.	Yes	Yes	Empty functions
Cb-Quick	Like Cb-Empty but with callback functions generated by QuickCheck.	Yes	Yes	QuickCheck-generated functions
Cb-Mined	Like Cb-Empty but with callback functions mined from existing code.	Yes	Yes	Mined functions
Cb-Writes	Like Cb-Empty but with callback functions generated based on a dynamic analysis of memory reads.	Yes	Yes	Functions with targeted writes

four approaches correspond to the callback generation techniques introduced in Section 5.2.2. The *Cb-Empty* approach infers which arguments expect callbacks, generates tests as sequences of function calls and passes empty callbacks to functions that expect them. *Cb-Mined* is like *Cb-Empty*, but with callback functions mined from existing code. *Cb-Quick* is also like *Cb-Empty*, but with callback functions generated by QuickCheck. Finally, *Cb-Writes* uses a dynamic analysis to determine relevant memory locations that are read by the method under test, and attempts to generate callbacks that write to those locations.

5.4.2 Effectiveness in Finding Behavioral Differences

Table 5.3 compares the different test generation approaches in terms of the number of behavioral differences exposed by 1,000 generated tests. The table includes the differences detected with test oracles listed in Section 5.3, except for differences in the warning messages reported by libraries (as discussed in more detail in Section 5.4.3).

In total, we find behavioral differences in 12 out of 13 libraries. The *Base* approach does not find any difference in any polyfill. *Cb-Empty* and *Cb-Quick* find differences in 7 libraries, *Cb-Mined* in 11 libraries, and *Cb-Writes* in 12 libraries. Overall, the *Cb-Writes* approach outperforms all other approaches. The average number of differences found by *Cb-Writes* is the largest across all libraries. The largest number of differences per single library is found in *Bluebird*, which is perhaps surprising as it is the most popular promise implementation[9]. Furthermore, we consider differences in warning messages as benign differences and exclude them as errors in Table 5.3.

5.4.3 Classification of Behavioral Differences

We are aware that some of the behavioral differences we find are likely to be due to the same root cause. To better understand the types of behavioral differences, we classify each of them into one or more categories based on the oracles defined in Section 5.3. In the following, we show a breakdown of differences based on the way they manifest.

Tables 5.4 and 5.5 show how many behavioral differences are found in each category for each feedback-directed testing approach. Based on these results we can draw the following conclusions:

- Warnings, error messages, and differences in execution summaries are the dominant kinds of behavioral differences. Because the exact warning messages are not specified, these differences can likely be considered as false positives.

- For the promise libraries, several differences arise from different states of callback arguments, showing that considering callback-related oracles helps identify more behavioral differences. Whether a callback is called or not is an important behavioral property and polyfills should agree with the native implementation.

[9]According to the star rating on `github.com`.

Table 5.3: Comparison of different test generation approaches in terms of the number of behavioral differences exposed by 1,000 generated tests. We show results only for those *mozilla.js* and *polyfill.io* polyfills where at least one of the testing approaches finds a behavioral difference.

Benchmark	Base	Cb-Empty	Cb-Quick	Cb-Mined	Cb-Writes
Polyfill.io (map)	0	0	0	0	28
Polyfill.io (find)	0	0	0	0	19
Mozilla (filter)	0	0	0	7	36
es5-shim	0	0	0	0	0
Bluebird	0	475	423	264	409
Q	0	3	1	60	151
when	0	374	331	49	309
then/promise	0	0	0	57	122
rsvp.js	0	21	15	23	164
native-promise-only	0	41	30	100	184
lie	0	28	24	82	135
pacta	0	0	0	57	120
es6-promises	0	0	0	57	149
bloodhound-promises	0	63	51	153	183

- Many promise libraries show equivalent behavior regarding their standard output and standard error. Because we test all libraries with the same generated tests, this is the reason for several identical numbers in the "Err" and "Warn" columns in Tables 5.4 and 5.5.

- Non-termination errors in promise libraries are detected by *Cb-Writes* only. The reason is that *Cb-Writes* uses objects created by previous method calls as possible return values for callbacks. This causes it to attempt to create circular promise chains, thus triggering non-termination errors. Since non-termination certainly is an undesirable property, the polyfills should not diverge from the native implementations w.r.t. this behavioral property.

- The *Cb-Writes* approach detects significantly more differences in array polyfills than other testing approaches. This result shows that using a more sophisticated approach for creating callbacks that interact with the method under test via shared state is worth the effort.

5.4.4 Array Polyfills Generated by Mimic

As another benchmark, in addition to the human-written libraries considered so far, we also apply *LambdaTester* to array polyfills generated by Mimic [HSC15]. Mimic is a tool for synthesizing models for a variety of array-manipulating functions. The current implementation provides models for JavaScript's built-in array methods, including higher-order functions such as `filter`, `find`, `every`, `some`, `forEach`, `map`, `reduce` and `reduceRight`.

Table 5.4: Behavioral differences found by *Cb-Empty* and *Cb-Quick* approaches in 1,000 generated tests per benchmark. Err = Error messages, Warn = Warning messages, NT = Non-termination errors, Stout = Standard output, Rec = Receiver objects, Ret = Return values, C.arg = Callback arguments, C.inv = Callback invocations.

Benchmark	Cb-Empty								Cb-Quick							
	Err	Warn	NT	Stout	Rec	Ret	C.arg	C.inv	Err	Warn	NT	Stout	Rec	Ret	C.arg	C.inv
Polyfill.io	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Mozilla	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
es5-shim	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Bluebird	381	0	0	0	0	0	296	255	317	0	0	0	0	0	269	227
Q	0	394	0	0	0	0	3	0	0	377	0	0	0	0	1	0
when	297	69	0	0	0	0	177	141	245	67	0	0	0	0	171	137
then/promise	0	394	0	0	0	0	0	0	0	377	0	0	0	0	0	0
rsvp.js	0	394	0	0	0	0	21	0	0	377	0	0	0	0	15	0
native-promise-only	0	394	0	0	0	0	41	0	0	377	0	0	0	0	30	0
lie	0	394	0	0	0	0	28	0	0	377	0	0	0	0	24	0
pacta	0	394	0	0	0	0	0	0	0	377	0	0	0	0	0	0
es6-promises	0	394	0	0	0	0	0	0	0	377	0	0	0	0	0	0
bloodhound-promises	0	394	0	0	0	0	63	0	0	377	0	0	0	0	51	0

We evaluate the effectiveness of *LambdaTester* on all Mimic-synthesized array polyfills. Table 5.6 shows the number of behavioral differences found by each testing approach. Interestingly, all tests generated by *Base* approach show errors in mimic polyfills. This is because Mimic's polyfill implementations do not throw errors when a non-function argument is passed at positions where a callback is expected. Furthermore, all tests generated by *Cb-Empty* expose errors in Mimic's `every` and `some` polyfills. The reason is that these polyfills are supposed to return a boolean value, but when receiving an empty callback, they always return `undefined`.

In general, the polyfills generated by Mimic have significantly more differences from the native implementation than the human-written polyfill libraries. Since we are not aware of any use of synthesized mimic models in real-world applications, we exclude these polyfills as a point of comparison in Table 5.3.

Future work might combine a test generator for higher-order functions, such as *LambdaTester*, with an approach for synthesizing polyfills, such as Mimic, so that the behavioral differences found with generated tests provide feedback on weaknesses of the synthesized code.

Table 5.5: Behavioral differences found by *Cb-Mined* and *Cb-Writes* approaches in 1,000 generated tests per benchmark. Err = Error messages, Warn = Warning messages, NT = Non-termination errors, Stout = Standard output, Rec = Receiver objects, Ret = Return values, C.arg = Callback arguments, C.inv = Callback invocations.

Benchmark	Cb-Mined								Cb-Writes							
	Err	Warn	NT	Stout	Rec	Ret	C.arg	C.inv	Err	Warn	NT	Stout	Rec	Ret	C.arg	C.inv
Polyfill.io (map)	0	0	0	0	0	0	0	0	0	0	0	0	6	27	7	0
Polyfill.io (find)	0	0	0	0	0	0	0	0	0	0	0	0	0	9	19	19
Mozilla (filter)	0	0	0	0	5	7	5	0	0	0	0	0	17	35	16	0
es5-shim	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Bluebird	7	0	0	1	1	1	262	252	211	0	0	0	0	0	316	276
Q	57	870	0	0	0	0	3	0	149	504	68	0	0	0	2	0
when	5	2	0	1	1	1	45	17	207	123	0	0	0	0	165	112
then/promise	57	870	0	0	0	0	0	0	149	504	0	0	0	0	1	0
rsvp.js	57	870	0	0	0	0	23	0	149	504	0	0	0	0	21	0
native-promise-only	57	870	0	0	0	0	47	0	149	504	0	0	0	0	48	0
lie	57	870	0	0	0	0	27	0	149	504	0	0	0	0	22	0
pacta	57	870	0	0	0	0	0	0	149	504	0	0	0	0	0	0
es6-promises	57	870	0	0	0	0	0	0	149	504	0	0	0	0	0	0
bloodhound-promises	57	870	0	0	0	0	106	1	149	504	0	0	0	0	51	0

Table 5.6: Behavioral differences in array polyfills generated by Mimic [HSC15].

