An Actionable Performance Profiler for Optimizing the Order of Evaluations

Marija Selakovic
TU Darmstadt
Germany
m.selakovic89@gmail.com

Thomas Glaser
TU Darmstadt
Germany
thomas.glaser@stud.tu-darmstadt.de

Michael Pradel
TU Darmstadt
Germany
michael@binaervarianz.de

ABSTRACT

The efficiency of programs often can be improved by applying relatively simple changes. To find such optimization opportunities, developers either rely on manual performance tuning, which is time-consuming and requires expert knowledge, or on traditional profilers, which show where resources are spent but not how to optimize the program. This paper presents a profiler that provides actionable advice, by not only finding optimization opportunities but by also suggesting code transformations that exploit them. Specifically, we focus on optimization opportunities related to the order of evaluating subexpressions that are part of a decision made by the program. To help developers find such reordering opportunities, we present DecisionProf, a dynamic analysis that automatically identifies the optimal order, for a given input, of checks in logical expressions and in switch statements. The key idea is to assess the computational costs of all possible orders, to find the optimal order, and to suggest a code transformation to the developer only if reordering yields a statistically significant performance improvement. Applying DecisionProf to 43 real-world JavaScript projects reveals 52 beneficial reordering opportunities. Optimizing the code as proposed by DecisionProf reduces the execution time of individual functions between 2.5% and 59%, and leads to statistically significant application-level performance improvements that range between 2.5% and 6.5%.

CCS CONCEPTS

•Software and its engineering →Software maintenance tools; Software testing and debugging;

KEYWORDS

JavaScript, Dynamic analysis, Performance, Profiler

ACM Reference format:
Marija Selakovic, Thomas Glaser, and Michael Pradel. 2017. An Actionable Performance Profiler for Optimizing the Order of Evaluations. In *Proceedings of 26th International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 2017 (ISSTA'17),* 11 pages.
DOI: 10.1145/3092703.3092716

1 INTRODUCTION

Optimizing the performance of software is important in various domains, e.g., for achieving high throughput, energy efficiency, responsiveness, and user satisfaction. Even relatively small performance improvements (measured in milliseconds) in applications such as web sites or search engines can positively influence the page traffic and user experience.

However, detecting and exploiting optimization opportunities is a cumbersome task that often requires significant human effort. Fortunately, many programs suffer from performance bottlenecks where a relatively simple source code change can make the program significantly more efficient [19, 34]. The challenge is to find and exploit such easy to use optimization opportunities.

Currently, there are three kinds of approaches to optimize performance. First, compiler optimizations automatically transform a program to a semantically equivalent yet more efficient program. Despite being very powerful for particular classes of optimizations, many other promising optimization opportunities are beyond the capabilities of a typical compiler. The main reason is that the compiler cannot ensure that a transformation preserves the semantics, a problem that is especially relevant for hard-to-analyze languages, such as JavaScript. Second, to complement compiler optimizations, developers use CPU [12] and memory profilers [18] to identify those code locations that use most resources. More recent approaches identify performance bottlenecks based on their symptoms, such as memory bloat [42], inefficient loops [29], and JIT unfriendliness [11]. While useful to understand why code is slow, these approaches do not show developers how to optimize the code. Finally, developers often fall back on manual performance tuning, which can be effective but is time-consuming and often requires expert knowledge.

This paper presents a novel automated approach to support developers in optimizing their programs, called *actionable performance profiling*. The key idea is to not only pinpoint where and why time is spent, but to also suggest concrete code transformations that speed up the code. A profiler following this idea is actionable in the sense that the developer can take immediate action based on the profiler's suggestions, by deciding whether to apply a suggested transformation. The reason why the profiler does not fully automatically optimize the program, as a compiler would, is that it does not guarantee to preserve the semantics, enabling it to address optimizations out-of-reach for compilers.

