Actionable Program Analyses for Improving Software Performance

Marija Selakovic

Abstract Nowadays, we have greater expectations of software than ever before. This is followed by the constant pressure to run the same program on smaller and cheaper machines. To meet this demand, the application's performance has become an 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 profilers report only where resources are spent, but not where resources are wasted. In this chapter, we investigate actionable performance analyses that help developers optimize their software by applying relatively simple code changes. We focus on optimizations that are *effective*, *exploitable*, *recurring*, and *out-of-reach for compilers*. These properties suggest that proposed optimizations lead to significant performance improvement, that they are easy to understand and apply, applicable across multiple projects, and that the compilers cannot guarantee that these optimizations always preserve the original program semantics. We implement our actionable analyses in practical tools and demonstrate their potential in improving software performance by applying relatively simple code optimizations.

1 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.

M. Selakovic (✉)
TU Darmstadt, Darmstadt, Germany

© The Author(s) 2020

M. Felderer et al. (eds.), *Ernst Denert Award for Software Engineering 2019*,
https://doi.org/10.1007/978-3-030-58617-1_7

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

- *Performance profiling*. Developers conduct performance testing in the form of CPU [16] and memory profiling [20] 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 [2] 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.
- *Manual tuning*. Finally, developers often rely on manual performance tuning [19] (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.

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 chapter addresses some of those limitations and proposes new approaches to help developers optimize their code with little effort.

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

To illustrate the potential of simple code transformations on software performance, Fig. 1 illustrates a performance issue and an associated optimization reported in *Underscore*, one of the most popular JavaScript utility libraries.

```
_.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)

```
_.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)

Fig. 1 Performance issue from Underscore library (pull request 1708). (**a**) Performance issue. (**b**) Optimized code

Figure 1 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 Fig. 1. This optimization improves performance because JavaScript engines are able to specialize the code in the for loop and execute it faster.

The optimization in Fig. 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 cannot guarantee that the code transformation is always semantics-preserving.

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

- *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.
- *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.
- *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.

In this chapter, we show 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 not only to pinpoint where and why time is spent but also to provide actionable advice on how to improve the application's performance.

1.3 Outline

The remaining sections of this chapter are organized as follows: Sect. 2 presents the first empirical study on performance issues and optimizations in JavaScript projects. Sections 3 and 4 present two actionable performance analyses that find reordering opportunities and method inlining optimizations. Section 5 gives an overview of the test generation approaches for higher-order functions in dynamic languages. Finally, Sect. 6 discusses conclusions and directions for future work.

2 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 section, we introduce an empirical study on performance issues and optimizations in real-world JavaScript projects. We chose JavaScript because it has become one of the most popular programming languages, used not only for client-side web applications but also for server-side applications, mobile applications, and even desktop applications.

Despite the effectiveness of highly optimizing just-in-time (JIT) compilers [13, 23, 18, 9, 1], developers still manually apply optimizations to address performance issues in their code. Furthermore, future improvements of JavaScript engines are unlikely to completely erase the need for manual performance optimizations.

To find optimizations amenable for actionable performance analyses, we first need to answer 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: Are there recurring optimization patterns, and can they be applied automatically?

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.

2.2 *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 ($\geq$ 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.

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

Table 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	7608	27
jQuery	Client-side library	Client	6348	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	1110	12
Underscore.string	String manipulation	Client and server	901	3
Backbone	MVC framework	Client and server	1131	5
EJS	Embedded templates	Client and server	354	3
Moment	Date manipulation library	Client and server	2359	3
NodeLruCache	Caching support library	Client and server	221	1
Q	Library for asynchronous promises	Client and server	1223	1
Cheerio	jQuery implementation for server-side	Server	1268	9
Chalk	Terminal string styling library	Server	78	3
Mocha	Testing framework	Server	7843	2
Request	HTTP request client	Server	1144	2
Socket.io	Real-time application framework	Server	703	2
Total			63, 951	98

2.3 Selection of Performance Issues

We select performance issues from bug trackers as follows:

1. *Keyword-based search or explicit labels*. One of the studied projects, Angular.js, explicitly labels performance issues, so we focus on them. For all other projects, we search the title, description, and comments 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 on unit tests and microbenchmarks allows us to assess the performance impact of the changed code while minimizing other confounding factors.