Polyfill	Base	Cb-Empty	Cb-Quick	Cb-Mined	Cb-Writes
Every	1,000	1,000	999	437	992
Some	1,000	1,000	993	195	974
ForEach	1,000	0	0	0	0
Filter	1,000	0	0	0	7
Map	1,000	0	0	0	28
Reduce	1,000	797	794	758	526
ReduceRight	1,000	797	794	783	809

5.4.5 Examples of Bugs and Other Inconsistencies

To illustrate the behavioral differences detected by *LambdaTester*, we discuss a few representative examples. For space reasons, we only include the relevant fragments of the generated tests that expose errors.

```
265 var p1 = Promise.resolve(18);
266 var p2 = Promise.reject(17);
267 p2.catch(function(){
268   return p1;
269 }, p1);
```

(a) *LambdaTester*-generated test.

```
270 Promise.prototype['catch'] = function(onRejected) {
271   if (arguments.length < 2) {
272     return origCatch.call(this, onRejected);
273   }
274   if (typeof onRejected !== 'function') {
275     return this.ensure(rejectInvalidPredicate);
276   }
277   return origCatch.call(this,
278   createCatchFilter(arguments[1],onRejected));
279 }
280
281 function createCatchFilter(handler, predicate) {
282   return function(e) {
283     return evaluatePredicate(e, predicate) ?
284         handler.call(this, e) : reject(e);
285   }
286 }
```

(b) Implementation of `catch` from *when* library.

Figure 5.6: Example of behavioral difference found in the *when* library.

Unexpected Types and Number of Arguments Figure 5.6a illustrates a test case that exposes a behavioral difference found in the *when* library caused by passing a non-function value as the second argument to the `catch` method. The buggy polyfill implementation is given in Figure 5.6b.

The library throws a `TypeError` because it tries to execute the `call` method on a non-function object at line 284. In contrast, the native implementation ignores the second argument and executes the method call without errors. This example illustrates a situation where a different output is written to the standard error stream.

Order of Executed Function Calls The example in Figure 5.7 illustrates a behavioral difference found in the *Q* library. The native implementation first executes calls to the `then` function at lines 289, 290, and 292 and then a call to `catch` at line 291. However, the *Q* library executes the method calls in the same order as presented in Figure 5.7. *LambdaTester* discovers this difference by inspecting the state of the callback arguments.

The cause of this bug appears to be in the library's queuing mechanism used for tracking unhandled rejections. Due to the complexity of the code, we were not able to fully diagnose the problem. In general, the complexity of callback-based code is a good reason for extensive testing, e.g., using our *LambdaTester* approach.

```
287 var p1 = Promise.resolve(18);
288 var p2 = Promise.reject(17);
289 var r1 = p1.then(function(){ return null; }, null);
290 var r2 = p2.then(function(){ return r1; });
291 var r3 = r2.catch(function(){ return p2; });
292 var r4 = r1.then(function(){ return p2; });
```

Figure 5.7: Example of *LambdaTester*-generated test that exposes a behavioral difference in the *Q* library.

Changing Receiver Object Inside a Callback Many of the behavioral differences found in the array polyfills are caused by changes made by the callback to properties of the receiver object or of other arguments. Figure 5.8a illustrates a test case that expose this type of problem, with the corresponding polyfill implementation in Figure 5.8b. In the test, when the method under test invokes the callback for the first time, the callback sets the `length` property of the receiver object to `false`. At line 307 in Figure 5.8b, the method under test performs a check to find whether the receiver object has a property named `index`. After changing the `length` property, the check always evaluates to `false`, which prevents further executions of the callback argument. As a result, the polyfill and the native implementation return the same value, `undefined`, but the native implementation executes the callback three times, whereas the polyfill executes it only once. This example illustrates a situation where a callback-related oracle helps detect behavioral differences that would be missed otherwise.

Despite the effectiveness of *LambdaTester* in detecting inconsistencies, we are aware that the developers of the tested libraries may find some of the generated tests more useful than others. The reason is that not every generated test represents a realistic usage scenario of the tested libraries. For example, method invocations with callbacks that change the state of the receiver or of the argument objects are unlikely to be a common use case. However, testing uncommon behavior helps with finding more bugs that would be missed otherwise.

5.4.6 Effectiveness in Covering Code Under Test

To assess the effectiveness of *LambdaTester* in covering code under test we measure statement coverage. For the array polyfills, we collect coverage data for each method implementation. However, since it is not straightforward to extract individual method implementations from the promise libraries, we chose to measure coverage of the entire promise libraries.

Table 5.7 lists the results for coverage measurements for each benchmark. The statement coverage of polyfills differs significantly between *Base* and the feedback-directed approaches, e.g., increasing from 20% to up to 100% for the `filter` method from the Mozilla library. Overall, *Cb-Writes* is the most effective approach in increasing statement coverage. For all benchmarks, feedback-directed approaches achieve better statement coverage compared to the baseline approach.

```
var base = ["w", "I", 126];
base.find(function(a,b,c){
  base['length'] = false;
  return a;
});
```

(a) *LambdaTester*-generated test

```
function find(callback) {
  ...
  var object = Object(this),
  scope = arguments[1],
  arraylike = object instanceof String ?
      object.split('') : object,
  index = -1;

  while (++index < length) {
    if (index in arraylike) {
      element = arraylike[index];
      if (callback.call(scope, element, index, object) {
        return element;
      }
    }
  }
}
```

(b) Implementation of `Array.prototype.find` from `polyfill.io`.

Figure 5.8: Example of a behavioral difference found in the *polyfill.io* library.

The statement coverage for promise libraries is relatively low, and the reason is that the tested methods comprise only a subset of the entire library code: In addition to the `then` and `catch` methods, the promise libraries define other functions not targeted by our generated tests.

5.4.7 Efficiency

To assess the performance of the test generation techniques, Table 5.8 shows, for each approach, the time needed to generate and execute 1,000 tests. All experiments are conducted on a 48-core machine with a 2.2GHz Intel Xeon CPU and 64GB of RAM. We use Node.js 8.5 and provide it with the default of 1GB of memory. The implementation of *LambdaTester* is single-threaded and while running the tool we effectively use a single core.

The execution time of the *Base* approach is dominated by the time needed to generate tests and collect their execution summaries. The execution time of the feedback-directed approaches is higher as they also include the time to generate multiple calls and to collect feedback. In particular, for the *Cb-Writes* approach, the time needed for dynamically analyzing each test's execution dominates the total execution time. The time spent to generate tests with *Cb-Writes* takes less than 1 hour on average, except for the generation of the promise tests, which

Table 5.7: Statement coverage for 1,000 generated tests.

Benchmark	Base	Cb-Empty	Cb-Quick	Cb-Mined	Cb-Writes
Polyfill.io (every)	40.0%	70.0%	80.0%	80.0%	90.0%
Polyfill.io (some)	40.0%	70.0%	80.0%	80.0%	90.0%
Polyfill.io (forEach)	44.4%	77.7%	77.7%	77.7%	88.8%
Polyfill.io (filter)	36.3%	72.7%	81.8%	81.8%	90.9%
Polyfill.io (map)	40.0%	80.0%	80.0%	80.0%	90.0%
Polyfill.io (find)	36.3%	72.7%	81.8%	81.8%	90.9%
Polyfill.io (reduce)	25.0%	81.2%	87.5%	81.2%	87.5%
Polyfill.io (reduceRight)	25.0%	81.2%	87.5%	81.2%	87.5%
Mozilla (every)	35.0%	80.0%	90.0%	90.0%	95.0%
Mozilla (some)	33.3%	75.0%	83.3%	83.3%	91.6%
Mozilla (forEach)	41.1%	88.2%	88.2%	88.2%	94.1%
Mozilla(filter)	20.0%	80.0%	93.3%	93.3%	100%
Mozilla (map)	35.0%	90.0%	90.0%	90.0%	95.0%
Mozilla (find)	40.0%	80.0%	86.6%	86.6%	93.3%
Mozilla (reduce)	19.0%	80.9%	85.7%	80.9%	85.7%
Mozilla (reduceRight)	23.5%	76.4%	82.3%	76.4%	82.3%
es5-shim (every)	56.5%	58.9%	59.6%	59.6%	63.5%
es5-shim (some)	56.5%	58.9%	59.6%	59.6%	63.5%
es5-shim (forEach)	56.4%	60.3%	60.3%	60.3%	64.1%
es5-shim (filter)	56.3%	60.1%	60.9%	60.9%	64.6%
es5-shim (map)	56.0%	60.6%	60.6%	60.6%	64.3%
es5-shim (reduce)	48.8%	58.6%	59.4%	58.6%	62.4%
es5-shim (reduceRight)	48.1%	59.2%	60.0%	59.2%	62.9%
Q	42.2%	43.8%	43.8%	44.2%	43.8%
bluebird	37.2%	39.6%	39.9%	40.0%	41.0%
when	51.5%	52.5%	52.8%	52.8%	53.2%
then/promise	48.9%	59.0%	62.1%	63.6%	64.6%
rsvp.js	41.9%	45.3%	46.5%	47.2%	47.9%
native-promise-only	65.1%	67.4%	68.0%	68.6%	69.1%
lie	41.2%	55.5%	55.5%	57.1%	62.3%
pacta	39.3%	54.3%	55.9%	56.6%	59.0%
es6-promises	58.6%	67.2%	68.8%	68.8%	68.8%
bloodhound-promises	29.9%	33.0%	33.3%	35.3%	36.4%

takes approximately 3 hours. Overall, since running *LambdaTester* requires minimal manual intervention and since the generated tests expose many behavioral differences, we consider the computational effort to be acceptable.