As motivating examples, Figure 1 shows two non-trivial to detect but easy to exploit optimization opportunities in popular JavaScript projects. The code in Figure 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 higher or equal to zero. This code can be optimized by swapping the first two checks (Figure 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,

ISSTA'17, Santa Barbara, CA, USA

DOI: 10.1145/3092703.3092716

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

Figures 1c, shows a performance issue found in Socket.io, a realtime application framework. The code 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 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. We call such a situation a *reordering opportunity*. 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 1, we found various other reordering optimizations in real-world code[1], including several reported by us to the respective developers.[2] Unfortunately, manually searching reordering opportunities is difficult and existing automated techniques do not support developers in detecting and exploiting them.

This paper presents DecisionProf, a profiler for detecting reordering opportunities. 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 evaluating those checks first 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 of 52 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 even yield application-level performance improvements, which range between 2.5% and 6.5%.

In summary, this paper contributes the following:

- *Actionable performance profiling.* We identify a new class of profilers that not only reports where and why resources are wasted but also how to optimize the code.
- *Profiling for reordering opportunities.* We present the first profiler that detects inefficiently ordered subexpressions of logical expressions and switch statements, and that suggests simple refactorings to optimize the code.
- *Empirical evidence.* We implement the approach into a practical tool and show that applying the suggested optimizations leads to performance improvements in widely used JavaScript projects and popular benchmarks.

2 PROBLEM STATEMENT

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

[1] E.g., see jQuery pull request #1560.
[2] E.g., see Underscore pull request #2496 and Moment pull request #3112.

2.1 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 1a shows a decision that consists of three checks that are combined into a conjunction. Figure 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 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. Which order of evaluation is most efficient 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.

2.2 Reordering Opportunities

The problem addressed in this paper is finding inefficiently ordered evaluations, called *reordering opportunities*, and suggesting a more efficient order to the developer. We consider arbitrarily complex decision, e.g., decisions that involve nested binary logical expressions. The goal of DecisionProf is to find a minimal order of the checks involved in a decision. Minimal 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 1 are reordering opportunities.

2.3 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 1a evaluates to `false`. In this case, the overall value of the expression is determined as `false`, without executing the other two checks.
- *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.

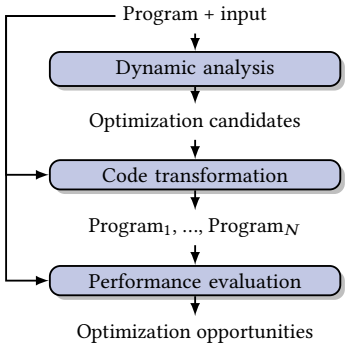

Figure 2: Overview of the approach.

3 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 2 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.

3.1 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 two steps. At first, it statically pre-processes the

```
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: Preprocessed logical expression from Figure 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 4: Preprocessed switch statement from Figure 1c.

source code of the analyzed program. Then, it dynamically analyzes the pre-processed program to collect runtime data.

3.1.1 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 4. Figure 3 shows the pre-processed code for the logical expression of Figure 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 4 shows the pre-processed code for the switch statement of Figure 1c.

3.1.2 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:

Table 1: Cost-value histories from executions of Figure 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)

Definition 3.1 (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 4).

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 1 shows a cost-value history gathered from three executions of the logical expression in Figure 1a. For example, when

[3] Section 6.4 evaluates the accuracy of the proxy metric.

Algorithm 1 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 H then
        e.left ← optimize(e.left)
    if e.right is not in H then
        e.right ← optimize(e.right)
    e' ← findOptimalOrder(e)
    return e'

function findOptimalOrder(e)
    c_orig ← computeCost(e.left, e.right, e.operator)
    c_swap ← computeCost(e.right, e.left, e.operator)
    if c_orig ≤ c_swap then
        e' ← e
    else
        e' ← e with left and right swapped
    h_e' ← cost-value history of optimized expression e'
    Add e ↦ h_e' to H
    return e'

function computeCost(e_left, e_right, op)
    h_left → H(e_left)
    h_right → H(e_right)
    c ← 1
    foreach i in 0 to length(h_left) do
        cv_left ← h_left[i]
        cv_right ← h_right[i]
        if op is && then
            if cv_left.value is true then
                c ← c + cv_left.cost + cv_right.cost
            else
                c ← c + cv_left.cost
        else if op is || then
            if cv_left.value is false then
                c ← c + cv_left.cost + cv_right.cost
            else
                c ← c + cv_left.cost
    return c
```

