Feature-Specific Profiling

LEIF ANDERSEN, PLT @ Northeastern University, United States of America
VINCENT ST-AMOUR, PLT @ Northwestern University, United States of America
JAN VITEK, Northeastern University and Czech Technical University
MATTHIAS FELLEISEN, PLT @ Northeastern University, United States of America

While high-level languages come with significant readability and maintainability benefits, their performance remains difficult to predict. For example, programmers may unknowingly use language features inappropriately, which cause their programs to run slower than expected. To address this issue, we introduce *feature-specific profiling*, a technique that reports performance costs in terms of linguistic constructs. Festure-specific profilers help programmers find expensive uses of specific features of their language. We describe the architecture of a profiler that implements our approach, explain prototypes of the profiler for two languages with different characteristics and implementation strategies, and provide empirical evidence for the approach's general usefulness as a performance debugging tool.

ACM Reference Format:
Leif Andersen, Vincent St-Amour, Jan Vitek, and Matthias Felleisen. 2018. Feature-Specific Profiling. 1, 1 (September 2018), 35 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 PROFILING WITH ACTIONABLE ADVICE

When programs take too long to run, programmers tend to reach for profilers to diagnose the problem. Most profilers attribute the run-time costs during a program's execution to *cost centers* such as function calls or statements in source code. Then they rank all of a program's cost centers in order to identify and eliminate key bottlenecks (Amdahl 1967). If such a profile helps programmers optimize their code, we call it *actionable* because it points to inefficiencies that can be remedied with changes to the program.

The advice of conventional profilers fails the actionable standard in some situations, mostly because their conventional choice of cost centers—e.g. lines or functions—does not match programming language concepts. For example, their advice is misleading in a context where a performance problem has a unique cause that manifests itself as a cost at many locations. Similarly, when a language allows the encapsulation of syntactic features in libraries, conventional profilers often misjudge the source of related performance bottlenecks.

Feature-specific profiling (FSP) addresses these issues with the introduction of linguistic features as cost centers. By "features" we specifically mean syntactic constructs with operational costs: functions and linguistic elements, such as pattern matching, keyword-based function calls, or

Authors' addresses: Leif Andersen, PLT, CCIS, PLT @ Northeastern University, Boston, Massachusetts, United States of America, leif@ccs.neu.edu; Vincent St-Amour, PLT, Department of Electrical Engineering and Computer Science, PLT @ Northwestern University, Evanston, Illinois, United States of America, stamourv@eecs.northwestern.edu; Jan Vitek, Northeastern University, Boston, Massachusetts, Czech Technical University, j.vitek@neu.edu; Matthias Felleisen, PLT, CCIS, PLT @ Northeastern University, Boston, Massachusetts, United States of America, matthias@ccs.neu.edu.

XXXX-XXXX/2018/9-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

behavioral contracts. This paper, an expansion of St-Amour et al.'s (2015) original report on this idea, explains its principles, describes how to turn them into reasonably practical prototypes, and presents evaluation results. While the original paper introduced the idea and used a Racket (Flatt and PLT 2010) prototype to evaluate its effectiveness, this paper confirms the idea with a prototype for the R programming language (R Development Core Team 2016). The creation of this second prototype confirms the validity of feature-specific profiling beyond Racket. It also enlarges the body of features for which programmers may benefit from a feature-specific profiler.

In summary, this expansion of the original conference paper into an archival one provides a definition for language features, feature instances, and feature-specific profiling, explains the components that make up a feature-specific profiler, describes two ingredients to make the idea truly practical, and evaluates prototypes for the actionability of its results, implementation effort, and run-time performance in the Racket and R contexts.

2 LINGUISTIC FEATURES AND THEIR PROFILES

An FSP attributes execution costs to instances of linguistic features, that is, any construct that has both a syntactic presence in code and a run-time cost that can be detected by inspecting the language's call stack. Because the computation associated with a particular instance of a feature can be dispersed throughout a program, this view can provide actionable information when a traditional profiler falls short. To collect this information an FSP comes with a slightly different architecture than a traditional profiler. This section gives an overview of our approach.