5.5 Summary

This section presented a framework for testing higher-order functions in dynamic programming languages. The approach consists of two phases: the *discovery phase* is concerned with discovering at which argument positions a function is expected,

Table 5.8: Time to generate 1,000 tests per API.

API	Base	Cb-Empty	Cb-Quick	Cb-Mined	Cb-Writes
every	6m 50s	27m 6s	27m 5s	33m 35s	53m 35s
forEach	6m 48s	27m 6s	27m 5s	33m 37s	53m 46s
some	6m 48s	27m 6s	27m 6s	33m 37s	53m 26s
filter	6m 47s	27m 6s	27m 6s	33m 34s	53m 7s
map	6m 47s	27m 6s	27m 6s	32m 30s	53m 34s
reduce	6m 46s	27m 5s	27m 7s	33m 1s	54m
reduceRight	6m 46s	27m 5s	27m 7s	33m 39s	53m 54s
find	6m 46s	27m 6s	27m 7s	33m 33s	53m 34s
then,catch	6m 31s	27m 10s	27m 6s	33m 23s	190m

and the *test generation phase* automatically creates tests that perform a sequence of method calls.

We have implemented the framework in a tool called *LambdaTester*, and evaluated several instances of the framework in which the generated callback functions consist of: (i) empty functions, (ii) functions that return random values [CH11], (iii) callbacks mined from a corpus of existing code, and (iv) functions that write to locations that are likely to be read, as determined using a feedback-directed dynamic analysis technique.

We apply *LambdaTester* to polyfills for array-related and promise functions taken from popular libraries. In our experimental evaluation we show that *LambdaTester* reveals various behavioral differences between polyfills and their corresponding native implementations, including previously unknown bugs in popular libraries. Overall, the approach detects differences in 12 of 13 libraries. We conclude that generating callbacks that modify program state in non-obvious ways is more effective in triggering non-trivial executions that expose behavioral differences than previous test generation techniques.

Related Work

6.1 Studies of Performance Issues

There has been extensive work on studying performance issues in real-world software. Jin et al. [JSS+12] study performance bugs in programs written in C and C++. They show that performance bugs occur frequently and that many of them can be addressed by relatively simple efficiency rules. Furthermore, Zaman et al. [ZAH12] suggest that performance bugs are more difficult to fix than correctness bugs and they account for a non-negligible amount of developer time. Liu et al. [LXC14] report on performance bugs in smartphone applications and propose specialized program analyses that detect common patterns of bugs identified in the study. By detecting 126 new instances of performance bug patterns, they support our observation that many performance issues are instances of recurring patterns. A study by Linares-Vasquez et al. [LBBC+14] illustrates how API usages on Android influence energy consumption, which is closely related to performance.

In Chapter 2, we presented the first empirical study on performance issues and optimizations in JavaScript, which differs from C, C++, and Java both on the language and the language implementation level. Furthermore, our work differs from the existing studies by investigating 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.

6.2 Approaches to Detect Performance Bottlenecks

Previous research has addressed various types of performance issues in real-world software. We distinguish between approaches that improve the execution time of a program and approaches that address memory-related issues.

Execution Time Table 6.1 gives an overview of relevant approaches proposed to improve the execution time of a program. The most common way to identify performance bottlenecks is CPU-time profiling [GKM82, CSL04]. CPU profiles usually report the code locations that are responsible for excessive resource computation for a particular run. Alternative approaches focus on *algorithmic profiling* [GAW07, ZH12], which empirically estimates the computational complexity that holds for multiple

Table 6.1: Approaches to detect performance issues.

Approach	Problem	Analysis
CPU-time profiling [GKM82, CSL04]	Excessive resource computation	Dynamic
Algorithmic profiling [GAW07, ZH12]	Computational complexity	Dynamic
Toddler [NSML13]	Repetitive loop computation	Dynamic
Maplesden et al. [MTHG15]	Repetitive patterns of method calls	Dynamic
Clarity [ODL15]	Redundant traversal bugs	Static
Caramel [NR14]	Wasted loop computation	Static
MemoizeIt [TPG15]	Redundant computation	Dynamic
JITProf[GPS15]	JIT-unfriendly code	Dynamic
JSweeter [XHZZ15]	JIT-unfriendly code	Dynamic
Optimization coaching [SAG15]	Missed compiler optimization	Dynamic
SyncProf [YP16]	Synchronization bottlenecks	Dynamic
DecisionProf [SGP17]	**Inefficient orders of evaluations**	**Dynamic**
Cross-language optimizations [SBMM18]	**Cross-runtime interactions**	**Static**

inputs. However, while traditional profiling techniques are effective when reporting how resources are spent, they provide very limited help in finding the causes of performance issues.

To overcome the limitations of traditional profilers, automated techniques have been proposed to help developers identify causes of performance issues. Toddler [NSML13] reports performance bugs caused by loops whose computation has repetitive and partially similar memory-access patterns across loop iterations. Work by Maplesden et al. [MTHG15] helps to better understand the performance characteristics of large-scale software by detecting repeated patterns of method calls. Such patterns are high potential candidates for optimizations that would lead to performance improvements. Clarity [ODL15] is a static analysis to detect a prevalent class of asymptotic performance bugs called *redundant traversal bugs*. A redundant traversal bug arises if a program repeatedly iterates over a data structure that has not been modified between successive traversals of the data structure. The optimization involves memoization and reuse of such computation.

Several approaches that identify performance bottlenecks also give developers a potential source-level fix for detected performance bugs. Caramel [NR14] is a static analysis that suggests code transformation to avoid wasting loop iterations. The key idea is to identify a condition under which the remainder of a loop can be skipped without changing the program outcome. MemoizeIt [TPG15] is a dynamic analysis to detect memoization opportunities in methods that repeatedly perform the same computation. To find optimization opportunities, the approach compares the input and output of method calls by repeatedly executing the program to increase the degree of detail collected by the dynamic analysis. For every memoization opportunity that MemoizeIt detects, it provides hints on how to implement memoization. JITProf [GPS15] is a profiling approach that identifies code locations that prohibit profitable JIT optimizations in JavaScript code. The approach simulates the execution of a JIT compiler by associating meta-information with JavaScript objects and code locations that are updated whenever a particular runtime event occurs. This information is then used to identify JIT-unfriendly code. JSweeter [XHZZ15] is an

approach that also finds JIT-unfriendly code locations but focuses on performance code smells related to type mutations. They use information on type update and deoptimzations to infer the reasons and number of deoptimizations, eventually reporting type-unstable code locations. JITProf and JSweeter both provide refactoring hints on how to optimize code, but they consider different JIT-unfriendly patterns. Another approach to help developers find JIT-unfriendly code focuses on optimizations that could almost be applied but that the compiler cannot apply due to lack of information or potential unsoundness: optimization coaching by [SAG15] searches for such missed optimizations and recommends program changes to trigger additional optimizations. Lastly, SyncProf [YP16] is a concurrency-based profiling approach that helps in detecting, localizing and optimizing synchronization bottlenecks. It finds performance bugs due to unnecessary or inefficient synchronization in concurrent programs. After detecing the root cause of a bottlenects, SyncProf suggests optimization strategies to the developer.

Our work differs from existing approaches by considering different types of performance issues: **inefficient orders of evaluation** and **cross-runtime interactions** in big data systems. *DecisionProf* automatically assesses whether an optimization opportunity improves performance and suggests concrete optimizations to the developer. Furthermore, our static analysis to find cross-language optimization opportunities suggests an optimization only if its application generates more native code from a non-relational part.

Other relevant approaches for diagnosing performance bottlenecks include a profiler for UI-related performance problems [JAH11], an analysis to diagnose idle times [AAFM10], mining of stacktraces [HDG+12] and execution traces [YHZX14], inefficient use of collections [GPC14], a systematic search for performance anti-patterns [WHH13], and an analysis of performance regressions [PHG14]. However, none of these approaches addresses the problem of inefficient orders of evaluations nor the problem of cross-runtime interactions in big data systems.

