HOW TO GENERATE ACTIONABLE ADVICE ABOUT PERFORMANCE PROBLEMS

Vincent St-Amour

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

College of Computer and Information Science
Northeastern University
Boston, Massachusetts

April 2015

NORTHEASTERN UNIVERSITY
GRADUATE SCHOOL OF COMPUTER SCIENCE
Ph.D. THESIS COMPLETION APPROVAL FORM

THESIS TITLE: How to generate actionable advice about performance problems

AUTHOR: Vincent St-Amour

Ph.D. Thesis Approved to complete all degree requirements for the Ph.D. Degree in Computer Science.

[illegible]	24 Apr 2015
Thesis Advisor	*Date*
[illegible]	4/24/15
Thesis Reader	*Date*
[illegible]	4/24/2015
Thesis Reader	*Date*
[illegible]	4/24/2015
Thesis Reader	*Date*
Olin Shivers	2015/4/24
Thesis Reader	*Date*

GRADUATE SCHOOL APPROVAL:

[illegible]	2015/4/24
Director, Graduate School	*Date*

COPY RECEIVED IN GRADUATE SCHOOL OFFICE:

[illegible]	4/24/2015
Recipient's Signature	*Date*

Distribution: Once completed, this form should be scanned and attached to the front of the electronic dissertation document (page 1). An electronic version of the document can then be uploaded to the Northeastern University-UMI website.

ABSTRACT

Performance engineering is an important activity regardless of application domain, as critical for server software as for mobile applications. This activity, however, demands advanced, specialized skills that require a significant time investment to acquire, and are therefore absent from most programmers' toolboxes.

My thesis is that tool support can make performance engineering both accessible and time-efficient for non-expert programmers. To support this claim, this dissertation introduces two novel families of performance tools that are designed specifically to provide actionable information to programmers: *optimization coaches* and *feature-specific profilers*. This dissertation presents blueprints for building tools in these families, and provides examples from tools that I have built.

ACKNOWLEDGMENTS

I would like to thank first and foremost my advisor Matthias Felleisen, who has taught me most of what I know today about the craft of research, writing, and design. He is the best advisor and ally I could have wished for, and I am incredibly grateful for of all the help and guidance he has provided me over the years.

This work would not have been possible without the help of my collaborators, with whom I have worked on the projects described in these pages and others along the way: Leif Andersen, Eric Dobson, Matthias Felleisen, Robby Findler, Matthew Flatt, Shu-yu Guo, Asumu Takikawa, Sam Tobin-Hochstadt, and Neil Toronto.

I would also like to thank my thesis committee for their valuable feedback and guidance: Matthias Felleisen, Kathryn McKinley, Olin Shivers, Sam Tobin-Hochstadt, David Van Horn.

I also owe a great deal of thanks to my colleagues and former colleagues whom I learned an immense amount from, at whom I bounced many ideas—some good, some questionable—and who often found solutions to problems I had been banging my head on. In particular, I would like to thank: Eli Barzilay, Michael Bebenita, Stephen Chang, Ryan Culpepper, Luca Della Toffola, Christos Dimoulas, Tony Garnock-Jones, Dave Herman, Shriram Krishnamurthi, Ben Lerner, Niko Matsakis, Jay McCarthy, Jonathan Schuster, Justin Slepak, Aaron Turon, Jesse Tov, and Jan Vitek.

J'aimerais finalement remercier ma famille, en particulier mes parents, mes grand-parents, mon oncle Jacques, ma tante France et mon oncle François pour avoir toujours encouragé ma curiosité et m'avoir poussé à me dépasser.

CONTENTS

Part I

TOOLING FOR THE DISCERNING PROGRAMMER

1 INTRODUCTION

For many application domains, performance is a critical requirement. High performance programs are responsive to their users, require modest resources, and can be used without concern as system building blocks. Low performance programs, in comparison, may cause users to switch to the competition, impose significant resource requirements, or be unsuitable for integration with other systems.

As it stands, performance engineering requires advanced, specialized skills for both diagnosis and treatment: working knowledge of profilers and disassembly tools, familiarity with compiler internals (especially optimization passes), memory hierarchies, etc. As a result, these skills are beyond the reach of many programmers. These programmers are left at the mercy of their languages, compilers and runtime systems. To make matters worse, many of these hard-earned skills are not transferable across platforms, and risk obsolescence with each new compiler or library update.

Specialized tooling can bring the benefits of performance engineering to a broad audience. The key is to mechanize the knowledge and experience of performance experts. In this spirit, my dissertation supports the following thesis:

> performance tools can use information from the compilation and execution processes to provide easy-to-follow recommendations that help programmers improve the performance of their programs with low effort and little knowledge about low-level details.

The evidence for this thesis comes in the form of two novel classes of performance tools: *optimization coaches* and *feature-specific profilers*. Optimization coaches provide programmers with insight into the optimization process of compilers and help them with recommendations of program changes to enable additional optimizations. Feature-specific profilers report how programs spend their execution time in terms of linguistic features—i.e., features provided by programming languages or libraries—rather than by program components, e.g., lines, procedures, or modules.

Both classes of tools leverage the compilation and execution processes to gather information: optimization coaches by instrumenting compilers' optimizers and feature-specific profilers by observing program execution. Both aim to provide information accessible to non-expert programmers: optimization coaches by providing targeted recommendations and feature-specific profilers by reducing the search space of program changes that programmers must explore.

As part of my thesis work, I have implemented instances of both classes of tools: an optimization coach for Racket (Flatt and PLT 2010) and one for the SpiderMonkey JavaScript (ECMA International 2011) engine, plus one feature-specific profiler for Racket. This dissertation describes these tools in detail and reports on evaluation experiments, both in terms of performance impact and programmer experience.

In addition to these instantiations, this work provides blueprints for building optimization coaches and feature-specific profilers that should apply beyond my specific implementations. The body of this dissertation presents general, language-agnostic techniques and provides specific sketches for certain extensions and instantiations.

1.1 Background

Performance engineering as an activity has existed for almost as long as programming has. In the early days of computing, the extreme performance limitations of computers meant that these two activities went hand in hand. Soon, researchers were developing empirical, systematic approaches to studying program performance (Knuth 1971).

In these early days, accurately predicting the performance of programs remained feasible. Computers were simple, programming languages were low-level, their semantics mapped to that of their host hardware in predictable ways, and system stacks were shallow, with programs mostly running on the bare metal.

Since then, the computing landscape has changed tremendously. Computer architectures have become much more complex and hard to reason about, programmers use high-level languages and frameworks with abstractions that do not have obvious low-level mappings and rely on complex runtime systems, and systems now include layers upon layers of abstraction—hardware abstraction layers, virtual memory, virtualization, etc. All this additional complexity, while invaluable for building large and complex applications, hinders reasoning about performance. Programmers constantly, and accidentally, run afoul of the unwritten rules that govern application performance (Kamp 2010).

Clearly, programmers need tools to help them tame this complexity and recover some ability to reason about—and improve—the performance of their programs. Researchers and practicioners alike have been tackling this problem for decades, with some measure of success. Tools such as profilers (Graham et al. 1982) provide an accounting of the execution costs of programs. Research into performance prediction (Ofelt and Hennessy 2000) aims to improve the programmers' understanding of hardware performance characteristics. Approaches such as vertical profiling (Hauswirth et al. 2004) help programmers see through the layers of their systems. Finally, performance style guides (Fog 2012) provide programmers with generic advice on avoiding pitfalls while writing their programs.

Within this landscape some key contributing factors to program performance are left unaddressed. Specifically, existing tools offer programmers no help in ensuring that compiler optimizations apply to their programs, or in helping them to avoid misusing potentially expensive linguistic features. In our experience and that of Racket programmers, both missed optimizations and feature misuses can significantly degrade program performance. For example, we have observed inlining failures slowing down programs by a factor of two, and overuse of behavioral contracts can account for over 80% of programs' execution time. Others, such as the JavaScript and R communities, have observed similar phenomena.

In addition to not covering these two aspects of program performance, existing tools are not suitable for use by non-expert programmers. Expert programmers (usually) know how to avoid performance pitfalls, and they can use existing diagnosis tools proficiently when they do encounter performance problems. Non-expert programmers, in contrast, may accidentally write underperforming programs, and they may not know how to improve them. Early experiences with the Typed Racket optimizer showed that even experienced programmers struggle to write programs that trigger the optimizations they expect—even when they are explicitly trying to please the optimizer.

These holes in the performance engineering landscape call for the construction of tools that can help programmers, both experts but especially non-experts, navigate the compiler optimization process and avoid misusing expensive linguistic features.

1.2 Scope

Program optimization and tuning is a broad topic. This work does not aim to be the final word on the topic. It rather studies some previously-unaddressed aspects. Specifically, my thesis work is concerned with performance improvements that come from the use and non-use of compiler optimizations and linguistic features. Furthermore, this work is focused on making these performance gains accessible to non-expert programmers, with low effort on their part.

In contrast, the following topics are explicitly out of the scope of this work:

- performance gains via algorithm and data structure selection
- performance gains via system architecture
- automatic application of optimization recommendations
- maximum performance extraction from programs
- performance tooling for performance experts

This is not to say, however, that the ideas and techniques presented in this dissertation are not suitable—or adaptable—for these purposes, only that I will not be considering these topics when describing or evaluating them.

1.3 Dissertation Outline

The bulk of this dissertation is divided in two parts, one for each family of tools it introduces.

Part II: optimization coaching

Part II begins with chapter 2, which presents optimization coaching in general terms and provides an overview of the architecture we propose for such tools.

Chapter 3 provides background on the host compilers of our prototype coaches. Because optimization coaches cooperate with their host compilers, some knowledge of the relevant compilers is necessary to understand the coaches' operation. This chapter presents the Typed Racket, Racket and SpiderMonkey compilers with an eye towards optimization coaching.

Chapter 4 describes techniques for gathering optimization information from compilers' optimizers. It covers instrumentation techniques suitable for ahead-of-time compilers as well as others for just-in-time compilers. To illustrate these ideas, it provides examples taken from our prototypes' host compilers.

Chapter 5 discusses optimization analysis, the process which our prototypes apply to generate actionable programmer-facing optimization reports from raw optimization logs. This chapter presents our techniques and heuristics for that purpose, which we group into four categories: *pruning*, to avoid false positives; *targeting*, to direct programmers' attention to the relevant program fragments; *ranking*, to assist programmers in prioritizing their tuning efforts; and *merging*, to consolidate repetitive information.

Chapter 6 describes the recommendation generation process, by which an optimization coach augments its reports with suggestions of program changes for programmers to follow. This chapter illustrates the process with examples taken from each of our prototypes.

Chapter 7 covers the user interfaces of our prototype coaches. It includes discussion of mechanisms they use to display ranking of recommendations and of user-facing filtering mechanisms.

Chapter 8 presents the results of evaluating our two optimization coaches. Our experiments evaluate three aspects of the coaches: the performance impact of recommendations, the effort required to follow them, and the quality of individual recommendations. Our results show that programmers can achieve significant speedups with low effort by using an optimization coach. Furthermore, speedups achieved by using our JavaScript prototype translate across all major JavaScript engines. This chapter additionally reports on successful user experiences with our Racket prototype.

Chapter 9 sketches how one could apply optimization coaching, and our techniques in particular, to optimizations beyond the ones covered by our prototypes.

Chapter 10 discusses research directions we pursued, but which did not ultimately lead to positive results. We present them to help future researchers avoid going down the same dead ends we did.

Part II concludes with a discussion of related work in chapter 11. It surveys topics ranging from traditional approaches to performance engineering, such as profiling, to recent trends in involving programmers in the optimization process, such as interactive optimization and automatic performance bug detection.

Part III: feature-specific profiling

Part III opens with chapter 12, which introduces the general idea of feature-specific profiling.

Chapter 13 presents the set of linguistic features that our prototype profiler supports. For each feature, it describes its use, explains how it may impact program performance and how its costs may be misunderstood. Finally, this chapter outlines the information our tool provides about each feature.

Chapter 14 explains the basic operation of a feature-specific profiler. It begins by providing background on continuation marks, the stack inspection mechanism which is key to our prototype's instrumentation. It then describes our proposed architecture, and presents the roles of each component: the core sampling profiler and the feature-specific plug-ins.

Chapter 15 describes how to extend the concepts from the previous chapter to leverage the information and structure present in certain features. The additional information from these rich features allows a feature-specific profiler to provide more detailed and targeted information to programmers.

Chapter 16 discusses another extension to the simple model, which gives programmers control over *when* and *where* intrumentation occurs. In turn, this provides control over the extent of the profiler's reports, as well as its overhead.

Chapter 17 presents the results of evaluating our prototype feature-specific profiler. It reports on three aspects of the tool: the impact of following its reports on performance, the effort required to implement feature plug-ins, and the overhead imposed by the profiler's instrumentation and sampling. Our experiments show that the use of a feature-specific profiler can yield significant performance improvements with low effort.

Chapter 18 discusses limitations of the particular approach we propose for feature-specifc profiling. In particular, it provides characterizations of classes of features which our approach does not support.

Chapter 19 sketches how one could apply feature-specific profiling to other language environments. It lists substitutes for the Racket mechanisms used to implement our tool. It also presents other potential instrumentation strategies which may be better suited to other contexts and may also address some of the limitations discussed in chapter 18.

Part III ends with a discussion of work related to feature-specific profiling. It includes comparisons between feature-specific profiling and other approaches to providing actionable views on profiling information.

Part IV offers concluding remarks and closes the dissertation.

1.4 Pronoun Conventions

To reflect the collaborative nature of this work,[1] the rest of this dissertation primarily uses first person plural prounouns. On occasion, when expressing personal beliefs, opinions, and conjectures, the prose uses first person singular pronouns.

1 Matthias Felleisen, Shu-yu Guo, and Sam Tobin-Hochstadt contributed to the optimization coaching work. Leif Andersen and Matthias Felleisen contributed to the feature-specific profiling work.

Part II

OPTIMIZATION COACHING

2 WHEN OPTIMIZERS FAIL

With optimizing compilers, programmers can create fast executables from high-level code. As Knuth (1971) observed, however,

> Programmers should be strongly influenced by what their compilers do; a compiler writer in his infinite wisdom may in fact know what is really good for the programmer and would like to steer him towards a proper course. This viewpoint has some merit, although it has often been carried to extremes in which programmers have to work harder and make unnatural constructions just so the compiler writer has an easier job.

Sadly, the communication between compilers and programmers has not improved in the intervening 40+ years. To achieve high-quality results, expert programmers learn to reverse-engineer the compiler's optimization behavior by looking at the object code generated for their programs. They can then write their programs to take advantage of compiler optimizations. Other programmers remain at the mercy of the compiler, which may or may not optimize their code properly. Worse, if the compiler fails to apply an optimization rule, it fails silently, and the programmer will never know.

Sometimes even experts cannot reliably predict the compiler's behavior. For example, during a recent discussion[1] about the performance of a ray tracer, the authors of the Racket compiler publicly disagreed on whether an inlining optimization had been performed, eventually resorting to a disassembly tool to determine the answer.

Such incidents can happen regardless of language or compiler. The original implementation of Shumway[2] provided an implementation of ActionScript's `Vector.forEach` method which performed poorly. Unbeknownst to its implementors, polymorphism in the method's implementation caused JavaScript engines to generate conservative, poorly-optimized code. Eventually, the Shumway engineers reverse engineered the compiler's opaque optimization decisions and diagnosed the problem. After manually removing the polymorphism, the method performed as expected.

Currently, programmers seeking improved performance from their compilers turn to style guides (Fog 2012; Hagen 2006; Zakas 2010) on how to write programs that play nicely with the optimizer. Such

1 See Racket bug report: `http://bugs.racket-lang.org/old/12518`

2 An open-source implementation of Adobe Flash in JavaScript. `https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Shumway`

static guides cannot offer targeted advice about individual programs and are instead limited to generic, program-agnostic advice.

In this part of the dissertation, we propose an alternative solution to this problem: *optimization coaching*. With a coach, a compiler engages programmers in a dialog about the optimization process, gives them insight into its decisions, and provides actionable recommendations of program changes to enable additional optimizations.

The rest of chapter 2 presents optimization coaching in general terms. Chapter 3 discusses the host compilers for which we prototyped optimization coaches. Chapters 4 through 7 walk through the phases of the optimization coaching process, explaining the techniques we developed for each phase. Chapter 8 presents the results of our experiments to evaluate the effectiveness of our prototypes. Then, chapter 9 sketches how optimization coaching may apply to optimizations beyond those we have studied so far, and chapter 10 lists some coaching techniques which were not successful. Finally, chapter 11 compares our approach to other threads of research that share similar goals.[3]

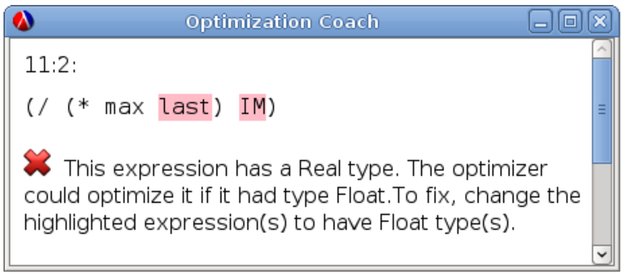
2.1 A Dialog Between Compilers and Programmers

A compiler with an optimization coach "talks back" to the programmer. It explains which optimizations it performs, which optimizations it misses, and suggests changes to the source that should trigger additional optimization. Figure 1 presents our prototype optimization coach for Racket (Flatt and PLT 2010), named Optimization Coach, in action. Here the tool points to a specific expression where the optimizer could improve the performance of the program, along with a particular recommendation for how to achieve the improvement. As the figure shows, a compiler with an optimization coach is no longer a capricious master but a programmer's assistant in search of optimizations.

An optimization coach gathers information during compilation, analyzes it, and presents it to programmers in an easily accessible manner. More concretely, an optimization coach should report two kinds of information:

- *successes*: optimizations that the compiler performed on the current program.
- *near misses*: optimizations that the compiler did *not* apply to the program—either due to a lack of information, or because it may change the program's behavior—but could apply safely if the source program were changed in a certain way.

3 Some of the material in part II of this dissertation appeared in: Vincent St-Amour, Sam Tobin-Hochstadt, and Matthias Felleisen. Optimization coaching. In *Proc. OOPSLA*, 2012. Some material will appear in: Vincent St-Amour and Shu-yu Guo. Optimization coaching for JavaScript. In *Proc. ECOOP*, 2015.



```
11:2:
(/ (* max last) IM)
```

This expression has a Real type. The optimizer could optimize it if it had type Float.To fix, change the highlighted expression(s) to have Float type(s).

Figure 1: Our Racket prototype in action

When possible, near miss reports should come with recommendations on how to change programs to resolve them. These modifications may simplify the compiler's analysis or rule out corner cases, so that the compiler may apply previously missed optimizations.

These recommendations are not required to preserve programs' semantics. In other words, coaches may recommend changes that would be beyond the reach of optimizing compilers, which are limited to semantics-preserving transformations. Worse, compilers are limited to transformations which they can *prove* preserve semantics. Transformations whose proofs require analysis beyond that performed by the compiler are also off-limits. Programmers remain in control and are free to veto any recommendation that would lead to any changes they deem unreasonable.

Including the programmer in the optimization process comes at a cost, however. Evaluating each recommendation takes time and effort. Avoiding false positives—recommendations that result in little performance benefit or that the programmer does not want to follow—and providing easy-to-follow recommendations are the main challenges an optimization coach faces.

2.2 Architecture

An optimization coach works by gathering information from the compiler's optimizer, processing it, and presenting the results to programmers. Specifically, an optimization coach operates in four main steps:

1. *Compiler instrumentation* (chapter 4): 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 does not apply. These logs include enough information to reconstruct the optimizer's reasoning post facto.

2. *Optimization analysis* (chapter 5): 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 events recorded in the logs. The coach optionally integrates information from other sources such as profilers to perform this task.

3. *Recommendation generation* (chapter 6): From the near miss reports, it 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.

4. *User interface* (chapter 7): The optimization coach presents reports and recommendations to programmers. The interface leverages optimization analysis metrics to visualize the coach's rankings of near misses or display high-estimated-impact recommendations only. The programmer reviews the recommendations and applies them if desired.

2.3 Prototypes

The following chapters illustrate optimization coaching with examples from the two instantiations that we developed: one for Racket and one for the SpiderMonkey[4] JavaScript (ECMA International 2011) engine, which is used by the Firefox[5] web browser.

The Racket prototype covers both the Racket and Typed Racket compilers. It is available as a DrRacket (Findler et al. 2002) plug-in and is[6] part of the main Racket distribution.[7] Its source is publicly available.[8] The tool is routinely used[9] [10] by Racket programmers to diagnose underperforming programs.

The SpiderMonkey prototype described here is currently available in source form[11] and requires a custom version of SpiderMonkey.[12] There is ongoing[13] work by Shu-yu Guo, Kannan Vijayan and Jordan Santell to include a version of the tool as part of the Firefox developer tools suite.[14]

4 `https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey`
5 `https://www.mozilla.org/en-US/firefox/`
6 At the time of this writing.
7 Available from: `http://racket-lang.org`
8 `https://github.com/stamourv/optimization-coach`
9 `http://blog.jverkamp.com/2013/04/16/adventures-in-optimization-re-typed-racket/`
10 `http://lists.racket-lang.org/users/archive/2015-January/065356.html`
11 `https://github.com/stamourv/jit-coach`
12 `https://github.com/stamourv/gecko-dev`
13 At the time of this writing.
14 `https://bugzilla.mozilla.org/show_bug.cgi?id=1143804`

3 HOST COMPILERS

Optimization coaches must interact closely with their host compiler to gather information about the optimization process. This chapter provides background about the three compilers we instrumented—Typed Racket, Racket, and SpiderMonkey—to provide context for the techniques described in the following chapters. In addition, this chapter describes the optimizations supported by our prototypes and illustrates their operation with examples.

3.1 The Typed Racket Compiler

The Typed Racket compiler (Tobin-Hochstadt et al. 2011) is a research compiler that compiles Typed Racket programs to Racket programs. It uses core Racket programs as a high-level intermediate representation. This representation is close to actual source programs, and most source-level information is still present. Typed Racket performs optimizations as source-to-source transformations, the most important of which are type-driven specializations of generic operations.

For example, a generic use of the multiplication function can be specialized if both its arguments are floating-point numbers, e.g., in definitions such as this one:

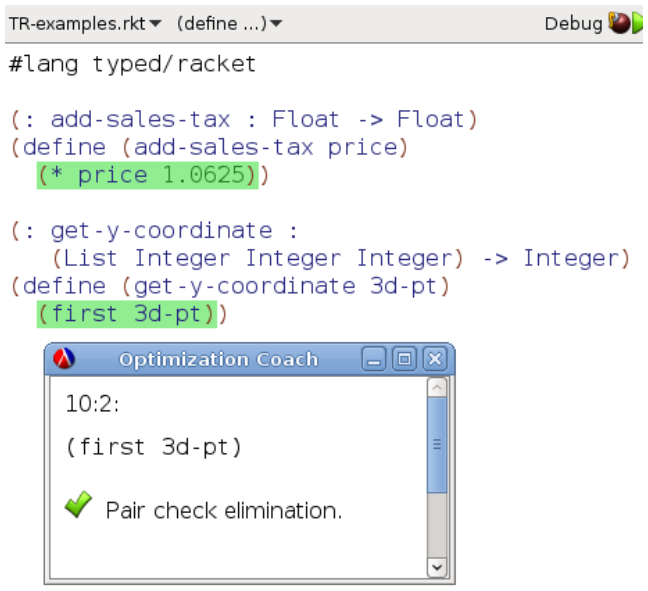
```
(: add-sales-tax : Float -> Float)
(define (add-sales-tax price)
  (* price 1.0625))
```

Typed Racket's type system validates that this multiplication always receives floating-point numbers as its arguments, and its compiler specializes it thus:

```
(: add-sales-tax : Float -> Float)
(define (add-sales-tax price)
  (unsafe-fl* price 1.0625))
```

The resulting code uses the `unsafe-fl*` operator, which operates on floating-point numbers only and has undefined behavior on other inputs. This operation is unsafe in general, but the typechecker proves it safe in this specific case. Similarly, when type information guarantees that a list is non-empty, the compiler may elide checks for null:

```
(: get-y-coordinate :
   (List Integer Integer Integer) -> Integer)
(define (get-y-coordinate 3d-pt)
  (first 3d-pt))
```



```
#lang typed/racket

(: add-sales-tax : Float -> Float)
(define (add-sales-tax price)
  (* price 1.0625))

(: get-y-coordinate :
   (List Integer Integer Integer) -> Integer)
(define (get-y-coordinate 3d-pt)
  (first 3d-pt))
```

```
10:2:

(first 3d-pt)

Pair check elimination.
```

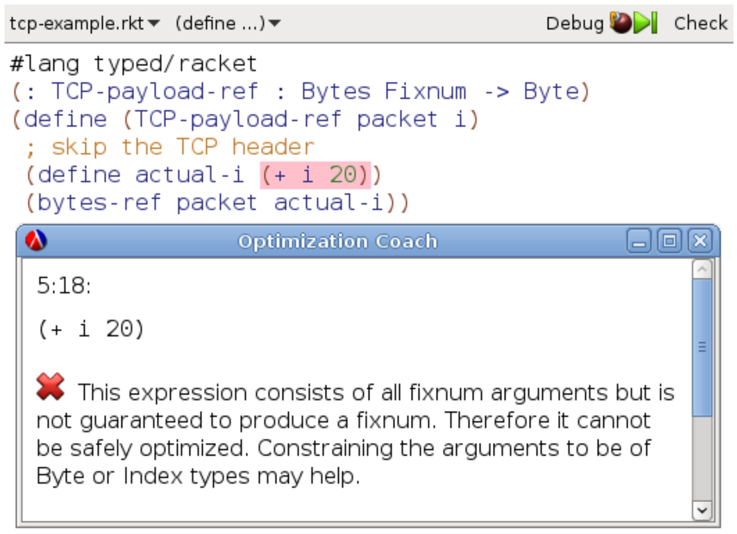
Figure 2: Optimization Coach's analysis of our example functions (with focus on `get-y-coordinate`'s body)

In this case, the type specifies that the input is a three-element list. Hence taking its first element is always safe.