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 Finding Optimization Candidates

Based on cost-value histories obtained through dynamic analysis, DecisionProf computes a minimal 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.

3.2.1 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 expressions, the history map contains a cost-value history h gathered during the dynamic analysis.

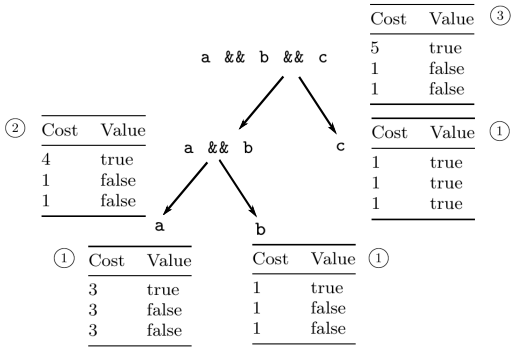

Figure 5: Example of finding the optimal order of checks.

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 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 1a and the cost-value history in Table 1. Figure 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.

3.2.2 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 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 cost-value history of 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.

3.3 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 1.

3.4 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 6.1). To determine whether there is a statistically significant difference, the approach applies the t-test on the collected measurements with a confidence level of 90%. Finally, DecisionProf suggests a change as a reordering opportunity if the change improves the execution time of a program.

3.5 Pruning False Positives

The approach described so far assumes that all checks are commutative, i.e., can be swapped without affecting the program's semantics. However, this assumption may not hold, e.g., because evaluating one check has a side effect that is a pre-condition for another check. One way to deal with this challenge would be to conservatively overapproximate all side effects and dependences between checks and to suggest reordering opportunities only if they are guaranteed to preserve the program's semantics. Unfortunately, this conservative approach is not practical, at least for a hard-to-analyze language, such as JavaScript (Section 6.5). Instead, DecisionProf relies on a set of heuristics to prune false positives, i.e., reordering opportunities that are spurious because the reordering would affect the program's semantics. We present these heuristics in the following.

Static pruning of non-commutative code idioms. DecisionProf ignores decisions that matches particular code patterns, which are likely non-commutative logical expressions. We identify two such patterns for JavaScript:

- A common idiom is to check whether a variable is defined before accessing its properties, e.g., `if (x && x.a)`. Changing the order of checks in such an `if` statement would lead to a runtime error when `x` is `undefined` or `null`. To avoid such false positives, DecisionProf excludes any logical expressions where the left-hand side checks a reference for `undefined` or `null` and where the right-hand side uses this reference.
- A common idiom for initializing variables is a logical expression of the form `var x = y || "abc"`, where the literal `"abc"` is the default value in case `y` is not defined. Reordering such a logical expressions would always assign the default value, i.e., break the program's semantics. Instead, DecisionProf excludes all `||`-expressions where the right-hand side is a literal.

Dynamic pruning of dependent checks. The checks in a logical expressions may be non-commutative because the side effects of one check influences the other check. For example, in `a() && b()`, check `a()` may write to a variable that check `b()` reads. Finding such dependences statically is challenging in JavaScript due to its dynamic features. Instead, DecisionProf identifies dependences at runtime and prunes interdependent checks. To this end, the analysis tracks reads and writes to variables and object properties performed by each check in a logical expression or switch statement. Based on these reads and writes, DecisionProf prunes decisions where two or more checks have read-write, write-read, or write-write conflicts.