Memory-Related Issues A number of approaches have been proposed to identify various symptoms of memory-related issues. MemInsight [JSSC15] is a browser-independent memory debugging tool for web applications that computes object lifetimes. The approach generates a trace of memory operations during an execution, capturing the uses of each object, variable declarations, and return calls. The trace is then used to to find memory leaks, drags, churns, and opportunities for stack allocation and object inlining. To detect unnecessary use of duplicate objects, Marinov et al. [MO03] propose dynamic analysis that records all the objects created during a particular program run. The analysis partitions the objects into equivalence classes, and uses collected timing information to determine when elements of an equivalence class could have been safely collapsed into a single representative object without affecting the behavior of that program.

Several approaches address the problem of *run-time bloat* or excessive memory usage. Xu et al. [XAM+09] introduce *copy profiling*, a technique that summarizes runtime activity in terms of chains of data copies. Based on profiled information, the approach builds a *copy graph*, an abstraction of chains of copies, which is used as an input to analyses that detect various patterns of run-time bloat. Another

approach by Xu et al. [XR10] addresses the problem of inefficiently-used containers in Java programs. The approach checks, for each container, whether it has enough data added and whether it is looked up sufficient number of times. The problems are further detected if a container is underutilized: it holds a very small number of elements during its lifetime or overpopulated: it holds many objects but is looked up only a few times. Yan et al. [YXR12] introduce *reference propagation profiling*, a dynamic analysis which tracks the propagation of object references, and produces a reference propagation graph. Based on reference propagation graph, several analyses are developed to detect problems such as never-used or excessive object allocations. Finally, JSWhiz [PH13] is a static analysis for finding memory leaks in JavaScript programs. The analysis identifies five common patterns that introduce memory leaks in JavaScript applications reliably, without any false positive.

6.3 Efficiency of JavaScript Engines

Because JavaScript was initially implemented through interpretation, it has long been perceived as a "slow" language. The increasing complexity of applications created a need to execute JavaScript code more efficiently. Due to recent advances in just-in-time (JIT) compilers, the performance of JavaScript has since been significantly improved. This section focuses on the most relevant compiler approaches to improve and assess the performance of JavaScript code.

6.3.1 JIT Compilation

Just-in-time (JIT) compilation has a long history of improving the execution time of a program by compiling it to efficient machine code at runtime [Ayc03], and most modern JavaScript engines use JIT compilation. For example, according to St-Amour et al. [SAG15], the SpiderMonkey JavaScript engine first interprets code without any compilation or optimization. Upon reaching a specific number of executions of a function, the baseline JIT compiler translates the function to native code. V8 engine skips the interpretation phase and instead compiles JavaScript code directly to native code. Both engines further optimize hot functions at runtime. Proposed optimizaiton techniques used in JavaScript JIT compilers include type specialization [GES+09, KRH15, HG12], specializing functions based on previously observed parameters [CASP13] and an improved object representation [ACS+14].

Type Specialization Due to the dynamically typed nature of JavaScript, JIT compilers do not have access to static type information. This lack of information makes the generation of efficient, type-specialized machine code difficult.

TraceMonkey [GES+09] was one of the first JIT compilers for JavaScript. Based on the observation that programs spend most of the time in hot loops and that most loops are type-stable, the compiler specializes the code for frequently executed loops at runtime. At the core of TraceMonkey is a dynamic analysis that gathers sequences of statements, called traces, along with their type information. The analysis represents frequently executed traces in a tree structure that encodes the

type conditions under which the code can be specialized. Despite the initial success of trace-based JIT compilation, the approach has since been mostly abandoned, e.g., in favor of hybrid (static and dynamic) type inference [HG12]. In this hybrid approach, a static analysis computes for each expression or heap value, a possibly incomplete set of types it may have at runtime. At runtime, the JIT engine checks for unexpected types and other special cases, such as arrays containing undefined values and integer overflows.

However, type specialization in JIT compilers is highly speculative, and when unexpected types are encountered, the compiler deoptimizes type-specialized code. To reduce the number of deoptimizations, Kedlaya et al. [KRH15] propose ahead-of-time profiling on the server side. Their profiler tracks when a function becomes hot, which types and shapes a function uses, and when it gets deoptimized. Based on information about type-unstable functions, the client-side engine prevents optimizing code that will likely be deoptimized later.

Function Specialization In contrast to the above approaches, which exploits dynamically observed types, Costa et al. [CASP13] propose to specialize functions based on dynamically observed values. Their approach is based on the empirical observation that 60% of all JavaScript functions are called only once or always with the same set of parameters. Based on this observation, they propose a JIT optimization that replaces the arguments passed to a function by previously observed runtime values.

Object Representation Ahn et al. [ACS+14] identify frequent changes to prototypes and method bindings as the main source of performance issues for website code. These issues make types very unpredictable, which hinders type specialization by the compiler. They address the problem by proposing three enhancements to the V8 compiler that effectively decouple prototypes and method bindings from the type definition.

As already discussed in Chapter 2, 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 optimizations. Furthermore, in Chapter 3, we proposed an analysis to find optimization opportunities in JavaScript code not addressed by today's JIT compilers.

6.3.2 Performance Benchmarks

Due to the complexity of JavaScript language, production engines have complex and different implementations and optimization strategies. This may cause different performance outputs when running the same code on multiple engines. To demonstrate and compare the performance of engines, vendors use performance benchmarks. The most commonly used JavaScript benchmark suites are SunSpider[1],

[1] https://webkit.org/perf/sunspider/sunspider.html

Octane[2], and Kraken[3]. Unfortunately, many of these benchmarks turn out to not be representative of real-world code, as shown by Ratanaworabhan et al. [RLZ10]. Therefore, focusing on benchmark behavior may result in overfitting and missing optimization opportunities that are present in real applications.

Motivated by the lack of representativeness of existing benchmarks, Richards et al. [RGEV11] propose JSBench to automate the creation of realistic and representative JavaScript benchmarks from existing web applications. JSBench instruments the original web code to generate a trace of JavaScript operations. The trace is used to generate a replayable JavaScript program which is then recombined with HTML from the original web application. [RGEV11] show that JSBench-generated benchmarks match the behavior of real web applications by using several metrics, such as memory usage, GC time, and event loop behavior, collected on several instrumented browsers.

6.4 Test Generation

To complement existing test generators, Chapter 5 presents a novel testing framework for higher-order functions. In Table 6.2, we outline relevant approaches for automated test generation and their underlying techniques.

Random testing has been shown to be very cost-effective [DN84, Nta01] and to detect a predictable number of bugs despite its random nature [CPO+11]. Furthermore, past experience [GHJ07] indicates that during the early stages of development, randomized testing with a high degree of automation offers much quicker paths to finding many bugs compared to more rigorous bug detection techniques.

Random test generators include JCrasher [CS04], which creates random arguments guided by test annotations, Randoop [PE07, PLB08b], which uses feedback from executions of previously generated partial tests, and RecGen [ZZLX10] which generates method calls by analyzing the object fields modified by the code under test. JSCrasher creates instances of different types to test the behavior of public methods under random data. It attempts to detect bugs by causing the program under test to "crash", that is, to throw an undeclared runtime exception. On the other hand, Randoop uses feedback to generate method sequences, by randomly selecting a method call to apply and selecting arguments from previously constructed sequences. Unlike other random testing approaches, RecGen analyzes object fields accessed by a method under test and recommends short method sequences that mutated these fields. Furthermore, Chen et al. [CLM04, CKMT10] propose adaptive random testing, an enhanced form of random testing. The idea is to increase the effectiveness of testing by equally distributing test inputs across the input domain. However, this approach appears to be less cost-effective than random testing [AB11, CLOM08].

Some test generators address the problem of testing higher-order functions. QuickCheck [CH11] randomly generates functions that return a type-correct value. However, the generated functions do not modify any other state beyond the return value. Koopman et al. [KP06] propose to improve QuickCheck by systematically

[2]https://developers.google.com/octane/
[3]http://krakenbenchmark.mozilla.org/

Table 6.2: Test generation approaches.