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

2.4 Main Findings

In this section, we discuss the main findings and provide detailed answers on the three research questions.

Root Causes of Performance Issues To address the first question, we identify eight root causes that are common among the 98 studied issues, and we assign each issue into one or more root causes. 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. Figure 2b further classifies these issues by the API that is used inefficiently. For example, the most commonly misused APIs are reflection APIs, followed by string and DOM APIs.

[2] For example, jsperf.com is a popular microbenchmarking website.

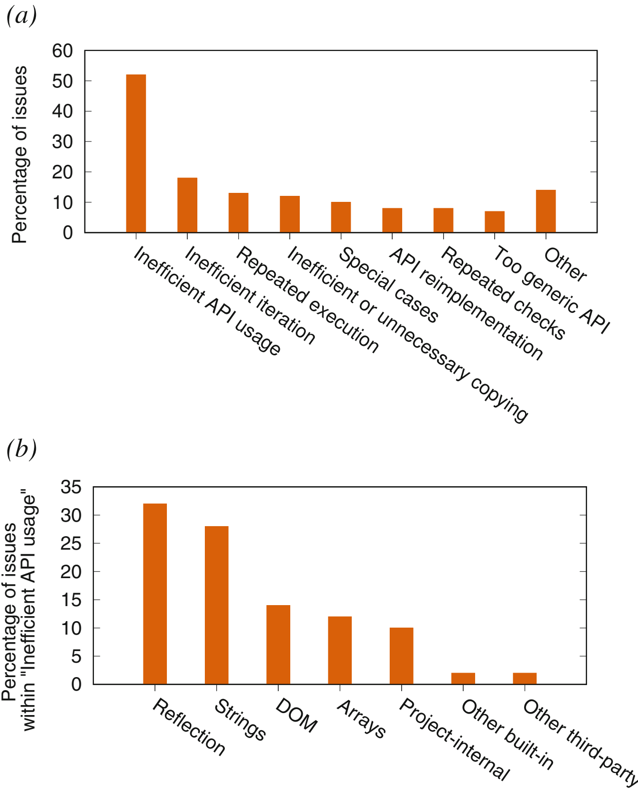

Fig. 2 Root causes of performance issues. (**a**) Most prevalent root causes. (**b**) APIs that are used inefficiently

Besides *inefficient API usage*, we identify seven other common causes of performance issues as illustrated by Fig. 2. The full description of these causes can be further found in [35], and some but not all of them have been addressed by existing approaches for automatically finding performance problems [15, 38, 39]. However, our results suggest that there is a need for additional techniques to help developers find and fix instances of other common performance issues.

Complexity of Changes To better 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 [24] of the program before and after each change. These metrics approximate the understandability and maintainability of the code.

Figure 3a, b summarizes our results. The graphs show what percentage of optimizations affect the number of statements and the cyclomatic complexity in a particular range. We find that a large portion of all optimizations do not affect the

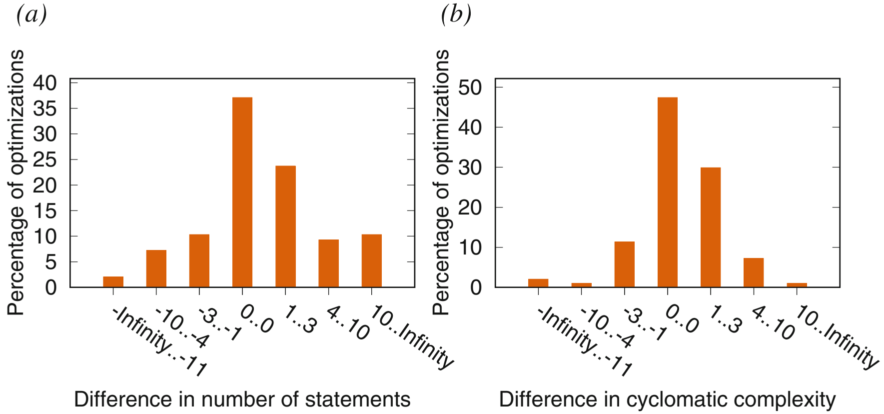

Fig. 3 Effect of applying an optimization on the cyclomatic complexity. (**a**) Effect on the number of statements. (**b**) Effect on cyclomatic complexity

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. 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 [22, 11]. We conclude from these results that many optimizations are possible without increasing the complexity of the optimized program.