Testing-based validation of optimized programs. As a best-effort approach to automatically validate whether optimizing a reordering candidate preserves the program's semantics, DecisionProf executes the optimized program. If the execution terminates without errors, an opportunity is reported to the developer.

4 SAFE CHECK EVALUATION

The profiling part of DecisionProf (Section 3.1) 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*.

```
1  var a = 0;
2  function foo(){
3      a++;
4      var b=1;
5      if (a===1) return true;
6      else return false;
7  }
8  startDecision;
9      startCheck: foo();
10     startCheck: someExpression;
11 undoSideEffects;
12 if (foo() && someExpression) ...
```

Figure 6: Changes in program behavior due to side effects.

As a motivating example, consider Figure 6. The profiler evaluates `foo()` before executing the `if` statement (line 9), 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 12 would evaluate to `false`, because `a` gets incremented again at line 3.

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.

4.1 Tracking Side Effects

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

Definition 4.1 (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 4.2 (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 [35]):

- 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 6, consider the evaluation of `foo()` at line 9. 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.

4.2 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 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.

5 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 [36]. We believe that our approach is applicable to other languages than JavaScript, e.g., based on existing dynamic analysis tools, such as Valgrind or PIN for x86 programs, and ASM or Soot for Java.

6 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 6.2)
- *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 6.2)
- *How efficient is the approach?* Compared to normal program execution, DecisionProf imposes a profiling overhead between 3x and 1,210x (median: 116). (Section 6.3)
- *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 6.4)
- *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 6.5)

Table 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	

6.1 Experimental Setup

Subject Programs and Inputs. We use 9 JavaScript libraries, listed in the upper part of Table 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 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. Tuning the performance at the unit-level is similar to the wide use of microbenchmarks.[6] The benchmarks come with inputs to execute the programs. 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 [3, 7, 13] is beyond the scope of this paper.

Performance Measurements. Reliably measuring the performance of executions is non-trivial [25]. DecisionProf follows the methodology by Georges et al. [9], both as part of the approach (Section 3.4) 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.

6.2 Detected Reordering Opportunities

In total, DecisionProf detects 52 reordering opportunities. The column "Opportunities" in Table 2 shows how many optimizations

[4] http://esprima.org/
[5] https://github.com/estools/escodegen
[6] E.g., see https://benchmarkjs.com/ and https://jsperf.com/ for popular microbenchmarking tools.

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 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 by DecisionProf.[7] 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 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 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.

6.3 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 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 [6, 11, 29, 42]. 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 intervention and reports actionable suggestions, we consider the computational effort to be acceptable for developers.

6.4 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 723 tests. The correlation coefficient for benchmarks and unit tests is 0.92 and 0.98, 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.

6.5 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.

7 RELATED WORK

Detecting performance problems. Studies show that performance problems occur frequently in practice [19], especially for JavaScript [34], and that they account for a non-negligible amount of developer time [46]. Various approaches to detect performance problems have been proposed, including CPU-time profiling [12],

[7] Pull request #2496 of Underscore.

Table 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%			

algorithms to discover parallelizable computations [33], a profiler to detect memoization opportunities [6], an analysis to find repeated executions of similar behavior in loops [29], an analysis to detect unnecessary use of duplicate objects [24], analyses for detecting unnecessarily high memory consumption [41–43], profilers for UI-related performance problems [20, 22, 40], an approach for predicting scalability-related performance problems [28], a sound static analysis to find redundant traversals [30], analyses that empirically estimate the computational complexity [10, 47], and a profiler to detect code locations that are hard to optimize for JavaScript engines [11]. Our work differs from these by considering a different kind of performance problem and by suggesting concrete optimizations to the developer.

Performance-guided test generation. Some test generators search performance problems, e.g., by triggering the worst-case complexity of a program [4], generating sequences of UI events to expose responsiveness problems [32], and automated performance regression testing [31]. Combining DecisionProf with automatic test generation may further improve the effectiveness of our approach.

