Optimization Coaching for JavaScript

Vincent St-Amour[1] and Shu-yu Guo[2]

1 **PLT @ Northeastern University**
Boston, Massachusetts, USA
stamourv@ccs.neu.edu
2 **Mozilla Research**
San Francisco, California, USA
shu@mozilla.com

Abstract

The performance of dynamic object-oriented programming languages such as JavaScript depends heavily on highly optimizing just-in-time compilers. Such compilers, like all compilers, can silently fall back to generating conservative, low-performance code during optimization. As a result, programmers may inadvertently cause performance issues on users' systems by making seemingly inoffensive changes to programs. This paper shows how to solve the problem of silent optimization failures. It specifically explains how to create a so-called *optimization coach* for an object-oriented just-in-time-compiled programming language. The development and evaluation build on the SpiderMonkey JavaScript engine, but the results should generalize to a variety of similar platforms.

1998 ACM Subject Classification D.2.3 [*Software Engineering*]: Coding Tools and Techniques; D.3.4 [*Programming Languages*]: Processors—Compilers

Keywords and phrases Optimization Coaching, JavaScript, Performance Tools

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2015.999

1 Optimization Coaching for the Modern World

An optimization coach [22] opens a dialog between optimizing compilers and programmers. It thus allows the latter to take full advantage of the optimization process. Specifically, coaches provide programmers with actionable recommendations of changes to their programs to trigger additional optimizations. Notably, the changes may not preserve the semantics of the program.

Our experiences with a prototype optimization coach for Racket show promising results. This prototype exploits Racket's [9] simple ahead-of-time byte-compiler,[1] which performs basic optimizations. The general idea of optimization coaching ought to apply beyond languages with functional cores and simple compilers.

Unsurprisingly, scaling coaching to object-oriented languages with advanced compilers presents challenges. An object-oriented programming style gives rise to *non-local optimization failures*, that is, the compiler may fail to optimize an operation in one part of the program because of properties of a different part of the program. Advanced just-in-time (JIT) compilers introduce a *temporal dimension* to the compilation and optimization process, that is, the compiler may compile the same piece of code multiple times, potentially performing different optimizations each time. Advanced compilers may also apply *optimization tactics* when

[1] Racket also includes a just-in-time code generator that does not perform many optimizations.

29th European Conference on Object-Oriented Programming (ECOOP'15).
Editor: John Tang Boyland; pp. 999–1023

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

optimizing programs, that is, they use batteries of related and complementary optimizations when compiling some operations.

This paper presents new ideas on optimization coaching that allow it to scale to dynamic object-oriented languages with state-of-the-art JIT compilers. Our prototype optimization coach works with the SpiderMonkey[2] JavaScript [7] engine, which is included in the Firefox[3] web browser.

In this paper, we

- describe optimization coaching techniques designed for object-oriented languages with state-of-the-art compilers
- present an evaluation of the recommendations provided by our optimization coach for SpiderMonkey.

The rest of the paper is organized as follows. Sections 2 and 3 provide background on optimization coaching and on the SpiderMonkey JavaScript engine. Section 4 describes the optimizations that our prototype supports. Section 5 sketches out its architecture. Section 6 outlines the challenges of coaching in an object-oriented setting and describes our solutions, and section 7 does likewise for the challenges posed by JIT compilation and optimization tactics. Section 8 presents coaching techniques that ultimately were unsuccessful. We then present evaluation results in section 9, compare our approach to related work and conclude.

Prototype Our prototype optimization coach is available in source form.[4] It depends on an instrumented version of SpiderMonkey whose source is also available.[5]

2 Background: Optimization Coaching

Because modern programming languages heavily rely on compiler optimizations for performance, failure to apply certain key optimizations is often the source of performance issues. To diagnose these performance issues, programmers need insight about what happens during the optimization process.

This section first discusses an instance of an optimization failure causing a hard-to-diagnose performance issue. The rest of the section then provides background on how optimization coaching provides assistance in these situations, and introduces some key technical concepts from previous work on coaching.

2.1 A Tale from the Trenches

The Shumway project,[6] an open-source implementation of Adobe Flash in JavaScript, provides an implementation of ActionScript's parametrically polymorphic `Vector` API,[7] which includes a `forEach` method. This method takes a unary kernel function `f` as its argument and calls it once for each element in the vector, passing that element as the argument to `f`. Initially, the Shumway implementors wrote a single implementation of the `forEach` method and used it for all typed variants of `Vector`.

[2] https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey
[3] https://www.mozilla.org/en-US/firefox/
[4] https://github.com/stamourv/jit-coach
[5] https://github.com/stamourv/gecko-dev/tree/profiler-opt-info
[6] https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Shumway
[7] http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/Vector.html

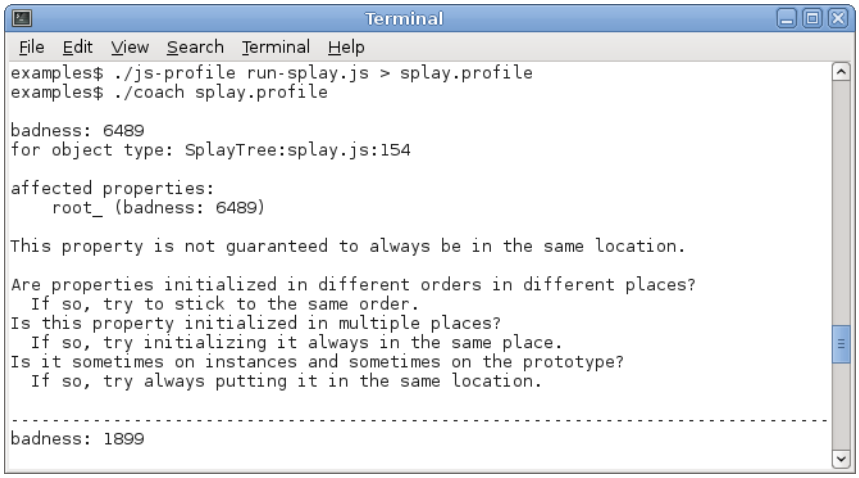

Figure 1 Excerpt from the coaching report for a splay tree implementation

This initial implementation performed poorly. Unbeknownst to Shumway's implementors, this implementation strategy triggers optimization failures inside JavaScript engines. If the compiler observes code to be polymorphic, it may not apply crucial optimizations. For instance, the compiler may be unable to determine a monomorphic context for the element accesses and the calling of the kernel function, prohibiting optimizations such as inlining.

Eventually, the Shumway engineers reverse engineered the JIT's opaque optimization decisions and could then diagnose the problem. They determined that performance could be recouped by cloning `forEach`'s implementation for variants that needed high performance (e.g., vectors of integers), as the JIT would then observe monomorphic accesses and call sites. While the compiler lacked the necessary context to make the appropriate tradeoff decision, the Shumway engineers were able to, once they understood the optimization decisions.

2.2 Optimization Coaching in a Nutshell

Failures such as those experienced by the first Shumway implementation are hard to diagnose and solve for two main reasons. First, optimizers fail silently; programmers are never informed that an optimization failed. Second, getting to the root causes of these failures requires skills and knowledge that are out of reach for most programmers. Those skills include auditing the compiler's output, reverse engineering the optimizer's decisions, etc.

Optimization coaches help programmers get more out of their optimizers without requiring such knowledge and with a minimum of effort. They achieve this feat by reporting *optimization near misses*. Near misses are optimizations that the compiler did not apply to their program—either due to a lack of information, or because doing so may be unsound in some cases—but could apply safely if the source program were changed in a certain way.