Recurring Optimization Patterns To identify performance optimizations that may apply in more than a single situation, we inspect all studied issues. First, we identify optimizations that occur repeatedly within the study. Furthermore, since some issues may not expose multiple instances of a pattern that would occur repeatedly in a larger set, we also identify patterns that may occur repeatedly. To find whether there are occurrences of the optimization patterns beyond the 98 studied optimizations, we develop a simple, AST-based, static analysis for each pattern and apply it to programs used in the study.

We find that the analyses cannot guarantee that the optimization patterns can be applied in a fully automated way without changing the program's semantics due to the following features of JavaScript language:

- *Dynamic types*: the types of identifiers and variables can be dynamically changed.
- *Dynamic changes of object prototypes*: properties of object prototype can be dynamically overridden.
- *Dynamic changes of native methods*: native or third-party functions can be dynamically overridden.

For example, in Fig. 1, the `obj` identifier must always have `Object` type, the `hasOwnProperty` property of `Object.prototype` must not be overridden, and both `hasOwnProperty()` and `keys()` must be built-in JavaScript functions.

To 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. In total, we find 139 new instances of recurring optimization patterns, not only across single project but also across multiple projects. These results motivate the research and the development of techniques that help developers apply an already performed optimizations at other code locations, possibly along the lines of existing work [26, 27, 3].

2.5 Practical Impact

The results of the empirical study can 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.

3 Performance Profiling for Optimizing Orders of Evaluation

The previous section discusses 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 optimization opportunities are challenging due to the dynamic features of the JavaScript language.

Reordering Opportunities In this section, we focus on a recurring and easy to exploit optimization opportunity called *reordering opportunity*. A reordering opportunity optimizes the orders of conditions that are part of a decision made by the program. As an example, Fig. 4 shows an instance of reported reordering optimization in a popular JavaScript project. The code in Fig. 4 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 expressions

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

(a)

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

(b)

Fig. 4 Performance issues from Underscore.string (pull request 471). (**a**) Optimization opportunity. (**b**) Optimized code

(Fig. 4) 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.

Once detected, such opportunities are easy to exploit by reordering the conditions so that the cost of overall evaluation has 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 Fig. 4, we found various other reordering optimizations in real-world code,[3] including several reported by us to the respective developers.[4]

3.1 An Analysis for Detecting Reordering Opportunities

Challenges Even though the basic idea of reordering conditions 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 conditions.* To identify reorderings of conditions that reduce the overall cost of evaluations, we must assess the cost of evaluating individual expressions and the likelihood that an expression 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 conditions.* To reason about all possible reorderings, we must gather cost and likelihood information for all conditions. However, dynamically analyzing all conditions may not be necessary in a normal execution. For example, consider that the first condition in Fig. 4 evaluates to `false`. In this case,

[3] For example, see jQuery pull request #1560.

[4] For example, see Underscore pull request #2496 and Moment pull request #3112.

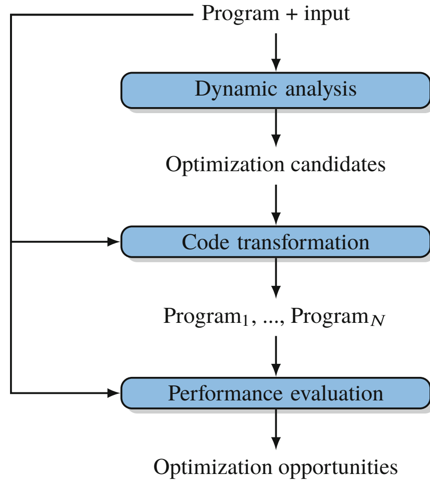

Fig. 5 Overview of *DecisionProf*

the overall value of the expression is determined as `false`, without executing the other two conditions.