2.1 Linguistic Features

We consider a language feature to be any syntactic construct that has an operational stack-based cost, such as a function calling protocol, looping constructs, or dynamic dispatch for objects. The features that a program uses are orthogonal to the actual algorithm it implements. For example, a program that implements a list traversal algorithm may use loops, comprehensions, or recursive functions. While the algorithms and resulting values are the same in all three cases, their implementation may have different performance costs.

The goal of feature-specific profiling is to find uses of features that are expensive and not expensive algorithms. Knowing which features are expensive in a program is not sufficient for programmers to know how to speed up their code. An expensive feature may appear in many places, some innocuous to performance, and may be difficult to remove from a program entirely. More precisely, a feature may not generally be expensive, but some uses may be inappropriate. For example, dynamic dispatch is not usually a critical cost component, but might be when used in a hot loop for a mega-morphic method. An FSP therefore points programmers to individual feature instances. As a concrete example, while all dynamic dispatch calls make up a single feature, every single use of dynamic dispatch is a unique feature instance, and one of them may come with a significant performance cost.

The cost of feature instances does not necessarily have a direct one-to-one mapping to their location in source code. One way this happens is when the cost centers of one feature may intersect with the cost centers of another feature. For example, a concurrent program may wish to attribute program costs in terms of its individual threads rather than the functions run by the threads. A traditional profiler correctly identifies the functions being run, but it fails to properly attribute them to their underlying threads. We call these *conflated costs*. An FSP properly attaches such costs to their appropriate threads.

In additional to having conflated costs, linguistic features may also come with non-local, *dispersed costs*, that is, costs that manifest themselves at a different point than their syntactic location in code. Continuing the previous example, dynamic dispatch is a language construct with non-local

```
1  #lang racket
2  (define (fizzbuzz n)
3    (for ([i (range n)])
4      (cond
5        [(divisible i 15) (printf "FizzBuzz\n")]
6        [(divisible i 5)  (printf "Buzz\n")]
7        [(divisible i 3)  (printf "Fizz\n")]
8        [else             (printf "~a\n" i)])))
9
10 (feature-profile
11  (fizzbuzz 10000000))
```

```
Feature Report
(Feature times may sum to more or less than 100% of the total running time)

Output accounts for 68.22% of running time
      (5580 / 8180 ms)
  4628 ms : fizzbuzz.rkt:8:24
  564 ms : fizzbuzz.rkt:7:24
  232 ms : fizzbuzz.rkt:6:24
  156 ms : fizzbuzz.rkt:5:24

Generic sequences account for 11.78% of running time
      (964 / 8180 ms)
  964 ms : fizzbuzz.rkt:3:11
```

Figure 1: Feature profile for FizzBuzz

costs. One useful way to measure dynamic dispatch is to attribute its costs to a specific method, rather than just its call sites. Accounting costs this way disambiguates time spent in the program's algorithm versus time spent dispatching. Traditional profilers attribute the dispatch cost only to the call site, which is misleading and suggests to programmers that the algorithm itself is costly, rather than the dispatch mechanism. An FSP solves this problem by attributing the cost of method calls to their declarations. Programmers may be able to use this information to avoid costly uses of dynamic dispatch, without having to change their underlying algorithm.

2.2 An Example Feature Profile

To illustrate the workings of an FSP, figure 1 presents a concrete example, the Fizzbuzz[1]program in Racket, and shows the report from the FSP for a call to the function with an input value of 10,000,000. The profiler report notes the use of two Racket features with a large impact on performance: output and iterations over generic sequences. Five seconds were spent on output. Most of this time is spent on printing numbers not divisible by either 3 or 5 (line 16), which includes most numbers. Unfortunately output is core to Fizzbuzz and it cannot be avoided. On the other hand, the `for`-loop spends about one second in generic sequence dispatch. Specifically, while the `range` function produces a list, the `for` construct iterates over all types of sequences and must therefore process its input generically. In Racket, this is actionable advice. A programmer can reduce this cost by using

[1]https://immanent.com/2007/01/24/using-fizzbuzz-to-find-developers-who-grok-coding/

`in-range`, rather than `range`, thus informing the compiler that the `for` loop iterates over a range sequence.