Our Racket prototype—Optimization Coach—reports successes and near misses for these type-driven specialization optimizations. Figure 2 shows how it informs the programmer that type specialization succeeds for the above functions. When programmers click on the colored region, the tool brings up a new window with extra information. Recommendations also become available when the optimizer cannot exploit type information. Consider the following function, which indexes into a TCP packet's payload, skipping the headers:

```
(: TCP-payload-ref : Bytes Fixnum -> Byte)
(define (TCP-payload-ref packet i)
  ; skip the TCP header
  (define actual-i (+ i 20))
  (bytes-ref packet actual-i))
```

This program typechecks, but the optimizer cannot eliminate the overhead from the addition. Racket's addition function implicitly promotes results to bignums on overflow, which may happen for the addition of `20` to a fixnum. Therefore, the Typed Racket compiler cannot safely specialize the addition to fixnum-only addition. Optimization Coach detects this near miss and reports it, as figure 3 shows. In addition, Optimization Coach suggests a potential solution, namely, to restrict the argument type further, ensuring that the result of the addition stays within fixnum range.



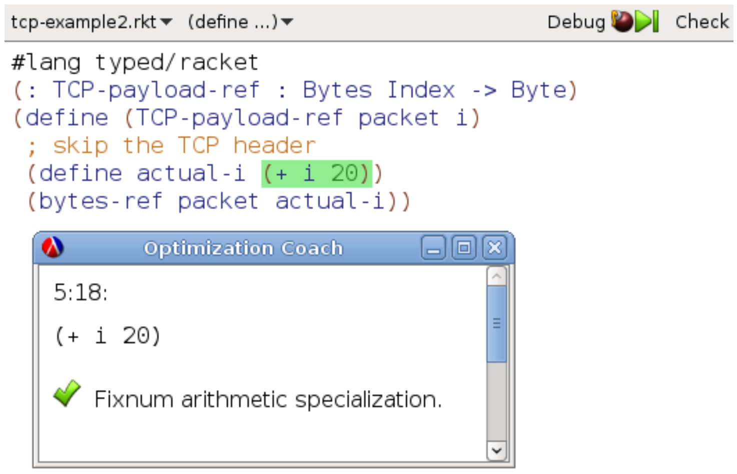
```
#lang typed/racket
(: TCP-payload-ref : Bytes Fixnum -> Byte)
(define (TCP-payload-ref packet i)
 ; skip the TCP header
 (define actual-i (+ i 20))
 (bytes-ref packet actual-i))
```

5:18:

```
(+ i 20)
```

This expression consists of all fixnum arguments but is not guaranteed to produce a fixnum. Therefore it cannot be safely optimized. Constraining the arguments to be of Byte or Index types may help.

Figure 3: Optimization near miss involving fixnum arithmetic



```
#lang typed/racket
(: TCP-payload-ref : Bytes Index -> Byte)
(define (TCP-payload-ref packet i)
 ; skip the TCP header
 (define actual-i (+ i 20))
 (bytes-ref packet actual-i))
```

5:18:

```
(+ i 20)
```

Fixnum arithmetic specialization.

Figure 4: Confirming that the optimization failure is now fixed

Figure 4 shows the result of following Optimization Coach's recommendation. Once the argument type is `Index`,[1] the optimizer inserts a fixnum addition for `+`, and Optimization Coach confirms the optimization.

Let us consider another optimization failure:

```
(define IM   139968)
(define IA     3877)
(define IC    29573)

(define last 42)
(define max  156.8)
(define (gen-random)
  (set! last (modulo (+ (* last IA) IC) IM))
  (/ (* max last) IM))
```

This code implements Lewis et al.'s (1969) pseudo-random number generator, using mixed-type arithmetic in the process. Racket allows mixing integers and floating-point numbers in arithmetic operations. This combination usually results in the coercion of the integer argument to a floating-point value and the return of a floating-point number. Therefore, the author of this code may expect the last expression of `gen-random` to be specialized for floating-point numbers.

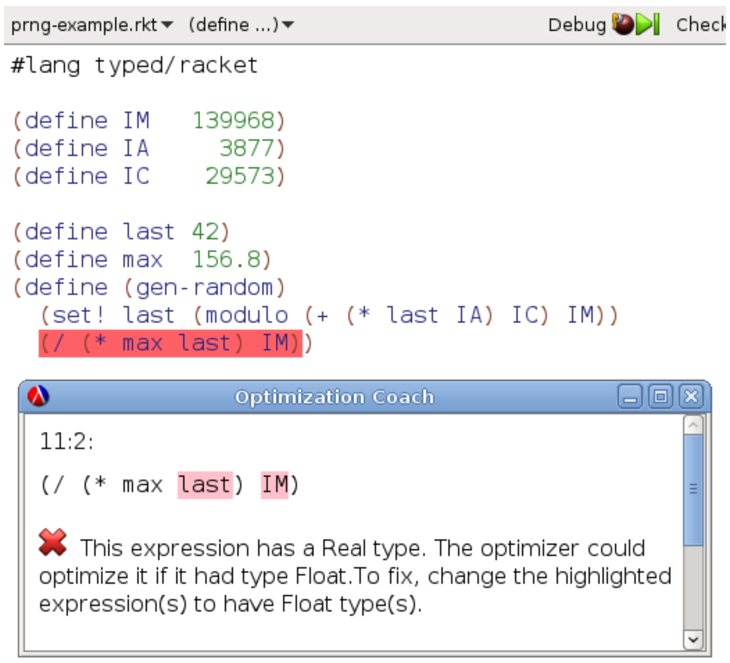
Unbeknownst to most programmers, however, this code suffers from a special case in Racket's treatment of mixed-type arithmetic. Integer-float multiplication produces a floating point number, unless the integer is `0`, in which case the result is the integer `0`. Thus the result of the above multiplication is a floating-point number most of the time, but not always, making floating-point specialization unsafe.

The Typed Racket optimizer knows this fact (St-Amour et al. 2012b) but most programmers fail to think of it when they program. Hence, this optimization failure may surprise them and is thus worth reporting. Figure 5 shows how Optimization Coach explains this failure. The programmer can respond to this recommendation with the insertion of explicit coercions:

```
(define (gen-random)
  (set! last (modulo (+ (* last IA) IC) IM))
  (/ (* max (exact->inexact last))
     (exact->inexact IM)))
```

Optimization Coach confirms that this change enables further optimization.

1 A fixed-width integer type whose range is more restricted than `Fixnum` and is bounded by the maximum heap size.



```
#lang typed/racket

(define IM   139968)
(define IA     3877)
(define IC    29573)

(define last 42)
(define max  156.8)
(define (gen-random)
  (set! last (modulo (+ (* last IA) IC) IM))
  (/ (* max last) IM))
```

```
11:2:
(/ (* max last) IM)
```

This expression has a Real type. The optimizer could optimize it if it had type Float.To fix, change the highlighted expression(s) to have Float type(s).

Figure 5: Optimization failure involving mixed-type arithmetic

3.2 The Racket Compiler

The Racket compiler is a mature ahead-of-time optimizing compiler that has been in development for 20 years. It compiles Racket programs to a custom bytecode, which the Racket virtual machine then translates to machine code just in time, at the first call to each function. This runtime code generation does not perform significant additional optimization.

The compiler is written in C and consists of several passes. The first pass, our focus here, features a large number of optimizations, including inlining as well as constant propagation, copy propagation, and constant folding. Subsequent passes perform closure conversion (Appel and Jim 1989) and lambda lifting (Johnsson 1985).

Our Racket prototype supports inlining, which is the most important of these optimizations. The Racket inliner is based on a design by Serrano (1997) and is a much more sophisticated optimizer than Typed Racket's. Its decision process uses a variety of heuristics to estimate whether inlining would be beneficial, the main one being *inlining fuel*. To avoid code size explosions, the inliner allocates a fixed amount of inlining fuel to each call site of the original program. Inlining a function at a given call site consumes an amount of fuel proportional to the size of the inlinee. Once a call site runs out of fuel, no further inlining is possible.

While effective in practice, these heuristics make the inlining process opaque to programmers—predicting whether a given function will be inlined is almost impossible. More complex inlining heuristics, such as *further benefit* (Sewe et al. 2011), would only exacerbate this problem.

Optimization Coach provides two kinds of information about inlining transformations:

- which functions are inlined and how often;
- which functions are not inlined, and why.

To illustrate, let us consider the function `in` from figure 6, that checks whether a point `p` is within a shape `s`. A shape can be either a circle, in which case the fuction calls to the `inC` helper function; a square, in which case it calls `inS`; or a union, in which case it calls itself recursively on the union's components. The author of the program introduced these functions to make the surrounding program readable, but it would be unfortunate if this factoring were to result in extra function calls, especially if they were to hurt the program's performance.

As figure 7 shows, the Racket inliner does inline `in` at most of its call sites. In contrast, the `inC` function is not inlined to a satisfactory level, as evidenced by the red highlight in figure 8. This may be indicative of low-hanging optimization fruit. One way to resolve the issue is to use inlining pragmas such as `define-inline` or `begin-encourage-inline`. When we follow this advice, Optimization Coach confirms that the inlining is now successful. Breaking up the function into smaller pieces might also work.

3.3 The SpiderMonkey JavaScript Engine

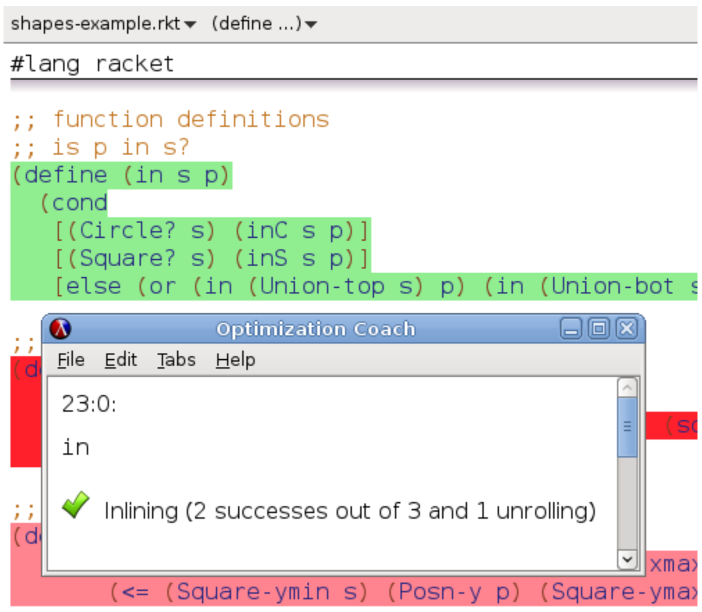
Like other modern JavaScript engines,[2] [3] [4] SpiderMonkey is a multi-tiered engine that uses type inference (Hackett and Guo 2012), type feedback (Chambers and Ungar 1990), and optimizing just-in-time compilation (Chambers et al. 1989) based on SSA form (Cytron et al. 1991), a formula proven to be well suited for JavaScript's dynamic nature. Specifically, it has three tiers: the interpreter, the baseline just-in-time (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,[5] the baseline JIT compiles methods to native code. Once methods become hotter still and reach a second threshold,[6] Ion com-

2 `https://developers.google.com/v8/intro`
3 `http://www.webkit.org/projects/javascript/`
4 `http://msdn.microsoft.com/en-us/library/aa902517.aspx`
5 At the time of this writing, 10.
6 At the time of this writing, 1000.

```
; is p in s?
(define (in s p)
  (cond
   [(Circle? s) (inC s p)]
   [(Square? s) (inS s p)]
   [else (or (in (Union-top s) p)
             (in (Union-bot s) p))]))

; is p in c?
(define (inC c p)
  (define m (Circle-center c))
  (<= (sqrt (+ (sqr (- (Posn-x m) (Posn-x p)))
               (sqr (- (Posn-y m) (Posn-y p)))))
      (Circle-r c)))

; is p in s?
(define (inS s p)
  (and (<= (Square-xmin s) (Posn-x p) (Square-xmax s))
       (<= (Square-ymin s) (Posn-y p) (Square-ymax s))))
```

Figure 6: Functions checking inclusion of points in geometric shapes

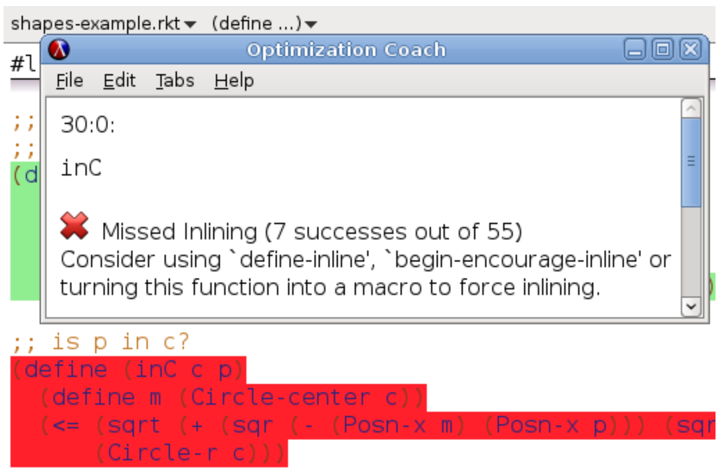


```
#lang racket

;; function definitions
;; is p in s?
(define (in s p)
  (cond
   [(Circle? s) (inC s p)]
   [(Square? s) (inS s p)]
   [else (or (in (Union-top s) p) (in (Union-bot s
```

```
23:0:

in

✓ Inlining (2 successes out of 3 and 1 unrolling)
```

```
      (<= (Square-ymin s) (Posn-y p) (Square-yma
```

Figure 7: Optimization Coach confirming that inlining is happening (with focus on `in`)



```
30:0:

inC

Missed Inlining (7 successes out of 55)
Consider using `define-inline', `begin-encourage-inline' or
turning this function into a macro to force inlining.
```

```
;; is p in c?
(define (inC c p)
  (define m (Circle-center c))
  (<= (sqrt (+ (sqr (- (Posn-x m) (Posn-x p))) (sqr
      (Circle-r c))))
```

Figure 8: The `inC` function, failing to be inlined (with focus on `inC`)

piles 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.

3.3.1 *The IonMonkey Optimizer*

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 purposes, 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 accesses 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 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 (Hölzle et al. 1992). 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 safe code generated by the baseline JIT.

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 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.

3.3.2 *Optimization Corpus*

Conventional wisdom among JavaScript compiler engineers points to property and element accesses as the most important operations to optimize. For this reason, our prototype focuses on these two classes of operations. The rest of this section describes the relevant optimizations with an eye towards optimization coaching.

3.3.2.1 *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 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[7] are observed to flow to `o`, Ion can emit a polymorphic inline cache (PIC) (Hölzle et al. 1991). 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 domain object model (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.

3.3.2.2 *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

7 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 prescribed for plain objects by the ECMAScript standard.

require contiguous chunks of memory addressable by offset. Semantically, JavaScript arrays are plain objects that map indices—*string* representation 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 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 dissertation, 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]`.

3.3.3 *A Near Miss Walkthrough*

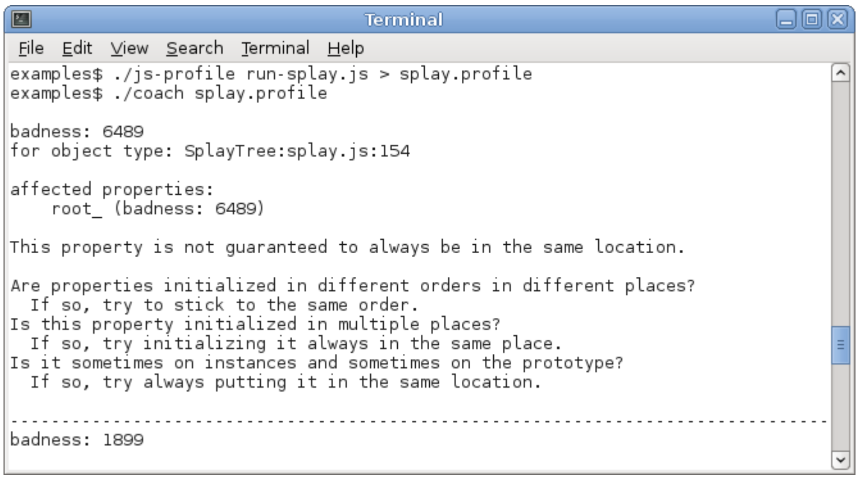
To make the above discussion more concrete, this section presents an example near miss and discusses the output of our SpiderMonkey prototype for it. Consider the excerpt from a splay tree implementation in figure 9. The `isEmpty` method may find the `root_` property either on `SplayTree` instances (if 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

```
// 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 instance or on prototype
  return !this.root_;
};
```

Figure 9: Splay tree implementation with an optimization near miss

instead generates code that can handle both of them. The generated code is thus much slower than necessary.

Figure 10 shows the coach's diagnosis and recommendations of program changes that may resolve this near miss. 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, in contrast, are free to do so and may rewrite the program to the version from figure 11, which consistently stores the property on instances, and does not suffer from the previous near miss.



```
examples$ ./js-profile run-splay.js > splay.profile
examples$ ./coach splay.profile

badness: 6489
for object type: SplayTree:splay.js:154

affected properties:
    root_ (badness: 6489)

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 always putting it in the same location.

--------------------------------------------------------------------------------
badness: 1899
```

Figure 10: Excerpt from the coaching report for a splay tree implementation

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

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

SplayTree.prototype.isEmpty = function() {
  // property always on instances
  return !this.root_;
};
```

Figure 11: Improved splay tree constructor, without near miss

4 OPTIMIZER INSTRUMENTATION

In order to explain an optimizer's results to programmers, a coach must discover what happens during optimization. One option is to reconstruct the optimizations via analysis of the compiler's output. Doing so would mimic the actions of highly-expert programmers with a thorough understanding of the compiler—with all their obvious disadvantages. In addition, it would forgo all the information that the compiler generates during the optimization phase.

Our proposed alternative is to equip the optimizer with instrumentation that gathers information as optimizations are performed or rejected. Debugging tools that explain the behavior of compilers or runtime systems (Clements et al. 2001; Culpepper and Felleisen 2010) have successfully used similar techniques in the past. Furthermore, optimization information obtained directly from the compiler has the advantage of being ground truth. A tool that attempts to reverse-engineer the optimization process risks imperfectly modeling the optimizer, or falling out of sync with its development.

The goal of the instrumentation code is to generate a complete log of the optimization decisions that the compiler makes as it compiles a given program. For this purpose, instrumentation code should record both *optimization successes*, i.e., optimizations that were successfully applied and *optimization failures*, i.e., optimizations that were attempted but failed. While most of these failures are of no interest to the programmer, curating the log is left to a separate phase.

The rest of this chapter describes the instrumentation for Typed Racket, Racket and SpiderMonkey.

4.1 The Typed Racket Optimizer

As mentioned, the Typed Racket compiler performs source-to-source optimizations. These optimizations are implemented using pattern matching and templating, e.g.:

```
(pattern
  (+ f1:float-expr f2:float-expr)
  #:with opt
  #'(unsafe-fl+ f1.opt f2.opt))
```

Each optimization is specified with a similar `pattern` construct. The only factors that affect whether a term is optimized are its shape—in this case, an AST that represents the application of the addition function—its type, and the type of its subterms.

Instrumenting the optimizer to log optimization successes is straightforward: we add a logging statement to each `pattern` describing which optimization takes place, the code involved, its source location, and the information that affects the optimization decision (the shape of the term, its type and the type of its subterms):

```
(pattern
  (+ f1:float-expr f2:float-expr)
  #:with opt
  (begin
   (log-optimization "binary float addition" this-syntax)
   #'(unsafe-fl+ f1.opt f2.opt)))
```

When translating the `add-sales-tax` and `get-y-coordinates` functions from figure 2, (page 16), the instrumented optimizer generates this log:

```
TR opt: TR-examples.rkt 5:2 (* price 1.0625)
  -- Float Float -- binary float multiplication
TR opt: TR-examples.rkt 10:2 (first 3d-pt)
  -- (List Integer Integer Integer)
  -- pair check elimination
```

It thus confirms that it applies the optimizations mentioned above. To log optimization failures, we add an additional `pattern` form that catches all non-optimized additions and does not perform any optimizations:

```
(pattern
  (+ n1:expr n2:expr)
  #:with opt
  (begin
   (log-optimization-failure "generic addition" this-syntax)
   this-syntax)) ; no change
```

To avoid generating excessive amounts of superfluous information, we restrict failure logging to terms that could at least conceivably be optimized. For instance, we log additions that are not specialized, but we do not log all non-optimized function applications.

As in the optimization success case, the logs contain the kind of optimization considered, the term involved, its type, and the types of its subterms. The following log entry shows evidence for the failed optimization in the `TCP-payload-ref` function of figure 3 (page 17):

```
TR opt failure: tcp-example.rkt 5:18 (+ i 20)
-- Fixnum Positive-Byte -- generic addition
```

Overall, the instrumentation is fairly lightweight. Each optimization clause is extended to perform logging in addition to optimizing; a few catch-all clauses log optimization failures. Most importantly, the structure of the optimizer is unchanged.

4.2 The Racket Inliner

As discussed in section 3.2, the Racket inliner uses multiple sources of information when making decisions, including but not limited to the size of inlining candidates, the remaining amount of inlining fuel, etc. This makes it unrealistic to log all the factors that contribute to any given optimization decision.

An additional constraint is that the inliner is a complex program, and instrumentation should not increase its complexity. Specifically, instrumentation should not add new paths to the inliner. Concretely, this rules out the use of catch-all clauses.

Finally, the Racket inliner is deep enough in the compiler pipeline that most source-level information is unavailable in the internal representation. As a result, it becomes difficult to correlate optimization decisions with the original program. These constraints on the available information are representative of production optimizers.

To record optimization successes, we identify all code paths that trigger an inlining transformation and, on each of them, add code that logs the name of the function being inlined and the location of the original call site. It is worth noting that precise information about the original call site may not be available; the call site may also have been introduced by a previous optimization pass, in which case a source location would be meaningless. In general, though, it is possible to locate the call site with at least function-level granularity; that is, the logger can usually determine the function body where the call site is located.

Inlining is a sufficiently general optimization that it could conceivably apply almost anywhere. This makes defining and logging optimization failures challenging. Since our ultimate goal is to enumerate a list of optimization near misses, we need to consider only optimization failures that directly contribute to near misses. For example, the failure of a large function to be inlined is an optimization failure that is unlikely to be linked to a near miss.

Consequently we consider as optimization failures only cases where the compiler considers inlining, but ultimately decides against it. We identify the code paths where inlining is decided against and add logging to them. As in the case of inlining successes, we log the name of the candidate and the call site. In addition, we also log the cause of failure. The most likely cause of failure is the inliner running out of fuel. If the inliner runs out of fuel for a specific instance, it is not performed and the optimization fails. For these kinds of failures, we also log the size of the function being considered for inlining, as well as the remaining fuel.

Figure 12 shows an excerpt from the inliner log produced when compiling the *binarytrees* benchmark from section 8.1.1.

```
mzc: no inlining, out of fuel #(for-loop 41:6)
   in #(main 34:0) size=77 fuel=8
mzc: inlining #(loop 25:2) in #(check 24:0)
mzc: inlining #(loop 25:2) in #(check 24:0)
mzc: no inlining, out of fuel #(loop 25:2)
  in #(check 24:0) size=28 fuel=16
mzc: inlining #(check 24:0) in #(for-loop 46:1)
mzc: no inlining, out of fuel #(check 24:0)
  in #(for-loop 46:18) size=31 fuel=24
```

Figure 12: Inliner log from the *binarytrees* benchmark

4.3 The IonMonkey Optimizer

The IonMonkey optimizer radically differs from the Racket and Typed Racket optimizers. As far as instrumentation is concerned, the main difference is that SpiderMonkey is a JIT compiler.[1] 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 ahead-of-time (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.

To address these challenges, 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. Like our Racket prototype, our SpiderMonkey prototype records optimization decisions and context information as the optimizer processes code. Where our Racket prototype emits that information on the fly, our SpiderMonkey prototype instead gathers all the informa-

1 As mentioned in section 3.2, Racket does have a just-in-time code generator, but it does not perform significant optimizations.

tion pertaining to a given invocation of the compiler, encodes it as a single profiler event and emits it all at once. The instrumentation code executes only when the profiler is active; its overhead is therefore almost entirely pay-as-you-go. The instrumentation code records information that 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, the successful attempts, the unsuccessful ones, etc.

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 in chapter 5. This additional piece of meta-information has negligible overhead and is present only when the profiler is active.

Figure 13 shows an excerpt from the optimization logs produced during the compilation of one of the functions in the *Richards* benchmark from section 8.2.1.

```
optimization info for compile #13

optimizing getprop currentTcb richards.js:226:2
  obj types: object[1] Scheduler:richards.js:94
  trying definite slot
    success

optimizing getprop run richards.js:332:2
  obj types: object[4] HandlerTask:richards.js:454
                       WorkerTask:richards.js:419
                       DeviceTask:richards.js:391
                       IdleTask:richards.js:363
  trying definite slot
    failure, 4 possible object types
  trying inline access
    failure, access needs to go through the prototype
  trying emitting a polymorphic cache
    success

optimizing getprop currentTcb richards.js:187:6
  obj types: object[1] Scheduler:richards.js:94
  trying definite slot
    success

optimizing setprop state richards.js:313:2
  obj types: object[1] TaskControlBlock:richards.js:255
  property types: int
  value types: Int32
  trying definite slot
    failure, property not in a fixed slot
  trying inline access
    success
```

Figure 13: Excerpt from optimization logs for the *Richards* benchmark

5 OPTIMIZATION ANALYSIS

To avoid overwhelming programmers with large numbers of low-level optimization failure 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 to be approved by programmers.

A coach uses four main classes of techniques for that purpose: pruning, targeting, ranking and merging. Some techniques apply uniformly regardless of optimization or compiler, while others make sense only for a subset. To assist its own strategies and heuristics, a coach can also incorporate information from other performance tools during this phase. In particular, profiling information can serve to enhance a coach's analyses.

This chapter discusses the techniques used by our prototypes in general terms. An in-depth presentation of full details would be out of place here; I invite the interested reader to consult our prototypes' source code instead.

5.1 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 those due to intentional design choices would be a waste of programmer time. Coaches therefore use heuristics to remove optimization failures from the coach's reports. The remaining optimization failures constitute near misses and are further refined via merging.

5.1.1 *Incomprehensible Failure Pruning*

Some optimization failures are considered *incomprehensible* because they affect code that is not present in the original program. Such code might be introduced by macro-expansion or earlier compiler passes, and it is thus not under the control of programmers. Reporting optimization failures that originate from this kind of code would be wasteful. Optimization coaches should prune such failures from their optimization logs.

5.1.2 *Irrelevant Failure Pruning*

Other optimization failures are considered *irrelevant* because the chosen semantics is likely to be the desired one. For example, reporting that an addition could not be specialized to floating-point numbers is irrelevant if the addition is used on integers only. Presenting recommendations that programmers are most likely to reject is unhelpful, so a coach should prune these kinds of reports.

One way of detecting such failures is to compute *optimization proximity* and prune failures that are not "close enough" to succeeding.

Some optimizations also lend themselves to domain-specific relevance heuristics. For example, optimization tactics in IonMonkey often include strategies that 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. These tactics are expected to fail most of the time. Such failures are irrelevant unless the value is in fact a string or a constant, respectively.

Irrelevant failures are not usually symptomatic of performance issues and should be pruned from the logs.

5.1.3 *Optimization Proximity*

Optimization proximity measures how close an optimization is from happening and is defined on an optimization-by-optimization basis. For a particular optimization, it might be derived from the number of program changes that would be necessary to trigger the optimization of interest. For another, it might correspond to the number of prerequisites of the optimization that are not satisfied. Log entries with close proximity are retained, others are pruned from the log.

Our Racket prototype computes optimization proximity for type specialization optimizations. To trigger specialization, all the arguments of a generic operation must be convertible to the same type, and that type must be one for which a specialized version of the operation is available. In this case, we define optimization proximity to be the number of arguments whose types would need to change to reach a state where optimization could happen.[1]

For example, addition can be specialized for `Float`s, but not for `Real` numbers. Therefore, the combination of argument types

```
(+ Float Real)
```

is 1-close to being optimized, while this one

```
(+ Real Real)
```

is 2-close, and thus further from being optimized. Only optimization failures within a specific proximity threshold are kept. Optimization

1 This can be seen as a "type edit distance".