For example, consider the excerpt from a splay tree implementation in figure 2. The `isEmpty` method may find the `root_` property either on `SplayTree` instances (if the `insert` method has been called) or on the `SplayTree` prototype (otherwise). Hence, the JavaScript engine cannot specialize the property access to either of these cases and instead generates code that can handle both of them. The generated code is thus much slower than necessary.

```
// constructor
function SplayTree() {};
// default value on the prototype
SplayTree.prototype.root_ = null;

SplayTree.prototype.insert = function(key, value) {
  // regular value on instances
  ... this.root_ = new SplayTree.Node(key, value); ...
};

SplayTree.prototype.isEmpty = function() {
  // property may be either on the instance or on the prototype
  return !this.root_;
};
```

Figure 2 Splay tree implementation with an optimization near miss

```
function SplayTree() {
    // default value on instances
    this.root_ = null;
};
```

Figure 3 Improved splay tree constructor, without near miss

Coaches supplement near miss reports with concrete recommendations of program changes that programmers can apply. These modifications may make the compiler's analysis easier or may rule out corner cases, with an end result of the compiler succeeding to apply previously missed optimizations. Figure 1 shows the coach's diagnosis and recommendations of program changes that may resolve the near miss.

These recommendations are not required to preserve programs' exact semantics. In other words, coaches may recommend changes that would be beyond the reach of optimizing compilers, which are limited to semantics-preserving transformations. Programmers remain in control and are free to veto any recommendation that would lead to semantic, or structural, changes that they deem unreasonable.

In our splay tree example, the compiler cannot move `root_`'s default value to instances; this would change the behavior of programs that depend on the property being on the prototype. Programmers, on the other hand, are free to do so and may rewrite the program to the version from figure 3, which consistently stores the property on instances, and does not suffer from the previous near miss.

2.3 Optimization Coaching Concepts

To provide the necessary background to describe this paper's technical contributions, we now provide an overview of existing optimization coaching concepts.

At a high level, an optimization coach operates in four phases.

- First, instrumentation code inside the optimizer logs optimization decisions during compilation. This instrumentation distinguishes between *optimization successes*, i.e.,

optimizations that the compiler applies to the program, and *optimization failures*, i.e., optimizations that it cannot apply. These logs include enough information to reconstruct the optimizer's reasoning post facto. Section 4 describes the information recorded by our prototype's instrumentation, and section 7.1 explains our approach to instrumentation in a JIT context.

- Second, after compilation, an offline analysis processes these logs. The analysis phase is responsible for producing high-level, human-digestible near miss reports from the low-level optimization failure events recorded in the logs. It uses a combination of optimization-agnostic techniques and optimization-specific heuristics. We describe some of these techniques below.
- Third, from the near miss reports, the coach generates recommendations of program changes that are likely to turn these near misses into optimization successes. These recommendations are generated from the causes of individual failures as determined during compilation and from metrics computed during the analysis phase.
- Finally, the coach shows reports and recommendations to programmers. The interface should leverage optimization analysis metrics to visualize the coach's rankings of near misses or display high-estimated-impact recommendations only.

To avoid overwhelming programmers with large numbers of low-level reports, an optimization coach must carefully curate and summarize its output. In particular, it must restrict its recommendations to those that are both likely to enable further optimizations and likely to be accepted by the programmer. A coach uses three main classes of techniques for that purpose: pruning, ranking and merging.

Pruning Not all optimization failures are equally interesting to programmers. For example, showing failures that do not come with an obvious source-level solution, or failures that are likely due to intentional design choices, would be a waste of programmer time. Coaches therefore use heuristics that decide to remove optimization failures from the coach's reports. Optimization failures that remain after pruning constitute near misses, and are further refined via merging.

Our previous work describes several pruning techniques, such as *irrelevant failure pruning*, which we discuss in section 7.2.1. Section 7.1.2 introduces a new form of pruning based on profiling information.

Ranking Some optimization failures have a larger impact on program performance than others. A coach must rank its reports based on their expected performance impact to allow programmers to prioritize their responses. In order to do so, the coach computes a *badness* metric for each near miss, which estimates its impact on performance.

Our previous work introduces static heuristics to compute badness. Section 7.1.2 introduces the new dynamic heuristic that our prototype uses.

Merging To provide a high-level summary of optimization issues affecting a program, a coach should consolidate sets of related reports into single summary reports. Different merging techniques use different notions of relatedness.

These summary reports have a higher density of information than individual near miss reports because they avoid repeating common information, which may include cause of failure, solution, etc. depending on the notion of relatedness. They are also more efficient in terms of programmer time. For example, merging reports with similar solutions or the same program location, allows programmers to solve multiple issues at the same time.

When merging reports, a coach must respect preservation of badness which, for summary reports, is the sum of that of the merged reports. The sum of their expected performance impacts is a good estimation of the estimated impact of the summary report. The increased badness value of summary reports causes them to rank higher than their constituents would separately. Because these reports have a higher impact-to-effort ratio, having them high in the rankings increases the actionability of the tool's output.

Our previous work introduces two merging techniques: *causality merging* and *locality merging*. This work introduces three additional kinds of merging, *by-solution merging* (section 6.3), *by-constructor merging* (section 6.4) and *temporal merging* (section 7.1.3).

3 Background: The SpiderMonkey JavaScript Engine

This section surveys the aspects of SpiderMonkey that are relevant to this work.

3.1 Compiler Architecture

Like other modern JavaScript engines,[8] [9] [10] SpiderMonkey is a multi-tiered engine that uses type inference [13], type feedback [1], and optimizing just-in-time compilation [2] based on the SSA form [5], a formula proven to be well suited for JavaScript's dynamic nature. Specifically, it has three tiers: the interpreter, the baseline JIT compiler, and the IonMonkey (Ion) optimizing JIT compiler.

In the interpreter, methods are executed without being compiled to native code or optimized. Upon reaching a certain number of executions,[11] the baseline JIT compiles methods to native code. Once methods become hotter still and reach a second threshold,[12] Ion compiles them. The engine's gambit is that most methods are short-lived and relatively cold, especially for web workloads. By reserving heavyweight optimization for the hottest methods, it strikes a balance between responsiveness and performance.

3.2 Optimizations in Ion

Because Ion performs the vast majority of SpiderMonkey's optimizations, our work focuses on coaching those. Ion is an optimistic optimizing compiler, meaning it assumes types and other observed information gathered during baseline execution to hold for future executions, and it uses these assumptions to drive the optimization process.

Types and layout For optimization, the information SpiderMonkey observes mostly revolves around type profiling and object layout inference. In cases where inferring types would require a heavyweight analysis, such as heap accesses and function calls, SpiderMonkey uses type profiling instead. During execution, baseline-generated code stores the result types for heap acccesses and function calls for consumption by Ion.

At the same time, the runtime system also gathers information to infer the layouts of objects, i.e., mappings of property names to offsets inside objects. These layouts are referred to as "hidden classes" in the literature. This information enables Ion to generate code for

[8] `https://developers.google.com/v8/intro`
[9] `http://www.webkit.org/projects/javascript/`
[10] `http://msdn.microsoft.com/en-us/library/aa902517.aspx`
[11] At the time of this writing, 10.
[12] At the time of this writing, 1000.

property accesses on objects with known layout as simple memory loads instead of hash table lookups.

The applicability of Ion's optimizations is thus limited by the information it observes. The observed information is also used to seed a number of time-efficient static analyses, such as intra-function type inference.

Bailouts To guard against changes in the observed profile information, Ion inserts dynamic checks [15]. For instance, if a single callee is observed at a call site, Ion may optimistically inline that callee, while inserting a check to ensure that no mutation changes the binding referencing the inlinee. Should such a dynamic check fail, execution aborts from Ion-generated code and resumes in the non-optimized—and therefore safe—code generated by the baseline JIT compiler.

Optimization tactics As a highly optimizing compiler, Ion has a large repertoire of optimizations at its disposal when compiling key operations, such as property accesses. These optimizations are organized into *optimization tactics*. When compiling an operation, the compiler attempts each known optimization strategy for that kind of operation in order—from most to least profitable—until one of them applies.