Approach	Technique	Callback Generation
JSCrasher [CS04]	Random testing	No
Randoop [PE07, PLB08b]	Feedback-directed random testing	No
RecGen [ZZLX10]	Random testing and static analysis of object fields	No
ARTGen [CLM04, CKMT10]	Adaptive random testing	No
QuickCheck [CH11]	Random testing	Functions with type-correct return values
Koopman et al. [KP06]	Systematic testing	Functions generated by a user-provided generator
Klein et al. [KFF10]	Random testing	Functions with random return values
Nguyen et al [NH15]	Symbolic execution	Functions with arbitrary computation
TStest [KM17]	Feedback-directed random testing	Functions with type-correct return values
LambdaTester [MS18]	**Feedback-directed random testing and dynamic analysis of memory reads**	**Functions with targeted writes and random return values**

generating functions based on the AST representation of a function argument. The basic idea of their approach is to represent functions as a data type and to systematically enumerate elements of this data type. However, their approach does not generate callback bodies. Instead, the user of the approach needs to provide a generator functions for callback bodies, which creates expressions to be used in the body. Klein et al. [KFF10] present a new algorithm for randomly testing Racket programs that use state and callbacks. The idea is to generate new subclasses of existing classes, guided by the types of functions and the environment, and guided by developer-provided contracts. Another line of work by Nguyen et al. [NH15] adopts symbolic execution to generate higher-order inputs to functional programs. The key insight of their work is that although the space of higher-order values is huge, it is only necessary to search for counterexamples from a subset of specific functions. Counterexamples are then used to reconstruct the potentially higher-order inputs needed to crash the program. TStest [KM17] is a recent test generator for JavaScript to check TypeScript interface declarations against the corresponding JavaScript implementations. Their approach provides support for higher-order functions by passing functions that return type-correct values. Finally, *LambdaTester* is our feedback-directed random test generation approach for testing higher-order functions. In contrast to other approaches, it analyzes reads in the code under test to direct the generation of callback functions toward writing to those locations. Another important difference is that all of the above test generators, except TStest [KM17], are guided by static type signatures, which are not available in the dynamic languages targeted by *LambdaTester*.

Other forms of test generation include symbolic testing [Kin76, VPK04, XMSN05, CDE08], combining symbolic testing with static analysis [TXT+11], concolic testing [GKS05b, SMA05, GLM08], UI-level test generation [Mem07, MBvD08, MTR08, ADJ+11, EP16, TLS+13], performance-guided test generation [BJS09, PSNS14] and bounded exhaustive test generation guided by pre- and post-conditions [BKM02]. To the best of our knowledge, none of these approaches address the problem of testing higher-order functions.

6.5 Other Program Analyses for JavaScript

As JavaScript has become one of the most popular programming languages, various static and dynamic program analyses have been proposed to analyze JavaScript applications.

Static Analyses JavaScript is a weakly typed programming languages and previous research has focused on improving the existing type system and finding type-related errors. Thiemann [Thi05] defines a new type system for JavaScript that tracks the possible traits of an object and flags suspicious type conversions. Jensen et al. [JMT09] presents a static analysis infrastructure that infers sound type information for JavaScript programs using abstract interpretation. The analysis can be used to detect common programming errors or for producing type information. For computing concrete types, the information from pointer analysis is especially useful. Sridharan et al. [SDC+12] introduce correlation tracking, a technique for addressing scalability problems in points-to-analysis for JavaScript, caused by dynamic property accesses. A similar line of work by Madsen et al. [MLF13] proposes a technique which combines pointer analysis with a novel use analysis to capture usages of objects returned and passed into libraries without analyzing library code. The combination of pointer analysis and use analysis is useful for variety of applications such as: code completion, call graph discovery and discovery of concrete types.

TypeScript, a superset of JavaScript, has been developed to add optional static typing to the language. Feldthaus et al [FM14] propose a new approach to check correctness of TypeScript declaration files with respect to JavaScript library implementations. It combines an analysis of the library initialization state with a light-weight static analysis of the library code.

Dynamic Analyses The dynamic nature of the JavaScript language often makes static analysis unscalable or impossible to apply. As an alternative, dynamic analyses [SKBG13] have been proposed to find errors that are out of scope for existing static analyses. DLint [GPSS15] is a dynamic analysis approach to check code quality rules. It consists of a generic framework and an extensible set of checkers that each address a particular rule. ConflictJS [PDP18] dynamically analyzes individual libraries to find pairs of potentially conflicting libraries. The approach validates potential conflicts by creating a client application that suffers from a conflict. To find inconsistent types, TypeDevil [PSS15] employs a dynamic analysis to gather type observations at runtime. The approach merges type observations into

a graph and warns developers about variable, properties, and functions that have multiple inconsistent types. The work by Mutlu et al. [MTL15] proposes a novel lightweight runtime symbolic exploration algorithm for finding races in JavaScript applications. The key advantages of the approach is that it requires only a single execution and reports only harmful races. Dynamic determinacy analysis by Schaefer et al. [SSDT13] analyzes more or one concrete executions to identify variables and expressions that have the same value at a given program point in any execution. This information can be exploited by other analyses and tools to, e.g., identify dead code or specialize uses of dynamic language constructs.

Other dynamic analyses support developers in program comprehension tasks. Clematis [ASMP14] is a technique for capturing low-level event-based interactions in a web application and mapping those to a higher-level behavioral model. This model is then visualized to illustrate episodes of triggered events, related JavaScript code executions and their impact on DOM state. Tochal [AMP15] is a DOM-sensitive event-aware change impact analysis technique for JavaScript. The approach creates a novel hybrid model to identify the impact set of a change in a given application.

6.6 Optimizations of Big Data Jobs

The analysis in Chapter 4 leverages cross-language optimization to reduce cross-runtime interactions in big data queries. Unfortunately, many powerful query optimizations [CS94, YL95, MP94, SHP+96, Kim82, Mur92, CKPS95, PL08, Sel88, BK89, FFN+08, HS93, CGK89] are not applicable in modern big-data processing systems because the traditional query optimizer treats non-relational code as a black-box.

To improve the performance of modern large-scale data processing systems, several optimization strategies have been proposed. Quincy [IPC+09] is a framework for scheduling concurrent distributed jobs with fine-grain resource sharing. LATE [ZKJ+08] addresses scheduling problems in heterogeneous environments by estimating the time needed to speculatively execute the task that hurt the response time the most. Hadoop++ [DQRJ+10] improves the query runtime of a Hadoop system by changing the internal layout of a *split*, a large horizontal partition of the data, and feeding Hadoop with appropriate user-written functions. Floratou et al. [FPST11] propose an approach to speed up MapReduce jobs by using column-oriented binary storage formats in Hadoop compatible with its replication and scheduling constraints. MRShare [NPM+10] is a sharing framework tailored to MapReduce programs. It merges jobs into groups and evaluates each group as a single query which enables more efficient execution of different jobs that perform similar work. Although some of the standard database optimizations, such as filter pushdown are implemented in Pig [ORSS08], the recent work by Jahani et al. [JCR11] suggests that many traditional query optimizations are not applicable to MapReduce [DG08] because of user-written operators. It further proposes several optimizations of *map()* functions by targeting data-centric programming idioms [JCR11]. These optimizations include data compression and eliminating unnecessary fields from files and indexing. In Chapter 4, we show that the time spent in non-relational code takes a large fraction

of data center time. Furthermore, we propose a novel cross-language optimization based on method inlining to improve the performance of MapReduce jobs.

Many MapReduce systems, such as Hadoop [DQRJ+10], provide facilities for monitoring cluster performance. The collected metrics usually represent cluster-level information from which the regular user does not benefit. Herodotou et al. [HB11] introduce a new dynamic binary instrumentation of the MapReduce framework to capture dataflows and costs during job execution at the task level or the phase level. Obtained profiles help developers apply a new class of optimization opportunities based on tuning of the configuration parameters. In contrast to the dynamic analysis, our approach to profiling big data jobs is purely static and based on the analysis of job artifacts. Doing this allows us to analyze a large number of jobs without introducing any additional overhead.

Conclusion

In this dissertation, we present actionable program analyses to improve software performance. More concretely, we focus on an empirical study of the most common performance issues in JavaScript programs (Chapter 2), analyses to find reordering opportunities (Chapter 3) and method inlining opportunities(Chapter 4) and a novel test generation technique for higher-order functions in dynamic languages (Chapter 5). These approaches aim to reduce manual effort by suggesting only beneficial optimization opportunities that are easy to understand and applicable across multiple projects.

7.1 Summary of Contributions

Chapter 2 presents an empirical study of 98 performance issues and optimizations in JavaScript projects. The results of the study show that many of the studied optimizations have the following key properties: they are *effective*, *exploitable*, *recurring*, and *out-of-reach for compilers*. To help developers find and exploit such optimization opportunities, Chapters 3 and 4 propose two actionable program analyses. The first analysis finds reordering opportunities in switch statements and logical expressions by dynamically analyzing and computing the optimal orders of evaluations. The second analysis statically analyzes programs for big data processing to find opportunities for method inlining, i.e., replacing a call to a user-written function with the logic of that function. Finally, Chapter 5 discusses how to improve state-of-the-art test generation approaches to generate effective tests for higher-order functions in dynamic languages such as JavaScript. Generated tests then can be used to drive the program execution during dynamic analysis and to reliably measure the performance impact of applied optimizations.

In this dissertation, we show that it is possible to automatically suggest effective, exploitable, recurring and out-of-reach for compilers optimization opportunities. In particular:

- By empirically studying performance issues and optimizations in real-world software, we show 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 JIT engines, we find that less than half of all optimizations improve performance consistently across all engines. Furthermore, we observe that many

optimizations are instances of patterns applicable across projects. These results motivate the development of performance-related techniques that address relevant performance problems.