```
(: sum : (Listof Real) -> Real)
(define (sum list-of-numbers)
  (if (null? list-of-numbers)
      0
      (+ (first list-of-numbers)
         (sum (rest list-of-numbers)))))
```

Figure 14: A generic sum function

Coach uses a threshold of 1 for this optimization, which has worked well in practice.

The sum function in figure 14 uses + in a generic way. The Typed Racket optimizer does not specialize this code, creating the following log entry:

```
TR opt failure: sum-example.rkt 5:6
(+ (first list-of-numbers) (sum (rest list-of-numbers)))
-- Real Real -- generic addition
```

The addition has a 2-close measure and is therefore ignored by Optimization Coach.

It would be possible to define optimization proximity for inlining failures, computing the metric based on the amount of remaining inlining fuel and the size of the potential inlinee, for example. In practice, however, this has not proven necessary; the merging techniques described in section 5.4 provide good enough results for inlining reports, which makes pruning less necessary.

Chapter 9 proposes optimization proximity metrics for various well-known optimizations beyond the scope of our prototypes.

5.1.4 *Harmless Failure Pruning*

Harmless optimization failures are associated with opportunities that are missed because other optimizations are performed. Harmless failures are pruned from the logs.

Consider a loop that is unrolled several times—eventually, unrolling must stop. This final decision is still reported in the log as an optimization failure, because the optimizer considers unrolling further but decides against it. It is a harmless failure, however, because loop unrolling cannot go on forever.

Since Racket is a mostly-functional language, loops are expressed as tail recursive functions; therefore, function inlining also expresses loop unrolling. An important role of optimization analysis for the

Racket inliner is to determine which log entries correspond to loop unrolling and which correspond to “traditional” inlining. This analysis can only be performed post-facto because the same code paths apply to both forms of inlining.

Separating unrolling from inlining has two benefits. First, it allows the coach to prune unrolling failures, as they are usually harmless. Second, it prevents unrolling successes from counting as inlining successes. A function that is unrolled but never inlined is problematic and should be reported to programmers. This case can be distinguished from that of a function that is sometimes inlined and sometimes not only if unrollings and inlinings are considered separately.

Similarly, optimizations that cancel each other can be the source of harmless failures. For example, it is possible to apply loop interchange twice, with the second transformation restoring the original loop ordering. To avoid oscillating forever, the optimizer should not apply the second interchange, which logging may report as a failure.

5.1.5 *Partial Success Short-Circuiting*

When faced with an array of optimization options, some compilers such as IonMonkey rely on optimization tactics to organize them. While a coach 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 all of a tactic’s options together.

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 property accesses cannot be optimized as well as monomorphic ones; 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 our SpiderMonkey prototype 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 SpiderMonkey 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.

5.1.6 *Profiling-Based Pruning*

Optimization failures in infrequently executed, "cold" code do not affect program performance as much as pitfalls in frequently executed, "hot" code. Recommendations with low expected performance impact offer a low return on programmer time investment. When profiling information is available, an optimization coach should use it to prune failures from cold code.

To enable this kind of pruning, a coach must compute a partition of code units (e.g., functions) into hot and cold units from profiling information (i.e., mappings from code units to cost). Our Racket prototype accomplishes this goal as follows.

After sorting functions in the profile in decreasing order of time cost, our prototype adds functions to the hot set starting from the most expensive, until functions in the set account for 90% of the program's total running time. All other functions are considered cold. This threshold was determined experimentally and produces small sets in practice,[2] which makes this kind of pruning quite effective. This partitioning scheme also puts an upper bound on the performance impact of pruned near misses via Amdahl's law; fixing them cannot improve program performance by more than 10%.

Our SpiderMonkey prototype does not rely on this kind of pruning. It instead uses profiling information for ranking (see section 5.3.2), then prunes based on ranking. Combining these two uses of profiling information, as our Racket prototype does, would be a straightforward extension.

2 This is consistent with the Pareto principle, i.e., 10% of a program's functions account for 90% of its running time.

5.2 Targeting

For the reports of an optimization coach to be actionable, they must point to the program location where their fix is likely to go. In many cases, an optimization failure and its solution will occur in the same location. In other cases, however, failures are non-local; the compiler may fail to optimize an operation in one part of the program because of properties of a different part of the program.

While it is not always possible to determine the location of fixes with perfect accuracy, it is possible to get good results in practice using heuristics based on optimization-specific domain knowledge and the specific failure causes of individual near misses. This section describes our heuristics.

5.2.1 *Type-Driven Specialization*

Optimization failures for specialization optimizations are always due to the types of their arguments. Therefore, these failures are local; adding assertions or coercions for problematic arguments can always solve the problem. A coach should therefore have reports point to the location of the failure.

Often, however, there may be other solutions that avoid assertions or coercions—and thus avoid run-time overhead—by changing types upstream from the relevant operation. Our Racket prototype does not infer these solutions, leaving that inference to programmers. This could potentially be addressed using program slicing and remains an open research question.

5.2.2 *Inlining*

Inlining failures, unlike specialization failures, are non-local. They are caused by properties of the inlining candidate, e.g., function size or mutability. The only way programmers can control the inlining process is through the definition of inlining candidates. Therefore, any information that would help the user get the most out of the inliner has to be explained in terms of the definition sites.

Some inlining failures are genuinely caused by the inlining site's context. For example, the inliner running out of fuel depends on the context. Such failures, however, are a result of the optimization working as intended and are therefore harmless.

5.2.3 *Property Access and Assignment*

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 from the property access considered for optimization.

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. This kind of failure can be solved only by changing the operation or its context, and the nature of the solution depends on the operation itself.

For a given near miss, the coach needs to determine 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. The rest of this section briefly describes two of our 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 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 considers it an operation near miss.

5.2.4 *Element Access and Assignment*

Like type specialization, element access and assignment operations suffer only from local optimization failures, which are due to properties of their arguments. Our SpiderMonkey prototype therefore reports near misses at the site of the operation.

5.3 Ranking

Some optimization failures have a larger impact on program performance than others. To be useful, a coach must rank its reports based on their expected performance impact to allow programmers to prioritize their responses. Our coaches compute a *badness* metric for each near miss. Badness scores are dimensionless quantities that estimate the performance impact of near misses. To compute them, our prototypes use a combination of static estimates based on the optimization and, when available, profiling information.

5.3.1 *Static Badness*

The initial badness score of a near miss is based purely on statically-available information from the optimization log. In the absence of profiling information, the coach reports this initial estimate directly.

Specialization near misses do not have much internal structure, and therefore have the constant 1 as their initial badness score. This value is arbitrary and, because badness is dimensionless, only important relative to that of other optimizations. Subsequent merging operations (section 5.4) update badness, which allows ranking between specialization near misses.

The badness scores of inlining near misses are closely tied to the results of locality merging (section 5.4.2), and their computation is delayed until that phase. After merging, each function that the optimizer attempted to inline is the subject of a single near miss, which summarizes all the inlining attempts that were made for it. The badness score of that near miss is equal to

$$\text{outright-failures} + \text{out-of-fuels} - \text{successes}$$

where outright-failures is the number of unavoidable failures (due, for example, to mutation of the function's binding), out-of-fuels is the number of inlining attempts that failed due to a lack of fuel, and successes is the number of successful inlinings. If the computed score is negative, the near miss is reported as a success instead—the number of successful inlinings is greater than the number of failures.

Initial badness for property and element near misses is, like specialization near misses, the constant 1. Because our SpiderMonkey prototype has always access to profiling information, this initial estimate is not as important and is skipped.

5.3.2 *Profiling-Based Badness*

When profiling information is available, an optimization coach can use it to refine its estimates of the performance impact of near misses. Indeed, all other things being equal, we would expect a near miss in hot code to have a larger impact than one in cold code.

Concretely, our prototypes compute the final badness score of a near miss by multiplying the initial badness score with the *profiling weight* of the surrounding function. We define the profiling weight of a function to be the fraction of the total execution time that is spent executing it.

The theory behind this metric is that the performance impact of a near miss should be proportional to the time spent executing the affected code, and that the execution time of the surrounding function is a good proxy for it. More fine-grained—e.g., line- or expression-level—profile information would improve the metric's accuracy, but function-level information has worked well in practice so far.

JITs and Badness State-of-the-art JIT compilers, such as SpiderMonkey, introduce additional complexity when computing badness. Most notably, they introduce a temporal dimension to the optimization process. They may compile the same code multiple times as they gather more information and possibly revise previous assumptions. In the process, they produce different *compiled versions* of that code, each of which may have a different set of near misses.

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 function is overall.

Because the profiler samples of our SpiderMonkey prototype include compilation identifiers, it associates each sample not only with particular functions, but with particular compiled versions of functions. This additional information enables the required distinctions discussed above.

5.4 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 estimate of the impact of the summary report. The increased badness value of summary reports causes them to rank higher than their constituents would separately, which increases the actionability of the tool's output.

5.4.1 *Causality Merging*

Some optimization events are related by causality: one optimization can open up further optimization opportunities, or an optimization failure may prevent other optimizations from happening. In such cases, presenting a combined report of the related optimizations yields denser information than reporting them individually.

Our notion of causality relies on the concept of *irritant*. For some optimizations, we can "blame" a failure affecting a term on one or several of its subterms, which we call irritants, because they may have the wrong shape or type and thus cause optimization to fail. This notion of irritants is also used for recommendation synthesis.

Since irritants are terms, they can themselves be the subjects of optimization failures, meaning they may contain irritants. These related optimization failures form a tree-shaped structure within which failures are nodes with their irritants as children. Irritants that do not contain irritants form the leaves of the tree; they are the initial causes of the optimization failure. The optimization failure that is caused by all the others becomes the root of the tree.

The entire tree can be merged into a single entry, with the root of the tree as its subject and the leaves as its irritants. The intermediate nodes of the tree can be safely ignored since they are not responsible for the failure; they merely propagate it. Causality merging reduces the number of reports and helps pinpoint the root causes of optimization failures.

Our Racket prototype applies causality merging to specialization near misses. Let us consider the last part of the pseudo-random number generator from section 3.1:

```
(/ (* max last) IM)
```

While `max` is of type `Float`, `last` and `IM` have type `Integer`. Therefore, the multiplication cannot be specialized, which causes the type of the multiplication to be `Real`. This type, in turn, causes the division to remain generic. Here is the relevant excerpt from the logs:

```
TR opt failure: prng-example.rkt 11:5
  (* max last)
  -- Float Integer -- generic multiplication
TR opt failure: prng-example.rkt 11:2
  (/ (* max last) IM)
  -- Real Integer -- generic division
```

The subject of the multiplication entry is an irritant of the division entry. Optimization Coach joins the two entries in a new one: the entire division becomes the subject of the new entry with `last` and `IM` as irritants.

Causality merging combines near misses that share the same solution, which allows programmers to solve multiple near misses at once. Section 5.4.5 presents another form of merging, *by-solution merging*, which also groups near misses with similar solutions, but determines grouping differently.

5.4.2 *Locality Merging*

Some optimization events are related by locality: multiple optimizations are applied to one piece of code or the same piece of code triggers optimizations in different places. An optimization coach should synthesize a single log entry per cluster of optimization reports that affect the same piece of code. Doing so helps formulate a coherent diagnosis about all the optimizations that affect a given piece of code.

In cases where these reports originate from the same kind of optimization, e.g., multiple inlining failures affecting the same function, the coach looks for patterns in the related reports. Reports that are related by locality but concern unrelated optimizations can still be grouped together, but the grouping step unions the reports without further summarization. These groupings, while they do not infer new information, improve the user interface by providing all the relevant information about a piece of code in a single location. This step also aggregates log entries that provide information too low-level to be of use to the programmer directly. New log entries report on these patterns and replace the low-level entries in the log.

This is the case for inlining failures. Optimization log entries from the Racket inliner provide two main pieces of information: which function is inlined, and at which call site it is inlined. This information is not especially enlightening to programmers as is. For widely used functions, however, it is likely that there is a large number of such reports; the number of inlining reports grows with the number of call sites of a function.

Clusters of log entries related to inlining the same function are a prime target for locality merging. Locality merging provides an aggregate view of a function's inlining behavior. Based on the ratio of inlining successes and failures, Optimization Coach decides whether the function as a whole is successful with regards to inlining or not, as section 5.3.1 explains.

Our SpiderMonkey prototype also applies locality merging to its reports. In addition, it applies other merging techniques which can be seen as variants of locality merging: *temporal merging* (section 5.4.3), which operates like locality merging but merges across the temporal dimension, and *by-constructor merging* (section 5.4.6), which uses a notion of locality based on the solution site.

5.4.3 *Temporal Merging*

As discussed in section 5.3.2, advanced JIT compilers introduce a temporal dimension to the compilation and optimization process. 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.

Even though a JIT compiler may optimize an operation differently each time it is 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 because object code is garbage collected.[3]

Identical near misses that originate from different invocations of the compiler necessarily have the same solution; they are symptoms of the same issue. Therefore, a coach merges groups of such reports into single reports. This technique—temporal merging—can be seen as an extension of locality merging which ignores version boundaries.

Our SpiderMonkey prototype uses temporal merging to combine 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.

5.4.4 *Same-Property Analysis*

The next two merging techniques, which are specific to property access near misses, 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 as starting points for merging would lead to incoherent reports that conflate unrelated near misses.

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[4] benchmark suite, boxed in figure 15. 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 fre-

3 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.

4 `https://developers.google.com/octane/`

```
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 15: Two different logical properties with name `run` in the *Richards* benchmark, one underlined and one boxed

quently 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 15 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. In contrast, `TaskControlBlock.prototype.run` 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.

5.4.5 *By-Solution Merging*

Property near misses that affect the same property and have similar solutions are related; a coach should merge them. That is, it should merge near misses that can be addressed either by the same program change or by analogous changes at multiple program locations.

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. This mirrors the work of the recommendation generation phase, which is described in chapter 6. In adition, the coach should ensure that the affected properties belong to the same logical property.

Once our SpiderMonkey prototype 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.

5.4.6 *By-Constructor Merging*

Multiple property 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, our SpiderMonkey prototype identifies which logical properties co-occur within at least one hidden class. To this end, it reuses knowledge about which logical properties occur within each hidden class from same-property analysis.

Because properties can be added to JavaScript objects dynamically, i.e., not inside the object's constructor, a property occuring within a given hidden class does not necessarily mean that it is 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 makes object layout consistent. Reporting these near misses along with those from properties from the constructor helps reinforce this connection.

By-constructor merging can be thought of as a variant of locality merging that groups reports according to solution site.

6 RECOMMENDATION GENERATION

After optimization analysis, near miss reports can be used to generate concrete recommendations. Recall that such recommendations represent program changes that eliminate optimization failures but may also alter the behavior of the program.

To identify expressions for which to recommend changes, an optimization coach should reuse the concept of irritant where available. Recall that irritants are terms that caused optimization to fail. If these terms were changed, the optimizer would be able to apply transformations. Irritants are thus ideal candidates for recommendations.

Determining the best program changes necessarily relies on optimization specific logic. Since each optimization has its own failure modes, general rules do not apply.

In some cases, generating recommendations is impossible, either due to the nature of an optimization, to the cause of the failure, or to properties of the context. Irritants are reported nonetheless; knowing the *cause* of a failure may still help programmers.

6.1 Recommendations for Typed Racket

In the context of Typed Racket, Optimization Coach's recommendations suggest changes to the types of irritants. The fixes are determined using optimization proximity. Recommendation generation for Typed Racket is therefore a straightforward matter of connecting the facts and presenting the result in an informative manner.

For example, when presented with the fixnum arithmetic example from figure 3 (page 17), Optimization Coach recommends changing the type of `i` to `Byte` or `Index`. In the case of the pseudo-random number generator from figure 5 (page 19), Optimization Coach recommends changing the types of the irritants to `Float` to conform with `max`.

Figure 16 provides examples of recommendations produced by Optimization Coach to accompany near misses from the Typed Racket optimizer.

6.2 Recommendations for Inlining

Inlining is not as easy to control through changes in the source program as Typed Racket's type-driven optimizations. Therefore, recommendations relating to inlining optimizations are less precise than those for Typed Racket.

- This expression consists of all fixnum arguments but is not guaranteed to produce a fixnum. Therefore it cannot be safely optimized. Constraining the arguments to be of Byte or Index types may help.
- This expression has a Real type. The optimizer could optimize it if it had type Float. To fix, change the highlighted expression(s) to have Float type(s).
- This expression compares values of different exactnesses, which prevents optimization. To fix, explicitly convert the highlighted expressions to be of type Float.
- According to its type, the highlighted list could be empty. Access to it cannot be safely optimized. To fix this, restrict the type to non-empty lists, maybe by wrapping this expression in a check for non-emptiness.

Figure 16: Example recommendations for Typed Racket

Since lack of fuel is the main reason for failed inlinings, reducing the size of functions is the simplest recommendation and applies in all cases. In some cases, it is possible to give the programmer an estimate of how much to reduce the size of the function, using its current size and the remaining fuel.

Another possibility is the use of inlining-related pragmas, such as Racket's `begin-encourage-inline` and `define-inline`, when they are provided by the host compiler. Alternatively, for languages with macros, an optimization coach can also recommend turning a function into a macro, i.e., inlining it unconditionally at all call sites. To avoid infinite expansion, Optimization Coach recommends this action only for non-recursive functions.

An optimization coach can also consider other factors, such as the shape of the function's body or commonly executed paths within the function, to recommend specialized responses to common patterns. For example, a function that includes both a short, commonly-taken fast-path and a longer, infrequently-used slow path may benefit from outlining the slow path. This transformation would facilitate inlining the now smaller function and may improve the performance of the fast path.

Figures 7 (page 21) and 8 (page 22) show examples of the Racket prototype's recommendations related to inlining.

- This property is not guaranteed to always be stored in the same location inside objects.

  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 always putting it in the same location.
- This object is a singleton. Singletons are not guaranteed to have properties in a fixed slot. Try making the object's properties global variables.
- This operation needs to walk the prototype chain to find the property. Try putting the property in the same location for all objects. For example, try always putting it directly on the object, or always on its direct prototype.
- This array operation saw an index that was not guaranteed to be an integer, and instead was an object. Try using an integer consistently. If you're already using an integer, trying doing `index|0` to help the JIT recognize its type.

Figure 17: Example recommendations for SpiderMonkey

6.3 Recommendations for SpiderMonkey

The SpiderMonkey coach computes recommendations using three main pieces of information: the inferred solution site, the cause of the failure and the types of the operands.

The inferred solution site both determines where the recommendation points and restricts the set of actions that can be recommended. For example, reordering fields is an action that makes sense only for constructor reports; it cannot be applied directly at the operation site. Similarly, recommending to make an operation monomorphic makes sense only in the context of operation reports.

The cause of the optimization failure, as well as the types of the expressions involved, feed into optimization-specific logic. This logic fills out recommendation templates selected based on the failure cause with source and type information.

Figure 17 shows example recommendations generated by our SpiderMonkey prototype.

7 USER INTERFACE

The integration of an optimization coach into the programmer's workflow requires a carefully designed tool for presenting relevant information. This chapter describes the user interfaces of our prototypes.

7.1 Racket Prototype

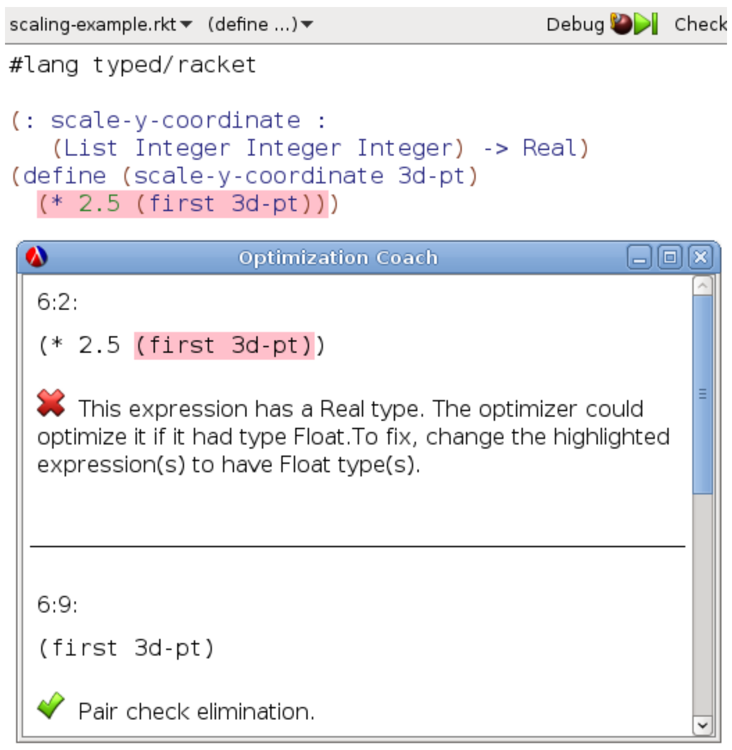
Optimization Coach is a plug-in tool for DrRacket (Findler et al. 2002). As such a tool, Optimization Coach has access to both the Racket and Typed Racket compilers and can collect instrumentation output in a non-intrusive fashion. At the press of a button, Optimization Coach compiles the program, analyzes the logs, and presents its results.

As the screenshots in previous chapters show, Optimization Coach highlights regions that are affected by either optimization successes or near misses. To distinguish the two, green boxes highlight successes and red boxes pinpoint areas affected by near misses. If a region is affected by both, a red highlight is used; near misses are more cause for concern than successes and should not be hidden. The `scale-y-coordinate` function from figure 18 contains both an optimization success—taking the first element of its input list—and an optimization near miss—scaling it by a floating-point factor.

The user interface uses different shades of red to express an ordering of near misses according to their badness score. Figure 19 shows ordering in action; the body of `add-sales-tax` contains a single near miss and is therefore highlighted with a lighter shader of red, distinct from the body of `add-discount-and-sales-tax`, which contains two near misses.

Clicking on a region brings up a tool-tip that enumerates and describes the optimization events that occurred in the region. The description includes the relevant code, which may be a subset of the region in the case of nested optimizations. It also highlights the irritants and explains the event. Finally, the window also comes with recommendations.

Optimization Coach allows programmers to provide profiling information, which is then fed to the optimization analysis phase to enable its profiling-based techniques. Programmers can collect profiling information by running their program with its entry point wrapped in the `optimization-coach-profile` form provided by the tool. This profiling wrapper enables Racket's statistical profiler and saves the results to a file. Optimization Coach then reads profiling information from that file.



```
#lang typed/racket

(: scale-y-coordinate :
   (List Integer Integer Integer) -> Real)
(define (scale-y-coordinate 3d-pt)
  (* 2.5 (first 3d-pt)))
```

6:2:

```
(* 2.5 (first 3d-pt))
```

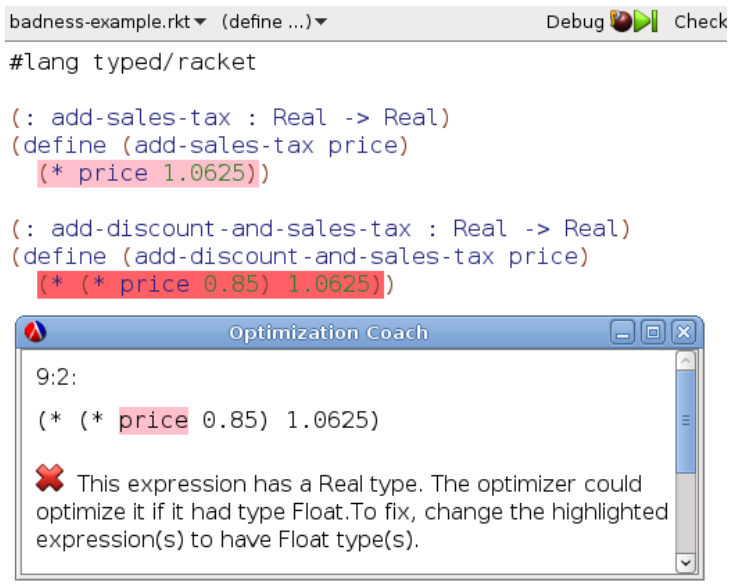
This expression has a Real type. The optimizer could optimize it if it had type Float.To fix, change the highlighted expression(s) to have Float type(s).

6:9:

```
(first 3d-pt)
```

Pair check elimination.

Figure 18: A function affected by both an optimization success and a near miss



```
#lang typed/racket

(: add-sales-tax : Real -> Real)
(define (add-sales-tax price)
  (* price 1.0625))

(: add-discount-and-sales-tax : Real -> Real)
(define (add-discount-and-sales-tax price)
  (* (* price 0.85) 1.0625))
```

```
9:2:

(* (* price 0.85) 1.0625)
```

This expression has a Real type. The optimizer could optimize it if it had type Float.To fix, change the highlighted expression(s) to have Float type(s).