Understanding performance problems. Approaches for diagnosing performance bottlenecks include a profiler to diagnose idle times [2], mining of stacktraces [15] and execution traces [45], a systematic search for performance anti-patterns [39], and statistical debugging [37]. A recent survey discusses more approaches [17]. In contrast to all of these, DecisionProf addresses the problem of finding reordering opportunities, which are missed by the existing approaches.

Fixing performance problems and automated program repair. MemoizeIt [6] identifies methods that can benefit from memoization and provide hints on how to implement the memoization. Caramel [27] suggests code transformation that avoid wasting loop iterations. SyncProf [44] detects, localizes, and optimizes synchronization bottlenecks. Orthogonal to performance problems, recent work focuses on automatically repairing correctness bugs [21, 26, 38]. DecisionProf differs by addressing a novel class of performance problems and by automatically assessing whether an optimization opportunities improves performance.

Just-in-time compilation. Modern JavaScript engines employ *just-in-time* compilers to efficiently execute JavaScript code. Recent advances include optimizing code into type-specialized code [8, 14, 23], an improved object representation [1], and specializing functions based on previously observed parameters [5]. DecisionProf proposes optimizations that are not addressed by today's JIT compilers, as checked in the performance evaluation step of our approach.

Transactional memory. DecisionProf's safe check evaluation is inspired by transactional memory [16]. Instead of simplifying concurrent programs, we use the idea to evaluate expressions without permanent effects on the program state.

8 CONCLUSION

This paper presents 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 beneficial and actionable: They are beneficial 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.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their comments. Our work has been supported by the DFG within ConcSys, and by the BMBF and the HMWK within CRISP.

REFERENCES

[1] Wonsun Ahn, Jiho Choi, Thomas Shull, Mara J. Garzarn, and Josep Torrellas. 2014. Improving JavaScript Performance by Deconstructing the Type System. In *Conference on Programming Language Design and Implementation (PLDI)*. 496–507.

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

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

[4] Jacob Burnim and Koushik Sen. 2009. Asserting and checking determinism for multithreaded programs. In *European Software Engineering Conference and Symposium on Foundations of Software Engineering (ESEC/FSE)*. ACM, 3–12.

[5] Igor Costa, Pricles Alves, Henrique Nazare Santos, and Fernando Magno Quinto Pereira. 2013. Just-in-time value specialization. In *CGO*. 1–11.

[6] Luca Della Toffola, Michael Pradel, and Thomas R. Gross. 2015. Performance Problems You Can Fix: A Dynamic Analysis of Memoization Opportunities. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. ACM, New York, NY, USA, 607–622. DOI: http://dx.doi.org/10.1145/2814270.2814290

[7] Monika Dhok and Murali Krishna Ramanathan. 2016. Directed Test Generation to Detect Loop Inefficiencies. In *FSE*.

[8] 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. 2009. Trace-based just-in-time type specialization for dynamic languages.. In *PLDI*. 465–478.

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

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

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

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

[13] Mark Grechanik, Chen Fu, and Qing Xie. 2012. Automatically Finding Performance Problems with Feedback-Directed Learning Software Testing. In *International Conference on Software Engineering (ICSE)*. 156–166.

[14] Brian Hackett and Shu-yu Guo. 2012. Fast and Precise Hybrid Type Inference for JavaScript. In *Conference on Programming Language Design and Implementation (PLDI)*. ACM, 239–250.

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

[16] Maurice Herlihy and J. Eliot B. Moss. 1993. Transactional Memory: Architectural Support for Lock-free Data Structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA '93)*. ACM, New York, NY, USA, 289–300. DOI: http://dx.doi.org/10.1145/165123.165164

[17] Olumuyiwa Ibidunmoye, Francisco Hernández-Rodriguez, and Erik Elmroth. 2015. Performance Anomaly Detection and Bottleneck Identification. *ACM Comput. Surv.* 48, 1 (2015), 4.