- Applying these optimizations in a fully automatic way is a challenging task: they are subject to preconditions that are hard to check or can be checked only at runtime. We propose two program analyses that prove to be powerful in finding optimization opportunities in complex programs. Even though our approaches do not guarantee that code transformations are semantics-preserving, the experimental results illustrate that suggested optimizations do not change program behavior.

- Reliably finding optimization opportunities and measuring their performance benefits require a program to be exercised with sufficient inputs. One possible solution to this problem is to use automated test generation techniques. We complement existing testing approaches by addressing the problem of test generation for higher-order functions. Finally, we show that generating effective tests for higher-order functions triggers behaviors that are usually not triggered by state-of-the-art testing approaches.

7.2 Future Research Directions

Assessing Performance Impact Across Engines Reliably assessing the performance benefits of applied optimizations is a challenging task, especially if a program runs in multiple environments. As discussed in Chapter 2, some of the manually applied optimizations can even cause performance degradation in some versions of JavaScript engines. Optimization strategies greatly differ across different engines and also across different versions of the same engine. To make sure that optimizations lead to positive performance improvements in all engines, future work should focus on techniques that monitor the performance effects of code changes across multiple execution environments.

Automatically Identifying Optimization Patterns Existing approaches that address performance bottlenecks either look for general performance properties, such as hot functions, or specific patterns of performance issues. As already shown in Chapters 3 and Chapter 4, finding and applying specific optimization opportunities can lead to significant performance improvements. However, this requires manually identifying optimization patterns and hard-coding them into the respective analysis. Manually studying instances of inefficient code and finding recurring patterns is a challenging task that often requires significant human effort. Even though we studied a significant number of performance problems and drew interesting conclusions in Chapter 2, the next interesting research question is: *how to automatically find optimization patterns that have significant performance benefits and are applicable across multiple projects?*

Analyses to Find Other Optimization Opportunities In this dissertation, we propose approaches that address two different types of optimizations: reordering opportunities and method inlining. However, in Chapter 2 we identify many optimization patterns that have the same properties as those we address. Therefore, it is an important research direction to propose novel approaches that address other kinds of performance issues and provide actionable advices to developers.

Bibliography

[AAFM10] Erik R. Altman, Matthew Arnold, Stephen Fink, and Nick Mitchell. Performance analysis of idle programs. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 739–753. ACM, 2010.

[AB11] Andrea Arcuri and Lionel Briand. Adaptive random testing: An illusion of effectiveness? In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 265–275, New York, NY, USA, 2011. ACM.

[ACS+14] Wonsun Ahn, Jiho Choi, Thomas Shull, María J. Garzarán, and Josep Torrellas. Improving JavaScript performance by deconstructing the type system. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 496–507, 2014.

[ADJ+11] Shay Artzi, Julian Dolby, Simon Holm Jensen, Anders Møller, and Frank Tip. A framework for automated testing of JavaScript web applications. In *ICSE*, pages 571–580, 2011.

[AGM+17] Esben Andreasen, Liang Gong, Anders Møller, Michael Pradel, Marija Selakovic, Koushik Sen, and Cristian-Alexandru Staicu. A survey of dynamic analysis and test generation for javascript. *ACM Computing Surveys*, 2017.

[AMN17] Esben Sparre Andreasen, Anders Møller, and Benjamin Barslev Nielsen. Systematic approaches for increasing soundness and precision of static analyzers. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, SOAP 2017, pages 31–36, New York, NY, USA, 2017. ACM.

[AMP15] Saba Alimadadi, Ali Mesbah, and Karthik Pattabiraman. Hybrid DOM-sensitive change impact analysis for JavaScript. In *ECOOP*, pages 321–345, 2015.

[ASMP14] Saba Alimadadi, Sheldon Sequeira, Ali Mesbah, and Karthik Pattabiraman. Understanding javascript event-based interactions. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, pages 367–377, 2014.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers. Principles, Techniques and Tools*. Addison Wesley, 1986.

[Ayc03] John Aycock. A brief history of just-in-time. pages 97–113, 2003.

[BGH07] Marat Boshernitsan, Susan L. Graham, and Marti A. Hearst. Aligning development tools with the way programmers think about code changes. In *CHI*, pages 567–576, 2007.

[BJS09] Jacob Burnim, Sudeep Juvekar, and Koushik Sen. WISE: Automated test generation for worst-case complexity. In *ICSE*, pages 463–473. IEEE, 2009.

[BK89] E. Bertino and W. Kim. Indexing techniques for queries on nested objects. *IEEE Trans. on Knowl. and Data Eng.*, 1(2):196–214, June 1989.

[BKM02] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: automated testing based on Java predicates. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 123–133, 2002.

[CASP13] Igor Costa, Péricles Alves, Henrique Nazare Santos, and Fernando Magno Quintão Pereira. Just-in-time value specialization. In *CGO*, pages 1–11, 2013.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 209–224. USENIX, 2008.

[CGK89] D. Chimenti, R. Gamboa, and R. Krishnamurthy. Towards an open architecture for ldl. In *Proceedings of the 15th International Conference on Very Large Data Bases*, VLDB '89, pages 195–203, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[CH11] Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. *SIGPLAN Not.*, 46(4):53–64, May 2011.

[CJL+08] Ronnie Chaiken, Bob Jenkins, Perake Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. Scope: Easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.*, 1(2):1265–1276, August 2008.

[CKMT10] Tsong Yueh Chen, Fei-Ching Kuo, Robert G. Merkel, and T. H. Tse. Adaptive random testing: The ART of test case diversity. *Journal of Systems and Software*, 83(1):60–66, 2010.

[CKPS95] Surajit Chaudhuri, Ravi Krishnamurthy, Spyros Potamianos, and Kyuseok Shim. Optimizing queries with materialized views. In *Proceedings of the Eleventh International Conference on Data Engineering*, ICDE '95, pages 190–200, Washington, DC, USA, 1995. IEEE Computer Society.

[CLM04] Tsong Yueh Chen, Hing Leung, and IK Mak. Adaptive random testing. In *Annual Asian Computing Science Conference*, pages 320–329. Springer, 2004.

[CLOM08] Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. ARTOO: adaptive random testing for object-oriented software. In *International Conference on Software Engineering (ICSE)*, pages 71–80. ACM, 2008.

[CPO+11] I. Ciupa, A. Pretschner, M. Oriol, A. Leitner, and B. Meyer. On the number and nature of faults found by random testing. *Software Testing, Verification and Reliability*, 21(1):3–28, 2011.

[CS94] Surajit Chaudhuri and Kyuseok Shim. Including group-by in query optimization. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 354–366, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[CS04] Christoph Csallner and Yannis Smaragdakis. JCrasher: an automatic robustness tester for Java. *Software Practice and Experience*, 34(11):1025–1050, 2004.

[CSL04] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '04, pages 2–2, Berkeley, CA, USA, 2004. USENIX Association.

[Dar78] John Darlington. A synthesis of several sorting algorithms. *Acta Inf.*, 11(1):1–30, March 1978.

[DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[DN84] Joe W. Duran and Simeon C. Ntafos. An evaluation of random testing. *IEEE Trans. Softw. Eng.*, 10(4):438–444, July 1984.

[DQRJ+10] Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz, Alekh Jindal, Yagiz Kargin, Vinay Setty, and Jörg Schad. Hadoop++: Making a yellow elephant run like a cheetah (without it even noticing). *Proc. VLDB Endow.*, 3(1-2):515–529, September 2010.

[DR16] Monika Dhok and Murali Krishna Ramanathan. Directed test generation to detect loop inefficiencies. In *FSE*, 2016.

[DRSS01] Reiner R. Dumke, Claus Rautenstrauch, Andreas Schmietendorf, and André Scholz, editors. *Performance Engineering, State of the Art and Current Trends*, London, UK, UK, 2001. Springer-Verlag.

[EP16] Markus Ermuth and Michael Pradel. Monkey see, monkey do: Effective generation of gui tests with inferred macro events. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 82–93, 2016.

[FA11] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*, pages 416–419, 2011.

[FFN+08] Yi Fang, Marc Friedman, Giri Nair, Michael Rys, and Ana-Elisa Schmid. Spatial indexing in microsoft sql server 2008. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1207–1216, New York, NY, USA, 2008. ACM.

[FM14] Asger Feldthaus and Anders 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.

[FPST11] Avrilia Floratou, Jignesh M. Patel, Eugene J. Shekita, and Sandeep Tata. Column-oriented storage techniques for mapreduce. *Proc. VLDB Endow.*, 4(7):419–429, April 2011.

[GAW07] Simon Goldsmith, Alex Aiken, and Daniel Shawcross Wilkerson. Measuring empirical computational complexity. In *European Software Engineering Conference and International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 395–404. ACM, 2007.

[GBE07] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous Java performance evaluation. In *Conference on Object-Oriented Programming, Systems, Languages, and Application (OOPSLA)*, pages 57–76. ACM, 2007.