Figure 19: Optimization Coach ordering near misses by predicted importance (with focus on `add-discount-and-sales-tax`); the most important near miss is in a darker shade of red and the least important one is in a lighter shade of red

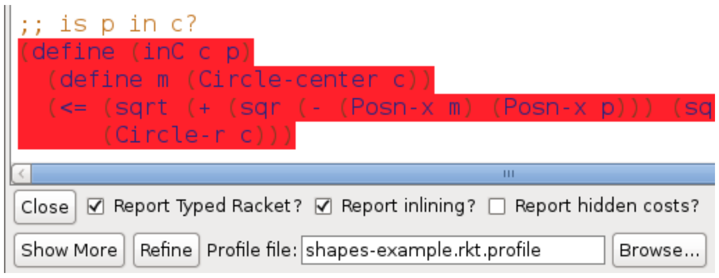
Figure 20 shows Optimization Coach's control panel, which includes profiling-related commands, as well as view filtering options. These filtering options control the classes of optimizations for which to show near misses. Additional filtering options would provide programmers with finer-grained control over reporting and are an area for future work. Potential extensions include user-specified black- and white-lists that limit reports to specific code regions, and tunable badness display thresholds.

Optimization Coach additionally provides a text-only mode at the command-line, suitable for batch usage. This version provides the same information as the graphical version, but without highlighting.

7.2 SpiderMonkey Prototype

Our prototype coach for SpiderMonkey, unlike Optimization Coach, provides a command-line interface only.

The tool's workflow is a two-step process. First, programmers run their program in profiling mode, either using the SpiderMonkey profiler's graphical front-end in the Firefox development tools suite or using a command-line interface. This profiling run produces a profile file, which also includes the optimization instrumentation log.



```
;; is p in c?
(define (inC c p)
  (define m (Circle-center c))
  (<= (sqrt (+ (sqr (- (Posn-x m) (Posn-x p))) (sq
      (Circle-r c)))
```

Close ☑ Report Typed Racket? ☑ Report inlining? ☐ Report hidden costs?

Show More Refine Profile file: shapes-example.rkt.profile Browse...

Figure 20: The Optimization Coach control panel

Second, programmers invoke the coach[1] with the profile file as input. This program executes the optimization analysis and recommendation generation phases, and emits reports and recommendations in text form.

At the time of this writing, the coach presents the five near misses with the highest badness value. This allows programmers to focus on high-priority reports only. As they resolve these near misses, they can repeat the coaching process if desired and receive reports for the remaining, lower-priority near misses.

The tool presents near misses in decreasing order of badness. Additionally, it sorts sub-components of reports, e.g., the fields in a constructor report, in decreasing order of badness. Figure 21 shows both forms of sorting in action using an excerpt from the coaching report for the *DeltaBlue* benchmark (discussed in section 8.2.1).

Each near miss report includes a brief description of the failure(s), an explanation of the cause, and recommendations. In addition, it includes source location information to allow programmers to cross-reference reports with their source code.

The use of a command-line interface makes coaching multi-file programs straightforward; reports for a program are presented all at once, regardless of their file of origin. In contrast, our Racket prototype supports coaching one file at a time because it is tied to DrRacket's one-file-at-a-time editing model. This is not, however, a general limitation of graphical interfaces for coaching.

At the time of this writing, there is ongoing work at Mozilla to integrate optimization coaching to the Firefox development tools suite, driven by Shu-yu Guo. Developing an easy-to-use browser interface for an optimization coach is still an open problem.

1 A command-line program written in Racket.

```
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.

--------------------------------------------------------------------------------
badness: 2533
for object types:
  ScaleConstraint:deltablue.js:452
  EditConstraint:deltablue.js:308
  StayConstraint:deltablue.js:290
  EqualityConstraint:deltablue.js:511
affected property: addToGraph
locations:
  deltablue.js:158:2 (badness: 2533)

This operation needs to walk the prototype chain to find the property.
Try putting the property in the same location for all objects.
For example, try always putting it directly on the object, or always
on its direct prototype.

--------------------------------------------------------------------------------
badness: 1753
for object types:
  Variable:deltablue.js:534
  ScaleConstraint:deltablue.js:452
  EditConstraint:deltablue.js:308
  StayConstraint:deltablue.js:290
  EqualityConstraint:deltablue.js:511
affected property: execute
locations:
  deltablue.js:773:4 (badness: 1747)
  deltablue.js:261:26 (badness: 6)

This operation needs to walk the prototype chain to find the property.
Try putting the property in the same location for all objects.
For example, try always putting it directly on the object, or always
on its direct prototype.
```

Figure 21: Coaching reports for the *DeltaBlue* benchmark (excerpt)

8 EVALUATION

For an optimization coach to be useful, it must provide actionable recommendations that improve the performance of a spectrum of programs. To validate this claim about our prototypes, we conducted three experiments on each of them. First, to measure the effectiveness of recommendations, we measured their impact on program performance. Second, to estimate the effort required to follow recommendations, we recorded the size and nature of code changes. Third, to measure the effect of optimization analysis, we looked at the quality of those recommendation, both in terms of usefulness and actionability. This chapter shows the results of these experiments.

8.1 Racket Prototype

We evaluated our Racket prototype on a series of programs of various sizes and origins. These range from small benchmarks from the Computer Language Benchmark Game[1] to full applications. Some of these programs are standard benchmarks, others are used inside the Racket standard library, while others yet were contributed by Optimization Coach users. Figure 22 lists the size of each program and provides brief descriptions. The full set of programs is available online.[2]

We performed our experiments as follows.[3] We ran our Racket prototype on each program and modified them by following the tool's recommendations, so long as those reports were truly actionable. That is, we rejected reports that did not suggest a clear course of action or would break a program's interface, as a programmer using the tool would do. In addition, we identified and rolled back recommendations that decreased performance, like a programmer would do. Finally, to simulate a programmer looking for "low-hanging fruit," we ran the coach only once on each program.[4]

Performance Impact We measured execution time of each program before and after applying recommendations. For six of the programs, we also had access to versions that had been hand-optimized by expert Racket programmers starting from the same baseline pro-

1 `http://benchmarksgame.alioth.debian.org`
2 `http://www.ccs.neu.edu/home/stamourv/dissertation/oc-racket-benchmarks.tgz`
3 With the exception of the perlin-noise and simplex-noise programs, whose methodology is described below.
4 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.

gram. In all cases, all three versions—baseline, coached, and hand-optimized—use the same algorithms. For these programs, we also measured how close the performance of the coached versions were from that of the hand-tuned versions.

Our plots show results normalized to the pre-coaching version of each program with error bars marking 95% confidence intervals. All our results represent the mean of 30 executions on a 6-core 64-bit x86 Debian GNU/Linux system with 12GB of RAM, using Racket version 6.1.1.6 (January 2015). We use execution times as our results, lower is better.

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 Quality To evaluate the usefulness of individual recommendations, we classified recommendations 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
- *undesirable* reports suggested changes that would break the program's intended behavior. Programmers can reject them easily.

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. Undesirable recommendations decrease the signal-to-noise ratio of the tool, but they can individually be dismissed pretty quickly by programmers. A small number of undesirable recommendations therefore does not contribute significantly to the programmer's workload. Large numbers of undesirable recommendations, however, would be cause for concern.

8.1.1 *Results and Discussion*

The results of our performance experiments are in figure 23. We observe significant[5] speedups for all but one of the programs we studied, ranging from 1.03× to 2.75×.

5 We consider speeedups to be significant when the confidence intervals of the baseline and coached versions do not overlap.

Benchmark	Size (SLOC)	Description
binarytrees	51	Allocation and GC benchmark
cantor	18	Cantor pairing and unpairing functions
heapsort	71	Heapsort sorting algorithm
mandelbrot	48	Rendering the Mandelbrot set
moments	72	Compute statistical moments of a dataset
nbody	166	Orbit simulation for the Jovian planets
ray-tracer	3,199	Ray tracer
pseudoknot	3,427	Nucleic acid 3D structure computation
video	860	Differential video codec
perlin-noise	175	Perlin noise generator
simplex-noise	231	Simplex noise generator

Figure 22: Racket benchmark descriptions

For the six programs where we can compare with hand-optimized versions, following recommendations can usually yield about half the improvement an expert can extract—after hours of work, for some of these programs. In the case of the *moments* benchmark, the performance of the coached version matches that of the hand-optimized version. In all cases, the hand-optimized versions include all the optimizations recommended by Optimization Coach.

Following recommendations is a low-effort endeavor. The maximum number of lines changed is 28, for the *pseudoknot* program. For all but 3 of the 11 programs, under 10 lines had to be changed. The quality of recommendations is also generally good. In all 11 programs, the coach produced only one negative recommendation, for *pseudoknot*, which resulted in a 0.95× slowdown.

Neutral and undesirable recommendations account for a significant proportion of the recommendations only in the *ray-tracer* program. The high number of overall recommendations for this program is not surprising, given its relative size. In depth analysis of those recommendations is available below. In general, recommendations that were classified as neutral had lower badness scores than positive recommendations in the same program. This suggests that our ranking heuristics are accurate.

The rest of this section presents detailed results for each program, comparing the recommended changes to those performed by experts where applicable.

BINARYTREES Optimization Coach does not provide any recommendations for this program. The coached version is therefore identical to the baseline. Prior to introducing the current definition of bad-

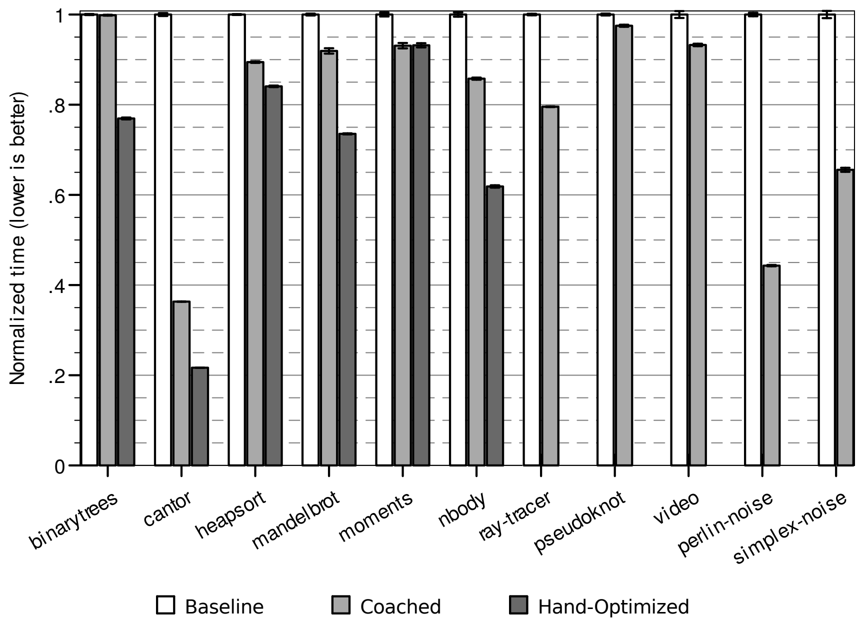


Figure 23: Benchmarking results for Racket

Benchmark	Lines changed (SLOC)		
	Added	Deleted	Edited
binarytrees	0	0	0
cantor	2	1	6
heapsort	0	0	5
mandelbrot	1	1	1
moments	0	0	9
nbody	0	0	1
ray-tracer	0	2	2
pseudoknot	24	1	3
video	1	0	2
perlin-noise	4	1	12
simplex-noise	4	1	19

Figure 24: Size of changes following recommendations

Benchmark	Recommendation impact (# recommendations)			
	Positive	Negative	Neutral	Undesirable
binarytrees	0	0	0	0
cantor	3	0	0	0
heapsort	1	0	0	0
mandelbrot	1	0	0	0
moments	3	0	3	0
nbody	1	0	0	0
ray-tracer	2	0	12	30
pseudoknot	1	1	0	0
video	2	0	0	0
perlin-noise	4	0	0	0
simplex-noise	7	0	2	0

Figure 25: Summary of recommendation quality

ness for inlining near misses,[6] however, the coach would produce one such report. Acting on that report led to a slowdown of 0.98×. With the new definition, this near miss is instead considered a success, which does not call for programmer action. This provides favorable evidence for the usefulness of our current definition of badness.

The performance gains observed in the hand-optimized versions come almost entirely from a single optimization: a switch from structures to vectors, which have a lower allocation overhead.

CANTOR This benchmark heavily exercises arithmetic operations. The straightforward implementation involves both exact rational arithmetic and complex number result types. The first recommendation suggests using integer division instead of regular division to avoid exact rationals. Since the original program truncates values to integers before returning them, this change in semantics did not affect the results of the program. The second recommendation involves replacing a use of the generic `sqrt` function, which may return complex results, with a real-number only equivalent. Finally, the coach also recommends inlining a helper function. All three recommendations lead to performance improvements, with a total speedup of 2.75×.

The expert-optimized version uses floating-point numbers instead of integers to benefit from additional specialization and unboxing optimizations, which do not apply to integer computations. This transformation radically affects the semantics of the program. This program performs Cantor pairing and unpairing, which relies on exact integer semantics. For large inputs, floating-point imprecision can

6 The previous definition only counted optimization failures, not successes.

cause the hand-optimized version to compute incorrect answers. The inputs used by the benchmark were not large enough to trigger this behavior, and thus the expert did not introduce any bugs in this particular case. This change remains undesirable in the general case, and Optimization Coach does not recommend it.

HEAPSORT This implementation of heapsort is written in Typed Racket and already benefits from type-driven optimizations. Optimization Coach recommends restricting a variable from type `Real` to `Float`, which requires a few upstream type changes as well. This change resulted in a 1.12× speedup. Manual vector bounds-check elimination explains the remaining performance gains in the hand-optimized version.

MANDELBROT The hand-optimized version benefits from two major optimizations: the manual inlining of a helper function too large to be inlined by the optimizer and the manual unrolling and specialization of an inner loop. Optimization Coach detects that the helper function is not being inlined and recommends using a macro to force inlining, which is exactly what the expert did. This change results in a speedup of 1.09×. Optimization Coach does not recommend manually unrolling the loop; its recommendation generation process does not extend to unrolling.

Furthermore, the author of the hand-optimized version also manually specialized a number of arithmetic operations. This duplicated work that the Typed Racket optimizer already performed—as Optimization Coach confirmed by reporting optimization successes. The hand-optimized version did feature some specialized fixnum operations that Typed Racket was not able to specialize, but those did not produce a noticeable effect on performance.

To improve the quality of the profiling information fed to the coach, we replaced two tail calls in the original benchmark with non-tail calls. These changes had no measurable impact on the performance of the program.

MOMENTS The baseline version of this mostly-numeric program uses types with low optimization potential. The coach points to a number of operations that failed to be optimized because of those, all of which can be traced back to six variables whose types are too conservative. Adding two type coercions and replacing an operation with a more precise equivalent allowed us to refine the types of all six, which led to a speedup of 1.07×. Those were the same changes the expert performed on the hand-optimized version.

As with the *mandelbrot* benchmark, we replaced one tail call with a non-tail call to improve the quality of the profiling information. This change also had no measurable impact on performance.

NBODY Optimization Coach identifies an expression where Typed Racket must conservatively assume that `sqrt` may produce complex numbers, leading to an optimization failure. This is the same kind of near miss as the one from the *cantor* program. We applied the same solution, namely replacing that use of `sqrt` by `flsqrt`. This change led to a 1.17× improvement.

The rest of the performance improvements in the hand-optimized version are due to replacing structures and lists with vectors, as in the binarytrees benchmark. In addition, the author of the hand-optimized version also replaced another instance of `sqrt` by `flsqrt`. This second replacement, however, did not actually have a noticeable impact on performance. Optimization Coach's profiling-based pruning recognized that the failure was located in cold code, and pruned the corresponding optimization failure from the logs.

RAY-TRACER This program is a ray tracer used to render logos and icons for DrRacket and the Racket website. It was written by Neil Toronto, an experienced Racket programmer. It supports a wide variety of features, such as refraction, multiple highlight and reflection modes, and text rendering. Over time, most icons in the Racket codebase have been ported to use this ray tracer, and it eventually became a bottleneck in the Racket build process. Its author spent significant time and effort[7] to improve its performance. To determine whether an optimization coach would have helped him, we attempted to optimize the original version of the ray tracer ourselves using Optimization Coach. For our measurements, we rendered a 600×600 pixel logo using each version of the ray tracer.

Optimization Coach identified two helper functions that Racket failed to inline. We followed its recommendations by turning the two functions into macros. The changes were local and did not require knowledge of the code base. Dr. Toronto had independently performed the same changes, which were responsible for most of the speedups over his original ray tracer. Optimization Coach successfully identified the same sources of the performance bugs as a Racket expert and provided solutions, making a strong case for the effectiveness of optimization coaching.

Dr. Toronto, being an expert Typed Racket user, had already written his code in a way that triggers type-driven specialization. The coach therefore did not find room for improvement there. We classified specialization-related recommendations as undesirable because they would have required restricting the public interface of the ray tracer to floating-point inputs instead of generic real numbers. These changes would have broken the ray tracer's interface.

7 Leading to the mailing list discussion mentioned in chapter 2 (page 11).

PSEUDOKNOT We used a version of the pseudoknot (Hartel et al. 1996) program originally written by Marc Feeley. This program computes the 3D structure of a nucleic acid from its primary structure and a set of constraints. The baseline version of this program is already highly optimized; there are few optimizations left for Optimization Coach to recommend.

The tool suggests inlining two functions: `dgf-base` and `pseudoknot-constraint?`. Inlining the first resulted in a 0.95× slowdown. Inlining the second produced a 1.03× speedup. The change involved converting the function to a macro, and modifying its calling context to use it in a first-order fashion. Forty additional optimization failures were pruned by Optimization Coach's profiling-based pruning.

This scenario highlights the importance of pruning in reducing the transformation space the programmer has to explore—trying out multiple combinations of recommendations is much more feasible when the tool presents only a small number of recommendations.

VIDEO This application is a simple video chat client, written by Tony Garnock-Jones, who worked hard to make the chat program efficient, meaning most performance low-hanging fruit had already been picked. We focused our effort on the simple differential video coder-decoder (codec) which is the core of the client. To measure the performance of each version of the codec, we timed the decoding of 600 pre-recorded video frames. The code mostly consists of bitwise operations on pixel values.

The decoding computation is spread across several helper functions. Optimization Coach confirmed that Racket inlines most of these helper functions, avoiding extra function call overhead and enabling other optimizations. However, two of them, `kernel-decode` and `clamp-pixel`, were either not inlined to a satisfactory level or not inlined at all. We followed these two recommendations, turning both helper functions into macros. Each change was local and did not require understanding the behavior of the function or its role in the larger computation.

PERLIN-NOISE This program is a Perlin noise[8] generator written by John-Paul Verkamp.[9] Unhappy with the performance of his program, Mr. Verkamp asked for help on the mailing list.[10] Following the resulting discussion, he used Optimization Coach to find optimization opportunities, and blogged about the process.[11]

The baseline version of this program used for our experiments corresponds to the original version of his program, and the coached ver-

8 Specifically, pseudo-randomly generated images often used for procedural generation of content in video games.

9 http://blog.jverkamp.com/2013/04/11/perlin-and-simplex-noise-in-racket/

10 http://lists.racket-lang.org/users/archive/2013-April/057245.html

11 http://blog.jverkamp.com/2013/04/16/adventures-in-optimization-re-typed-racket/

sion corresponds to the final version he derived, on his own, using Optimization Coach. Following the coach's recommendations led to a speedup of 2.26×.

The changes all involved replacing types with low optimization potential with types with higher potential, which enabled type-driven specialization. This process involved converting the internals of the noise generator to always use floating-point numbers (instead of operating on real numbers), and providing an adapter function to satisfy clients that use the generator with non-floating-point inputs.

SIMPLEX-NOISE Like the Perlin noise generator, this program is also due to John-Paul Verkamp. The two programs were, in fact, developed together. Our experimental methodology was the same for the two programs.

Following Optimization Coach's recommendations resulted in a 1.53× speedup. The changes followed the same patterns as those for the Perlin noise generator.

8.2 SpiderMonkey Prototype

To evaluate our SpiderMonkey prototype, we used a subset of the widely-used Octane benchmark suite. Our experimental protocol was the same as for the Racket prototype, with the following exceptions.

First, we applied the five top-rated recommendations only—which are those shown by the tool. As with the Racket experiments, we rolled back negative recommendations.[12]

Second, because a web page's JavaScript code is likely to be executed by multiple engines, we used three of the major JavaScript engines: SpiderMonkey[13], Chrome's V8[14] and Webkit's JavaScript-Core[15]. For the impact of individual recommendations, however, we measured only the performance on SpiderMonkey.

Third, 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] Therefore, our plots show normalized Octane scores, and higher results are better. To eliminate confounding factors due to interference from other browser components, we ran our experiments in standalone JavaScript shells.

Our chosen subset of the Octane suite focuses on benchmarks that use property and array operations in a significant manner. It excludes,

12 This methodology differs from that used in the published version of this work (St-Amour and Guo 2015), for which we did not roll back negative recommendations. Consequently, the results presented here differ slightly from the published version.

13 Revision `f0f846d875acaced6ce9c9af484096ed2670eef1` (September 2014).

14 Version 3.30.5 (October 2014).

15 Version 2.4.6 (September 2014).

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.

Benchmark	Size (SLOC)	Description
Richards	538	OS simulation
DeltaBlue	881	Constraint solver
RayTrace	903	Ray tracer
Splay	422	Splay tree implementation
NavierStokes	415	2D Navier-Stokes equation solver
PdfJS	33,053	PDF reader
Crypto	1,698	Encryption and decryption benchmark
Box2D	10,970	2D physics engine

Figure 26: Octane benchmark descriptions

for example, the *Regexp* benchmark because it exercises nothing but an engine's regular expression subsystem. 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*. The full set of programs is available online.[19] Figure 26 provides size information and a brief description for each program. Unlike for our Racket experiments, these baseline programs are already tuned by hand. It is therefore impossible to compare the coach against plain and tuned versions.

8.2.1 *Results and Discussion*

As figure 27 shows, following the coach's recommendations leads to significant speedups on six of our eight benchmarks when run on SpiderMonkey. These speedups range from 1.01× to 1.17×. For the other two, we observe no significant change; in no case do we observe a slowdown.

The results are similar for the other engines: on both V8 (figure 28) and JavaScriptCore (figure 29), we observe speedups on two and four benchmarks, respectively, ranging from 1.02× to 1.20×. These speedups differ from those observed using SpiderMonkey, but are of similar magnitude. These results provide evidence that, even though

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.

19 `http://www.ccs.neu.edu/home/stamourv/dissertation/oc-js-benchmarks.tgz`

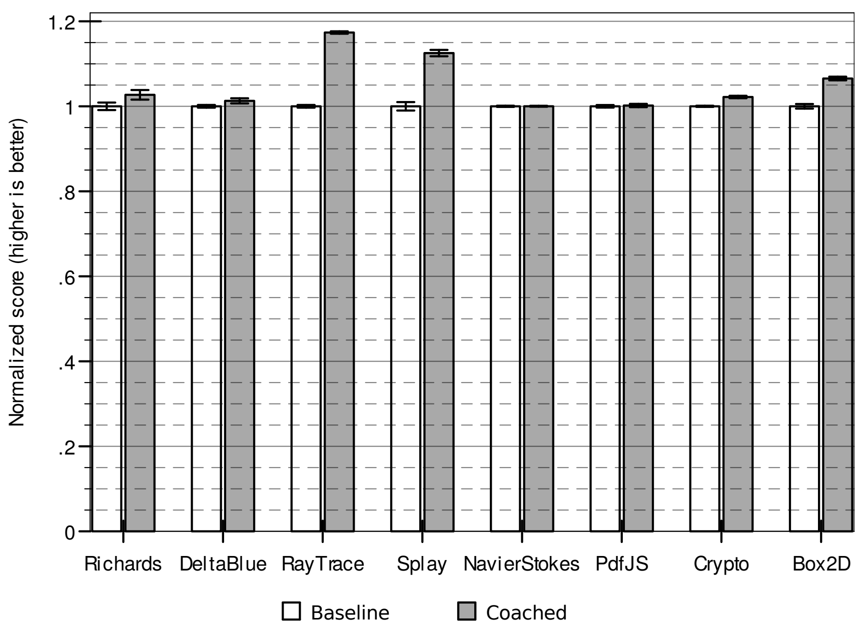


Figure 27: Benchmarking results on SpiderMonkey

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 promising. We conjecture that our prototype (or an extension of it) could yield even larger speedups on other, regular programs.

Figure 30 presents our results for the effort dimension. For all programs, the total number of lines changed is at most 33. 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.

Figure 31 presents the results of studying the usefulness of recommendations. We classified 17 out of 35 reports as positive, and only one as negative. We classified 12 reports as undesirable, 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 recommendations for individual benchmarks.