[18] Simon Holm Jensen, Manu Sridharan, Koushik Sen, and Satish Chandra. 2015. 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*. 345–356.

[19] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and Detecting Real-world Performance Bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, New York, NY, USA, 77–88. DOI: http://dx.doi.org/10.1145/2254064.2254075

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

[21] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches.. In *International Conference on Software Engineering (ICSE)*. 802–811.

[22] Yepang Liu, Chang Xu, and S.C. Cheung. 2014. Characterizing and Detecting Performance Bugs for Smartphone Applications. In *ICSE*. 1013–1024.

[23] Francesco Logozzo and Herman Venter. 2010. RATA: Rapid Atomic Type Analysis by Abstract Interpretation—Application to JavaScript Optimization. In *CC*. 66–83.

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

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

[26] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: program repair via semantic analysis. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*. 772–781.

[27] Adrian Nistor, Po-Chun Chang, Cosmin Radoi, and Shan Lu. 2015. Caramel: Detecting and Fixing Performance Problems That Have Non-intrusive Fixes. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 902–912. http://dl.acm.org/citation.cfm?id=2818754.2818863

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

[29] Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. 2013. Toddler: Detecting Performance Problems via Similar Memory-Access Patterns. In *International Conference on Software Engineering (ICSE)*. 562–571.

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

[31] Michael Pradel, Markus Huggler, and Thomas R. Gross. 2014. Performance Regression Testing of Concurrent Classes. In *International Symposium on Software Testing and Analysis (ISSTA)*. 13–25.

[32] Michael Pradel, Parker Schuh, George Necula, and Koushik Sen. 2014. EventBreak: Analyzing the Responsiveness of User Interfaces through Performance-Guided Test Generation. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

[33] Martin C. Rinard and Pedro C. Diniz. 1997. Commutativity Analysis: A New Analysis Technique for Parallelizing Compilers. *ACM Trans. Program. Lang. Syst.* 19, 6 (Nov. 1997), 942–991. DOI: http://dx.doi.org/10.1145/267959.269969

[34] Marija Selakovic and Michael Pradel. 2016. Performance Issues and Optimizations in JavaScript: An Empirical Study. In *International Conference on Software Engineering (ICSE)*.

[35] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. 2013. Jalangi: A Selective Record-Replay and Dynamic Analysis Framework for JavaScript. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 488–498.

[36] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. 2013. 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)*. ACM, New York, NY, USA, 488–498. DOI: http://dx.doi.org/10.1145/2491411.2491447

[37] Linhai Song and Shan Lu. 2014. Statistical debugging for real-world performance problems. In *Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*. ACM, 561–578.

[38] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically Finding Patches Using Genetic Programming. In *International Conference on Software Engineering (ICSE)*. 363–374.

[39] Alexander Wert, Jens Happe, and Lucia Happe. 2013. Supporting Swift Reaction: Automatically Uncovering Performance Problems by Systematic Experiments. In *International Conference on Software Engineering (ICSE)*. 552–561.

[40] Xiao Xiao, Shi Han, Charles Zhang, and Dongmei Zhang. 2015. Uncovering JavaScript Performance Code Smells Relevant to Type Mutations. (November 2015), 335–355. https://www.microsoft.com/en-us/research/publication/uncovering-javascript-performance-code-smells-relevant-type-mutations/

[41] Guoqing Xu and Atanas Rountev. 2010. Detecting Inefficiently-used Containers to Avoid Bloat. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10)*. ACM, New York, NY, USA, 160–173. DOI: http://dx.doi.org/10.1145/1806596.1806616

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

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

[44] Tingting Yu and Michael Pradel. 2016. SyncProf: Detecting, Localizing, and Optimizing Synchronization Bottlenecks. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*. ACM, New York, NY, USA, 389–400. DOI: http://dx.doi.org/10.1145/2931037.2931070

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

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

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