2.3 A Four Part Profiler

Feature-specific profiling relies on one optional and three required ingredients. First, the language's run-time system must support a way to keep track of dynamic extents. Second, the language must also support statistical or sampling profiling. Third, the author of features must be able to modify the code of their features so that they mark their dynamic extent following an FSP-specific protocol. Finally, optional feature-specific plugins augment the protocol by turning the FSP's collected data into useful information.

Dynamic Extent. An FSP relies on a language's ability to track the dynamic extent of features. Our approach is to place annotations on the call stack. A feature's implementation adds a mark to the stack at the begining of its extent. The mark carries information that identifies both the feature and its specific instance. When an instance's execution ends, the annotation is removed from the stack. Many features contain "callbacks" to user code, such as the `for`-loop located at line 11 of the Fizzbuzz example in figure 1. The cost of running these callbacks should not be accounted as part of the feature's cost. Our way to handle this situation is to add an additional annotation to the stack. When the callback finishes, this annotation is popped off the stack, which indicates that the program has gone back to executing feature code. Some languages such as Racket directly support stack annotations. Racket refers to these as continuation marks (Clements et al. 2001), which are similar to stack annotations. Others, such as R, do not, but we show that adding stack annotations is straightforward (section 8).

Sampling Profiler. An FSP additionally requires its host language to support sampling profiling. Such a profiler collects samples of the stack and its annotations at fixed intervals during program execution. It uses these samples to determine what features, if any, are being executed. After the program has finished, these collected samples are analyzed and presented, as in figure 1. The total time spent in features tends to differ from the program's total execution time. These differences stem from the distribution of annotations in the collected samples. Any individual sample may contain the cost of multiple features, meaning a sample with multiple annotations is associated with multiple features. Likewise, in the case of an annotation-free stack, a sample is not associated with any features. The cost of a feature is composed entirely of all of its specific instances. That is, a feature is only executing when exactly one of its instances are running.

Feature annotations. Every feature comes with a different notion about what costs are related to that feature, and which dynamic extent the profiler should track. Features also have different notions about what code is not related to the feature, and thus the profiler should not track. For example, the `for`-loop in figure 1 must account for the time spent generating and iterating over the list as a part of its feature, but it is not responsible for the time spent in its body. Because every feature has a unique notion of cost, its authors are responsible for modifying their libraries to add annotating indicating feature code. While modifying a feature's implemenation code puts some burden on authors, we show that adding these annotations is manageable.

Feature Plugins. While annotations denote a feature's dynamic extent, a plugin denotes the profile with the interpretation. Specifically, a plugin enables features to report their cost centers even when multiple instances have overlapping and non-local cost centers. This plugin is completely optional and many features rely entirely on the protocol.