Richards The coach provides three reports. Two of those point out an inconsistency in the layout of `TaskControlBlock` objects. Figure 32 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 as-

20 `http://arewefastyet.com`

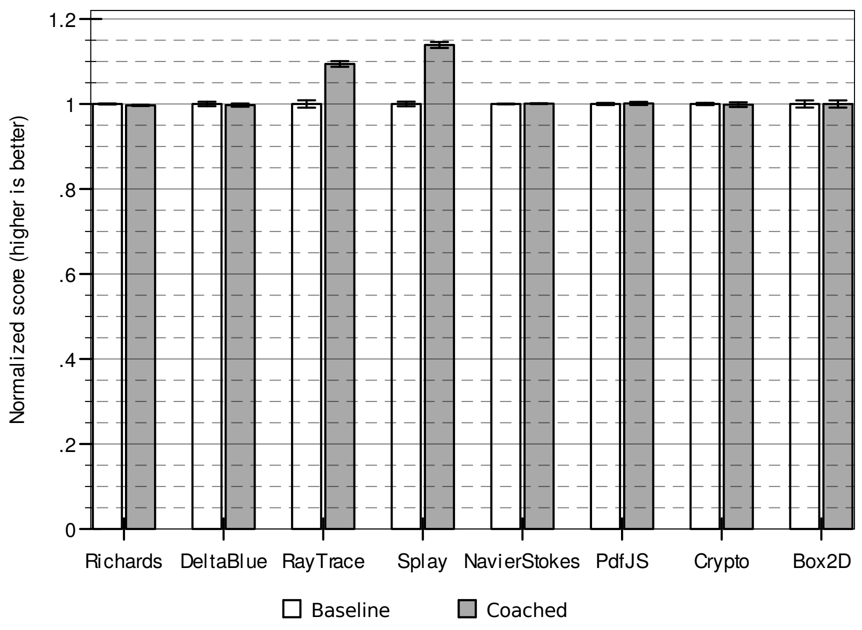


Figure 28: Benchmarking results on V8

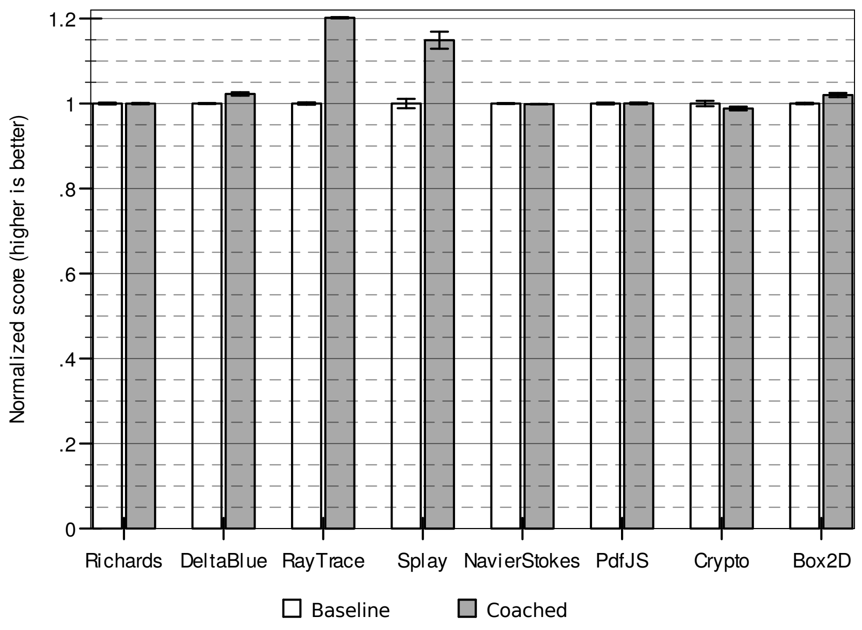


Figure 29: Benchmarking results on JavaScriptCore

Benchmark	Lines changed (SLOC)		
	Added	Deleted	Edited
Richards	1	5	0
DeltaBlue	6	3	24
RayTrace	10	11	0
Splay	3	3	0
NavierStokes	0	0	4
PdfJS	2	1	0
Crypto	2	0	1
Box2D	8	0	0

Figure 30: Size of changes following recommendations

Benchmark	Recommendation impact (# recommendations)			
	Positive	Negative	Neutral	Undesirable
Richards	2	0	0	1
DeltaBlue	2	1	1	1
RayTrace	5	0	0	0
Splay	2	0	1	2
NavierStokes	0	0	1	0
PdfJS	0	0	1	4
Crypto	4	0	0	1
Box2D	2	0	0	3

Figure 31: Summary of recommendation quality

```
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 32: 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 33: Making object layout consistent in the *Richards* benchmark

signments into one, as figure 33 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 undesirable.

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 the first report in figure 21 (page 57). 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%. We rolled it back. 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.

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 3.3.3. 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 change may cause additional space usage, the time/space tradeoff is worthwhile and leads to speedups on all three engines. Two of the three changes yield speedups, with the third having no 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 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

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.

this large codebase. Programmers more familiar with *PdfJS*'s internals may find these reports more actionable.

Crypto Four 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 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 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 large 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.

9 COACHING BEYOND RACKET AND SPIDERMONKEY

The general idea of optimization coaching—and some of the specific techniques decribed in previous chapters—should apply beyond the optimizations covered by our two prototypes. In this chapter, we discuss several standard optimizations and describe how optimization coaching may apply. For each optimization, we explain its purpose and prerequisites. We then propose a way of detecting near misses of this kind of optimization. Finally, we sketch ideas for how a programmer could react to a near miss.

9.1 Common Subexpression Elimination

Common expression elimination (Muchnick 1997 §13.1) is only valid if the candidate expressions do not depend on variables that may be mutated.

An optimization coach can detect cases where an expression is computed multiple times—and is a candidate for common subexpression elimitation—but a reference to a mutated variable prevents the optimization from happening. The optimization coach could recommend that the programmer reconsider mutating the relevant variable.

9.2 Test Reordering

Conditionals with multiple conjuncts can be optimized by performing the less expensive tests (e.g., integer comparisons) before more expensive ones (e.g., unknown function calls). This optimization (Muchnick 1997 §12.3) is valid only if the reordered tests are pure.

An optimization coach could identify near misses by counting the number of impure tests. Specifically, the lower the ratio of impure tests to the total number of tests, the closer the optimizer is to triggering the reordering. To further rank missed optimizations, an optimization coach can take into account the size of the body of the conditional when computing its badness metric. The cost of the condition matters more for small conditionals. When reporting near misses, the coach can recommend making the problematic tests pure or reordering them manually.

9.3 Scalar Replacement

A scalar replacement optimization (Muchnick 1997 §20.3) breaks up aggregates, e.g., structures or tuples, and stores each component separately. Like inlining, scalar replacement is mostly useful because it opens up further optimization possibilities and reduces allocation. Optimizers perform scalar replacement only when the target aggregate does not escape.

The Typed Racket optimizer performs scalar replacement on complex numbers. Our prototype for Racket reports when and where the optimization triggers, but it does not currently detect near misses or provide recommendations.

An optimization coach could use the ratio of escaping use sites to non-escaping use sites of the aggregate as a possible optimization proximity metric. This metric can be refined by considering the size of the aggregate: breaking up larger aggregates may enable more optimizations than breaking up smaller ones. When it does discover near misses, the optimization coach can recommend eliminating escaping uses or manually breaking up the aggregate.

9.4 Loop-Invariant Code Motion

A piece of code is loop-invariant (Muchnick 1997 §13.2) if all reaching definitions of variables in the piece of code come from outside the loop. If one or more relevant definitions come from *inside* the loop, the optimization does not apply.

A potential proximity metric for loop-invariant code motion would measure how many problematic assignments in the body of the loop could potentially be avoided. This can be refined by also considering the ratio of assignments and references inside the loop for problematic variables. Variables that are used often and mutated rarely may be more easily made loop-invariant than those that are mutated often. When presenting such near misses, the optimization coach could recommend avoiding the problematic assignments or performing the code motion manually. Assignments to loop index variables cannot be avoided, however; attempting to make them loop-invariant would defeat the purpose of using a loop. Such necessary assignments can be recognized and should not count towards the optimization proximity metric.

9.5 Reducing Closure Allocation

Efficient treatment of closures is a key optimization for languages that support higher-order functions. When a compiler decides whether—and where—to allocate closures, it takes into account a number of factors (Adams et al. 1986). A closed function requires only a code pointer. A non-escaping function can have its environment allocated on the stack, which avoids heap allocation. In the worst case, closures must be allocated on the heap.

An optimization coach could warn programmers when the compiler needs to allocate closures on the heap. For instance, storing a function in a data structure almost always forces it to be allocated on the heap. An optimization coach can remind programmers of that fact, and encourage them to avoid storing functions in performance-sensitive code.

To reduce the number of false positives, the optimization coach can discard reports that correspond to common patterns where heap allocation is desired, such as functions that mutate variables from their lexical context.

9.6 Specialization of Polymorphic Containers

Polymorphic containers, such as lists, usually require a uniform representation for their contents in order for operations such as `map` to operate generically. In some cases, however, a specialized representation, e.g., a list of unboxed floating-point numbers, can be more efficient. Leroy (1992) and Shao and Appel (1995) propose optimizations that allow specialized representations to be used where possible and to fall back on uniform representations when not. To avoid excessive copying when converting between representations, these optimizations are typically applied only when the element types are sufficiently small. For example, a list of 4-tuples may be specialized, but not a list of 5-tuples.

An optimization coach could communicate to programmers which datatypes are not specialized and report how much they need to be shrunk—or split up—to enable optimization.

9.7 Anchor Pointing

Anchor pointing (Allen and Cocke 1972)—also known in the Haskell world as the case-of-case transformation (Peyton Jones 1996)—rewrites nested conditionals of the form

```
(if (if test1 then1 else1)
    then2 else2)
```

to this form

```
(if test1
    (if then1 then2 else2)
    (if else1 then2 else2))
```

which potentially exposes additional optimization opportunities.

This transformation may lead to code duplication—just as with inlining—and may generate extra closures for join points. Whether it is worth performing depends on the optimizations it enables, and compilers must resort to heuristics. As with inlining, an optimization coach can report cases where this transformation is possible but ultimately not performed, along with the cause of the failure.

10 DEAD ENDS

The previous chapters describe successful coaching techniques. Along the way, we also implemented other techniques that ultimately did not prove to be useful and that we removed from our prototypes. 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 chapter describes two kinds of reports that we studied without success: *hidden costs* and *temporal patterns*.

10.1 Hidden Costs

We investigated having our Racket prototype report operations with hidden costs in addition to reporting optimization near misses. Such operations have performance characteristics that are easily misunderstood and may depend on their context, their mode of use, etc. Misusing, or overusing, such operations may harm program performance.

Examples of such operations that we looked into are:

- *structure allocation*, which can be significantly more expensive than vector allocation in some cases;
- *exact rational arithmetic*, which may use unbounded amounts of memory and can be introduced accidentally and silently in programs, in ways that may not be detectable by observing final results only;
- *implicit parameter dereference*, which may cause functions to consult their context redundantly;
- and *generic sequence dispatch*, which allows polymorphic handling of sequence-like datatypes, e.g. lists, vectors and hashtables, at the cost of run-time dispatch.

Because operations with hidden costs are often desirable—or even necessary—despite their cost, and because they are problematic only in hot code, they may cause a coach to generate numerous false positives. In contrast, optimization failures are more often symptomatic of actual performance issues, and fixing them usually requires light program changes. Integrating profiling information, as discussed in section 5.3.2, does alleviate the problem somewhat, but not entirely. The number of false positives remained high.

Optimization Coach optionally displays hidden costs. In practice, this option has mostly been used when auditing performance-critical programs to ensure they do not use any potentially expensive features. For this use case, false positives are not as problematic.

Treating operations with hidden costs as an accounting problem—a class of problems that profiling handles well—as opposed to a performance issue detection problem served as inspiration for feature-specific profiling, which is the topic of part III of this dissertation.

10.2 Temporal Patterns

As discussed in section 5.4.3, JIT compilers introduce a temporal dimension to the optimization process. We investigated whether detecting optimization success/failure patterns across this temporal dimension could produce actionable reports. Specifically, we looked into two classes of patterns: *regressions* and *flip-flops*. 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.

Regression Reports A regression occurs when an operation that was optimized well during one compilation failed to be optimized as well during a later compilation. This pattern occurred only rarely in the JavaScript programs we studied, and when it did, it was either 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.

Flip-Flop Reports As mentioned, SpiderMonkey discards object code and type information during major collections. When this happens, the engine must start gathering type information and compiling methods from scratch. In some cases, the new 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 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 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 be 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.

11 RELATED WORK

Optimization coaching assists programmers with finding missed optimizations and gives suggestions on how to enable them. This chapter briefly surveys other efforts that pursue optimization using programmer assistance.

11.1 Profilers

When they encounter performance issues, programmers often reach for a profiler (Graham et al. 1982). Profilers answer the question

Which pieces of the program have a high cost?

where that cost may be expressed in terms of execution time, space, I/O, or other metrics such as cache misses. Programmers must still determine which parts of a program have an *abnormally* high cost before they can address performance issues.

Optimization coaches answer a different question, namely

Which pieces could be optimized further?

and the answers identify the location of potential code improvements.

Profilers also do not interpret their results or provide recommendations, both of which are key goals of optimization coaches. The Zoom[1] system-wide profiler is an exception, providing hints to programmers about how to replace possibly slow operations. However, Zoom describes these operations at the assembly level, which makes it challenging for programmers—especially for non-experts—to act on these recommendations using a high-level language.

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.

Finally, for profilers to produce meaningful results, they need to run programs with representative input, but performance-heavy inputs may appear only after deployment, at which point fixing performance bugs becomes significantly more expensive than during development. In contrast, optimization coaches for AOT compilers can operate entirely statically, and do not require representative inputs. Optimization coaches for JIT compilers, however, share this limitation of profilers.

To summarize, the two classes of tools cover different use cases and are complementary.

1 `http://www.rotateright.com/zoom.html`

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 optimization decisions.

Several compilers provide optimization logging. GCC (The Free Software Foundation 2012) supports the `-ftree-vectorizer-verbose` flag for its vectorizer, for example. Similarly, GHC (The GHC Team 2011) provides the `-ddump-rule-firings` flag that lists the rewriting rules that it applies to its input program.

In addition to logging optimization successes, Common Lisp compilers such as SBCL (The SBCL Team 2012) and LispWorks (LispWorks Ltd. 2013) also report some optimization failures, such as failures to specialize generic operations or to allocate objects on the stack. The Cray XMT C and C++ compilers (Cray inc. 2011) report both successful optimimizations and parallelization failures.[2] Steele (1981) describes an "accountable" source-to-source transformation system, which reports both transformations it applies and those it fails to.[3]

JIT inspector (Hackett 2013) and IRHydra (Egorov 2014) report similar kinds of information, 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. These two tools, unlike the compilers listed above, provide user interfaces that are separate from their associated compilers, which makes them easier to use by programmers.

The Open Dylan IDE (Dylan Hackers 2015, chapter 10) reports optimizations such as inlining and dispatch optimizations using highlights in the IDE's workspace. Its use of highlights is similar to that of our Racket prototype.

The FeedBack compiler (Binkley et al. 1998), based on `lcc` (Fraser and Hanson 1995), improves on raw logging by visualizing the optimizations it applies to programs. It was primarily designed to help students and compiler writers understand how specific optimizations work. Implicitly, it also informs programmers of the optimizations applied to their programs. The Feedback Compiler's visualizations illustrate two optimizations: a stepper-like interface that replays common subexpression elimination events and a graphical representation of array traversals affected by *iteration space reshaping* (Wolfe 1986).

In all these cases, the information provided is similar to the result of the logging phase of an optimization coach, without any ranking, pruning or merging. Expert programmers knowledgeable about compiler internals may find this information actionable and use it as a starting point for their tuning efforts. In constrast, optimization coaches target programmers who may not have the necessary knowl-

2 Thanks to Preston Briggs for the pointer.
3 Thanks to Guy Steele for the pointer.

edge and expertise to digest such raw information, and do so by providing recommendations that only require source-level knowledge.

Furthermore, due to the lack of pruning or ranking, using these tools is time consuming even for experts. They must manually determine which reports are actionable, and manually prioritize those.

11.3 Analysis Visualization

A large number of tools exist for visualizing analysis results, of which MrSpidey (Flanagan et al. 1996) is an early example. Some of these tools focus on helping programmers understand and debug their programs; others help compiler writers understand and debug analyses.

A recent effort additionally aims to help programmers optimize their programs. Lhoták's work (Lhoták et al. 2004; Shaw 2005) introduces a plug-in for the Eclipse IDE that displays the results of static analyses computed by Soot (Vallée-Rai et al. 2000), an optimization framework for Java bytecode. While most of these visualizations are targeted at compiler researchers or students learning compilers, their visualization (Shaw (2005), page 87) of Soot's array bounds check analysis (Qian et al. 2002) informs programmers about provably safe array accesses. Similarly, their visualization of loop invariants (Shaw (2005), page 99) highlights expressions that, according to Soot's analysis, can safely be hoisted out of the loop by the compiler.

Although these visualizations are relevant for programmers concerned with optimization, they differ from those of an optimization coach in two major ways. First, they report the results of an *analysis*, not those of the *optimizer*. The two are closely related, but analysis information is only a proxy for the decisions of the optimizer. Furthermore, as implemented, the plug-in uses its own instance of Soot (Shaw (2005), page 15), that may not reflect the analyses performed by the compiler.

Second, while Lhoták's tool reports potentially unsafe array accesses, and explains which bounds are to blame, it does not attempt to distinguish between expected failures and near misses. In contrast to an optimization coach, it also fails to issue recommendations pointing to the code that needs to change to eliminate the bounds check.

11.4 Interactive Optimization

Interactive optimization systems allow programmers to assist optimizers, which allows them to perform optimizations optimizers could not perform on their own. Some of these systems allow users to override the results of analysis, and some support applying arbitrary user-supplied transformations during compilation.

The Parascope system provides an editor that displays the results of dependence analysis for programmers to inspect (Cooper et al.

1993). It further allows programmers to override overly conservative analysis results. Programmers can, for example, mark specific dependences discovered by dependence analysis as infeasible. These dependences are then ignored by the compiler when determining whether optimizations are safe to perform, which allows it to perform optimizations it could not otherwise. The editor also assists programmers by applying source-to-source parallelization and dependence satisfaction transformations at their request.

SUIF Explorer (Liao et al. 1999) is an interactive parallelization tool that allows programmers to bring in domain knowledge to assist the SUIF system's interprocedural parallelization. This combination makes parallelization both less error-prone than doing it manually, and more broadly applicable than relying solely on fully automatic techniques.

Like an optimization coach, SUIF explorer was designed to be easy to use by programmers who may lack knowledge about program analysis or compiler interals. The tool accomplishes this by providing a "wizard"-style interface—the *parallelization guru*—that guides programmers throught the parallelization process, and by using program slicing (Weiser 1981) to direct programmers' attention to the relevant code only.

The HERMIT (Farmer et al. 2012) GHC plugin provides an interactive shell that allows programmers to visualize and apply transformations to the internal representation of programs as they are being compiled. This allows programmers to apply optimizations that may be too specialized to be useful in a general-purpose compiler, but may be worthwhile for specific programs. Programmers specify transformations using rewriting rules, which HERMIT then applies to GHC's internal representation.

VISTA (Zhao et al. 2002) is an interactive code improvement tool that targets embedded system developers. It leverages user interaction to help compilers make use of specific architectural features that would be hard for compilers to exploit on their own—e.g., zero-overhead loop buffers or modular address arithmetic—or to optimize for application-specific size or power consumption goals. Like HERMIT, VISTA allows programmers to visualize intermediate compilation stages and apply custom transformations during optimization.

Interactive optimization systems, while very flexible and powerful, are only really suitable for use by expert programmers. With the exception of SUIF explorer, using the above systems require familiarity with some combination of program analysis, compiler intermediate representations, and optimization. SUIF explorer is significantly better in that regard, but still requires its users to be knowledgeable about parallel programming.

Such systems, especially those that allow programmers to override analysis results or perform arbitrary transformations, may invalidate safety guarantees provided by their host compiler, causing unsound optimizations in the process. Optimization coaches may also encourage programmers to make incorrect changes to their programs but, because the modified programs must still go through the compiler, changes cannot break the compiler's internal invariants, and therefore cannot introduce unsoundness. This also holds for source-to-source interactive optimization systems.

Finally, all of the above tools target different domains from optimization coaches, and are therefore complementary.

11.5 Rule-Based Performance Bug Detection

Several tools use rule-based approaches to detect code patterns that may be symptomatic of performance bugs. This section describes rule-based tools that are closely related to optimization coaching.

Jin et al. (2012) extract source-based rules from known performance bugs in existing applications, then use these rules to detect new performance bugs in other applications. These rules encode API misuses and algorithmic issues that were responsible for performance woes. Their tool provides recommendations based on the existing manual fixes that resolved the original bugs. Their work focuses on algorithmic and API-related performance bugs and is complementary to optimization coaching.

JITProf (Gong et al. 2014) 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 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. Our experiments show, however, that engines behave similarly enough in practice that a coach's recommendations, despite originating from a specific engine, lead to cross-engine performance improvements (section 8.2.1).

Chen et al. (2014) 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.

11.6 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.

Kremlin (Garcia et al. 2011) is a tool that analyses program executions and generates recommendations concerning parallelization efforts. Like an optimization coach, Kremlin issues program-specific recommendations. Kremlin requires representative input to produce meaningful results, which exposes it to the same limitations as profilers. Finally, Kremlin is a special-purpose tool; it is unclear whether the underlying techniques would apply to other domains.

Similarly, Larsen et al. (2012) 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. Like Kremlin, Larsen et al.'s tool is specifically designed for parallelization.

Precimonious (Rubio-González et al. 2013) 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. (2010) 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.

11.7 Auto-Tuning

Auto-tuning tools (Frigo and Johnson 2005; Püschel et al. 2004; Whaley and Dongarra 1998) automatically adapt programs to take advantage of specific characteristics of their execution environment—e.g., hardware or operating system—or of their inputs—e.g., size or matrix shape—and improve their performance.

Like optimization coaches, these tools improve program performance via transformations that traditional optimizing compilers do not perform. Auto-tuning tools perform these transformations automatically, while optimization coaches rely on programmers to approve and apply them.

Each of these tools is specialized for a specific domain, e.g., Fourier transforms or linear algebra. To apply auto-tuning to a new domain, one must write a new tool, which is a significant endeavor. In constrast, optimization coaches are domain-agnostic.

11.8 Refactoring Tools

Some refactoring tools, in addition to performing transformations at the programmer's request, can also look for transformation opportunities on their own. They can then either apply the transformations directly, or request action from the programmer.

The workflow for these tools resembles that of an optimization coach: the tool identifies an opportunity for improvement, and the program is edited to incorporate it. The difference between refactoring tools and optimization coaches lies in who transforms the program. Some refactoring tools can perform transformations automatically, while optimization coaches rely on programmer action.

Refactoring tools with detection capabilities tend to focus on a particular domain—parallel data structures (Dig et al. 2009), API usage patterns (Holmes and Murphy 2005), code smells (Nongpong 2012), or access-control in web applications (Son et al. 2013), for example—and are complementary to optimization coaches. Adding automatic refactoring support to optimization coaches while keeping programmers in control is an open problem.

Part III

FEATURE-SPECIFIC PROFILING

12 WEIGHING LANGUAGE FEATURES

Many linguistic features[1] come with difficult-to-predict performance costs. First, the cost of a specific use of a feature depends on its context. For instance, use of reflection may not observably impact the execution time of some programs but may have disastrous effects on others. Second, the cost of a feature also depends on its mode of use; a higher-order type coercion tends to be more expensive than a first-order coercion (see chapter 13).

When cost problems emerge, programmers often turn to performance tools such as profilers. A profiler reports costs, e.g., time or space costs, in terms of location, which helps programmers focus on frequently executed code. Traditional profilers, however, do little to help programmers find the cause of their performance woes or potential solutions. Worse, some performance issues may have a unique cause and yet affect multiple locations, spreading costs across large swaths of the program. Traditional profilers fail to produce actionable observations in such cases.

To address this problem, we propose *feature-specific profiling*, a technique that reports time spent in linguistic features. Where a traditional profiler may break down execution time across modules, functions, or lines, a feature-specific profiler assigns costs to instances of features—a specific type coercion, a particular software contract, or an individual pattern matching form—whose actual costs may be spread across multiple program locations.

Feature-specific profiling complements the view of a conventional profiler. In many cases, this orthogonal view makes profiling information actionable. Because these profilers report costs in terms of specific features, they point programmers towards potential solutions, e.g., using a feature differently or avoiding it in a particular context.

The following chapters introduce the idea of feature-specific profiling and illustrate it using examples from our Racket-based prototype. Chapter 13 describes the features that we chose to support. Chapter 14 outlines the architecture of our framework, provides background on its instrumentation technique, and explains how to profile structurally simple features. Chapters 15 and 16 describe two extensions to the basic framework, to profile structurally rich features, and to control instrumentation, respectively. Then, chapter 17 presents evaluation results, chapter 18 discusses limitations of our approach,

1 With "linguistic feature" we mean the constructs of a programming language itself, combinator-style DSLs such as those common in the Haskell world, or "macros" exported from libraries, such as in Racket or Rust.

and chapter 19 sketches how feature-specific profiling would apply beyond our prototype. Finally, chapter 20 discusses related work.[2]

12.1 Prototype

Our prototype tool is available from the Racket package catalog,[3] and its source is also publicly available.[4] Leif Andersen contributed to the development of recent versions of the prototype.

Our feature-specific profiler has successfully been used, among others, to diagnose and resolve performance issues in gradually typed programs (Takikawa et al. 2015).

2 Some of the material in part III of this dissertation appeared in Vincent St-Amour, Leif Andersen, and Matthias Felleisen. Feature-specific profiling. In *Proc. CC*, 2015.

3 `https://pkgs.racket-lang.org/`

4 `https://github.com/stamourv/feature-profile`

13 FEATURE CORPUS

In principle, a feature-specific profiler should support all the features that a language offers or that the author of a library may create. This section presents the Racket features that our prototype feature-specific profiler supports, which includes features from the standard library, and from three third-party libraries. The choice is partially dictated by our underlying instrumentation technology, which can deal with linguistic features whose dynamic extent obeys a stack-like behavior.

The list introduces each feature and outlines the information the profiler provides about each. We have identified the first six features below as causes of performance issues in existing Racket programs. Marketplace processes hinder reasoning about performance while not being expensive themselves. The remaining constructs are considered expensive and are often first on the chopping block when programmers optimize programs, but our tool does not discover a significant impact on performance in ordinary cases. A feature-specific profiler can thus dispel the myths surrounding these features by providing measurements.

13.1 Contracts

Behavioral software contracts are a linguistic mechanism for expressing and dynamically enforcing specifications. They were introduced in Eiffel (Meyer 1992) and have since spread to a number of platforms including Python, JavaScript, .NET and Racket.

When two components—e.g., modules or classes—agree to a contract, any value that flows from one component to the other must conform to the specification. If the value satisfies the specification, program execution continues normally. Otherwise, an exception is raised. Programmers can write contracts using the full power of the host language. Because contracts are checked dynamically, however, computationally intensive specifications can have a significant impact on program performance.

For specifications on objects (Strickland and Felleisen 2010), structures (Strickland et al. 2012) or closures (Findler and Felleisen 2002), the cost of checking contracts is non-local. The contract system defers checking until methods are called or fields are accessed, which happens after crossing the contract boundary. To predict how often a given contract is checked, programmers must understand where the contracted value may flow. Traditional profilers attribute costs to the driver.rkt

```
(require "http-client.rkt" "crawler.rkt")
(define fetch (make-fetcher "fetcher/1.0"))
(define crawl (make-crawler fetch))
... (crawl "racket-lang.org") ...
```

http-client.rkt

```
(provide
  (contract-out
    [make-fetcher (-> user-agent? (-> safe-url? html?))]))
(define (make-fetcher user-agent) (lambda (url) ...))
(define (safe-url? url) (member url whitelist))
```

Figure 34: Contract for an HTTP client

location where contracts are checked, leaving it to programmers to trace those costs to specific contracts.

Figure 34 shows an excerpt from an HTTP client library. It provides `make-fetcher`, which accepts a user agent and returns a function that performs requests using that user agent. The HTTP client accepts only those requests for URLs that are on a whitelist, which it enforces with the underlined contract. The `driver` module creates a crawler that uses a fetching function from the `http-client` module. The crawler then calls this function to access web pages, triggering the contract each time. Because checking happens while executing crawler code, a traditional profiler attributes contract costs to `crawl`, but it is the contract between `http-client` and `driver` that is responsible.

Because of the difficulty of reasoning about the cost of contracts, performance-conscious programmers often avoid them. This, however, is not always possible. First, the Racket standard library uses contracts pervasively to preserve its internal invariants and provide helpful error messages. Second, many Racket programs combine untyped components written in Racket with components written in Typed Racket. To preserve the invariants of typed components, Typed Racket inserts contracts at boundaries between typed and untyped components (Tobin-Hochstadt and Felleisen 2006). Because these contracts are necessary for Typed Racket's safety and soundness, they cannot be avoided.

To provide programmers with an accurate view of the costs of contracts and their actual sources, our profiler provides several contract-related reports and visualizations.

13.2 Output

Programs that interact heavily with files, the console or the network may spend a significant portion of their running time in the I/O subsystem. In these cases, optimizing the program's computation is of limited usefulness; instead programmers must be aware of I/O costs.

Our tool profiles the time programs spend in Racket's output subsystem and traces it back to individual output function[1] call sites.

13.3 Generic Sequence Dispatch

Racket's `for` loops operate on any sequence datatype. This includes built-in types such as lists, vectors, hash-tables and strings as well as user-defined datatypes that implement the `sequence` interface. For example, the first loop in figure 35 checks for the existence of all files contained in `source-files`, regardless of which type of sequence it is. While this genericity encourages code reuse, it introduces runtime dispatch. For loops whose bodies do not perform much work, the overhead from dispatch can dwarf the cost of the loop's actual work.

To alleviate this cost, programmers can manually specialize their code for a specific type of sequence by providing type hints to iterations forms. The second loop in figure 35 uses the `in-list` type hint and is only valid if `source-files` is a list; it throws an exception if `source-files` is, e.g., a vector. This eliminates dispatch overhead, but is not always desirable from a software engineering perspective. Nevertheless, specializing sequences is often one of the first changes experienced Racket programmers perform when optimizing programs.

Our feature-specific profiler reports which iteration forms spend significant time in sequence dispatch to pinpoint which uses would benefit most from specialization. The use of our feature-specific profiler rejects the prejudice that sequence dispatch is always expensive.

13.4 Type Casts and Assertions

Typed Racket, like other typed languages, provides type casts as an "escape hatch". They can help programmers get around the constraints of the type system. Figure 36 shows one case where such an escape hatch is necessary. The program reads (what it expects to be) a list of numbers from standard input. Because users can type in anything they want, the call to `read` is not guaranteed to return a list of numbers, meaning its result must be cast to get around the typechecker.

Like Java's casts, Typed Racket's casts are safe. Runtime checks ensure that a cast's target is of a compatible type, otherwise it throws

1 Racket, like Scheme before it, uses the notion of *ports* to unify different kinds of output, such as files or sockets, and use the same functions for all of them.