- *Side effect-free evaluation of condition.* Evaluating conditions may have side effects, such as modifying a global variable or an object property. Therefore, naively evaluating all conditions, 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.

DecisionProf: An Overview To address the aforementioned challenges, we propose *DecisionProf*, a profiling approach that automatically finds reordering opportunities at runtime and proposes them to the developer. Figure 8 gives an overview of the approach. The input to *DecisionProf* is an executable program and the output is the list of reordering opportunities. The first step of the profiler is a dynamic analysis that identifies optimization candidates. In the second step, for each candidate, the approach applies the optimization via source-to-source transformation. Finally, for the modified version of the program, *DecisionProf* checks whether an optimization reduces the execution time of a program. If and only if the changes lead to statistically significant performance improvement, the approach suggests them as reordering opportunities to the developer (Fig. 5).

3.2 Dynamic Analysis

The main component of *DecisionProf* is the runtime analysis that collects two pieces of information about every dynamic occurrence of a condition: the computational *cost* of evaluating the condition and the *value*, i.e., whether the Boolean

expression evaluates to `true` or `false`. *DecisionProf* gathers these runtime data in two steps. At first, it statically preprocesses the source code of the analyzed program. More details on a preprocessing step can be found in [34]. After collecting runtime data, the approach associates with each condition a cost-value history:

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

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

- When the analysis observes a beginning of a new conditional statement, it pushes the upcoming evaluation onto a stack $evaluations$ of currently evaluated statement.
- When the analysis observes new condition, it pushes the condition that is going to be evaluated onto a stack $conditions$ of currently evaluated conditions. 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 condition in $conditions$. We use the number of executed branching points as a proxy measure for wall clock execution time, avoiding the challenges of reliably measuring short-running code.
- When the analysis observes the end of conditional evaluation, it pops the corresponding condition from $conditions$. Furthermore, the analysis appends (c, v) to h, where h is the cost-value history of the condition as stored in the history map $\mathcal{H}$ of $top(evaluations)$, c is the cost of the current condition evaluation, and v is the Boolean outcome.
- When reaching the end of conditional statements, the analysis pops the corresponding statement from $evaluations$.

The reason for using stacks to represent the currently evaluated conditions 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 approach refines the described analysis in two ways. First, the analysis monitors runtime exceptions that might occur during the evaluation of the Boolean expression. If an exception is thrown, the analysis catches the error, restores the program state, and excludes the expression from further analysis. Such exceptions typically occur because the evaluation of one condition depends on the evaluation of another condition. 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 conditions corresponding to the cases that are evaluated together into a single condition.

Based on cost-value histories obtained through dynamic analysis, *DecisionProf* computes an optimal order of conditions for each executed conditional statement in the program. The computed order is optimal in the sense that it minimizes the overall cost of the analyzed executions.

Table 2 Cost-value histories from executions of Fig. 4

Check	Execution		
	First	Second	Third
`/[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)

To illustrate the algorithm for finding the optimal order of conditions, consider Table 2 that shows a cost-value history gathered from three executions of the logical expression in Fig. 4. 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. Based on these histories, the algorithm computes the optimal cost of the first, innermost logical expression, `/[def]/.test(match[8]) && match[3]`. The costs in the three executions with the original order are 4, 4, and 3. In contrast, the costs when swapping the conditions are 4, 1, and 1. That is, swapping the subexpressions reduces the overall cost. Therefore, the optimal order for the first subexpression is `math[3] && /[def]/.test(match[8])`. Next, the algorithm moves up in the expression tree and optimizes the order of `math[3] && /[def]/.test(match[8])` and `arg`. Comparing their costs shows that swapping these subexpressions is not beneficial, so the algorithm computes the history of the subexpression, and finally it returns `math[3] && /[def]/.test(match[8]) && arg` as the optimized expression.