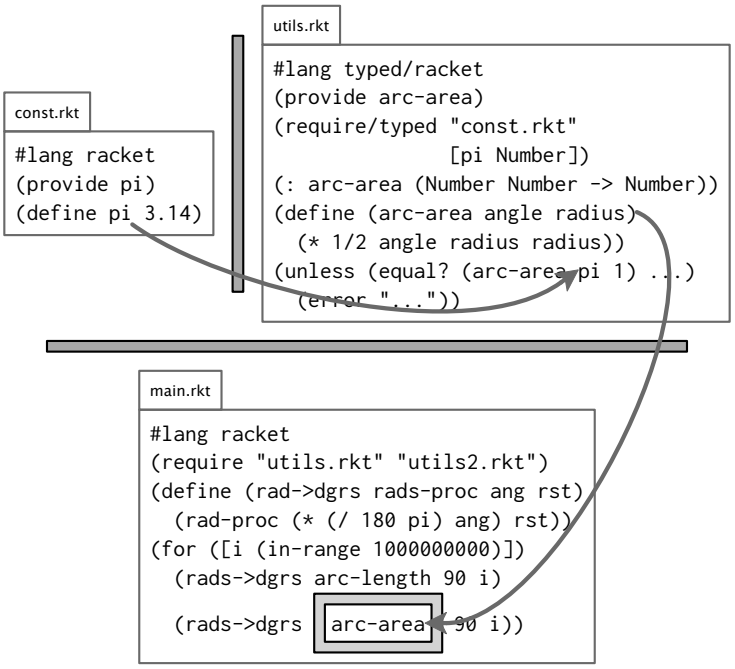


Figure 2: Flat (top) and higher-order (bottom) contracts for typed and untyped modules

3 PROFILING RACKET CONTRACTS

The Fizzbuzz example is simplistic and does not necessitate a new type of profiling. To motivate a feature-centric reporting of behavioral costs, this section illustrates the profiling of contracts (Findler and Felleisen 2002), a feature with dispersed costs.

In Racket, contracts are used to monitor the flow of values across module boundaries. One common use case is to ensure that statically typed modules interact safely with untyped modules. The left half of figure 2 shows an untyped module `"const.rkt"` and a typed module `"utils.rkt"`. The untyped module defines and exports `pi` as `3.14`. That value is used in a test for `arc-area` to convert the radius of an arc to its area. The value `pi` passes through a contract (represented by the gray box), as it passes to the typed module. If `pi` is not a number, the contract prevents the value from passing through. Likewise, if `pi` is a number, the computation of `"utils.rkt"` may safely rely on the fact that `pi` is a number and can compile accordingly. Not all contracts can be checked immediately when values cross boundaries, especially contracts for higher-order functions or first-class objects. These contracts, shown in the right half of figure 2, are implemented as wrappers that check the arguments and results for every function or method call. Here, the module defines a function `rads->dgrs`, which converts a function that operates on radians into one that operates on degrees. The `arc-area` function is used in a higher-order manner. As such, the contract boundary must wrap the function, represented as a gray box surrounding `arc-area`, to ensure that the function meets the type it is given.

Traditional profilers properly track the costs of flat contracts but fail to properly track the delayed checking of higher-order contracts. The left side of figure 3 shows the results when profiling the program in figure 2 with a traditional profiler. This profiler is able to detect that the program spends roughly 10% of execution time checking contracts, but it is unable to determine the time spent in

```
Total cpu time: 23186ms
Number of samples: 421

Idx  Total   Self  Name+src
[1] 100.0%   0.0%  [traversing imports]
[2] 100.0%   0.0%  [running body]
[3] 100.0%   0.0%  profile-thunk16
[4] 100.0%   0.0%  run
[5] 100.0%  17.7%  temp1
[6]  82.3%  71.6%  for-loop
[7]  10.6%  10.6%  ??? (contract)
```

```
Feature Report
(Feature times may sum to more
  or less than 100% of the total
  running time)

1144 samples
Contracts: 25.92% of run time
3386/13061 ms

(-> Number Number any)  3386 ms
    arc-length          1836 ms
    arc-area            1550 ms
```

Figure 3: Output Traditional Profiler (left) and Feature-Specific Profiler (right)

individual contract instances. Worse still, the profiler associates the costs of checking contracts with the for loop rather than where the contracts are actually introduced, at the typed-untyped boundaries. This behavior does not help programmers solve performance problems with their code.

An FSP properly attributes the run-time costs of contracts. The right side of figure 3 shows the result when running the same program in a feature-specific profiler. The profiler determines that contracts account for roughly 25% of execution time. Additionally, the profiler determines that the `arc-area` and `arc-length` contracts take comparable time to check.

The FSP's output is broken down into distinct features and instances of features. In the case of figure 3, only one feature takes a noticeable amount of time: contracts. It additionally notices two particular instances of contracts and reports the amount of time each spent.