```
(define source-files ...)

(for ([file source-files]) ; generic
  (unless (file-exists? file)
    (error "file does not exist" file)))

(for ([file (in-list source-files)]) ; specialized
  (unless (file-exists? file)
    (error "file does not exist" file)))
```

Figure 35: Generic and specialized sequence operations

```
(: mean : (Listof Number) → Number)
(define (mean l) (/ (sum l) (length l)))

(mean (cast (read) (Listof Number)))
```

Figure 36: Taking the mean of a user-provided list of numbers

an exception. As a result, casts can have a negative impact on performance. It can be especially problematic for casts to higher-order types that must wrap their target, causing extra indirections and imposing an overhead similar to that of higher-order contracts.

Typed Racket also provides type assertions as a lighter-weight but less powerful alternative to casts. Assertions also perform dynamic checks but do not support higher-order types and therefore cannot introduce wrapping. Because of this lack of wrapping, assertions are usually preferable to casts when it comes to performance.

Our feature-specific profiler reports time spent in each cast and assertion site in the program.

13.5 Parser Backtracking

The Parsack parsing library[2] provides a disjunction operator that attempts to parse alternative non-terminals in sequence. The operator backtracks in each case unless the non-terminal successfully matches. When the parser backtracks, however, any work it did for matching that non-terminal does not contribute to the final result and is wasted.

Hence, ordering non-terminals within disjunctions to reduce backtracking, e.g., by putting infrequently matched non-terminals at the

2 https://github.com/stchang/parsack

end, can significantly improve parser performance. Our feature-specific profiler reports time spent on each disjunction branch from which the parser ultimately backtracks.

13.6 Shill Security Policies

The Shill language (Moore et al. 2014) restricts how scripts can use system resources according to user-defined security policies. Shill enforces policies dynamically, by interposing dynamic checks on each security-sensitive operation. These checks incur run-time overhead.

Because Shill is implemented as a Racket extension, it is an ideal test case for our feature-specific profiler. Our tool succeeds in reporting time spent enforcing each policy.

13.7 Marketplace Processes

The Marketplace library (Garnock-Jones et al. 2014) allows programmers to express concurrent systems functionally as trees of sets of processes grouped within task-specific virtual machines (VMs)[3] that communicate via a publish and subscribe mechanism. Marketplace is especially suitable for building network services; it has been used as the basis of an SSH (Ylönen and Lonvick 2006) server (see section 17.1.4) and a DNS (Mockapteris 1987) server. While organizing processes in a hierarchy of VMs has clear software engineering benefits, deep VM nesting hinders reasoning about performance. Worse, different processes often execute the same code, but because these processes do not map to threads, traditional profilers may attribute all the costs to one location.

Our feature-specific profiler overcomes both of these problems. It provides process accounting for their VMs and processes and maps time costs to individual processes, e.g., the authentication process for an individual SSH connection, rather than the authentication code shared among all processes. For VMs, it reports aggregate costs and presents their execution time broken down by children.

13.8 Pattern Matching

Racket sports an expressive and extensible pattern matcher (Tobin-Hochstadt 2011); users can write pattern-matching plug-ins to implement, e.g., custom views on data structures or new types of binding patterns.

Racket programmers often worry that pattern matching introduces more predicate tests and control transfers than handwritten code

3 These VMs are process containers running within a Racket OS-level process. The relationship with their more heavyweight cousins such as VirtualBox, or the JVM, is one of analogy only.

would, resulting in less efficient programs. In reality, Racket's pattern matching library uses a sophisticated pattern compiler (Le Fessant and Maranget 2001) that normally generates highly efficient code. However, some features of the pattern matcher, such as manual backtracking, carry additional costs and must be used with caution.

Our feature-specific profiler reports time spent in individual pattern matching forms, excluding time in the user-provided form bodies. With the pattern matching plug-in, we confirm that uses of pattern matching usually run efficiently.

13.9 Method Dispatch

On top of its functional core, Racket supports object-oriented programming with first-class classes (Flatt et al. 2006). In comparison to function calls, though, method calls have a reputation for being expensive.

In practice, the overhead of method calls is usually dwarfed by the cost of the work inside method bodies. Since most operations, even those inside method bodies, are performed using function calls, method dispatch is not as omnipresent in Racket as in most object-oriented languages. As a consequence, it is rarely a bottleneck in Racket programs.

Our tool profiles the time spent performing method dispatch for each method call site, reporting the rare cases where dispatch costs are significant.

13.10 Optional and Keyword Argument Functions

Like many languages, Racket offers functions that support optional positional arguments as well as keyword-based non-positional arguments. These features allow for transparently extending APIs without breaking backwards compatibility, and for writing highly configurable interfaces.[4]

Figure 37 provides two examples of API extension. The `member?` function, which checks whether a value is contained in a list, optionally takes an equality function for list elements. Racket's `sort` function accepts an optional `#:key` keyword argument, which specifies the part of the elements that should be used for comparison.

To support these additional modes, the Racket compiler provides a special function call protocol, distinct from and far less efficient than the regular protocol. As a result, some Racket programmers are reluctant to use optional and keyword argument functions in performance-sensitive code.

4 The `serve/servlet` function—one of the main entry points of the Racket web server (Krishnamurthi et al. 2007)—supports 27 optional keyword arguments, which control various aspects of web server configuration.

```
(define books
  (list war-and-peace les-misérables don-quixote))


> (member? anna-karenina books)
#f
> (member? anna-karenina books same-author?)
#t
> (sort books > #:key book-page-number)
(list
 (book "Les Misérables" "Victor Hugo" 1488)
 (book "War and Peace" "Leo Tolstoy" 1440)
 (book "Don Quixote" "Miguel de Cervantes" 1169))
```

Figure 37: Example use of optional and keyword arguments

The reality is less bleak than imagined. The special protocol is only necessary when calling "unknown"[5] functions with optional or keyword arguments. In all other cases, uses of the protocol can be eliminated statically. Unknown uses account for only a small portion of all uses of optional and keyword argument functions, and the special protocol is therefore rarely a problem in practice.

To inform programmers about the true cost of optional and keyword argument functions, our feature-specific profiler reports time spent performing the special function call protocol for individual functions.

5 Unknown functions are typically functions used in a higher-order way or those bound to mutated variables.

14 PROFILING SIMPLE FEATURES

Because programmers may create new features, our profiler consists of two parts: the core framework and feature-specific plug-ins. The core is a sampling profiler with an API that empowers the implementors of linguistic features to create plug-ins for their creations.

The core part of our profiler employs a sampling-thread architecture to detect when programs are executing certain pieces of code. When a programmer turns on the profiler, the program spawns a separate sampling thread, which inspects the stack of the main thread at regular intervals. Once the program terminates, an offline analysis deals with the collected stack information, looking for feature-specific stack markers and producing programmer-facing reports.

The feature-specific plug-ins exploit this core by placing markers on the control stack that are unique to their construct. Each marker indicates when a feature executes its specific code. The offline analysis can then use these markers to attribute specific slices of time consumption to a feature.

For our Racket-based prototype, the plug-in architecture heavily relies on Racket's continuation marks (Clements et al. 2001), an API for stack inspection. Since this API differs from stack inspection protocols in other languages, the first section of this chapter provides background on continuation marks. The second explains how the implementer of a feature uses continuation marks to interact with the profiler framework for structurally simple constructs. The last section presents the offline analysis.

14.1 Inspecting the Stack with Continuation Marks

Any program may use continuation marks to attach key-value pairs to stack frames and retrieve them later. Racket's API provides two main operations:

- `(with-continuation-mark key value expr)`, which attaches (`key`, `value`) to the current stack frame and evaluates `expr`.
- `(current-continuation-marks [thread])`, which walks the stack and retrieves all key-value pairs from the stack of an optionally specified thread, which defaults to the current thread. This allows one thread to inspect the stack of another.

Programs can also filter marks to consider only those with relevant keys using

```
; Tree = Number | [List Number Tree Tree]
; paths : Tree -> [Listof [Listof Number]]
(define (paths t)
  (cond
    [(number? t)
     (list (cons t (continuation-mark-set->list
                    (current-continuation-marks)
                    'paths)))]
    [else
     (with-continuation-mark 'paths (first t)
       (append (paths (second t)) (paths (third t))))]))

> (paths '(1 (2 3 4) 5))
'((3 2 1) (4 2 1) (5 1))
```

Figure 38: Recording paths in a tree with continuation marks

- `(continuation-mark-set->list marks key)`, which returns the list of values with that key contained in `marks`.

Outside of these operations, continuation marks do not affect a program's semantics.[1]

Figure 38 illustrates the working of continuation marks with a function that traverses binary trees and records paths from leaves to the root. Whenever the function reaches an internal node, it leaves a continuation mark recording that node's value. When it reaches a leaf, it collects those marks, adds the leaf to the path and returns the completed path.

Continuation marks are extensively used in the Racket ecosystem, notably for the generation of error messages in DrRacket (Findler et al. 2002), an algebraic stepper (Clements et al. 2001), the DrRacket debugger, for thread-local dynamic binding (Dybvig 2009), and for exception handling. Serializable continuations in the PLT web server (McCarthy 2010) are also implemented using continuation marks.

The built-in Racket statistical profiler, written by Eli Barzilay, relies on continuation marks and sampling. We reused some of the techniques it uses in the process of building our feature-specific profiling prototype.

1 Continuation marks also preserve proper tail call behavior.

```
(define-syntax (assert stx)
  (syntax-case stx ()
    [(assert v p) ; the compiler rewrites this to:
     (quasisyntax
       (let ([val v] [pred p])
         (with-continuation-mark
           'TR-assertion (unsyntax (source-location stx))
           (if (pred val) val (error "assertion")))))]))
```

Figure 39: Instrumentation of assertions (excerpt)

14.2 Feature-specific Data Gathering

During program execution, feature plug-ins leave feature markers on the stack. The core profiler gathers these markers concurrently, using a sampling thread.

Marking The author of a feature-specific plug-in must change the implementation of the feature so that instances mark themselves with *feature marks*. These marks allow the profiler to observe whether a thread is currently executing code related to a feature. It suffices to wrap the relevant code with `with-continuation-mark`.

Figure 39 shows an excerpt from the instrumentation of Typed Racket assertions. The underlined conditional is responsible for performing the actual assertion. The feature mark's key should uniquely identify the feature. In this case, we use the symbol `'TR-assertion` as key. Unique choices avoid false reports and interference by distinct plug-ins. By using distinct keys for each feature, plug-ins compose naturally. As a consequence, our feature-specific profiler can present a unified report to users; it also implies that users need not select in advance the constructs they deem problematic.

The mark value—or *payload*—can be anything that identifies the instance of the feature to which the cost should be assigned. In figure 39, the payload is the source location of a specific assertion, which allows the profiler to compute the cost of individual assertions.

Writing such plug-ins is simple and involves only non-instrusive, local code changes, but it does require access to the implementation of the feature of interest. Because it does not require any specialized profiling knowledge, however, it is well within the reach of the authors of linguistic constructs.

Antimarking Features are seldom "leaves" in a program; feature code usually runs user code whose execution time may not have to count towards the time spent in the feature. For example the pro-

```
(define-syntax (lambda/keyword stx)
  (syntax-case stx ()
    [(lambda/keyword formals body)
     ; the compiler rewrites this to:
     (quasisyntax
       (lambda (unsyntax (handle-keywords formals))
         (with-continuation-mark
           'kw-opt-protocol (unsyntax (source-location stx))
           (; parse keyword arguments
            ; compute default values
            ; ...
            (with-continuation-mark
              'kw-opt-protocol 'antimark
              body)))))])) ; body is use-site code
```

Figure 40: Use of antimarks in instrumentation

filer must not count the time spent in function bodies towards the function call protocol for keyword arguments.

To solve this problem, a feature-specific profiler expects *antimarks* on the stack. Such antimarks are continuation marks with a distinguished value that delimit a feature's code. Our protocol dictates that the continuation mark key used by an antimark be the same as that of the feature it delimits and that they use the `'antimark` symbol as payload. Figure 40 illustrates the idea with code that instruments a simplified version of Racket's optional and keyword argument protocol. In contrast, assertions do not require antimarks because no non-feature code executes inside the marked region.

The analysis phase recognizes antimarks and uses them to cancel out feature marks. Time is attributed to a feature only if the most recent mark is a feature mark. If it is an antimark, the program is currently executing user code, which should not be counted.

Sampling During program execution, a sampling thread periodically collects and stores continuation marks from the main thread, along with timestamps. The sampling thread has knowledge of the keys used by feature marks and collects marks for all features at once.

14.3 Analyzing Feature-specific Data

After the program execution terminates, the core profiler analyzes the data collected by the sampling thread to produce a cost report.

Cost assignment The profiler uses a standard sliding window technique to assign a time cost to each sample. The time cost of a sample is the sum of the duration of the time interval between it and its predecessor and that of the one between it and its successor, divided by two. Only samples with a feature mark as the most recent mark contribute time towards features.

Payload grouping As explained in section 14.2, payloads identify individual feature instances. Our accounting algorithm groups samples by payload and adds up the cost of each sample; the sums correspond to the cost of each feature instance. Our profiler then generates reports for each feature, using payloads as keys and time costs as values.

Report composition Finally, after generating individual feature reports, our profiler combines them into a unified report. Features absent from the program or too inexpensive to ever be sampled are pruned to avoid clutter. The report lists features in descending order of cost and does likewise for instances within feature reports.

Figure 41 shows a feature profile for a Racket implementation of the FizzBuzz[2] program with an input of 10,000,000. Most of the execution time is spent printing numbers not divisible by either 3 or 5 (line 16), which includes most natural numbers. About a second is spent in generic sequence dispatch; the `range` function produces a list, but the `for` iteration form accepts all sequences and must therefore process its input generically.

2 `http://imranontech.com/2007/01/24/`

```
10  (define (fizzbuzz n)
11    (for ([i (range n)])
12      (cond
13       [(divisible i 15) (printf "FizzBuzz\n")]
14       [(divisible i 5)  (printf "Buzz\n")]
15       [(divisible i 3)  (printf "Fizz\n")]
16       [else             (printf "~a\n" i)])))
17
18  (feature-profile
19    (fizzbuzz 10000000))
```

```
Output accounts for 68.22% of running time
      (5580 / 8180 ms)
  4628 ms : fizzbuzz.rkt:16:24
  564 ms : fizzbuzz.rkt:15:24
  232 ms : fizzbuzz.rkt:14:24
  156 ms : fizzbuzz.rkt:13:24

Generic sequences account for 11.78% of running time
      (964 / 8180 ms)
  964 ms : fizzbuzz.rkt:11:11
```

Figure 41: Feature profile for FizzBuzz

15 EXTENSION: PROFILING STRUCTURE-RICH FEATURES

The basic architecture assumes that the placement of a feature and the location where it incurs run-time costs are the same or in one-to-one correspondence. In contrast to such *structurally simple* features, others cause time consumption in many different places or several different instances of a construct may contribute to a single cost center. We refer to the latter two kinds of linguistic features as *structurally rich*.

While the creator of a structurally rich feature can use a basic plug-in to measure some aspects of its cost, it is best to adopt a different strategy for evaluating such features. This section shows how to go about such an adaptation. Section 17.2 illustrates with an example how to migrate from a basic plug-in to one appropriate for a structurally rich feature.

15.1 Custom Payloads

Instrumentation for structure-rich features uses arbitrary values as mark payloads instead of locations.

Contracts Our contract plug-in uses *blame objects* as payloads. A blame object explains contract violations and pinpoints the faulty party; every time an object traverses a higher-order contract boundary, the contract system attaches a blame object. Put differently, a blame object holds enough information—the contract to check, the name of the contracted value, and the names of the components that agreed to the contract—to reconstruct a complete picture of contract checking events. The contract system creates blame objects as part of its regular operation, and uses those to generate error messages.

Marketplace processes The Marketplace plug-in uses process names as payloads. Since `current-continuation-marks` gathers all the marks currently on the stack, the sampling thread can gather *core samples*.[1] Because Marketplace VMs are spawned and transfer control using function calls, these core samples include not only the current process but also all its ancestors—its parent VM, its grandparent, etc.

Parser backtracking The Parsack plug-in combines three values into a payload: the source location of the current disjunction, the index of the active branch within the disjunction, and the offset in

1 In analogy to geology, a core sample includes marks from the entire stack.

the input where the parser is currently matching. Because parsing a term may require recursively parsing sub-terms, the Parsack plug-in gathers core samples that allow it to attribute time to all active non-terminals.

While storing rich payloads is attractive, plug-in writers must avoid excessive computation or allocation when constructing payloads. Even though the profiler uses sampling, payloads are constructed every time feature code is executed, whether or not the sampler observes it.

15.2 Analyzing Structure-Rich Features

Programmers usually cannot directly digest information generated via custom payloads. If a feature-specific plug-in uses such payloads, its creator should implement an analysis pass that generates user-facing reports.

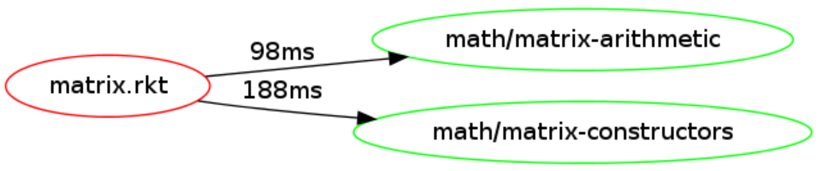
Contracts The goal of the contract plug-in is to report which pairs of parties impose contract checking, and how much the checking costs. Hence, the analysis aims to provide an at-a-glance overview of the cost of each contract and boundary.

To this end, our analysis generates a *module graph* view of contract boundaries. This graph shows modules as nodes, contract boundaries as edges and contract costs as labels on edges. Because typed-untyped boundaries are an important source of contracts, the module graph distinguishes typed modules (in green) from untyped modules (in red). To generate this view, our analysis extracts component names from blame objects. It then groups payloads that share pairs of parties and computes costs as discussed in section 14.3. The middle part of figure 42 shows the module graph for a program that constructs two random matrices and multiplies them. This code resides in an untyped module, but the matrix functions of the `math` library reside in a typed module. Hence linking the client and the library introduces a contract boundary between them.

In addition to the module graph, our feature-specific profiler provides two other views as well. The bottom portion of figure 42 shows the *by-value* view, which provides fine-grained information about the cost of individual contracted values. The *boundary view* provides the same information as the *by-value* view, but grouped graphically by component. It is useful when identifying functions or components that could benefit from being moved across a boundary.

Marketplace Processes The goal of feature-specific analysis for Marketplace processes is to assign costs to individual processes and VMs, as opposed to the code they execute. Marketplace feature

```
(define (random-matrix)
  (build-matrix 200 200
    (lambda (i j) (random))))

(feature-profile
 (matrix* (random-matrix) (random-matrix)))
```



```
Contracts account for 47.35% of running time
      (286 / 604 ms)
  188 ms : build-matrix
           (-> Int Int (-> any any any) Array)
  88 ms : matrix-multiply-data
          (-> Array Array (values any any any any any
                                  (-> any))))
  10 ms : make-matrix-multiply
          (-> Int Int Int (-> any any any) Array)
```

Figure 42: Module graph and by-value views of a contract boundary

```
==============================================================
Total Time  Self Time      Name                         Local%
==============================================================
100.0%      32.3%       ground
                          (tcp-listener 5999 ::1 53588)  33.7%
                          tcp-driver                      9.6%
                          (tcp-listener 5999 ::1 53587)   2.6%
                          [...]
33.7%       33.7%       (tcp-listener 5999 ::1 53588)
2.6%        2.6%        (tcp-listener 5999 ::1 53587)
[...]
```

Figure 43: Marketplace process accounting (excerpt)

marks use the names of processes and VMs as payloads, which allows the plug-in to distinguish separate processes executing the same code.

Our analysis uses full core samples to attribute costs to VMs based on the costs of their children. These core samples record the entire ancestry of processes in the same way the call stack records function calls. We exploit that similarity and reuse standard edge profiling techniques to attribute costs to the entire ancestry of a process.

Figure 43 shows the accounting from a Marketplace-based echo server. The first entry of the profile shows the ground VM, which spawns all other VMs and processes. The rightmost column shows how execution time is split across the ground VM's children. Of note are the processes handling requests from two clients. As reflected in the profile, the client on port 53588 is sending ten times as much input as the one on port 53587.

Parser backtracking The feature-specific analysis for Parsack determines how much time is spent backtracking for each branch of each disjunction. The source locations and input offsets in the payload allows the plug-in to identify each unique visit that the parser makes to each disjunction during parsing.

We detect backtracking as follows. Because disjunctions are ordered, the parser must have backtracked from branches 1 through $n-1$ once it reaches the nth branch of a disjunction. Therefore, whenever the analysis observes a sample from branch n of a disjunction at a given input location, it attributes backtracking costs to the preceding branches. It computes that cost from the samples taken in these branches at the same input location. As with the Marketplace plug-in, the Parsack plug-in uses core samples and edge profiling to handle the recursive structure of the parsing process.

Figure 44 shows a simple parser that first attempts to parse a sequence of `b`s followed by an `a`, and in case of failure, backtracks in order to parse a sequence of `b`s. The bottom portion of figure 44

```
(define $a (compose $b (char #\a)))
(define $b (<or> (compose (char #\b) $b)
                 (nothing)))
(define $s (<or> (try $a) $b))