[GES+09] 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 *PLDI*, pages 465–478, 2009.

[GFX12] Mark Grechanik, Chen Fu, and Qing Xie. Automatically finding performance problems with feedback-directed learning software testing.

In *International Conference on Software Engineering (ICSE)*, pages 156–166, 2012.

[GHJ07] Alex Groce, Gerard Holzmann, and Rajeev Joshi. Randomized differential testing as a prelude to formal verification. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 621–631, Washington, DC, USA, 2007. IEEE Computer Society.

[GKM82] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. Gprof: A call graph execution profiler. In *SIGPLAN Symposium on Compiler Construction*, pages 120–126. ACM, 1982.

[GKS05a] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 213–223. ACM, 2005.

[GKS05b] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. *SIGPLAN Not.*, 40(6):213–223, June 2005.

[GLM08] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Automated whitebox fuzz testing. In *Network and Distributed System Security Symposium (NDSS)*, 2008.

[GPC14] Irene Lizeth Manotas Gutiérrez, Lori L. Pollock, and James Clause. Seeds: a software engineer's energy-optimization decision support framework. In *ICSE*, pages 503–514, 2014.

[GPS15] Liang Gong, Michael Pradel, and Koushik Sen. JITProf: Pinpointing JIT-unfriendly JavaScript code. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 357–368, 2015.

[GPSS15] Liang Gong, Michael Pradel, Manu Sridharan, and Koushik Sen. DLint: Dynamically checking bad coding practices in JavaScript. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 94–105, 2015.

[HB11] Herodotos Herodotou and Shivnath Babu. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. 4:1111–1122, 01 2011.

[HDG+12] Shi Han, Yingnong Dang, Song Ge, Dongmei Zhang, and Tao Xie. Performance debugging in the large via mining millions of stack traces. In *International Conference on Software Engineering (ICSE)*, pages 145–155. IEEE, 2012.

[HG12] Brian Hackett and Shu-yu Guo. Fast and precise hybrid type inference for JavaScript. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 239–250. ACM, 2012.

[HNS09] Albert Hartono, Boyana Norris, and P. Sadayappan. Annotation-based empirical performance tuning using orio. *2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–11, 2009.

[HS93] Joseph M. Hellerstein and Michael Stonebraker. Predicate migration: Optimizing queries with expensive predicates. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD '93, pages 267–276, New York, NY, USA, 1993. ACM.

[HSC15] Stefan Heule, Manu Sridharan, and Satish Chandra. Mimic: computing models for opaque code. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 710–720, 2015.

[IPC+09] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 261–276, New York, NY, USA, 2009. ACM.

[JAH11] Milan Jovic, Andrea Adamoli, and Matthias Hauswirth. Catch me if you can: performance bug detection in the wild. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 155–170. ACM, 2011.

[JCR11] Eaman Jahani, Michael J. Cafarella, and Christopher Ré. Automatic optimization for mapreduce programs. *Proc. VLDB Endow.*, 4(6):385–396, March 2011.

[JMT09] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for JavaScript. In *Symposium on Static Analysis (SAS)*, pages 238–255. Springer, 2009.

[JSS+12] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and detecting real-world performance bugs. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 77–88. ACM, 2012.

[JSSC15] Simon Holm Jensen, Manu Sridharan, Koushik Sen, and Satish Chandra. Meminsight: platform-independent memory debugging for javascript. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 345–356, 2015.

[KFF10] Casey Klein, Matthew Flatt, and Robert Bruce Findler. Random testing for higher-order, stateful programs. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 555–566. ACM, 2010.

[Kim82] Won Kim. On optimizing an sql-like nested query. *ACM Trans. Database Syst.*, 7(3):443–469, September 1982.

[Kin76] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[KM17] Erik Krogh Kristensen and Anders Møller. Type test scripts for typescript testing. *PACMPL*, 1(OOPSLA):90:1–90:25, 2017.

[Knu74] Donald E. Knuth. Computer programming as an art. *Commun. ACM*, 17(12):667–673, December 1974.

[KP06] Pieter Koopman and Rinus Plasmeijer. Automatic testing of higher order functions. In *Asian Symposium on Programming Languages and Systems (APLAS)*, pages 148–164, 2006.

[KRH15] Madhukar N. Kedlaya, Behnam Robatmili, and Ben Hardekopf. Server-side type profiling for optimizing client-side javascript engines. In *Proceedings of the 11th Symposium on Dynamic Languages*, DLS 2015, pages 140–153, New York, NY, USA, 2015. ACM.

[LBBC+14] Mario Linares Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Rocco Oliveto, Massimiliano Di Penta, and Denys Poshyvanyk. Mining energy-greedy api usage patterns in android apps: an empirical study. In *MSR*, pages 2–11, 2014.

[LSS+15] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: A manifesto. *Commun. ACM*, 58(2):44–46, January 2015.

[LV10] Francesco Logozzo and Herman Venter. RATA: Rapid atomic type analysis by abstract interpretation—application to JavaScript optimization. In *CC*, pages 66–83, 2010.

[LXC14] Yepang Liu, Chang Xu, and S.C. Cheung. Characterizing and detecting performance bugs for smartphone applications. In *ICSE*, pages 1013–1024, 2014.

[MBvD08] Ali Mesbah, Engin Bozdag, and Arie van Deursen. Crawling Ajax by inferring user interface state changes. In *International Conference on Web Engineering (ICWE)*, pages 122–134, 2008.

[McC76] Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, December 1976.

[McK98] William M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

[MDHS09] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing wrong data without doing anything obviously wrong! In *ASPLOS*, pages 265–276, 2009.

[Mem07] Atif M. Memon. An event-flow model of GUI-based applications for testing. *Softw. Test., Verif. Reliab.*, pages 137–157, 2007.

[MKM11] Na Meng, Miryung Kim, and Kathryn S. McKinley. Systematic editing: generating program transformations from an example. In *PLDI*, pages 329–342, 2011.

[MKM13] Na Meng, Miryung Kim, and Kathryn S. McKinley. Lase: locating and applying systematic edits by learning from examples. In *ICSE*, pages 502–511, 2013.

[MLF13] Magnus Madsen, Benjamin Livshits, and Michael Fanning. Practical static analysis of JavaScript applications in the presence of frameworks and libraries. In *ESEC/SIGSOFT FSE*, pages 499–509, 2013.

[MO03] Darko Marinov and Robert O'Callahan. Object equality profiling. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 313–325, 2003.

[MP94] Inderpal Singh Mumick and Hamid Pirahesh. Implementation of magic-sets in a relational database system. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, SIGMOD '94, pages 103–114, New York, NY, USA, 1994. ACM.

[MS18] Rezwana Karim Frank Tip Marija Selakovic, Michael Pradel. Test generation for higher-order functions in dynamic languages. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 2018.

[MTHG15] David Maplesden, Ewan D. Tempero, John G. Hosking, and John C. Grundy. Subsuming methods: Finding new optimisation opportunities in object-oriented software. In *ICPE*, pages 175–186, 2015.

[MTL15] Erdal Mutlu, Serdar Tasiran, and Benjamin Livshits. Detecting javascript races that matter. In *European Software Engineering Conference and International Symposium on Foundations of Software Engineering (ESEC/FSE)*, 2015.

[MTR08] Alessandro Marchetto, Paolo Tonella, and Filippo Ricca. State-based testing of Ajax web applications. In *ICST*, pages 121–130. IEEE Computer Society, 2008.

[Mur92] M. Muralikrishna. Improved unnesting algorithms for join aggregate sql queries. In *Proceedings of the 18th International Conference on Very Large Data Bases*, VLDB '92, pages 91–102, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc.

[MvD09] Ali Mesbah and Arie van Deursen. Invariant-based automatic testing of Ajax user interfaces. In *ICSE*, pages 210–220, 2009.

[NCRL15] Adrian Nistor, Po-Chun Chang, Cosmin Radoi, and Shan Lu. Caramel: Detecting and fixing performance problems that have non-intrusive fixes. In *ICSE*, 2015.

[NH15] Phuc C. Nguyen and David Van Horn. Relatively complete counterexamples for higher-order programs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 446–456, 2015.

[NPM+10] Tomasz Nykiel, Michalis Potamias, Chaitanya Mishra, George Kollios, and Nick Koudas. Mrshare: Sharing across multiple queries in mapreduce. *Proc. VLDB Endow.*, 3(1-2):494–505, September 2010.

[NR14] Adrian Nistor and Lenin Ravindranath. Suncat: helping developers understand and predict performance problems in smartphone applications. In *ISSTA*, pages 282–292, 2014.

[NSML13] Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. Toddler: Detecting performance problems via similar memory-access patterns. In *International Conference on Software Engineering (ICSE)*, pages 562–571, 2013.

[Nta01] S. C. Ntafos. On comparisons of random, partition, and proportional partition testing. *IEEE Transactions on Software Engineering*, 27(10):949–960, Oct 2001.