Many features run simultaneously, such as pattern matching and function calls. In these cases, the profiler collects information for all running features or none in cases where no features are running. As a result, not all of the features put together may not add up to 100% of the execution time. In this case, contracts are the only feature the profile tracked, and they account for roughly 26% of the run time. In contrast, a feature's total cost is the sum of all instances. As such, all instances for a particular feature will make up 100% of that feature's total cost.

4 PROFILER ARCHITECTURE

An FSP consists of four parts (shown in figure 4): a sampling profiler, an analysis to process the raw samples, a protocol for features to mark the extent of feature execution, and optional analysis plug-ins for generating reports on individual features. The architecture allows programmers to add profiler support for features on an incremental basis. In this section, we describe our implementation of an FSP for Racket[2] in detail. We illustrate it with features that do not require custom analysis plug-ins, such as output, type casts, and optional function arguments. In the next section we discuss the optional analysis plug-ins and features that benefit from them.

The profiler employs a sampling-thread architecture to detect when programs execute certain pieces of code. When a programmer turns on the profiler, a run of the program spawns a separate sampling thread, which inspects the main thread's stack at regular intervals on the order of one sample per 50 milliseconds. Once the program terminates, an offline analysis deals with the collected samples and produces programmer-facing reports.

The sample analysis relies on a protocol between itself and the feature implementations. The protocol is articulated in terms of markers on the control stack. Each marker indicates when a

[2]https://github.com/stamourv/feature-profile

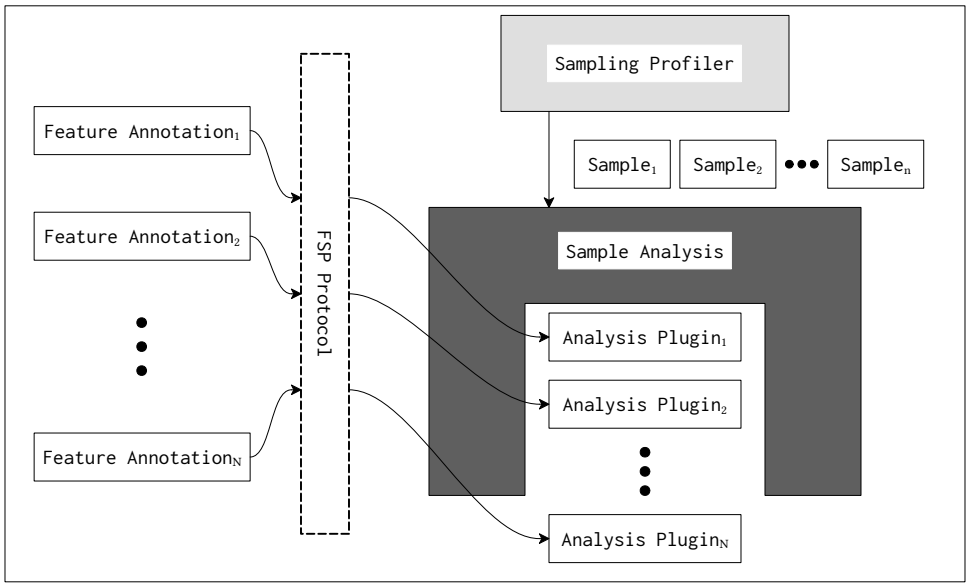


Figure 4: Architecture for an FSP

feature executes its specific code. The offline analysis can thus use these markers to attribute specific slices of time consumption to a feature.

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

4.1 Inspecting the Stack with Continuation Marks

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

- `(with-continuation-mark key value expr)`, which attaches a (`key`, `value`) pair to the current stack frame and then evaluates `expr`. The markers automatically disappear when the evaluation of `expr` terminates.
- `(current-continuation-marks thread)`, which walks the stack and retrieves all key-value pairs from the stack of a specified thread.