A tactic's first few strategies are typically highly specialized optimizations that generate extremely efficient code, but apply only in limited circumstances, e.g., accessing a property of a known constant object. As compilation gets further into a tactic, strategies become more and more general and less and less efficient, e.g., polymorphic inline caches, until it reaches fallback strategies that can handle any possible situation but carry a significant performance cost, e.g., calling into the VM.

4 Optimization Corpus

Conventional wisdom among JavaScript compiler engineers points to property and element accesses as the most important operations to be optimized. For this reason, our prototype focuses on these two classes of operations.

For both, the instrumentation code records similar kinds of information. The information uniquely identifies each operation affected by optimization decisions, i.e., source location, type of operation and parameters. Additionally, it records information necessary to reconstruct optimization decisions themselves, i.e., the sets of inferred types for each operand, the sequence of optimization strategies attempted, which attempts were successful, which were not and why. This information is then used by the optimization analysis phase to produce and process near miss reports.

The rest of this section describes the relevant optimizations with an eye towards optimization coaching.

4.1 Property Access and Assignment

Conceptually, JavaScript objects are open-ended maps from strings to values. In the most general case, access to an object property is at best a hash table lookup, which, despite being amortized constant time, is too slow in practice. Ion therefore applies optimization tactics when compiling these operations so that it can optimize cases that do not require the full generality of maps. We describe some of the most important options below.

Definite slot Consider a property access `o.x`. In the best case, the engine observes `o` to be monomorphic and with a fixed layout. Ion then emits a simple memory load or store for the slot where `x` is stored. This optimization's prerequisites are quite restrictive. Not only must all objects that flow into `o` come from the same constructor, they must also share the same fixed layout. An object's layout is easily perturbed, however, for example by adding properties in different orders.

Polymorphic inline cache Failing that, if multiple types of plain JavaScript objects[13] are observed to flow to `o`, Ion can emit a polymorphic inline cache (PIC) [14]. The PIC is a self-patching structure in JIT code that dispatches on the type and layout of `o`. Initially, the PIC is empty. Each time a new type and layout of `o` flows into the PIC during execution, an optimized *stub* is generated that inlines the logic needed to access the property `x` for that particular layout of `o`. PICs embody the just-in-time philosophy of not paying for any expensive operation ahead of time. This optimization's prerequisites are less restrictive than that of definite slots, and it applies for the majority of property accesses that do not interact with the DOM.

VM call In the worst case, if `o`'s type is unknown to the compiler, either because the operation is in cold code and has no profiling information, or because `o` is observed to be an exotic object, then Ion can emit only a slow path call to a general-purpose runtime function to access the property.

Such slow paths are algorithmically expensive because they must be able to deal with any aberration: `o` may be of a primitive type, in which case execution must throw an error; `x` may be loaded or stored via a native DOM accessor somewhere on `o`'s prototype chain; `o` may be from an embedded frame within the web page and require a security check; etc. Furthermore, execution must leave JIT code and return to the C++ VM. Emitting a VM call is a last resort; it succeeds unconditionally, requires no prior knowledge, and is capable of handling all cases.

4.2 Element Access and Assignment

JavaScript's element access and assignment operations are polymorphic and operate on various types of indexable data, such as arrays, strings and `TypedArray`s. This polymorphism restricts the applicability of optimizations; most of them can apply only when the type of the indexed data is known in advance.

Even when values are known to be arrays, JavaScript semantics invalidate common optimizations in the general case. For example, JavaScript does not require arrays in the C sense, that is, it does not require contiguous chunks of memory addressable by offset. Semantically, JavaScript arrays are plain objects that map indices—*string* representations of unsigned integers—to values. Element accesses into such arrays, then, are semantically (and perhaps surprisingly) equivalent to property lookups and are subject to the same set of rules, such as prototype lookups.

As with inferring object layout, SpiderMonkey attempts to infer when JavaScript arrays are used as if they were dense, C-like arrays, and optimize accordingly. Despite new APIs such

[13] The restriction on plain JavaScript objects is necessary because properties may be accessed from a variety of exotic object-like values, such as DOM nodes and proxies. Those objects encapsulate their own logic for accessing properties that is free to deviate from the logic perscribed for plain objects by the ECMAScript standard.

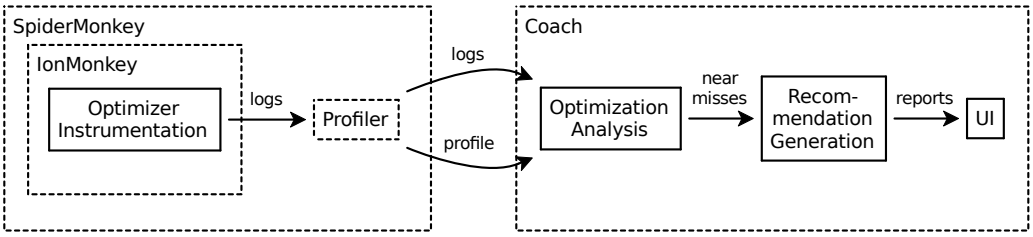

Figure 4 Our prototype's architecture

as `TypedArray`s offering C-like arrays directly, SpiderMonkey's dense array optimizations remain crucial to the performance of the web.

To manage all possible modes of use of element accesses and the optimizations that apply in each of them, Ion relies on optimization tactics. We describe the most important optimization strategy—dense array access—below. The PIC and VM call cases are similar to the corresponding cases for property access. Other, specialized strategies heavily depend on SpiderMonkey's data representation and are beyond the scope of this paper, but are handled by the prototype.

Dense array access Consider an element access `o[i]`. In the best case, if `o` is determined to be used as a dense array and `i` an integer, Ion can emit a memory load or a store for offset `i` plus bounds checking. For this choice to be valid, all types that flow into `o` must be plain JavaScript objects that have dense indexed properties. An object with few indexed properties spread far apart would be considered sparse, e.g., if only `o[0]` and `o[2048]` were set, `o` would not be considered dense. Note that an object may be missing indexed properties and still be considered dense. SpiderMonkey further distinguishes dense arrays—those with allocated dense storage—from packed arrays—dense arrays with no holes between indexed properties. Ion is able to elide checking whether an element is a hole, or a missing property, for packed arrays. Furthermore, the object `o` must not have been observed to have prototypes with indexed properties, as otherwise accessing a missing indexed property `j` on `o` would, per specification, trigger a full prototype walk to search for `j` when accessing `o[j]`.

5 Architecture

As section 2.3 explains, our optimization coach operates in four phases. Figure 4 illustrates how these phases interact. In the first phase, instrumentation inside IonMonkey's optimizer logs optimization successes and failures and sends that information to the SpiderMonkey profiler (section 7.1.1). Next, the optimization analysis phase applies pruning heuristics (sections 7.2.1 and 7.2.2), determines solution sites (section 6.1), computes badness scores (section 7.1.2), and finally merges reports (sections 6.3, 6.4, and 7.1.3). Its end result is a list of ranked near misses.

The third phase, recommendation generation, fills out textual recommendation templates—selected based on failure causes—with inferred solution sites, failure causes, type information, and source information. Finally, the tool's user interface presents the five highest-ranked recommendations to programmers.

6 Coaching for Object-Oriented Languages

Dispatch optimizations for property operations fundamentally depend on non-local information. For example, the optimizer must know the layout of objects that flow to a property access site to determine whether it can be optimized to a direct dereference. That information is encoded in the *constructor* of these objects, which can be arbitrarily far away in source text from the property access considered for optimization.

In turn, this gap causes optimization failures to be non-local; a failure at one program location—the property access site—can be resolved by program changes at a different location—the constructor. To provide actionable feedback to programmers, a coach must connect the two sites and link its reports to the solution site.

Not all failures, however, are non-local in this manner. For example, failing to specialize a property access that receives multiple different types of objects is a purely local failure; it fails because the operation itself is polymorphic, which can only be solved by changing the operation itself. An optimization coach must therefore distinguish between local and non-local failures and target its reports accordingly. Our prototype accomplishes this using *solution site inference*.