(feature-profile (parse $s input))
```

```
Parsack Backtracking
====================================
Time (ms / %)   Disjunction   Branch
====================================
2076    46%     ab.rkt:29:12    1
```

Figure 44: A Parsack-based parser and its backtracking profile

shows the output of the feature-specific profiler when running the parser on a sequence of 9,000,000 bs. It confirms that the parser had to backtrack from the first branch after spending almost half of the program's execution attempting it. Swapping the `$a` and `$b` branches in the disjunction eliminates this backtracking.

16 EXTENSION: INSTRUMENTATION CONTROL

As described so far, plug-ins insert continuation marks regardless of whether a programmer wishes to profile or not. We refer to this mode of operation as *active marking*. For features where individual instances perform a significant amount of work, such as contracts, the overhead of feature marks is usually not significant (section 17.3). For other features, such as fine-grained console output where the aggregate cost of individually inexpensive instances is significant, the overhead of marks can be problematic. In such situations, programmers want to control *when* marks are applied on a by-execution basis.

In addition, programmers may also want to control *where* mark insertion takes place to avoid reporting costs in code that they cannot modify or wish to ignore. For instance, reporting that some function in the standard library performs a lot of pattern matching is useless to most programmers; they cannot fix it.

To establish control over the *when* and *where* of continuation marks, our framework introduces the notion of *latent marks*. A latent mark is an annotation that can be turned into an active mark by a preprocessor or a compiler pass. We distinguish between *syntactic latent marks* for use with features implemented using meta-programming and *functional latent marks* for use with library or runtime functions.

16.1 Syntactic Latent Marks

Syntactic latent marks exist as annotations on the intermediate representation (IR) of programs. To add a latent mark, the implementation of a feature leaves tags[1] on the residual program's IR instead of directly inserting feature marks. These tags are discarded after compilation and thus have no run-time effect on the program. Other metaprograms or the compiler can observe latent marks and turn them into active marks.

Our implementation uses Racket's *compilation handler* mechanism to interpose the activation pass between macro-expansion and the compiler's front end with a command-line flag that enables the compilation handler. The compilation handler then traverses the input program, replacing any syntactic latent mark it finds with an active mark. Because latent marks are implicitly present in programs, no library recompilation is necessary. The programmer must merely recompile the code to be profiled.

1 We use Racket's syntax-property mechanism, but any IR tagging mechanism works.

This method applies only to features implemented using meta-programming. Because Racket relies heavily on syntactic extension, most of our plug-ins use syntactic latent marks.

16.2 Functional Latent Marks

Functional latent marks offer an alternative to syntactic latent marks. Instead of tagging the programmer's code, a compiler pass recognizes calls to feature-related functions and rewrites the programmer's code to wrap such calls with active marks. The definition of plug-ins that use functional latent marks must therefore include the list of functions that the activation pass should recognize. To avoid counting the evaluation of arguments to feature-related functions towards their feature, the activation pass adds antimarks around arguments.

Like syntactic latent marks, functional latent marks require recompilation of profiled programs that use the relevant functions. They do not, however, require recompiling libraries that *provide* feature-related functions, which makes them appropriate for functions provided as runtime primitives. Our plug-in for output profiling uses functional latent marks.

Because the activation pass activates functional latent marks at call sites, only first-order uses are counted towards feature time. Higher-order uses are not visible to the profiler. An alternative, wrapper-based implementation—conceptually similar to that of higher-order contracts—would enable profiling higher-order uses.

17 EVALUATION

Our evaluation addresses two promises concerning feature-specific profiling: that measuring in a feature-specific way supplies useful insights into performance problems, and that it is easy to implement new plug-ins. This chapter first presents case studies that demonstrate how feature-specific profiling improves the performance of programs. Then it reports on the amount of effort required to implement plug-ins.

To reduce observer effect, profilers should not add significant overhead to programs. Section 17.3 presents the results from our overhead measurements.

All the execution time results in this chapter are the mean of 30 executions on a 6-core 64-bit Debian GNU/Linux system with 12GB of RAM, with the exception of results for the *grade* benchmark. Because Shill runs only on FreeBSD, results for grade are from a 6-core FreeBSD system with 6GB of RAM. On both systems, we used Racket version version 6.1.1.6 (January 2015). Our plots show error bars marking 95% confidence intervals.

17.1 Case Studies

To be useful, a feature-specific profiler must accurately identify specific uses of features that are responsible for significant performance costs in a given program. Furthermore, an ideal profiler must provide *actionable* information, that is, its reports must point programmers towards solutions. Ideally, it also provides *negative* information, i.e., it confirms that some constructs need not be investigated.

We present five case studies suffering from the overhead of specific features. Each subsection describes a program, summarizes the feedback from the feature-specific profiler, and explains the changes that directly follow from the report. Figure 45 summarizes the features present in each of the case studies. Figure 46 presents the results of comparing execution times before and after the changes. The full set of programs is available online.[1]

17.1.1 *Sound Synthesis Engine*

Our first case study is a sound synthesis engine that I wrote in 2012. The engine uses arrays provided by Racket's `math` library to represent sound signals. It consists of a `mixer` module that handles most of

1 `http://www.ccs.neu.edu/home/stamourv/dissertation/fsp-case-studies.tgz`

Program	Size (SLOC)	Problem feature(s)	Negative Infomation
synth	452	Contracts	Output, generic sequences
maze	758	Output	Casts
grade	330	Security policies	-
ssh	3,762	Contracts, marketplace processes	Generic sequences, pattern matching
markdown	4,058	Backtracking	Pattern matching

Figure 45: Feature usage by case study

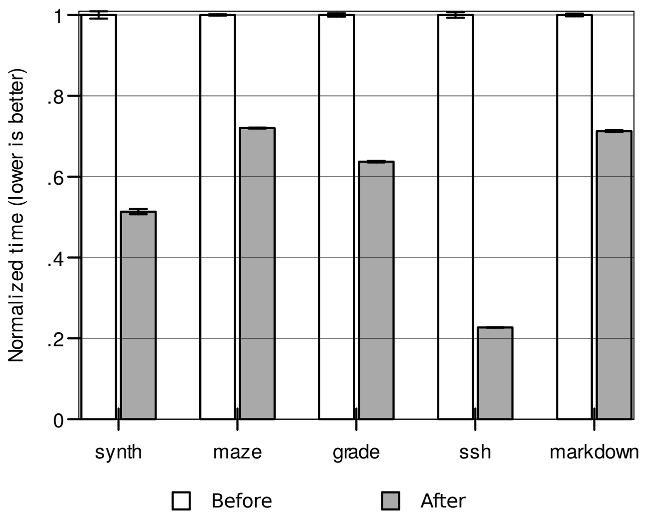


Figure 46: Execution time after profiling and improvements

```
===============================================
Self time            Source location
===============================================
[...]
23.6%    math/typed-array-transform.rkt:207:16
22.5%    basic-lambda9343 (unknown source)
17.8%    math/untyped-array-pointwise.rkt:43:39
14.0%    synth.rkt:86:2
[...]
```

Figure 47: Traditional profile for the synthesizer (excerpt)

```
Contracts account for 73.77% of running time
     (17568 / 23816 ms)
  6210 ms : Array-unsafe-proc
            (-> Array (-> (vectorof Int) any))
  3110 ms : array-append*
            (->* ((listof Array)) (Int) Array)
  2776 ms : unsafe-build-array
            (-> (vectorof Int) [...] Array)
  [...]

Generic sequences account for 0.04% of running time
     (10 / 23816 ms)
  10 ms : wav-encode.rkt:51:16
```

Figure 48: Feature profile for the synthesizer (excerpt)

the interaction with the `math` library, as well as a number of specialized synthesis modules that interface with the mixer, such as function generators, sequencers, and a drum machine. The `math` library uses Typed Racket, and a contract boundary separates it from the untyped synthesis engine. To profile the synthesis engine, the program is run to synthesize ten seconds of music.[2]

Figure 47 shows an excerpt from the output of Racket's traditional statistical profiler, listing the functions in which the program spends most of its time. Two of those, accounting for around 40% of total execution time, are in the `math` library. Such profiling results suggest two potential improvements: optimizing the `math` library or avoiding it altogether. Either solution would be a significant undertaking.

Figure 48 shows a different view of the same execution, provided by our feature-specific profiler. According to the feature profile, al-

2 It synthesizes the opening of Lipps Inc.'s 1980 disco hit Funkytown, specifically.

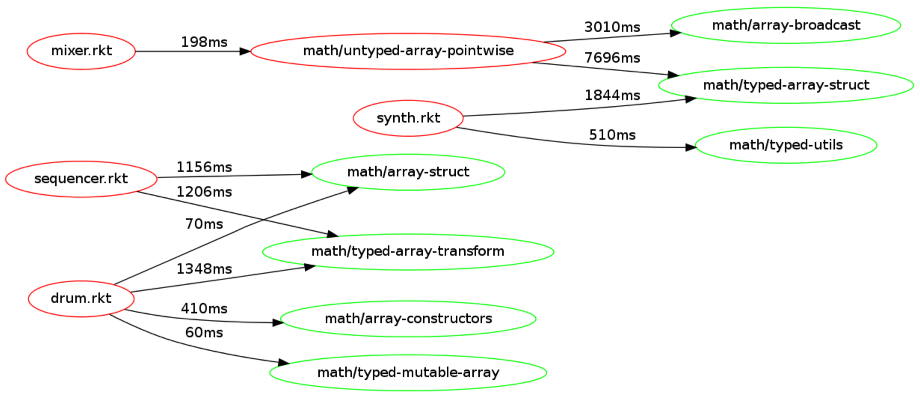


Figure 49: Module graph view for the synthesizer

most three quarters of the program's execution time is spent checking contracts, the most expensive being attached to the `math` library's array functions. Consequently, any significant performance improvements must come from reducing the use of those generated contracts. Optimizing the `math` library itself, as hinted at by the original profile, may not yield an improvement. Because contract checking happens at the typed-untyped boundary, moving code across the boundary to reduce the frequency of crossings may improve matters.

For this, a programmer turns to the module graph view in figure 49 provided by our feature-specific analysis for contracts. According to this view, almost half the total execution time lies between the untyped interface to the `math` library used by the `mixer` module (in red) and the typed portions of the library (in green). This suggests a conversion of the `mixer` module to Typed Racket. This conversion, a 15-minute effort,[3] improves the performance by 1.95×.

Figure 48 also shows that generic sequence operations, while often expensive, do not impose a significant cost in this program despite being used pervasively. Manually specializing sequences would be a waste of time. Similarly, since the report does not feature file output costs, optimizing how the generated signal is emitted as a WAVE file—which is currently done naively—would also be useless.

17.1.2 *Maze Generator*

Our second case study employs a Typed Racket version of a maze generator whose original version is due to Olin Shivers. The program generates a maze on a hexagonal grid, ensures that it is solvable, and prints it.

3 I demonstrated this conversion at RacketCon 2013. A video is available online at `http://www.youtube.com/watch?v=D7uPm3J-o6g`

```
Output accounts for 55.31% of running time
      (1646 / 2976 ms)
  386 ms : maze.rkt:730:2
  366 ms : maze.rkt:731:2
  290 ms : maze.rkt:732:2
  [...]
```

Figure 50: Feature profile for the maze generator (excerpt)

```
; BEFORE
730  (display (if sw #\\ #\space))
731  (display (if s  #\_ #\space))
732  (display (if se #\/ #\space))

; AFTER
(display
  (cond [(and sw       s       se)       "\\_/"]
        [(and sw       s       (not se)) "\\_ "]
        [(and sw       (not s) se)       "\\ /"]
        [(and sw       (not s) (not se)) "\\  "]
        [(and (not sw) s       se)        " _/"]
        [(and (not sw) s       (not se))  " _ "]
        [(and (not sw) (not s) se)        "  /"]
        [(and (not sw) (not s) (not se))  "   "]))
```

Figure 51: Fusing output operations in the maze generator

Figure 50 shows the top portion of the output of our feature-specific profiler. According to the feature profile, 55% of the execution time is spent performing output. Three calls to `display`, each responsible for printing part of the bottom of hexagons, stand out as especially expensive. Printing each part separately results in a large number of single-character output operations. This report suggests fusing all three output operations into one, as figure 51 shows. Following this advice results in a 1.39× speedup.

Inside an inner loop, a dynamic type assertion enforces an invariant that the type system cannot guarantee statically. Even though this might raise concerns with a cost-conscious programmer, the profile reports that the time spent in the cast is negligible.

17.1.3 *Shill-Based Grading Script*

Our third case study involves a Shill script that tests students' OCaml code; it is due to Scott Moore.

According to the feature profile, security policy enforcement accounts for 68% of execution time. Overhead from calling external programs causes most of that slowdown. The script calls out to three external programs, one being OCaml and the other two being text manipulation utilities.

Reimplementing the two text manipulation utilities in Shill avoids crossing security boundaries, which reduces the time spent in policy enforcement. This change results in a 1.57× increase in performance.

17.1.4 *Marketplace-Based SSH Server*

Our fourth case study involves an SSH server[4] written using the Marketplace library. To exercise it, a driver script starts the server, connects to it, launches a Racket read-eval-print-loop on the host, evaluates the arithmetic expression `(+ 1 2 3 4 5 6)`, disconnects and terminates the server.

As figure 52 shows, our feature-specific profiler brings out two useful facts. First, two *spy* processes—the `tcp-spy` process and the boot process of the `ssh-session` VM—account for over 25% of the execution time. In Marketplace, spies are processes that observe others for logging purposes. The SSH server spawns these spy processes even when logging is ignored, resulting in unnecessary overhead.

Second, contracts account for close to 67% of the running time. The module view, of which figure 53 shows an excerpt, reports that the majority of these contracts lie at the boundary between the typed Marketplace library and the untyped SSH server. We can selectively remove these contracts in one of two ways: by adding types to the SSH server or by disabling typechecking in Marketplace.

Disabling spy processes and type-induced contracts[5] results in a speedup of 4.41×. In addition to these two areas of improvement, the feature profile also provides negative information: pattern matching and generic sequences, despite being used pervasively, account for only a small fraction of the server's running time.

4 `https://github.com/tonyg/marketplace-ssh`
5 This change has since been integrated into the Marketplace library.

```
Marketplace Processes
===============================================================
Total Time  Self Time     Name                          Local%
===============================================================
100.0%      3.8%        ground
                          ssh-session-vm                 51.2%
                          tcp-spy                        19.9%
                          (tcp-listener 2322 ::1 44523)  19.4%
                          [...]
51.2%       1.0%        ssh-session-vm
                          ssh-session                    31.0%
                          (#:boot-process ssh-session-vm) 14.1%
                          [...]
19.9%       19.9%       tcp-spy
7.2%        7.2%        (#:boot-process ssh-session-vm)
[...]

Contracts account for 66.93% of running time
     (3874 / 5788 ms)
  1496 ms : add-endpoint
            (-> pre-eid? role? [...] add-endpoint?)
  1122 ms : process-spec
            (-> (-> any [...]) any)
  [...]

Pattern matching accounts for 0.76% of running time
     (44 / 5788 ms)
  [...]

Generic sequences account for 0.35% of running time
     (20 / 5788 ms)
  [...]
```

Figure 52: Profiling results for the SSH server (excerpt)

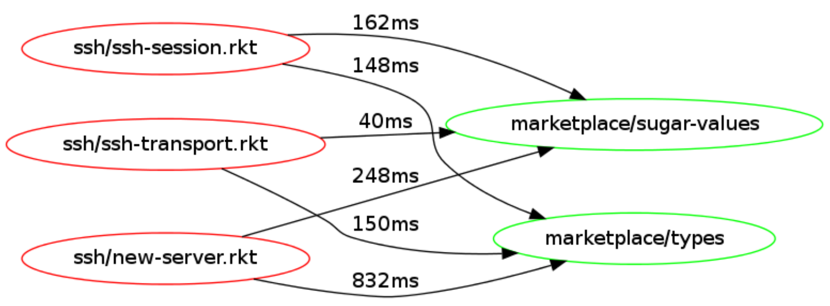


Figure 53: Module graph view for the SSH server (excerpt)

```
Parsack Backtracking
==================================================
Time (ms / %)   Disjunction                 Branch
==================================================
5809.5  34%     markdown/parse.rkt:968:7      8
366.5    2%     parsack/parsack.rkt:449:27    1
313.5    2%     markdown/parse.rkt:670:7      2
[...]