3.3 Experimental 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. 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 [17, 5, 10] is further discussed in Sect. 5.

Table 3 illustrates the libraries and benchmarks used in the evaluation. In total, *DecisionProf* detects 52 reordering opportunities. The column "Optimizations" in Table 3 shows how many optimizations the approach suggests in each project and function-level performance improvements after applying these optimizations. To the best of our knowledge, none of the optimizations detected by *DecisionProf* have been previously reported. Furthermore, after manually inspecting all suggested optimizations, we find that all of them are semantics-preserving, i.e., the approach has no false positives in our evaluation. Further details on detected opportunities, examples, and their performance impact can be found in [34].

Table 3 Projects used for the evaluation of *DecisionProf*

Project	Tests	LoC	Optimizations	Perf. improvements (%)
Libraries				
Underscore	161	1110	2	3.7–14
Underscore.string	56	905	1	3–5.8
Moment	441	2689	1	3–14.6
Minimist	50	201	1	4.2–6.5
Semver	28	863	5	3.5–10.6
Marked	51	928	1	3–4.4
EJS	72	549	2	5.6–6.7
Cheerio	567	1396	9	6.2–40
Validator	90	1657	3	3–10.9
Total	1516	10, 928	23	
Benchmarks				
float-m		3972	3	2.5
crypto-aes		295	3	5.2
deltablue		483	2	6.5
gbemu		9481	18	5.8
Total		14, 231	26	

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.

4 Cross-Language Optimizations in Big Data Systems

Sections 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 section, 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 [7], a language for big data processing queries that combines SQL-like declarative language with C# expressions.

To demonstrate the effectiveness of method inlining, Fig. 6 illustrates two semantically equivalent SCOPE programs that interleave relational logic with C# expressions. Figure 6a shows the situation where the user implements the predicate

Fig. 6 Examples of SCOPE programs. (**a**) Predicate invisible to optimizer. (**b**) Predicate visible to optimizer

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

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

(a)

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

(b)

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, Fig. 6 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` is the column 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 *intrinsic*. 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 .NET virtual machine.

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

4.1 Performance Issues in SCOPE Language

SCOPE [7] 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*). Each operator,

however, must execute either entirely in C# or in 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. But when a script contains a C# expression that cannot be converted to a C++ function, such as in Fig. 6a, the .NET runtime must be started and each row in the input table must be converted to a C# representation.

Data conversions to and from .NET runtime poses a significant cost in the overall system. To alleviate some of these inefficiencies, 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*. Wherever possible, the SCOPE compiler emits calls to the (C++) intrinsics instead of C# functions. However, the optimization opportunity presented in Fig. 6 is outside the scope of SCOPE compiler: user-written functions are compiled as a black box: no analysis or optimization is performed at this level.

4.2 Static Analysis to Find Method Inlining Opportunities

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 [7]. 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 a million jobs every day.

Finding optimization opportunities in such a large number of diverse jobs is a challenging problem. We can hope to find interesting conclusions only if our analysis infrastructure is scalable. To achieve this, we analyze the following artifacts that are produced after the execution of each SCOPE program:

- *Job Algebra* The job algebra is a graph representation of the job execution plan. Each vertex in a graph contains operators that run either inside native (C++) or .NET runtime.
- *Runtime Statistics* The runtime statistics provide information on the CPU time for every job vertex and every operator inside the vertex.
- *Generated Code* The SCOPE compiler generates both C# and C++ codes 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 artifact containing the C# code includes implementations of non-native operators and user-written classes and functions defined inside the script.

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. If in the class 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 non-

native 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*, and *user-written operators*.

To find method inlining opportunities, we are particularly interested in the second category. Among user-written functions, we find inlineable ones as per the following definition:

Definition 2 (Inlineable Method) Method m is *inlineable* if it has the following properties:

- It contains only calls to *intrinsics* methods
- 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).

By optimizing inlineable methods, the SCOPE compiler is now able to run the operator contains the method completely in native runtime. In the next section, we further discuss the empirical results of performance improvements due to method inlining opportunities. The complete description of our analysis infrastructure can be found in [33].