In addition, a coach should also merge near misses that have the same, or similar, solutions and report them together. Our prototype uses *by-solution merging* and *by-constructor merging* for this purpose. It also reuses the notion of *locality merging* (see [22]).

6.1 Solution Site Inference

The goal of solution site inference is to determine, for a given near miss, whether it could be resolved by changes at the site of the failing optimization or whether changes to the receivers' constructors may be required. We refer to the former as *operation near misses* and to the latter as *constructor near misses*. To reach a decision, the coach follows heuristics based on the cause of the failure, as well as on the types that flow to the affected operation. We briefly describe two of these heuristics.

Monomorphic operations If an optimization fails for an operation to which a single receiver type flows, then that failure must be due to a property of that type, not of the operation's context. The coach infers these cases to be constructor near misses.

Property addition When a property assignment operation for property `p` receives an object that lacks a property `p`, the operation instead adds the property to the object. If the same operation receives both objects with a property `p` and objects without, that operation cannot be specialized for either mode of use. This failure depends on the operation's context, and the coach infers it to be an operation near miss.

6.2 Same-Property Analysis

The merging techniques we describe below both depend on grouping near misses that affect the same property. The obvious definitions of "same property," however, do not lead to satisfactory groupings. If we consider two properties with the same name to be the same, the coach would produce spurious groupings of unrelated properties from different parts of the program, e.g., grouping `canvas.draw` with `gun.draw`. Using these spurious groupings for merging would lead to incoherent reports that conflate unrelated near misses.

```
Scheduler.prototype.schedule = function () {
  // this.currentTcb is only ever a TaskControlBlock
  ...
  this.currentTcb = this.currentTcb.run();
  ...
};

TaskControlBlock.prototype.run = function () {
  // this.task can be all four kinds of tasks
  ...
  return this.task.run(packet);
  ...
};

IdleTask.prototype.run = function (packet) { ... };
DeviceTask.prototype.run = function (packet) { ... };
WorkerTask.prototype.run = function (packet) { ... };
HandlerTask.prototype.run = function (packet) { ... };
```

Figure 5 Two different logical properties with name `run` in the Richards benchmark, one underlined and one boxed.

In contrast, if we considered only properties with the same name and the same hidden class, the coach would discriminate too much and miss some useful groupings. For example, consider the `run` property of various kinds of tasks in the Richards benchmark from the Octane[14] benchmark suite, boxed in figure 5. These properties are set independently for each kind of task and thus occur on different hidden classes, but they are often accessed from the same locations and thus should be grouped by the coach. This kind of pattern occurs frequently when using inheritance or when using structural typing for ad-hoc polymorphism.

To avoid these problems, we introduce another notion of property equivalence, *logical properties*, which our prototype uses to guide its near-miss merging. We define two concrete properties `p1` and `p2`, which appear on hidden classes `t1` and `t2` respectively, to belong to the same logical property if they

- have the same name `p`, and
- co-occur in at least one operation, i.e., there exists an operation `o.p` or `o.p = v` that receives objects of both class `t1` and class `t2`

As figure 5 shows, the four concrete `run` properties for tasks co-occur at an operation in the body of `TaskControlBlock.prototype.run`, and therefore belong to the same logical property. `TaskControlBlock.prototype.run`, on the other hand, never co-occurs with the other `run` properties, and the analysis considers it separate; near misses that are related to it are unrelated from those affecting tasks' `run` properties and should not be merged.

6.3 By-Solution Merging

In addition to linking near-miss reports with the likely location of their solution, an optimization coach should group near misses with related solutions. That is, it should merge near misses that can be addressed either by same program change or by performing analogous changes at multiple program locations.

[14] `https://developers.google.com/octane/`

Detecting whether multiple near misses call for the same kind of corrective action is a simple matter of comparing the causes of the respective failures and their context, as well as ensuring that the affected properties belong to the same logical property. This mirrors the work of the recommendation generation phase, as described in section 2.3.

Once the coach identifies sets of near misses with related solutions, it merges each set into a single summary report. This new report includes the locations of individual failures, as well as the common cause of failure, the common solution and a badness score that is the sum of those of the merged reports.

6.4 By-Constructor Merging

Multiple near misses can often be solved at the same time by changing a single constructor. For example, inconsistent property layout for objects from one constructor can cause optimization failures for multiple properties, yet all of those can be resolved by editing the constructor. Therefore, merging constructor near misses that share a constructor can result in improved coaching reports.

To perform this merging, the coach must identify which logical properties co-occur within at least one hidden class. To do this, it reuses knowledge about which logical properties occur within each hidden class from same-property analysis.

Because, in JavaScript, properties can be added to objects dynamically—i.e., not inside the object's constructor—a property occuring within a given hidden class does not necessarily mean that it was added by the constructor associated with that class. This may lead to merging reports affecting properties added in a constructor with others added elsewhere. At first glance, this may appear to cause spurious mergings, but it is in fact beneficial. For example, moving property initialization from the outside of a constructor to the inside often helps keeping object layout consistent. Reporting these near misses along with those from properties from the constructor helps reinforce this connection. We discuss instances of this problem in section 9.3.

7 Coaching for an Advanced Compiler

Advanced compilers such as IonMonkey operate differently from simpler compilers, such as the ahead-of-time portion of the Racket compiler, which we studied previously. An optimization coach needs to adapt to these differences. This section presents the challenges posed by two specific features of Ion that are absent in a simple compiler—JIT compilation and optimization tactics—and describes our solutions.

7.1 JIT Compilation

From a coaching perspective, JIT compilation poses two main challenges absent in an ahead-of-time (AOT) setting. First, compilation and execution are interleaved in a JIT system; there is no clear separation between compile-time and run-time, as there is in an AOT system. The latter's separation makes it trivial for a coach's instrumentation to not affect the program's execution; instrumentation, being localized to the optimizer, does not cause any runtime overhead and emitting the optimization logs does not interfere with the program's I/O proper. In a JIT setting, however, instrumentation may affect program execution, and a coach must take care when emitting optimization information.

Second, whereas an AOT compiler compiles a given piece of code once, a JIT compiler may compile it multiple times as it gathers more information and possibly revises previous

assumptions. In turn, a JIT compiler may apply—and fail to apply—different optimizations each time. Hence, the near misses that affect a given piece of code may evolve over time, as opposed to being fixed, as in the case of an AOT compiler. Near misses therefore need to be ranked and merged along this new, temporal axis.

Our prototype coach addresses both challenges via the use of a novel, profiler-driven instrumentation strategy and by applying *temporal merging*, an extension of the locality merging technique we presented in previous work.

7.1.1 Profiler-Driven Instrumentation

Our prototype coach uses SpiderMonkey's profiling subsystem as the basis for its intrumentation. The SpiderMonkey profiler, as many profilers, provides an "event" API in addition to its main sampling-based API. The former allows the engine to report various kinds of one-off events that may be of interest to programmers: Ion compiling a specific method, garbage collection, the execution bailing out of optimized code, etc.

This event API provides a natural communication channel between the coach's instrumentation inside Ion's optimizer and the outside world. As with an AOT coach, our prototype records optimization decisions and context information as the optimizer processes code. Where an AOT coach would emit that information on the fly, our prototype instead gathers all the information pertaining to a given invocation of the compiler, encodes it as a profiler event and emits it all at once. Our prototype's instrumentation executes only when the profiler is active; its overhead is therefore almost entirely pay-as-you-go.

In addition to recording optimization information, the instrumentation code assigns a unique identifier to the compiled code resulting from each Ion invocation. This identifier is included alongside the optimization information in the profiling event. Object code that is instrumented for profiling carries meta-information (e.g. method name and source location) that allows the profiler to map the samples it gathers back to source code locations. We include the compilation identifier as part of this meta-information, which allows the coach to correlate profiler samples with optimization information, which in turn enables heuristics based on profiling information as discussed below. This additional piece of meta-information has negligible overhead and is present only when the profiler is active.