Programs can also filter marks with `(continuation-mark-set->list marks key)`. This operation returns a filtered list of `marks` whose keys match `key`. Outside of these operations, continuation marks do not affect a program's behavior.[3]

Figure 5 illustrates the working of continuation marks with a function that traverses binary trees and records paths from roots to leaves. The top half of the figure shows the code that performs the traversal. 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

[3]Continuation marks also preserve the proper implementation of tail calls.

```
(struct tree ())
(struct leaf tree (n))
(struct node tree (l n r))

; paths : Tree -> [Listof [Listof Number]]
(define (paths t)
  (cond
    [(leaf? t)
     (list (cons (leaf-n t)
                 (continuation-mark-set->list
                  (current-continuation-marks)
                  'paths)))]
    [(node? t)
     (with-continuation-mark 'paths (node-l t)
       (append (paths (node-n t)) (paths (node-r t))))]))

(check-equal? (paths (node 1 (node 2 (leaf 3) (leaf 4)) (leaf 5)))
              '((3 2 1) (4 2 1) (5 1)))
```

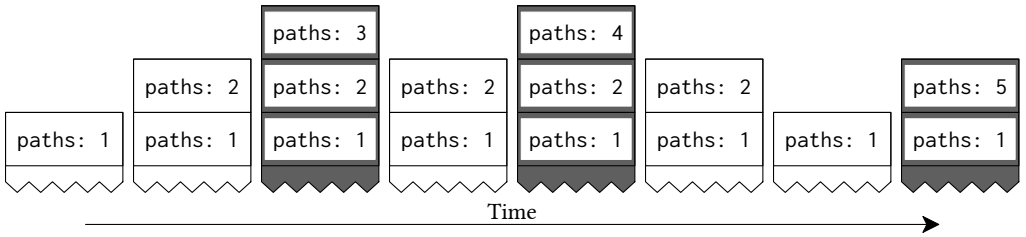


Figure 5: Recording paths in a tree with continuation marks

returns the completed path. A trace of the continuation mark stack is shown in the bottom half of the figure. It highlights the execution points where the stack is reported to the user.

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

Beyond Racket, continuation marks have also been added to Microsoft's CLR (Pettyjohn et al. 2005) and JavaScript (Clements et al. 2008). Other languages provide similar mechanisms, such as stack reflection in Smalltalk and the stack introspection used by the GHCi debugger (Marlow et al. 2007) for Haskell.

4.2 Feature-specific Data Gathering : The Protocol

The stack-sample analysis requires that a feature implementation places a marker with a certain key on the control stack when it begins to evaluate feature-specific code.

Marking. Feature authors who wish to enable feature-specific profiling for their features must change the implementation of the feature so that instances mark their dynamic extents with *feature marks*. It suffices to wrap the relevant code with `with-continuation-mark`. These marks, added

```
(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 failed."))))))]))
```

Figure 6: Instrumentation of assertions (excerpt)

to the call stack, allow the profiler to observe whether a thread is currently executing code related to a feature.

Figure 6 shows an excerpt from the instrumentation of type assertions in Typed Racket, a variant of Racket that is statically type checked (Tobin-Hochstadt and Felleisen 2008). The underlined conditional is responsible for performing the actual assertion. The mark's key should uniquely identify the construct. In this case, we use the symbol `'TR-assertion` as the key. Unique choices avoid false reports and interference by distinct features. In addition, choosing unique keys also permits the composition of arbitrary features. As a consequence, the analysis component of the FSP 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 feature instance to which the cost should be assigned. In figure 6, the payload is the source location of a specific assertion in the program, which allows the profiler to compute the cost of individual instances of `assert`.

Annotating features is simple and involves only non-instrusive, local code changes, but it does require access to the implementation for 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; i.e., they usually run user code whose execution time may not have to count towards the time spent in the feature. For example, the profiler must not count the time spent in function bodies towards the cost of the language's function call protocol.