4.3 Experimental Evaluation

To understand the potential of method inlining opportunities, we analyze over 3,000,000 SCOPE jobs over a period of six days that run on five data centers at Microsoft. To quantify the amount of CPU time that can be optimized by applying method inlinings, we consider job vertices that have as the only source of managed code an *inlineable* method. By optimizing such a method, we expect an entire vertex to run as native code, which should significantly improve the vertex execution time. Figure 7 shows the proportion of CPU time of optimizable vertices relative to data center time and total time spent in .NET runtime. We observe that with the current list of intrinsics, we can optimize a relatively small proportion of both, data center time and non-native 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 or cosmos15, where we can optimize up to 0.15% of data center time. However, taking into account the scale of big data processing at Microsoft, this percentage amounts to almost 40,000 h of optimizable time.

The crucial observation is that the results illustrate only the time in data centers that can be affected by inlining method calls. To measure the actual performance improvement, it is necessary to rerun every optimized job.

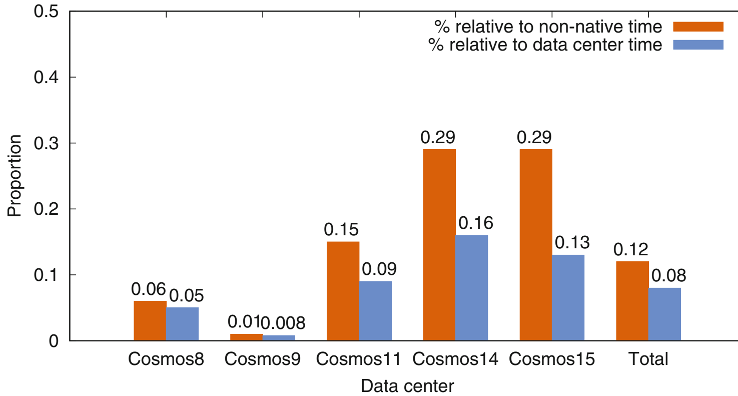

Fig. 7 Execution time affected by method inlining

Table 4 Summary of case studies. The reported changes are percentage of 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%

4.3.1 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 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 are not available, we had to contact the job owners and ask them to rerun the job with a manually inlined version of their script. We were able to have 6 jobs rerun by their owners, categorized by their total CPU time: short, medium, and long.

In total, we looked at 6 rerun jobs, summarized in Table 4. For one job (D), the optimization did not trigger C++ translation of an inlined operator because the operator called to a non-intrinsic 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 this section.

For jobs A and B, we were able to perform 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 user-written functions. In both cases, inlining the function resulted in the entire vertex being executed in C++. The values are normalized by the average of the unoptimized execution times; the optimized version of the job A saves approximately 60% of the execution time. In similar fashion, we find that the normalized vertex CPU time in Job B does not show any consistent improvement. 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 in line 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.

By presenting six case studies of big data processing tasks, we show that method inlinig is a promising optimization strategy for triggering more generation of native code in SCOPE programs, which yields significant performance improvements.

5 Test Generation of Higher-Order Functions in Dynamic Languages

In Sect. 3, we present *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 [29, 30], symbolic execution [21, 6], concolic execution [14, 37], bounded exhaustive testing [4], evolutionary test generation [12], UI-level test generation [25, 28, 34], and concurrency testing [31, 32].

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 [40]. 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 [8].

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.

In this section, we give a brief overview of *LambdaTester*, a novel framework for testing higher-order functions in dynamic languages. The complete details of the framework and proposed solution can be found in [36].

5.1 Overview of the Framework

In the *LambdaTester* framework, test generation proceeds in two phases. The *discovery phase* is concerned with discovering, for a given method 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. The test generation phase uses a form of feedback-directed, random testing [29] to incrementally extend and execute tests. We augment feedback-directed, random testing with four techniques to create callback arguments. 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.

The basic ingredient of generated tests is method calls:

Definition 3 (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, $var_{arg1}, \ldots, var_{argk}$ 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.