7.1.2 Profiling-Based Badness Metric

One of the key advantages of an optimization coach over raw optimization logs is the pruning and ranking of near misses that a coach provides based on expected performance impact. An AOT coach uses a number of static heuristics to estimate this impact.

Our prototype incorporates profiling-based heuristics, which has two main advantages. First, even in an AOT setting, actionable prioritization of near misses benefits from knowing where programs spend their time; near misses in hot methods are likely to have a larger impact on performance than those in cold code.

Second, state-of-the-art JIT compilers may compile the same code multiple times—producing different *compiled versions* of that code—potentially with different near misses each time. A coach needs to know which of these compiled versions execute for a long time and which are short-lived. Near misses from compiled versions that execute only for a short time cannot have a significant impact on performance across the whole execution, regardless of the number or severity of near misses, or how hot the affected method is overall. Because profiler samples include compilation identifiers, our prototype associates each sample not

only with particular methods, but with particular compiled versions of methods. It then enables the required distinctions discussed above.

Concretely, our prototype uses the *profiling weight* of the compiled version of the function that surrounds a near miss as its badness score. We define the profiling weight of a compiled version to be the fraction of the total execution time that is spent executing it. Combined with temporal merging, this design ensures that near misses from hot compiled versions rise to the top of the rankings.

To avoid overwhelming programmers with large numbers of potentially low-impact recommendations, our prototype prunes reports based on badness and shows only the five reports with the highest badness scores. This threshold has been effective in practice but is subject to adjustment.

7.1.3 Temporal Merging

Even though a JIT compiler may optimize methods differently each time they get compiled, this is not always the case. It is entirely possible to have an operation be optimized identically across multiple versions or even all of them. It happens, for instance, when recompilation is due to the optimizer's assumptions not holding for a different part of the method or as a result of object code being garbage collected.[15]

Identical near misses that originate from different invocations of the compiler necessarily have the same solution; they are symptoms of the same underlying issue. To reduce redundancy in the coach's reports, we extend the notion of locality merging—which merges reports that affect the same operation—to operate across compiled version boundaries. The resulting technique, *temporal merging*, combines near misses that affect the same operation or constructor, originate from the same kind of failure and have the same causes across multiple compiled versions.

7.2 Optimization Tactics

When faced with an array of optimization options, Ion relies on optimization tactics to organize them. While we could consider each individual element of a tactic as a separate optimization and report near misses accordingly, all of a tactic's elements are linked. Because the entire tactic returns as soon as one element succeeds, its options are mutually exclusive; only the successful option applies. To avoid overwhelming programmers with multiple reports about the same operation and provide more actionable results, a coach should consider a tactic's options together.

7.2.1 Irrelevant Failure Pruning

Ion's tactics often include strategies that only apply in narrow cases—e.g. indexing into values that are known to be strings, property accesses on objects that are known to be constant, etc. Because of their limited applicability, failure to apply these optimizations is not usually symptomatic of performance issues; these optimizations are expected to fail most of the time.

[15] In SpiderMonkey, object code is collected during major collections to avoid holding on to object code for methods that may not be executed anymore. While such collections may trigger more recompilation than strictly necessary, this tradeoff is reasonable in the context of a browser, where most scripts are short-lived.

In these cases, we reuse the Racket coach's *irrelevant failure pruning* technique. Failures to apply optimizations that are expected to fail do not provide any actionable information to programmers, and thus we consider them irrelevant. The coach prunes such failures from the logs and does not show them in its reports.

7.2.2 Partial Success Shortcircuiting

While some elements of a given tactic may be more efficient than others, it is not always reasonable to expect that all code be compiled with the best tactic elements. For example, polymorphic call sites cannot be optimized as well as monomorphic call sites; polymorphism notably prevents fixed-slot lookup. Polymorphism, however, is often desirable in a program. Recommending that programmers eliminate it altogether in their programs is preposterous and would lead to programmers ignoring the tool. Clearly, considering all polymorphic operations to suffer from near misses is not effective.

We partition a tactic's elements according to source-level concepts—e.g., elements for monomorphic operations vs polymorphic operations, elements that apply to array inputs vs string inputs vs typed array inputs, etc.—and consider picking the best element from a group to be an optimization success, so long as the operation's context matches that group.

For example, the coach considers picking the best possible element that is applicable to polymorphic operations to be a success, as long as we can infer from the context that the operation being compiled is actually used polymorphically. Any previous failures to apply monomorphic-only elements to this operation would be ignored.

With this approach, the coach reports polymorphic operations that do not use the best possible polymorphic element as near misses, while considering those that do to be successes. In addition, because the coach considers only uses of the best polymorphic elements to be successes if operations are actually polymorphic according to their context, monomorphic operations that end up triggering them are reported as near misses—as they should be.

In addition to polymorphic property operations, our prototype applies partial success shortcircuiting to array operations that operate on typed arrays and other indexable datatypes. For example, Ion cannot apply dense-array access for operations that receive strings, but multiple tactic elements can still apply in the presence of strings, some more performant than others.

8 Dead Ends

The previous sections describe successful coaching techniques, which result in actionable reports. Along the way, we also implemented other techniques that ultimately did not prove to be useful and which we removed from our prototype. These techniques either produced reports that did not lead programmers to solutions or pointed out optimization failures that did not actually impact performance.

In the interest of saving other researchers from traveling down the same dead ends, this section describes two kinds of optimization failures that we studied without success: *regressions* and *flip-flops*. Both are instances of *temporal patterns*, that is, attempts by the coach to find optimization patterns across time. None of our attempts at finding such patterns yielded actionable reports, but there may be other kinds of temporal patterns that we overlooked that would.

8.1 Regression Reports

The coach would report a regression when an operation that was optimized well during a compilation failed to be optimized as well during a subsequent one. This pattern occurred only rarely in the programs we studied, and when it did, it either was inevitable (e.g. a call site becoming polymorphic as a result of observing a sentinel value in addition to its usual receiver type) or did not point to potential improvements.

8.2 Flip-Flop Reports

As mentioned, SpiderMonkey discards object code and all type information during major collections. When this happens, the engine must start gathering type information and compiling methods from scratch. In some cases, this new set of type information may lead the engine to be optimistic in a way that was previously invalidated, then forgotten during garbage collection, leading to excessive recompilation. Engine developers refer to this process of oscillating back and forth between optimistic and conservative versions as *flip-flopping*.

For example, consider a method that almost always receives integers as arguments, but sometimes receives strings as well. Ion may first optimize it under the first assumption, then have to back out of this decision after receiving strings. After garbage collection, type information is thrown away and this process starts anew. As a result, the method may end up being recompiled multiple times between each major collection.

Engine developers believe that this behavior can cause significant performance issues, mostly because of the excessive recompilation. While we observed instances of flip-flopping in practice, modifying the affected programs to eliminate these recompilations often required significant reengineering and did not yield observable speedups.

9 Evaluation

For an optimization coach to be useful, it must provide actionable recommendations that improve the performance of a spectrum of programs. This section shows the results of evaluating our prototype along two axes: performance improvements and programmer effort.

9.1 Experimental Protocol

For our evaluation, we chose a subset of the widely-used Octane benchmark suite. We ran these programs using our prototype and modified them by following the coach's recommendations. For each program, we applied all of the five top-rated recommendations, so long as the advice was directly actionable. That is, we rejected reports that did not suggest a clear course of action, as a programmer using the tool would do.

To simulate a programmer looking for "low-hanging fruit," we ran the coach only once on each program. Re-running the coach on a modified program may cause the coach to provide different recommendations. Therefore, it would in principle be possible to apply recommendations up to some fixpoint.

For each program and recommendation, we measured a number of attributes to assess three dimensions of optimization coaching:

Performance Impact Our primary goal is to assess the effect of recommended changes on program performance. Because a web page's JavaScript code is likely to be executed by multiple engines, we used three of the major JavaScript engines: SpiderMonkey, Chrome's V8 and Webkit's JavaScriptCore.