Pattern matching accounts for 0.04% of running time
      (6 / 17037 ms)
  6 ms : parsack/parsack.rkt:233:4
```

Figure 54: Profiling results for the Markdown parser (excerpt)

17.1.5 *Markdown Parser*

Our last case study involves a Parsack-based Markdown parser,[6] due to Greg Hendershott. To profile the parser, we ran it on 1,000 lines of sample text.[7]

As figure 54 shows, backtracking from three branches took noticeable time and accounted for 34%, 2%, and 2% of total execution time, respectively. Based on the tool's report, we moved the problematic branches further down in their enclosing disjunction, which produced a speedup of 1.40×.

For comparison, Parsack's author, Stephen Chang, manually optimized the same version of the Markdown parser using ad-hoc, low-level instrumentation and achieved a speedup of 1.37×. Using our tool, Leif Andersen, with no knowledge of the parser's internals, was able to achieve a similar speedup in only a few minutes of work.

The feature-specific profiler additionally confirmed that pattern matching accounted for a negligible amount of the total running time.

17.2 Plug-in Implementation Effort

Writing feature-specific plug-ins is a low-effort endeavor. It is easily within reach for the authors of linguistic libraries because it does not require advanced profiling knowledge. To support this claim, we start with anecdotal evidence from observing the author of the Marketplace library implement feature-specific profiling support for it.

Mr. Garnock-Jones, an experienced programmer, implemented the plug-in himself, with myself acting as interactive documentation of

6 `https://github.com/greghendershott/markdown`
7 The input is an excerpt from "The Time Machine" by H.G. Wells.

Feature	Instrumentation LOC	Analysis LOC
Contracts	183	672
Output	11	-
Generic sequence dispatch	18	-
Type casts and assertions	37	-
Parser backtracking	18	60+506
Shill security policies	23	-
Marketplace processes	7	9+506
Pattern matching	18	-
Method dispatch	12	-
Optional and keyword arguments	50	-

Figure 55: Instrumentation and analysis LOC per feature

the framework. Implementing the first version of the plug-in took about 35 minutes. At that point, Mr. Garnock-Jones had a working process profiler that performed the basic analysis described in section 14.3. Adding feature-specific analysis took an additional 40 minutes. Less experienced library authors may require more time for a similar task. Nonetheless, we consider this amount of effort to be quite reasonable.

For the remaining features, we report the number of lines of code for each plug-in in figure 55. The third column reports the number of lines of domain-specific analysis code. The basic analysis is provided as part of the framework. The line counts for Marketplace and Parsack count the portions of Racket's edge profiler that are re-linked into the plug-ins separately. They account for 506 lines. With the exception of contract instrumentation—which covers multiple kinds of contracts and is spread across the 16,421 lines of the contract system—instrumentation is local and non-intrusive.

17.3 Instrumentation Overhead

Our feature-specific profiler imposes an acceptably low overhead on program execution. For a summary, see figure 56, which reports overhead measurements.

We use the programs listed in figure 57 as benchmarks. They include three of the case studies from section 17.1,[8] two programs that make heavy use of contracts (*lazy* and *ode*), and six programs from the Computer Language Benchmarks Game[9] that use features supported by our prototype. The full set of programs is available online.[10]

The first column of figure 56 corresponds to programs executing without any feature marks and serves as our baseline. The second

8 We performed the other two case studies after our overhead experiments.
9 `http://benchmarksgame.alioth.debian.org`
10 `http://www.ccs.neu.edu/home/stamourv/dissertation/fsp-overhead.tgz`

column reports results for programs that include only marks that are active by default: contract marks and Marketplace marks. This bar represents the default mode for executing programs without profiling. The third column also includes all activated latent marks. The fourth column includes all of the above as well as the overhead from the sampling thread;[11] it is closest to the user experience when profiling.

With all marks activated, the overhead is lower than 6% for all but two programs, *synth* and *maze*, where it accounts for 16% and 8.5% respectively. The overhead for marks that are active by default is only noticeable for two of the four programs that include such marks, *synth* and *ode*, and account for 16% and 4.5% respectively. Total overhead, including sampling, ranges from 3% to 33%.

Based on these experiments, we conclude that the overhead from instrumentation is quite reasonable in general. The one exception, the *synth* benchmark, involves a large quantity of contract checking for cheap contracts, which is the worst case scenario for contract instrumentation. Further engineering effort could lower this overhead. The overhead from sampling is similar to that of state-of-the-art sampling profilers as reported by Mytkowicz et al. (2010).

Threat to validity We identify one threat to validity. Because instrumentation is localized to feature code, its overhead is also localized. This may cause feature execution time to be overestimated. Because these overheads are low in general, we conjecture this problem to be insignificant in practice. In contrast, sampling overhead is uniformly[12] distributed across a program's execution and should not introduce such biases.

11 We use green threads provided by the Racket runtime system for sampling.
12 This assumes random sampling, which we did not verify.

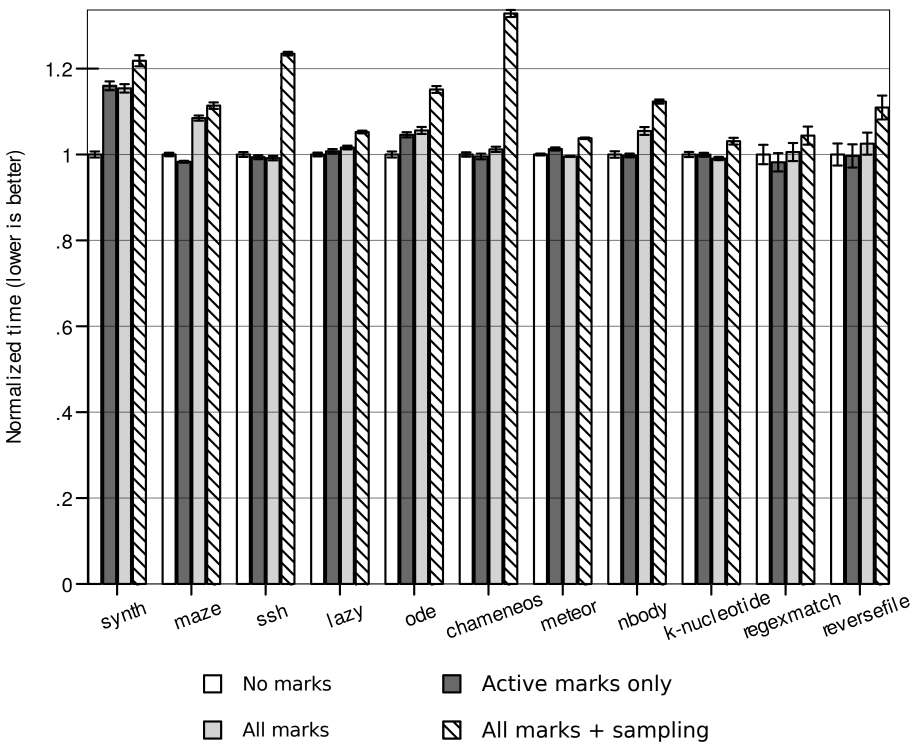


Figure 56: Instrumentation and sampling overhead

Benchmark	Description	Features
synth	Sound synthesizer	Contracts, output, generic sequences, keyword protocol
maze	Maze generator	Output, casts
ssh	SSH server	Contracts, output, generic sequences, casts, marketplace processes, pattern matching, keyword protocol
lazy	Computer vision algorithm	Contracts
ode	Differential equation solver	Contracts
chameneos	Concurrency game	Pattern matching
meteor	Meteor puzzle	Pattern matching
nbody	N-body problem	Casts
k-nucleotide	K-nucleotide frequencies	Generic sequences
regexmatch	Matching phone numbers	Casts, pattern matching
reversefile	Reversing the lines of a file	Output

Figure 57: Benchmark descriptions

18 LIMITATIONS

Our particular approach to feature-specific profiling applies only to certain kinds of linguistic constructs. This chapter describes cases that our feature-specific profiler should but cannot support. Those limitations are not fundamental to the idea of feature-specific profiling and could be addressed by different approaches to data gathering.

18.1 Control Features

Because our instrumentation strategy relies on continuation marks, it does not support features that interfere with marks. This rules out non-local control features that unroll the stack, discarding continuation marks along the way. This makes our approach unsuitable for measuring the cost of, e.g., exception raising.

18.2 Non-Observable Features

The sampler must be able to observe a feature in order to profile it. This rules out uninterruptible features, e.g., struct allocation or FFI calls, which do not allow the sampling thread to be scheduled during their execution. Other obstacles to observability include sampling bias (Mytkowicz et al. 2010) and instances that execute too quickly to be reliably sampled.

18.3 Diffuse features

Some features, such as garbage collection, have costs that are diffused throughout the program. This renders mark-based instrumentation impractical. An event-based approach, such as Morandat et al.'s (2012), would fare better.

19 FEATURE-SPECIFIC PROFILING BEYOND RACKET

The previous chapters provide evidence that feature-specific profiling is both feasible and useful in the context of Racket. This chapter sketches how the idea could be applied beyond Racket, and briefly discusses ongoing efforts in that direction.

19.1 Continuation Marks Beyond Racket

The key components of the framework described in the previous chapters are a sampling thread and feature marks. Therefore, to instantiate the framework in another language, both components must be expressible. Sampling threads only require multi-threading support and timers, both of which are commonplace in mainstream programming languages.

Continuation marks, in contrast, are not as widespread. In addition to Racket, they have been implemented in the context of Microsoft's CLR (Pettyjohn et al. 2005) and JavaScript (Clements et al. 2008). Both of these implementations are sufficient to express feature marks.

Clement's dissertation (2006) argues that continuation marks are easy to add to existing language implementations. Transitively, this would make it easy to instantiate our framework. At the time of this writing, there is ongoing work by Leif Andersen to implement continuation marks for the R language,[1] with the goal of subsequently implementing a feature-specific profiler.

In the absence of continuation marks per se, other related features may be subtitutable instead. For example, stack reflection mechanisms such as those provided by Smalltalk systems should suffice to implement feature marks. Similarly, the stack introspection used by the GHCi debugger (Marlow et al. 2007) may also be suitable.

19.2 Profiling Beyond Stack Inspection

While our proposed framework relies on feature marks and sampling for its instrumentation, other profiling approaches could be substituted instead.

Event-based profiling looks particularly promising. By event-based profiling, we mean emitting events—e.g., log messages—whenever some event of interest occurs, along with a timestamp. In the context of feature-specific profiling, these events of interest would be entering and leaving feature code and would replace the use of feature marks.

1 `http://www.r-project.org/`

Antimarking would be trivially supported by emitting exit and entry events before and after entering and exiting user code, respectively. Aside from the computation of time costs, which would become even more straightforward than with our sampling-based approach, the analysis side of the framework would remain unchanged.

Because most languages provide logging libraries or equivalents, this approach should make feature-specific profiling broadly accessible. In addition, an event-based approach would address all three limitations described in chapter 18, which our framework inherits from sampling-based profiling.

An event-based approach, however, may impose a higher overhead than a sampling-based approach. Logging, for example, may involve allocating a message and adding it to an internal buffer. In turn, the latter operation may require synchronization if the logging system can operate across multiple threads. These operations are likely to be more expensive than pushing a key-value pair on the stack. Furthermore, like marking but unlike sampling, this overhead would be incurred every time feature code is entered and exited. An analog to latent marks may help reduce this overhead when the profiler is not in use, but the problem otherwise remains. Dynamic analyses that rely on exhaustive instrumentation face similar problems; an event-based feature-specific profiler may be able to reuse solutions designed for that domain context (Ha et al. 2009).

Of course, it should be possible to implement a feature-specific profiler that uses continuation marks and sampling for some features and events for others. This would allow plug-in authors to pick the instrumentation strategy that is best suited for their feature.

Hauswirth et al.'s *software performance monitors* (Hauswirth et al. 2004) are another alternative to sampling. Software performance monitors are analogous to hardware performance monitors but record software-related performance events. These monitors could also possibly be used to implement feature-specific profiling by tracking the execution of feature code.

20 RELATED WORK

Programmers already have access to a wide variety of performance tools that are complementary to feature-specific profilers. This section compares our work to those approaches that are closely related.

20.1 Traditional Profiling

Profilers have been successfully used to diagnose performance issues for decades. They most commonly report on the consumption of time, space and I/O resources. Traditional profilers group costs according to program organization, be it static—e.g., per function—or dynamic—e.g., per HTTP request. Feature-specific profilers group costs according to linguistic features and specific feature instances.

Each view is useful in different contexts. For example, a feature-specific profiler's view is most useful when non-local feature costs make up a significant portion of a program's running time. Traditional profilers may not provide actionable information in such cases.

Furthermore, by identifying costly features, feature-specific profilers point programmers towards potential solutions, namely correcting feature usage. In contrast, traditional profilers often report costs without helping find solutions. Conversely, traditional profilers may detect a broader range of issues than feature-specific profilers, such as inefficient algorithms, which are invisible to feature-specific profilers.

Traditional profilers that report costs in terms of dynamic program organization can provide precise accounting in cases where multiple instances of specific pre-selected features, e.g., threads, share the same code. This is similar to the accounting provided by a feature-specific profiler for features that follow this cost allocation pattern, such as Marketplace processes. These traditional profilers, however, support a fixed set of such features, selected by the tools' authors. In contrast, feature-specific profilers can be extended by library authors to support any such feature.

20.2 Vertical Profiling

A vertical profiler (Hauswirth et al. 2004) attempts to see through the use of high-level language features. It therefore gathers information from multiple layers—hardware performance counters, operating system, virtual machine, libraries—and correlates them into a gestalt of program performance.

Vertical profiling focuses on helping programmers understand how the interaction between layers affects their program's performance. By comparison, feature-specific profiling focuses on helping them understand the cost of features per se. Feature-specific profiling also presents information in terms of features and feature instances, which is accessible to non-expert programmers, whereas vertical profilers report low-level information, which requires a thorough understanding of the compiler and runtime system.

20.3 Alternative Profiling Views

A number of profilers offer alternative views to the traditional attribution of time costs to program locations. Most of these views focus on particular aspects of program performance and are complementary to the view offered by a feature-specific profiler. We briefly mention recent pieces of relevant work that provide alternative profiling views.

Singer and Kirkham (2008) use the notion of concepts (Biggerstaff et al. 1994)—programmer-annotated program portions that are responsible for a given task—to assign costs when profiling programs. Because programmers must identify concepts on their own, they must have prior hypotheses regarding which portions of their programs are likely to be expensive and must therefore be profiled. In contrast, with feature-specific profiling, language and library authors annotate the code relevant to particular features, which frees programmers from having to know what may or may not be expensive.

Listener latency profiling (Jovic and Hauswirth 2011) helps programmers tune interactive applications by reporting high-latency operations, as opposed to operations with long execution times.

Tamayo et al. (2012) present a tool that provide programmers with information on the cost of database operations in their programs and helps them optimize multi-tier applications.

Chang and Felleisen (2014) introduce a profiling-based tool that reports "laziness potential", a measure of how much a given expression would benefit from being evaluated lazily.

20.4 Dynamic Instrumentation Frameworks

Dynamic instrumentation frameworks such as ATOM (Srivastava and Eustace 1994), PIN (Patil et al. 2004), Javana (Maebe et al. 2006) or Valgrind (Nethercote and Seward 2007) serve as the basis for profilers and other kinds of performance tools. These frameworks resemble the use of continuation marks in our framework and could potentially be used to build feature-specific profilers. These frameworks are much more heavy-weight than continuation marks and, in turn, allow more thorough instrumentation, e.g., of the memory hierarchy, of hardware performance counters, etc., and they have not been used to measure the cost of linguistic features.

20.5 Domain-Specific Debugging

Chiş et al. (2014) introduce the idea of *moldable debugging*. A moldable debugger allows programmers to extend it with domain-specific plug-ins, which in turn enable debugging at the level of domain concepts, as opposed to generic implementation-level concepts such as functions and lines of code. These plug-ins may provide domain-specific views and operations to be accessible when certain activation conditions are met during debugging.

Chiş et al. present a prototype moldable debugger for the Pharo Smalltalk system and extend it with plug-ins for four different domains: unit testing, synchronous notifications, parser generation, and browser scripting.

Domain-specific debugging shares goals with feature-specific profiling, namely that tools should provide information at the abstraction level at which programmers think about their programs, i.e., linguistic features or domain concepts. The two approaches, however, target different activities—debugging and profiling, respectively—and are complementary.

From an architectural perspective, the two approaches are also similar. Both are structured around a framework that is extensible using feature/domain-specific plug-ins.

Part IV

CLOSING REMARKS

21 CONCLUSION

Specialized tools can successfully bring the benefits of performance engineering to an audience of non-expert programmers. This dissertation demonstrates this thesis with two classes of such tools: optimization coaches and feature-specific profilers.

Optimization coaching In high-level languages, the performance of programs depends heavily on whether compilers optimize them well. Optimizing compilers are black boxes, however, whose inner workings are opaque to most programmers, who remain at their mercy and may end up with underperforming programs without warning and without obvious solutions.

Optimization coaches open up these black boxes and provide programmers with insight into the optimization process. Coaches further help programmers harness their compilers' optimizers with recommendations that make programs more amenable to optimization.

The construction of optimization coaches relies on general principles: optimization logging (chapter 4), optimization analysis (chapter 5), and recommendation generation (chapter 6). Within the scope of these principles, we identified general concepts and techniques which apply beyond our specific instantiations: optimization failures, near misses, irritants, pruning, targeting, ranking, and merging. Furthermore, we have identified profiling information as a key additional source of information that critically complements a coach's metrics and heuristics.

We have confirmed the applicability of these principles by building two prototype coaches: one for the Racket and Typed Racket compilers, and another for the SpiderMonkey JavaScript engine. A tool derived from the latter prototype is about to be released as part of Firefox. The existence of two distinct prototypes provides evidence for the generality of these principles.

Our experiments with these prototypes show that, using an optimization coach, programmers can improve the performance of their programs significantly. We observed speedups of up to 2.75×. Furthermore, these speedups are achievable with simple, non-intrusive program changes. The most significant changes we recorded consisted of modifying 33 lines of code. And finally, none of these changes required any actual knowledge about the implementation of the optimization process.

Feature-specific profiling Misuse of features provided by languages and libraries is a significant source of performance issues in practice. This problem is exacerbated by the fact that modern programming languages are fundamentally extensible. Any programmer may introduce additional features via new library APIs or, in some languages, even provide new syntactic forms altogether. This state of affairs calls for tools that not only report costs in terms of existing linguistic features, but can also be extended to support new features as programmers introduce them.

Feature-specific profilers are such tools. By reporting the costs that originate from individual feature instances, feature-specific profilers can uncover the source of distributed costs, that is, instances of features which spread their costs across large swaths of the program. Furthermore, this focus on features reduces the solution search space programmers must explore; only program changes that affect the use of a problematic feature can address the costs that it engenders.

Generally speaking, feature-specific profiling relies on a cooperation between feature plug-ins and a core profiler. Plug-ins are responsible for recording, during execution, when the code related to a particular feature is executing. They can perform this task in one of multiple ways, for example by leaving markers on the control stack—as our prototype does—or by emitting entry and exit events. The core profiler is responsible for observing the information produced by feature plug-ins, for instance using sampling, tallying the time spent in each feature, and producing reports for programmers. This separation of responsibilities allows for an extensible collection of plug-ins, which can be added as new features are implemented in languages or libraries.

We have instantiated this blueprint to build a prototype feature-specific profiler for Racket. Our experience using it to diagnose day-to-day performance issues, as well as our experiments, show successful results. We have observed performance improvements of close to 5× as a result of addressing feature misuses pointed out by our tool.

Outlook The ideas and techniques behind optimization coaching and feature-specific profiling should apply beyond the specific instantiations presented in this dissertation. We imagine that our experiences may inspire others to create similar tools for their languages, compilers, and libraries of choice. Indeed, the techniques described in this dissertation can serve as a starting point.

There is much work to be done beyond our initial forays into optimization coaching and feature-specific profiling. For instance, the optimizations and features we studied are but a small subset of those programmers use. Further work is needed to confirm whether optimization coaching can apply to other commonly-used optimizations such as common subexpression elimination or loop-invariant code motion. Similarly, we have yet to see whether feature-specific profiling applies to other features in common use, such as exception handling. We also do not know whether feature-specific profiling is useful beyond Racket, though preliminary results from Leif Andersen's work on bringing it to the R language are promising.

More generally, our methodology of gathering information from compilers and runtime systems and involving programmers in the optimization process should generalize to other performance engineering topics, beyond compiler optimization and linguistic feature usage. One could imagine using a similar paradigm to involve programmers in system-level aspects of program performance—perhaps by recording information at the operating system level to advise programmers on virtual memory usage, or by gathering data from the network stack to recommend more efficient network usage patterns—or to assist them in reducing the power consumption of their programs—for example by identifying code responsible for excessive processor wakeups.

Outside of performance engineering, programmers could benefit from targeted advice and higher-level reporting in other aspects of programming as well. Writing secure programs, for instance, requires just as much hard-earned expertise—if not more—than writing efficient programs. Tools that mechanize the knowledge and experience of security experts, and rely on compile- or run-time instrumentation to fill in program-specific information, could go a long way towards helping programmers avoid security pitfalls. Other program requirements such as robustness and portability are similarly hard to fulfill, and programmers could potentially benefit from tool assistance to achieve them.

Automated tools and programmers have different and complementary strengths. Tools can handle large quantities of information and analyze them to look for many different properties. Programmers are aware of the context their programs operate in, and they can exercise judgement as to whether particular changes make sense in that context. Only by combining these strengths—e.g., via the use of programmer-assisted tools—can we effectively tackle the complex challenges of software development.

BIBLIOGRAPHY

Norman Adams, David Kranz, Richard Kelsey, Jonathan Rees, Paul Hudak, and James Philbin. ORBIT: an optimizing compiler for Scheme. In *Proc. Symp. on Compiler Construction*, pp. 219–233, 1986.

Frances E. Allen and John Cocke. A catalogue of optimizing transformations. *Design and Optimization of Compilers*, pp. 1–30, 1972.

Andrew Appel and Trevor Jim. Continuation-passing, closure-passing style. In *Proc. Symp. on Principles of Programming Languages*, pp. 293–302, 1989.

Ted J. Biggerstaff, Bharat G. Mitbander, and Dallas E. Webster. Program understanding and the concept assignment problem. *Commun. ACM* 37(5), pp. 72–82, 1994.

David Binkley, Bruce Duncan, Brennan Jubb, and April Wielgosz. The FeedBack compiler. In *Proc. International Works. on Program Comprehension*, pp. 198–206, 1998.

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

Craig Chambers, David Ungar, and Elgin Lee. An efficient implementation of SELF. In *Proc. Conf. Object-Oriented Programming Systems, Languages, and Applications*, pp. 49–70, 1989.

Stephen Chang and Matthias Felleisen. Profiling for laziness. In *Proc. Symp. on Principles of Programming Languages*, pp. 349–360, 2014.

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. International Conf. on Software Engineering*, pp. 1001–1012, 2014.

Andrei Chiş, Tudor Gîrba, and Oscar Nierstrasz. The moldable debugger: A framework for developing domain-specific debuggers. In *Proc. Conf. on Software Language Engineering*, pp. 102–121, 2014.

John Clements. *Portable and High-Level Access to the Stack with Continuation Marks*. PhD dissertation, Northeastern University, 2006.

John Clements, Matthew Flatt, and Matthias Felleisen. Modeling an algebraic stepper. In *Proc. European Symp. on Programming*, pp. 320–334, 2001.

John Clements, Ayswarya Sundaram, and David Herman. Implementing continuation marks in JavaScript. In *Proc. Scheme and Functional Programming Works.*, pp. 1–10, 2008.

Keith D. Cooper, Mary W. Hall, Robert T. Hood, Ken Kennedy, Kathryn S. McKinley, John M. Mellor-Crummey, Linda Torczon, and Scott K. Warren. The ParaScope parallel programming environment. *Proc. of the IEEE* 81(2), pp. 244–263, 1993.

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

Ryan Culpepper and Matthias Felleisen. Debugging hygienic macros. *Science of Computer Programming* 75(7), pp. 496–515, 2010.

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. *Transactions of Programming Languages and Systems* 13(4), pp. 451–490, 1991.

Danny Dig, Cosmin Radoi, Mihai Tarce, Marius Minea, and Ralph Johnson. ReLooper: Refactoring for loop parallelism. University of Illinois, 2142/14536, 2009.

R. Kent Dybvig. *Chez Scheme Version 8 User's Guide*. Cadence Research Systems, 2009.

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

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

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

Andrew Farmer, Andy Gill, Ed Komp, and Neil Schulthorpe. The HERMIT in the machine. In *Proc. Haskell Symp.*, pp. 1–12, 2012.

Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. DrScheme: a programming environment for Scheme. *J. of Functional Programming* 12(2), pp. 159–182, 2002.

Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *Proc. International Conf. on Functional Programming*, pp. 48–59, 2002.

Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Stephanie Weirich, and Matthias Felleisen. Catching bugs in the web of program invariants. In *Proc. Conf. on Programming Language Design and Implementation*, pp. 23–32, 1996.

Matthew Flatt, Robert Bruce Findler, and Matthias Felleisen. Scheme with classes, mixins, and traits. In *Proc. Asian Conf. on Programming Languages and Systems*, pp. 270–289, 2006.

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

Agner Fog. Software optimization resources. 2012. `http://www.agner.org/optimize/`

Christopher W. Fraser and David R. Hanson. *A Retargetable C Compiler: Design and Implementation*. Addison-Wesley, 1995.

Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proc. of the IEEE, Special issue on "Program Generation, Optimization, and Platform Adaptation"* 93(2), pp. 216–231, 2005.

Saturnino Garcia, Donghwan Jeon, Chris Louie, and Michael Bedford Taylor. Kremlin: Rethinking and rebooting gprof for the multicore age. In *Proc. Conf. on Programming Language Design and Implementation*, pp. 458–469, 2011.

Tony Garnock-Jones, Sam Tobin-Hochstadt, and Matthias Felleisen. The network as a language construct. In *Proc. European Symp. on Programming*, pp. 473–492, 2014.

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

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.

Jungwoo Ha, Matthew Arnold, Stephen M. Blackburn, and Kathryn S. McKinley. A concurrent dynamic analysis framework for multicore hardware. In *Proc. Conf. Object-Oriented Programming Systems, Languages, and Applications*, pp. 155–174, 2009.

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

Brian Hackett and Shu-yu Guo. Fast and precise type inference for JavaScript. In *Proc. Conf. on Programming Language Design and Implementation*, pp. 239–250, 2012.

William von Hagen. *The Definitive Guide to GCC*. Apress, 2006.

Pieter H. Hartel, Marc Feeley, Martin Alt, Lennart Augustsson, Peter Baumann, Marcel Beemster, Emmanuel Chailloux, Christine H. Flood, Wolfgang Grieskamp, John H. G. van Groningen, Kevin Hammond, Bogumił Hausman, Melody Y. Ivory, Richard E. Jones, Jasper Kamperman, Peter Lee, Xavier Leroy, Rafael D. Lins, Sandra Loosemore, Niklas Röjemo, Manuel Serrano, Jean-Pierre Talpin, Jon Thackray, Stephen Thomas, Pum Walters, Pierre Weis, and E.P. Wentworth. Benchmarking implementations of functional languages with "pseudoknot" a float-intensive benchmark. *J. of Functional Programming* 6(4), pp. 621–655, 1996.

Matthias Hauswirth, Peter F. Sweeney, Amer Diwan, and Michael Hind. Vertical profiling: Understanding the behavior of object-oriented applications. In *Proc. Conf. Object-Oriented Programming Systems, Languages, and Applications*, pp. 251–269, 2004.

Reid Holmes and Gail C. Murphy. Using structural context to recommend source code examples. In *Proc. International Conf. on Software Engineering*, pp. 117–125, 2005.

Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proc. European Conf. on Object-Oriented Programming*, pp. 21–38, 1991.

Urs Hölzle, Craig Chambers, and David Ungar. Debugging optimized code with dynamic deoptimization. In *Proc. Conf. on Programming Language Design and Implementation*, pp. 32–43, 1992.

Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and detecting real-world performance bugs. In *Proc. Conf. on Programming Language Design and Implementation*, pp. 77–88, 2012.

Thomas Johnsson. Lambda lifting: Transforming programs to recursive equations. In *Proc. Functional Programming Languages and Computer Architecture*, pp. 190–203, 1985.

Milan Jovic and Matthias Hauswirth. Listener latency profiling: Measuring the perceptible performance of interactive Java applications. *Science of Computer Programming* 19(4), pp. 1054–1072, 2011.

Poul-Henning Kamp. You're doing it wrong: Think you've mastered the art of server performance? Think again. *ACM Queue* 8(6), 2010.

Donald E. Knuth. An empirical study of FORTRAN programs. *Software—Practice and Experience* 1, pp. 105–133, 1971.

Shriram Krishnamurthi, Peter Walton Hopkins, Jay McCarthy, Paul T. Graunke, Greg Pettyjohn, and Matthias Felleisen. Implementation and use of the PLT Scheme web server. *Higher-Order and Symbolic Computing* 20(4), pp. 431–460, 2007.

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.

Fabrice Le Fessant and Luc Maranget. Optimizing pattern-matching. In *Proc. International Conf. on Functional Programming*, pp. 26–37, 2001.

Xavier Leroy. Unboxed objects and polymorphic typing. In *Proc. Symp. on Principles of Programming Languages*, pp. 177–188, 1992.

Peter A. W. Lewis, A. S. Goodman, and J. M. Miller. A pseudo-random number generator for the System/360. *IBM Systems Journal* 8(2), pp. 136–146, 1969.

Jennifer Lhoták, Ondřej Lhoták, and Laurie J. Hendren. Integrating the Soot compiler infrastructure into an IDE. In *Proc. Conf. on Compiler Construction*, pp. 281–297, 2004.

Shih-Wei Liao, Amer Diwan, Robert P. Bosch Jr., Anwar Ghuloum, and Monica S. Lam. SUIF Explorer: An interactive and interprocedural parallelizer. In *Proc. Symp. on Principles and Practice of Parallel Programming*, pp. 37–48, 1999.

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

Jonas Maebe, Dries Buytaert, Lieven Eeckhout, and Koen De Bosschere. Javana: A system for building customized Java program analysis tools. In *Proc. Conf. Object-Oriented Programming Systems, Languages, and Applications*, pp. 153–168, 2006.

Simon Marlow, José Iborra, Bernard Pope, and Andy Gill. A lightweight interactive debugger for Haskell. In *Proc. Haskell Works.*, pp. 13–24, 2007.

Jay McCarthy. The two-state solution: Native and serializable continuations accord. In *Proc. Conf. Object-Oriented Programming Systems, Languages, and Applications*, pp. 567–582, 2010.

Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

Paul V. Mockapteris. Domain names—implementation and specification. IETF RFC 1035, 1987.

Scott Moore, Christos Dimoulas, Dan King, and Stephen Chong. SHILL: A secure shell scripting language. In *Proc. Symp. on Operating Systems Design and Implementation*, pp. 183–199, 2014.

Floréal Morandat, Brandon Hill, Leo Osvald, and Jan Vitek. Evaluating the design of the R language. In *Proc. European Conf. on Object-Oriented Programming*, pp. 104–131, 2012.

Stephen S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan-Kaufmann, 1997.

Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Evaluating the accuracy of Java profilers. In *Proc. Conf. on Programming Language Design and Implementation*, pp. 187–197, 2010.

Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proc. Conf. on Programming Language Design and Implementation*, pp. 89–100, 2007.

Kwankamol Nongpong. *Integrating "Code Smells" Detection with Refactoring Tool Support*. PhD dissertation, University of Wisconsin-Milwaukee, 2012.

David Ofelt and John L. Hennessy. Efficient performance prediction for modern microprocessors. In *Proc. International Conf. on Measurement and Modeling of Computer Systems*, pp. 229–239, 2000.

Harish Patil, Robert Cohn, Mark Charney, Rajiv Kapoor, Andrew Sun, and Anand Karunanidhi. Pinpointing representative portions of large Intel® Itanium® programs with dynamic instrumentation. In *Proc. Symp. on Microarchitecture*, pp. 81–92, 2004.

Greg Pettyjohn, John Clements, Joe Marshall, Shriram Krishnamurthi, and Matthias Felleisen. Continuations from generalized stack inspection. In *Proc. International Conf. on Functional Programming*, pp. 216–227, 2005.

Simon L Peyton Jones. Compiling Haskell by program transformation: A report from the trenches. In *Proc. European Symp. on Programming*, pp. 18–44, 1996.

Markus Püschel, José M. F. Moura, Bryan Singer, Jianxin Xiong, Jeremy Johnson, David Padua, Manuela Veloso, and Robert W. Johnson. Spiral: A generator for platform-adapted libraries of signal processing applications. *J. of High Performance Computing Applications* 18(1), pp. 21–45, 2004.

Feng Qian, Laurie J. Hendren, and Clark Verbrugge. A comprehensive approach to array bounds check elimination for Java. In *Proc. Conf. on Compiler Construction*, pp. 325–342, 2002.

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.

Manuel Serrano. Inline expansion: When and how. In *Proc. International Symp. on Programming Language Implementation and Logic Programming*, pp. 143–147, 1997.

Andreas Sewe, Jannik Jochem, and Mira Mezini. Next in line, please! In *Proc. Works. on Virtual Machines and Intermediate Languages*, pp. 317–328, 2011.

Zhong Shao and Andrew Appel. A type-based compiler for standard ML. In *Proc. Conf. on Programming Language Design and Implementation*, pp. 116–129, 1995.

Jennifer Elizabeth Shaw. *Visualisation Tools for Optimizing Compilers*. MS dissertation, McGill University, 2005.

Jeremy Singer and Chris Kirkham. Dynamic analysis of Java program concepts for visualization and profiling. *Science of Computer Programming* 70(2-3), pp. 111–126, 2008.

Sooel Son, Kathryn S. McKinley, and Vitaly Shmatikov. Fix me up: Repairing access-control bugs in web applications. In *Proc. Symp. on Network and Distributed System Security*, 2013.

Amitabh Srivastava and Alan Eustace. ATOM: a system for building customized program analysis tools. In *Proc. Conf. on Programming Language Design and Implementation*, pp. 196–205, 1994.

Vincent St-Amour, Leif Andersen, and Matthias Felleisen. Feature-specific profiling. In *Proc. Conf. on Compiler Construction*, pp. 49–68, 2015.

Vincent St-Amour and Shu-yu Guo. Optimization coaching for JavaScript. In *Proc. European Conf. on Object-Oriented Programming*, 2015.

Vincent St-Amour, Sam Tobin-Hochstadt, and Matthias Felleisen. Optimization coaching: Optimizers learn to communicate with programmers. In *Proc. Conf. Object-Oriented Programming Systems, Languages, and Applications*, pp. 163–178, 2012a.

Vincent St-Amour, Sam Tobin-Hochstadt, Matthew Flatt, and Matthias Felleisen. Typing the numeric tower. In *Proc. International Symp. on Practical Aspects of Declarative Languages*, pp. 289–303, 2012b.

Barbara Sue Kerne Steele. *An Accountable Source-to-Source Transformation System*. MS dissertation, Massachusetts Institute of Technology, 1981.

T. Stephen Strickland and Matthias Felleisen. Contracts for first-class classes. In *Proc. Dynamic Languages Symp.*, pp. 97–112, 2010.

T. Stephen Strickland, Sam Tobin-Hochstadt, Robert Bruce Findler, and Matthew Flatt. Chaperones and impersonators: Run-time support for reasonable interposition. In *Proc. Conf. Object-Oriented Programming Systems, Languages, and Applications*, pp. 943–962, 2012.

Asumu Takikawa, Daniel Feltey, Earl Dean, Robert Bruce Findler, Matthew Flatt, Sam Tobin-Hochstadt, and Matthias Felleisen. Towards practical gradual typing. In *Proc. European Conf. on Object-Oriented Programming*, 2015.

Juan M. Tamayo, Alex Aiken, Nathan Bronson, and Mooly Sagiv. Understanding the behavior of database operations under program control. In *Proc. Conf. Object-Oriented Programming Systems, Languages, and Applications*, pp. 983–996, 2012.

The Free Software Foundation. *GCC 4.7.0 Manual*. 2012.

The GHC Team. *The Glorious Glasgow Haskell Compilation System User's Guide, Version 7.4.1*. 2011.

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

Sam Tobin-Hochstadt. Extensible pattern matching in an extensible language. `arXiv:1106.2578 [cs.PL]`, 2011.

Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage refactoring: From scripts to programs. In *Proc. DLS*, pp. 964–974, 2006.

Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. Languages as libraries. In *Proc. Conf. on Programming Language Design and Implementation*, pp. 132–141, 2011.

Raja Vallée-Rai, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing Java bytecode using the Soot framework: is it feasible? In *Proc. Conf. on Compiler Construction*, pp. 18–34, 2000.

Mark Weiser. Program slicing. In *Proc. International Conf. on Software Engineering*, pp. 439–449, 1981.

R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Proc. High Performance Networking and Computing*, pp. 1–27, 1998.

Michael Wolfe. Loop skewing: The wavefront method revisited. *J. of Parallel Programming* 15(4), pp. 279–293, 1986.

Guoqing Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, Edith Schonberg, and Gary Sevitsky. Finding low-utility data structures. In *Proc. Conf. on Programming Language Design and Implementation*, pp. 174–186, 2010.

Tatu Ylönen and Chris Lonvick. The Secure Shell (SSH) protocol architecture. IETF RFC 4251, 2006.

Nicholas C. Zakas. *High Performance JavaScript*. O'Reilly, 2010.

Wankang Zhao, Baosheng Cai, David Whalley, Mark W. Bailey, Robert van Engelen, Xin Yuan, Jason D. Hiser, Jack W. Davidson, Kyle Gallivan, and Douglas L. Jones. VISTA: A system for interactive code improvement. In *Proc. Languages, Compilers, and Tools for Embedded Systems and Software and Compilers for Embedded Systems*, pp. 155–164, 2002.