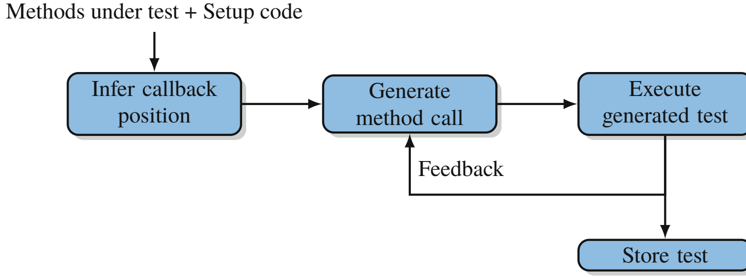

Fig. 8 Overview of *LambdaTester*

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

Definition 4 (Test) A test *test* is a sequence $(setup, c_i, \ldots, c_n)$, where *setup* is the setup code and $c_i, \ldots, c_n$ are generated method calls.

Figure 8 illustrates the process of test generation. For each method under test, the approach attempts to infer the positions of callback arguments. Afterward, 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 as an input to the dynamic analysis or for bug finding [36].

5.2 *Callback Generation Approaches*

The key idea of *LambdaTester* is a feedback-directed test generation with a novel generation of callback inputs. Our framework currently supports four techniques for generating callback functions, which we present below.

Empty Callbacks The most simple approach for creating callbacks is to simply create an empty function that does not perform any computation and does not explicitly return any value. Figure 9 gives an example of an empty callback.

Callbacks by QuickCheck QuickCheck [8] 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. Figure 9 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

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

(a)

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

(b)

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

(c)

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

(d)

Fig. 9 Examples of generated callbacks. (**a**) Empty callback ("Cb-Empty"). (**b**) Callback generated by QuickCheck ("Cb-QuickCheck"). (**c**) Callback mined from existing code ("Cb-Mined"). (**d**) Callback generated based on dynamically analyzing the method under test ("Cb-Writes")

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.

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 (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.

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

Based on detected memory locations, *LambdaTester* generates a callback body that interacts with the function under test. To this end, the approach first infers how many arguments a callback function receives. 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 9 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.

As further discussed in [36], all callback generation techniques are more effective in finding programming errors than state-of-the-art test generation approaches that do not consider the generation of function inputs. Moreover, among proposed techniques, generating callbacks that modify program state in non-obvious ways is more effective in triggering non-trivial executions than other callback generation techniques.

6 Conclusions

In this chapter, 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 (Sect. 2), analyses to find reordering opportunities (Sect. 3) and method inlining opportunities (Sect. 4), and a novel test generation technique for higher-order functions in dynamic languages (Sect. 5). These approaches aim to reduce manual effort by suggesting only beneficial optimization opportunities that are easy to understand and applicable across multiple projects.

6.1 Summary of Contributions

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. 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.

6.2 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. 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 for specific patterns of performance issues. As already shown in Sects. 3 and 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 are challenging tasks that often require significant human effort. Even though we studied a significant number of performance problems and drew interesting conclusions in Chap. 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 We propose approaches that address two different types of optimizations: reordering opportunities and method inlining. However, in Sect. 2, we discuss 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.

References

1. W. Ahn, J. Choi, T. Shull, M. J. Garzarán, and J. Torrellas. Improving JavaScript performance by deconstructing the type system. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 496–507, 2014.
2. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers. Principles, Techniques and Tools*. Addison Wesley, 1986.
3. M. Boshernitsan, S. L. Graham, and M. A. Hearst. Aligning development tools with the way programmers think about code changes. In *CHI*, pages 567–576, 2007.
4. C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on Java predicates. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 123–133, 2002.
5. J. Burnim, S. Juvekar, and K. Sen. WISE: Automated test generation for worst-case complexity. In *ICSE*, pages 463–473. IEEE, 2009.
6. C. Cadar, D. Dunbar, and D. 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.
7. R. Chaiken, B. Jenkins, P. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. Scope: Easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.*, 1(2):1265–1276, Aug. 2008.
8. K. Claessen and J. Hughes. Quickcheck: A lightweight tool for random testing of Haskell programs. *SIGPLAN Not.*, 46(4):53–64, May 2011.
9. I. Costa, P. Alves, H. N. Santos, and F. M. Q. Pereira. Just-in-time value specialization. In *CGO*, pages 1–11, 2013.
10. M. Dhok and M. K. Ramanathan. Directed test generation to detect loop inefficiencies. In *FSE*, 2016.
11. R. R. Dumke, C. Rautenstrauch, A. Schmietendorf, and A. Scholz, editors. *Performance Engineering, State of the Art and Current Trends*, London, UK, UK, 2001. Springer-Verlag.
12. G. Fraser and A. 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.
13. A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based just-in-time type specialization for dynamic languages. In *PLDI*, pages 465–478, 2009.