The Octane suite measures performance in terms of an *Octane Score* which, for the benchmarks we discuss here, is inversely proportional to execution time.[16] Our plots show scores normalized to the pre-coaching version of each program with error bars marking 95% confidence intervals. All our results represent the mean score of 30 executions on a 6-core 64-bit x86 Debian GNU/Linux system with 12GB of RAM. To eliminate confounding factors due to interference from other browser components, we ran our experiments in standalone JavaScript shells.

Programmer Effort As a proxy for programmer effort, we measured the number of lines changed in each program while following recommendations. We also recorded qualitative information about the nature of these changes.

Recommendation Usefulness To evaluate the usefulness of individual recommendations, we classified them into four categories:

- *positive* recommendations led to an increase in performance,
- *negative* recommendations led to a decrease in performance,
- *neutral* recommendations did not lead to an observable change in performance, and
- *non-actionable* reports did not suggest a clear course of action.

For this aspect of the evaluation, we measured the impact of individual recommendations under SpiderMonkey alone.

Ideally, a coach should give only positive recommendations. Negative recommendations require additional work on the part of the programmer to identify and reject. Reacting to neutral recommendations is also a waste of programmer time, and thus their number should be low, but because they do not harm performance, they need not be explicitly rejected by programmers. Non-actionable recommendations decrease the signal-to-noise ratio of the tool, but they can individually be dismissed pretty quickly by programmers. A small number of non-actionable recommendations therefore does not contribute significantly to the programmer's workload. Large numbers of non-actionable recommendations, however, would be cause for concern.

9.2 Program Selection

Our subset of the Octane suite focuses on benchmarks that use property and array operations in a significant manner. It excludes, for example, the Regexp benchmark because it exercises nothing but an engine's regular expression subsystems. Coaching these programs would not yield any recommendations with our current prototype. It also excludes machine-generated programs from consideration. The output of, say, the Emscripten C/C++ to JavaScript compiler[17] is not intended to be read or edited by humans; it is therefore not suitable for coaching.[18] In total, the set consists of eight programs: Richards, DeltaBlue, RayTrace, Splay, NavierStokes, PdfJS, Crypto and Box2D.

[16] The Octane suite also includes benchmarks whose scores are related to latency instead of execution time, but we did not use those for our experiments.
[17] `https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Emscripten`
[18] It would, however, be possible to use coaching to improve the code generation of Emscripten or other compilers that target JavaScript, such as Shumway. This is a direction for future work.

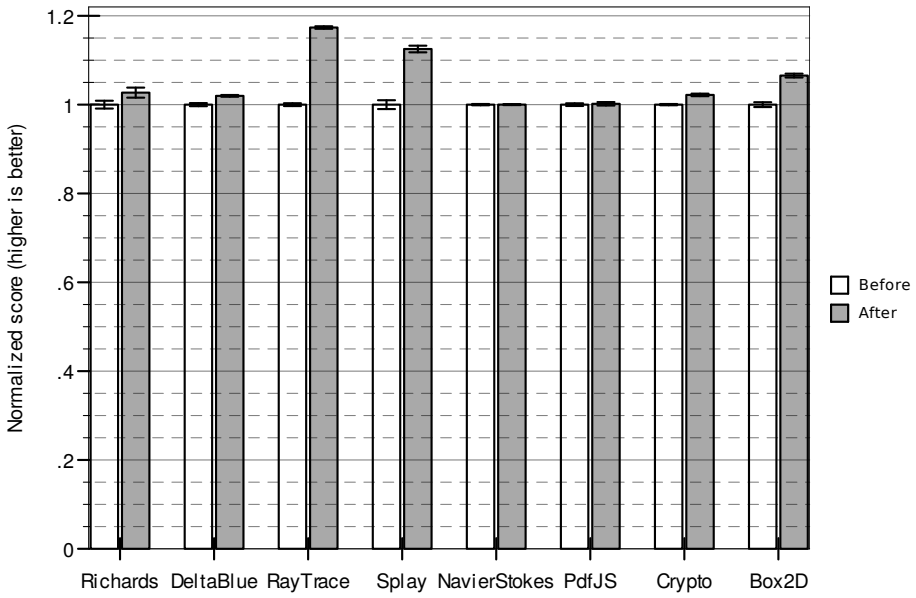

Figure 6 Benchmarking results on SpiderMonkey

9.3 Results and Discussion

As figure 6 shows, following the coach's recommendations leads to significant[19] speedups on six of our eight benchmarks when run on SpiderMonkey. These speedups range from 1.02× to 1.17×. For the other two benchmarks, we observe no significant change; in no case do we observe a slowdown.

The results are similar for the other engines, see figure 7. On both V8 and JavaScriptCore, we observe significant speedups on two and three benchmarks, respectively, ranging from 1.02× to 1.20×. These speedups differ from those observed using SpiderMonkey, but are of similar magnitude. Only in the case of the DeltaBlue benchmark on JavaScriptCore is there a significant slowdown. These results provide evidence that, even though coaching recommendations are derived from the optimization process of a *single engine*, they can lead to *cross-engine* speedups.

Keeping in mind that JavaScript engines are tuned to perform well on those benchmark programs,[20] we consider these results quite promising. We conjecture that our prototype (or an extension of it) could yield even larger speedups on other, regular programs for which the engine is not specifically tuned.

Figure 8 presents our results for the effort and usefulness dimensions. For all programs, the total number of lines changed is at most 42. Most of these changes are also fairly mechanical in nature—moving code, search and replace, local restructuring. Together, these amount to modest efforts on the programmer's part.

[19] We consider speedups to be significant when the confidence intervals of the baseline and coached versions do not overlap.

[20] `http://arewefastyet.com`

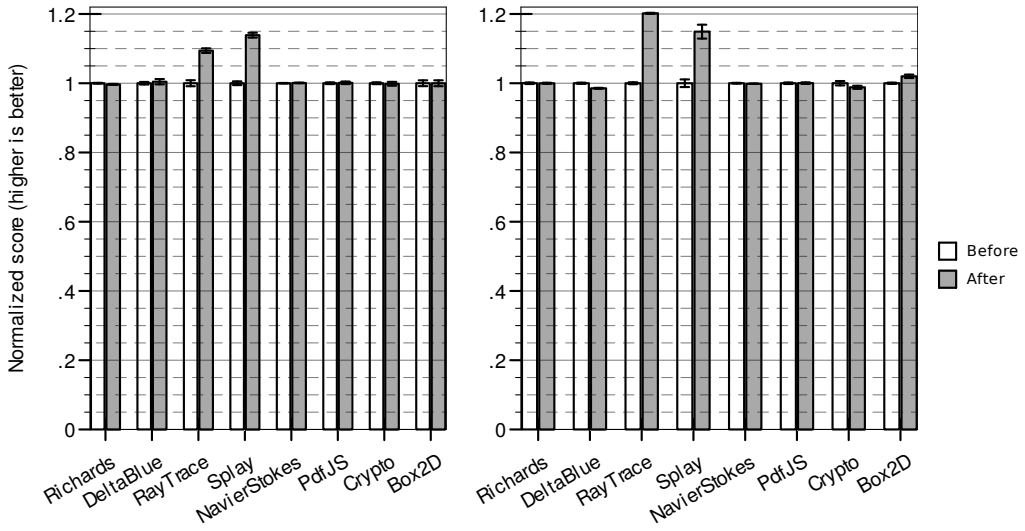

Figure 7 Benchmarking results on V8 and JavaScriptCore

We classified 17 out of 35 reports as positive, and only one as negative. We classified 12 reports as non-actionable, which we consider acceptably low. As discussed above, those reports can be dismissed quickly and do not impose a burden. The remainder of the section presents the coach's recommendations for individual benchmarks.