[ODL15] Oswaldo Olivo, Isil Dillig, and Calvin Lin. Static detection of asymptotic performance bugs in collection traversals. In *PLDI*, pages 369–378, 2015.

[ORR+15] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. Making sense of performance in data analytics frameworks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 293–307, Oakland, CA, 2015. USENIX Association.

[ORSS08] Christopher Olston, Benjamin Reed, Adam Silberstein, and Utkarsh Srivastava. Automatic optimization of parallel dataflow programs. In *USENIX 2008 Annual Technical Conference*, ATC'08, pages 267–273, Berkeley, CA, USA, 2008. USENIX Association.

[PDP18] Jibesh Patra, Pooja N. Dixit, and Michael Pradel. Conflictjs: Finding and understanding conflicts between javascript libraries. In *ICSE*, 2018.

[PE07] Carlos Pacheco and Michael D. Ernst. Randoop: Feedback-directed random testing for java. In *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*, OOPSLA '07, pages 815–816, New York, NY, USA, 2007. ACM.

[PG12] Michael Pradel and Thomas R. Gross. Fully automatic and precise detection of thread safety violations. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 521–530, 2012.

[PH13] Jacques A. Pienaar and Robert Hundt. Jswhiz: Static analysis for javascript memory leaks. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2013, Shenzhen, China, February 23-27, 2013*, pages 11:1–11:11, 2013.

[PHG14] Michael Pradel, Markus Huggler, and Thomas R. Gross. Performance regression testing of concurrent classes. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 13–25, 2014.

[PL08] Thomas Phan and Wen-Syan Li. Dynamic materialization of query views for data warehouse workloads. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, ICDE '08, pages 436–445, Washington, DC, USA, 2008. IEEE Computer Society.

[PLB08a] Carlos Pacheco, Shuvendu K. Lahiri, and Thomas Ball. Finding errors in .NET with feedback-directed random testing. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 87–96. ACM, 2008.

[PLB08b] Carlos Pacheco, Shuvendu K. Lahiri, and Thomas Ball. Finding errors in .net with feedback-directed random testing. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 87–96, New York, NY, USA, 2008. ACM.

[PSNS14] Michael Pradel, Parker Schuh, George Necula, and Koushik Sen. EventBreak: Analyzing the responsiveness of user interfaces through performance-guided test generation. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 33–47, 2014.

[PSS15] Michael Pradel, Parker Schuh, and Koushik Sen. TypeDevil: Dynamic type inconsistency analysis for JavaScript. In *International Conference on Software Engineering (ICSE)*, 2015.

[RBVK16] Veselin Raychev, Pavol Bielik, Martin T. Vechev, and Andreas Krause. Learning programs from noisy data. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 761–774, 2016.

[RGEV11] Gregor Richards, Andreas Gal, Brendan Eich, and Jan Vitek. Automated construction of JavaScript benchmarks. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 677–694, 2011.

[RLZ10] Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin G. Zorn. Jsmeter: Comparing the behavior of javascript benchmarks with real web applications. In *USENIX Conference on Web Application Development, WebApps'10, Boston, Massachusetts, USA, June 23-24, 2010*, 2010.

[SAG15] Vincent St-Amour and Shu-yu Guo. Optimization coaching for javascript. In *ECOOP*, 2015.

[SBMM18] Marija Selakovic, Michael Barnett, Madan Musuvathi, and Todd Mytkowicz. Cross-language optimizations in big data systems: A case study of scope. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '18, pages 45–54, New York, NY, USA, 2018. ACM.

[SDC+12] Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and Frank Tip. Correlation tracking for points-to analysis of javascript. In *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings*, pages 435–458, 2012.

[Sel88] Timos K. Sellis. Intelligent caching and indexing techniques for relational database systems. *Information Systems*, 13(2):175 – 185, 1988.

[SGP17] Marija Selakovic, Thomas Glaser, and Michael Pradel. An actionable performance profiler for optimizing the order of evaluations. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 170–180, 2017.

[SHP+96] Praveen Seshadri, Joseph M. Hellerstein, Hamid Pirahesh, T. Y. Cliff Leung, Raghu Ramakrishnan, Divesh Srivastava, Peter J. Stuckey, and S. Sudarshan. Cost-based optimization for magic: Algebra and implementation. *SIGMOD Rec.*, 25(2):435–446, June 1996.

[SKBG13] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for javascript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 488–498, New York, NY, USA, 2013. ACM.

[SMA05] Koushik Sen, Darko Marinov, and Gul Agha. Cute: A concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. ACM.

[SP16] Marija Selakovic and Michael Pradel. Performance issues and optimizations in JavaScript: An empirical study. In *International Conference on Software Engineering (ICSE)*, pages 61–72, 2016.

[SPL18] Cristian-Alexandru Staicu, Michael Pradel, and Ben Livshits. Understanding and automatically preventing injection attacks on Node.js. In *Network and Distributed System Security Symposium (NDSS)*, 2018.

[SR14] Malavika Samak and Murali Krishna Ramanathan. Multithreaded test synthesis for deadlock detection. In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 473–489, 2014.

[SSDT13] Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. Dynamic determinacy analysis. In *PLDI*, pages 165–174, 2013.

[Thi05] Peter Thiemann. Towards a type system for analyzing JavaScript programs. In *European Symposium on Programming (ESOP)*, pages 408–422, 2005.

[TLS+13] Suresh Thummalapenta, K. Vasanta Lakshmi, Saurabh Sinha, Nishant Sinha, and Satish Chandra. Guided test generation for web applications. In *International Conference on Software Engineering (ICSE)*, pages 162–171. IEEE, 2013.

[TPG15] Luca Della Toffola, Michael Pradel, and Thomas R. Gross. Performance problems you can fix: A dynamic analysis of memoization opportunities. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 607–622, 2015.

[TXT+11] Suresh Thummalapenta, Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Zhendong Su. Synthesizing method sequences for high-coverage testing. In *OOPSLA*, pages 189–206, 2011.

[VPK04] Willem Visser, Corina S. Pasareanu, and Sarfraz Khurshid. Test input generation with Java PathFinder. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 97–107. ACM, 2004.

[WHH13] Alexander Wert, Jens Happe, and Lucia Happe. Supporting swift reaction: Automatically uncovering performance problems by systematic experiments. In *International Conference on Software Engineering (ICSE)*, pages 552–561, 2013.

[XAM+09] Guoqing (Harry) Xu, Matthew Arnold, Nick Mitchell, Atanas Rountev, and Gary Sevitsky. Go with the flow: profiling copies to find runtime bloat. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 419–430. ACM, 2009.

[XHZZ15] Xiao Xiao, Shi Han, Charles Zhang, and Dongmei Zhang. Uncovering JavaScript performance code smells relevant to type mutations. In *APLAS*, volume 9458, pages 335–355, 2015.

[XMA+10] Guoqing (Harry) Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, Edith Schonberg, and Gary Sevitsky. Finding low-utility data structures. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 174–186, 2010.

[XMSN05] Tao Xie, Darko Marinov, Wolfram Schulte, and David Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 365–381. Springer, 2005.

[XR10] Guoqing Xu and Atanas Rountev. Detecting inefficiently-used containers to avoid bloat. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 160–173. ACM, 2010.

[YHZX14] Xiao Yu, Shi Han, Dongmei Zhang, and Tao Xie. Comprehending performance from real-world execution traces: a device-driver case. In *ASPLOS*, pages 193–206, 2014.

[YL95] Weipeng P. Yan and Perake Larson. Eager aggregation and lazy aggregation. In *Proceedings of the 21th International Conference on Very Large Data Bases*, VLDB '95, pages 345–357, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[YP16] Tingting Yu and Michael Pradel. Syncprof: Detecting, localizing, and optimizing synchronization bottlenecks. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 389–400, 2016.

[YXR12] Dacong Yan, Guoqing (Harry) Xu, and Atanas Rountev. Uncovering performance problems in Java applications with reference propagation profiling. In *International Conference on Software Engineering, (ICSE)*, pages 134–144. IEEE, 2012.

[ZAH12] Shahed Zaman, Bram Adams, and Ahmed E. Hassan. A qualitative study on performance bugs. In *Working Conference on Mining Software Repositories (MSR)*, pages 199–208. IEEE, 2012.

[ZH12] Dmitrijs Zaparanuks and Matthias Hauswirth. Algorithmic profiling. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 67–76, 2012.

[ZKJ+08] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 29–42, Berkeley, CA, USA, 2008. USENIX Association.

[ZZK16] Hua Zhong, Lingming Zhang, and Sarfraz Khurshid. Combinatorial generation of structurally complex test inputs for commercial software

applications. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 981–986, New York, NY, USA, 2016. ACM.

[ZZLX10] Wujie Zheng, Qirun Zhang, Michael Lyu, and Tao Xie. Random unit-test generation with mut-aware sequence recommendation. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 293–296, New York, NY, USA, 2010. ACM.