14. P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 213–223. ACM, 2005.
15. L. Gong, M. Pradel, and K. Sen. JITProf: Pinpointing JIT-unfriendly JavaScript code. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 357–368, 2015.
16. S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A call graph execution profiler. In *SIGPLAN Symposium on Compiler Construction*, pages 120–126. ACM, 1982.
17. M. Grechanik, C. Fu, and Q. Xie. Automatically finding performance problems with feedback-directed learning software testing. In *International Conference on Software Engineering (ICSE)*, pages 156–166, 2012.
18. B. Hackett and S. Guo. Fast and precise hybrid type inference for JavaScript. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 239–250. ACM, 2012.
19. A. Hartono, B. Norris, and P. Sadayappan. Annotation-based empirical performance tuning using orio. *2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–11, 2009.
20. S. H. Jensen, M. Sridharan, K. Sen, and S. 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.
21. J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.
22. D. E. Knuth. Computer programming as an art. *Commun. ACM*, 17(12):667–673, Dec. 1974.
23. F. Logozzo and H. Venter. RATA: Rapid atomic type analysis by abstract interpretation—application to JavaScript optimization. In *CC*, pages 66–83, 2010.
24. T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, Dec. 1976.
25. A. M. Memon. An event-flow model of GUI-based applications for testing. *Softw. Test., Verif. Reliab.*, pages 137–157, 2007.
26. N. Meng, M. Kim, and K. S. McKinley. Systematic editing: generating program transformations from an example. In *PLDI*, pages 329–342, 2011.
27. N. Meng, M. Kim, and K. S. McKinley. Lase: locating and applying systematic edits by learning from examples. In *ICSE*, pages 502–511, 2013.
28. A. Mesbah and A. van Deursen. Invariant-based automatic testing of Ajax user interfaces. In *ICSE*, pages 210–220, 2009.
29. C. Pacheco and M. 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.
30. C. Pacheco, S. K. Lahiri, and T. Ball. Finding errors in .NET with feedback-directed random testing. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 87–96. ACM, 2008.
31. M. Pradel and T. R. Gross. Fully automatic and precise detection of thread safety violations. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 521–530, 2012.
32. M. Samak and M. K. Ramanathan. Multithreaded test synthesis for deadlock detection. In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 473–489, 2014.
33. M. Selakovic, M. Barnett, M. Musuvathi, and T. 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.

34. M. Selakovic, T. Glaser, and M. Pradel. An actionable performance profiler for optimizing the order of evaluations. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 170–180, 2017.
35. M. Selakovic and M. Pradel. Performance issues and optimizations in JavaScript: An empirical study. In *International Conference on Software Engineering (ICSE)*, pages 61–72, 2016.
36. M. Selakovic, M. Pradel, R. Karim, and F. Tip. Test generation for higher-order functions in dynamic languages. *Proceedings of the ACM on Programming Languages*, 2:1–27, 10 2018.
37. K. Sen, D. Marinov, and G. 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.
38. L. D. Toffola, M. Pradel, and T. R. Gross. Performance problems you can fix: A dynamic analysis of memoization opportunities. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 607–622, 2015.
39. G. H. Xu, N. Mitchell, M. Arnold, A. Rountev, E. Schonberg, and G. Sevitsky. Finding low-utility data structures. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 174–186, 2010.
40. H. Zhong, L. Zhang, and S. 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.

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.