Richards The coach provides three reports. Two of those point out an inconsistency in the layout of `TaskControlBlock` objects. Figure 9 shows one of them. The `state` property is initialized in two different locations, which causes layout inference to fail and prevents optimizations when retrieving the property. Combining these two assignments into one, as figure 10 shows, solves the issue and leads to a speedup of 1.03× on SpiderMonkey. The third report points to an operation that is polymorphic by design; it is not actionable.

DeltaBlue Two of the five reports have a modest positive impact. The first involves replacing a singleton object's properties with global variables to avoid dispatch; it is shown in figure 11. The second recommends duplicating a superclass's method in its subclasses, making them monomorphic in the process.

These changes may hinder modularity and maintainability in some cases. They clearly illustrate the tradeoffs between performance and software engineering concerns, which coaching tends to bring up. Which of those is more important depends on context, and the decision of whether to follow a recommendation must remain in the programmer's hands. With a coach, programmers at least know *where* these tradeoffs may pay off by enabling additional optimization.

One of the recommendations (avoiding a prototype chain walk) yields a modest slowdown of about 1%. This report has the lowest badness score of the five. We expect programmers tuning their programs to try out these kinds of negative recommendations and revert them after observing slowdowns.

Benchmark	Size (SLOC)	Lines changed (SLOC)			Recommendation impact (# recommendations)			
		Added	Deleted	Edited	Positive	Negative	Neutral	Non-act.
Richards	538	1	5	0	2	0	0	1
DeltaBlue	881	12	6	24	2	1	1	1
RayTrace	903	10	11	0	5	0	0	0
Splay	422	3	3	0	2	0	1	2
NavierStokes	415	0	0	4	0	0	1	0
PdfJS	33,053	2	1	0	0	0	1	4
Crypto	1,698	2	0	1	4	0	0	1
Box2D	10,970	8	0	0	2	0	0	3

Figure 8 Summary of changes following recommendations

```
badness: 24067
for object type: TaskControlBlock:richards.js:255

affected properties:
    state (badness: 24067)

This property is not guaranteed to always be in the same location.

Are properties initialized in different orders in different places?
  If so, try to stick to the same order.
Is this property initialized in multiple places?
  If so, try initializing it always in the same place.
Is it sometimes on instances and sometimes on the prototype?
  If so, try using it consistently.
```

Figure 9 Report of inconsistent property order in the Richards benchmark

```
// before coaching
if (queue == null) {
  this.state = STATE_SUSPENDED;
} else {
  this.state = STATE_SUSPENDED_RUNNABLE;
}

// after coaching
this.state = queue == null ? STATE_SUSPENDED : STATE_SUSPENDED_RUNNABLE;
```

Figure 10 Making object layout consistent in the Richards benchmark

```
badness: 5422
for object type: singleton
affected properties:
    WEAKEST (badness: 2148)
    REQUIRED (badness: 1640)
    STRONG_DEFAULT (badness: 743)
    PREFERRED (badness: 743)
    NORMAL (badness: 147)

This object is a singleton.
Singletons are not guaranteed to have properties in a fixed slot.
Try making the object's properties globals.
```

Figure 11 Recommendation to eliminate a singleton object in the DeltaBlue benchmark

RayTrace All five of the coach's reports yield performance improvements, for a total of 1.17× on SpiderMonkey, 1.09× on V8 and 1.20× on JavaScriptCore. The proposed changes include reordering property assignments to avoid inconsistent layouts, as well as replacing a use of prototype.js's class system with built-in JavaScript objects for a key data structure. All these changes are mechanical in nature because they mostly involve moving code around.

Splay This program is the same as the example in section 2.2. Of the five reports, three recommend moving properties from a prototype to its instances. These properties are using a default value on the prototype and are sometimes left unset on instances, occasionally triggering prototype chain walks. The fix is to change the constructor to assign the default value to instances explicitly. While this may cause additional space usage by making instances larger, the time/space tradeoff is worthwhile and leads to speedups on all three engines. Two of the three changes yield speedups, with the third one not having a noticeable effect.

NavierStokes The coach provides a single recommendation for this program. It points out that some array accesses are not guaranteed to receive integers as keys. Enforcing this guarantee by bitwise or'ing the index with `0`, as is often done in asm.js codebases, solves this issue but does not yield noticeable performance improvements. It turns out that the code involved only accounts for only a small portion of total execution time.

PdfJS One of the coach's reports recommends initializing two properties in the constructor, instead of waiting for a subsequent method call to assign them, because the latter arrangement results in inconsistent object layouts. As with the recommendation for the NavierStokes benchmark, this one concerns cold code[21] and does not lead to noticeable speedups.

We were not able to make changes based on the other four recommendations, which may have been due to our lack of familiarity with this large codebase. Programmers more familiar with PdfJS's internals may find these reports more actionable.

Crypto Four of the five reports are actionable and lead to speedups. Three of the four concern operations that sometimes add a property to an object and sometimes assign an existing one, meaning that they therefore cannot be specialized for either use. Initializing

[21] PdfJS's profile is quite flat in general, suggesting that most low-hanging fruit has already been picked, which is to be expected from such a high-profile production application.

those properties in the constructor makes the above operations operate as assignments consistently, which solves the problem. The last positive recommendation concerns array accesses; it is similar to the one discussed in conjunction with the NavierStokes benchmark, with the exception that this one yields speedups.

Box2D Two of the reports recommend consistently initializing properties, as with the PdfJS benchmark. Applying those changes yields a speedup of 1.07× on SpiderMonkey. The other three recommendations are not actionable due to our cursory knowledge of this codebase. As with PdfJS, programmers knowledgeable about Box2D's architecture may fare better.

For reference, the Octane benchmark suite uses a minified version of this program. As discussed above, minified programs are not suitable for coaching so we used a non-minified, but otherwise identical, version of the program.

10 Related Work

This work is not the only attempt at helping programmers take advantage of their compilers' optimizers. This section discusses tools with similar goals and compares them with our work.

10.1 Optimization Logging

From an implementation perspective, the simplest way to inform programmers about the optimizer's behavior on their programs is to provide them with logs recording its optimization decisions. This is the approach taken by tools such as JIT inspector [12] and IRHydra [8], both of which report optimization successes and failures, as well as other optimization-related events such as dynamic deoptimizations. JIT inspector reports optimizations performed by IonMonkey, while IRHydra operates with the V8 and Dart compilers.

Similar facilities also exist outside of the JavaScript world. For instance, Common Lisp compilers such as SBCL [23] and LispWorks [18] report both optimization successes and optimization failures, such as failures to specialize generic operations or to allocate objects on the stack. The Cray XMT C and C++ compilers [4] report both successful optimimizations and parallelization failures. The Open Dylan IDE [6, chapter 10] reports optimizations such as inlining and dispatch optimizations using highlights in the IDE's workspace.

These tools provide reports equivalent to the raw output of our prototype's instrumentation without any subsequent analysis, interpretation or recommendations. Expert programmers knowledgeable about compiler internals may find this information actionable and use it as a starting point for their tuning efforts. In constrast, our prototype coach targets programmers who may not have the necessary knowledge and expertise to digest such raw information, and it does so by providing recommendations that only require source-level knowledge.

10.2 Rule-Based Performance Bug Detection

Some performance tools use rule-based approaches to detect code patterns that may be symptomatic of performance bugs.

JITProf [10] is a dynamic analysis tool for JavaScript that detects code patterns that JavaScript JIT compilers usually do not optimize well. The tool looks for six dynamic patterns during program execution, such as inconsistent object layouts and arithmetic operations on the `undefined` value, and reports instances of these patterns to programmers.

The JITProf analysis operates independently from the host engine's optimizer; its patterns essentially constitute a model of a typical JavaScript JIT compiler. As a result, JITProf

does not impose any maintenance burden on engine developers, unlike a coach whose instrumentation must live within the engine itself. Then again, this separation may cause the tool's model to be inconsistent with the actual behavior of engines, either because the model does not perfectly match an engine's heuristics, or because engines may change their optimization strategies as their development continues. In contrast, an optimization coach reports ground truth by virtue of getting its optimization information from the engine itself.

By not being tied to a specific engine, JITProf's reports are not biased by the implementation details of that particular engine. Section 9 shows, however, that engines behave similarly enough in practice that a coach's recommendations, despite originating from a specific engine, usually lead to cross-engine performance improvements.

Jin et al. [16] distill performance bugs found in existing applications to source-level patterns which can then be used to detect similar latent bugs in other applications. Their tool suggests fixes for these new bugs based on those used to resolve the original bugs. Their work focuses on API usage and algorithms, and is complementary to optimization coaching.

Chen et al. [3] present a tool that uses static analysis to detect performance anti-patterns that result from the use of object-relational mapping in database-backed applications. The tool detects these anti-patterns using rules that the authors synthesized from observing existing database-related performance bugs. To cope with the large number of reports, the tool estimates the performance impact of each anti-pattern, and uses that information to prioritize reports. This is similar to the use of ranking by optimization coaches.

10.3 Profilers

When they encounter performance issues, programmers often reach for a profiler [11, 19, 21, 24]. Unlike an optimization coach, a profiler does not point out optimization failures directly. Instead, it identifies portions of the program where most of its execution time is spent, some of which may be symptomatic of optimization failures. That inference, however, is left to programmers.

Profilers also cannot distinguish between code that naturally runs for a long time from code that runs for an abnormally long time. Again, the programmer is called upon to make this distinction. In contrast, coaches distinguish between optimization failures that are expected from those that are not. In addition, coaches aim to provide actionable recommendations to programmers, whereas profilers report data without pointing towards potential solutions.

Note, though, that profilers can point to a broader range of performance issues than optimization coaches. For example, a profiler would report code that runs for a long time due to an inefficient algorithm, which an optimization coach could not detect. To summarize, the two classes of tools cover different use cases and are complementary.

10.4 Assisted Optimization

A number of performance tools are aimed at helping programmers optimize specific aspects of program performance. This section discusses the ones most closely related to this work.

Larsen et al. [17] present an interactive tool that helps programmers parallelize their programs. Like an optimization coach, their tool relies on compiler instrumentation to reconstruct the optimization process—specifically automatic parallelization—and discover the causes of parallelization failures. Larsen et al.'s tool is specifically designed for parallelization and is thus complementary to optimization coaching.

Precimonious [20] is a tool that helps programmers balance precision and performance in floating-point computations. It uses dynamic program analysis to discover floating-point

variables that can be converted to use lower-precision representations without affecting the overall precision of the program's results. The tool then recommends assignments of precisions to variables that programmers can apply. This workflow is similar to that of an optimization coach, but applied to a different domain.

Xu et al. [25] present a tool that detects data structures that are expensive to compute, but that the program either does not use, or only uses a small portion of. Based on the tool's reports, programmers can replace the problematic structures with more lightweight equivalents that only store the necessary data. The tool relies on a novel program slicing technique to detect those low-utility data structures. This tool is also complementary to optimization coaches.

11 Conclusion

In this paper, we present an adaptation of optimization coaching to the world of dynamic object-oriented languages with advanced JIT compilers. The additional constraints imposed by these languages and their compilers require novel coaching techniques such as profiler-based instrumentation and solution-site inference.

We additionally provide evidence, in the form of case studies using well-known benchmark programs, that optimization coaching is an effective means of improving the performance of JavaScript programs. The evaluation also shows that its usage is well within the reach of JavaScript programmers.

Acknowledgments We would like to thank Niko Matsakis, Dave Herman, and Michael Bebenita for discussions and suggestions about the tool's design and development. Kannan Vijayan, Luke Wagner, and Nicolas Pierron helped with the design of the profiler-driven instrumentation. Finally, we thank Matthias Felleisen, Sam Tobin-Hochstadt and Jan Vitek for their comments on previous drafts.

This work was partially supported by Darpa, NSF SHF grants 1421412, 1421652, and Mozilla.

References

1 Craig Chambers and David Ungar. Iterative type analysis and extended message splitting. *Lisp and Symbolic Computation* 4(3), pp. 283–310, 1990.

2 Craig Chambers, David Ungar, and Elgin Lee. An efficient implementation of SELF. In *Proc. OOPSLA*, pp. 49–70, 1989.

3 Tse-Hun Chen, Weiyi Shang, Zhen Ming Jiang, Ahmed E. Hassan, Mohammed Nasser, and Parminder Flora. Detecting performance anti-patterns for applications developed using object-relational mapping. In *Proc. ICSE*, pp. 1001–1012, 2014.

4 Cray inc. *Cray XMT™ Performance Tools User's Guide*. 2011.

5 Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *TOPLAS* 13(4), pp. 451–490, 1991.

6 Dylan Hackers. *Getting Started with the Open Dylan IDE*. 2015. `http://opendylan.org/documentation/getting-started-ide/GettingStartedWithTheOpenDylanIDE.pdf`

7 ECMA International. *ECMAScript® Language Specification*. Standard ECMA-262, 2011.

8 Vyacheslav Egorov. *IRHydra Documentation*. 2014. `http://mrale.ph/irhydra/`

9 Matthew Flatt and PLT. Reference: Racket. PLT Inc., PLT-TR-2010-1, 2010. http://racket-lang.org/tr1/

10 Liang Gong, Michael Pradel, and Koushik Sen. JITProf: Pinpointing JIT-unfriendly JavaScript code. University of California at Berkeley, UCB/EECS-2014-144, 2014.

11 Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. Gprof: a call graph execution profiler. In *Proc. Symp. on Compiler Construction*, pp. 120–126, 1982.

12 Brian Hackett. *JIT Inspector Add-on for Firefox*. 2013. https://addons.mozilla.org/en-US/firefox/addon/jit-inspector/

13 Brian Hackett and Shu-yu Guo. Fast and precise type inference for JavaScript. In *Proc. PLDI*, pp. 239–250, 2012.

14 Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proc. ECOOP*, pp. 21–38, 1991.

15 Urs Hölzle, Craig Chambers, and David Ungar. Debugging optimized code with dynamic deoptimization. In *Proc. PLDI*, pp. 32–43, 1992.

16 Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and detecting real-world performance bugs. In *Proc. PLDI*, pp. 77–88, 2012.

17 Per Larsen, Razya Ladelsky, Jacob Lidman, Sally A. McKee, Sven Karlsson, and Ayal Zaks. Parallelizing more loops with compiler guided refactoring. In *Proc. International Conf. on Parallel Processing*, pp. 410–419, 2012.

18 LispWorks Ltd. *LispWorks© 6.1 Documentation*. 2013.

19 Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Evaluating the accuracy of Java profilers. In *Proc. PLDI*, pp. 187–197, 2010.

20 Cindy Rubio-González, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, David H. Bailey, Costin Iancu, and David Hough. Precimonious: Tuning assistant for floating-point precision. In *Proc. Conf. for High Performance Computing, Networking, Storage and Analysis*, pp. 1–12, 2013.

21 Aibek Sarimbekov, Andreas Sewe, Walter Binder, Philippe Moret, and Mira Mezini. JP2: Call-site aware calling context profiling for the Java virtual machine. *SCP* 79(EST 4), pp. 146–157, 2014.

22 Vincent St-Amour, Sam Tobin-Hochstadt, and Matthias Felleisen. Optimization coaching: optimizers learn to communicate with programmers. In *Proc. OOPSLA*, pp. 163–178, 2012.

23 The SBCL Team. *SBCL 1.0.55 User Manual*. 2012.

24 Guoqing Xu. Resurrector: A tunable object lifetime profiling technique for optimizing real-world programs. In *Proc. OOPSLA*, pp. 111–130, 2013.

25 Guoqing Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, Edith Schonberg, and Gary Sevitsky. Finding low-utility data structures. In *Proc. PLDI*, pp. 174–186, 2010.