To account for user code, features place *antimarks* on the stack. Such antimarks are continuation marks with a distinguished value, a payload of `'antimark`, that delimit a feature's code. The analysis phase recognizes antimarks and uses them to cancel out feature marks. Cost 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. An antimark only cancels marks for its original feature. Marks and antimarks, for the same or different features can be nested.

Figure 7 illustrates the idea with code that instruments a simplified version of Racket's optional and keyword argument protocol (Flatt and Barzilay 2009). The simplified implementation appears in the top half of the figure and a sample trace of a function call using keyword arguments is displayed in the bottom half. When the function call begins, a `'kw-protocol` mark is placed on the stack (annotated in DARK GRAY) with a source location as its payload. Once evaluation of the function begins, an antimark is placed on the stack (annotated in LIGHT GRAY). Once the antimark has been removed from the stack, cost accounting is again attributed towards keyword arguments.

```
1  (define-syntax (lambda/keyword stx)
2    (syntax-case stx ()
3      [(lambda/keyword formals body) ; the compiler rewrites this to:
4       (quasisyntax
5         (lambda (unsyntax (handle-keywords formals))
6           (with-continuation-mark 'kw-protocol
7                                   (unsyntax (source-location stx))
8             ···parse keyword arguments, compute default values···
9             (with-continuation-mark 'kw-protocol 'antimark
10              body))))])) ; body is use-site code
```

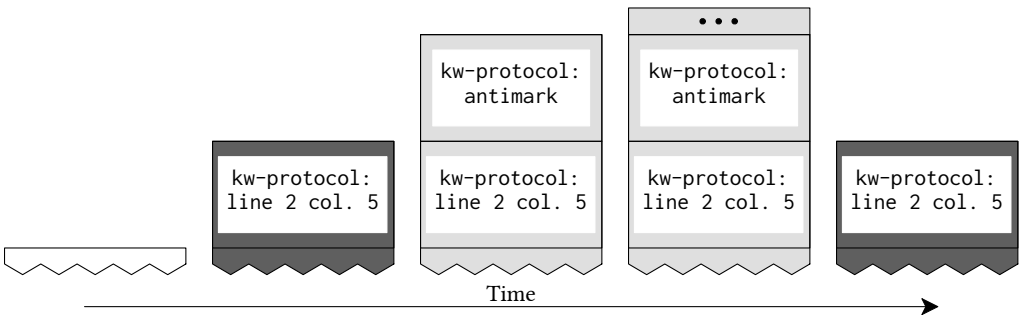


Figure 7: Use of antimarks in instrumentation

In contrast, the assertions from figure 6 do not require antimarks because user code evaluation happens exclusively outside the marked region (line 8). Another feature that has this behavior is program output, which also never calls user code from within the feature.

Sampling. During program execution, the FSP's sampling thread periodically collects and stores continuation marks from the main thread. The sampling thread knows which keys correspond to features it should track, and collects marks for all features at once.[4]

4.3 Analyzing Feature-specific Data

After the program execution terminates, the analysis component processes the data collected by the sampling thread to produce a feature cost report. The tool analyses each feature separately, then combines the results into a unified report.

Cost assignment. The profiler uses a standard sliding window technique to assign a time cost to each sample based on the elapsed time between the sample, its predecessor and its successor. Only samples with a feature mark as the most recent mark contribute time towards features.

Payload grouping. 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. Payloads can be grouped in arbitrary equivalence classes. Our profiler currently groups them based on equality, but library authors can implement grouping according to any criteria they desire. The FSP then generates reports for each feature, using payloads as keys and time costs as values.

[4]In general, the sampling thread could additionally collect samples of all marks and sort the marks in the analysis phase.

```
1  #lang racket
2  (require feature-profile "utils.rkt")
3
4  (define 2pi (* 2 pi))
5  (feature-profile (for ([i (in-range 1000000)])
6                     (printf "Radius: ~a~n" i)
7                     (printf "Area: ~a~n" (arc-area 2pi i))
8                     (printf "Circ.: ~a~n~n" (arc-length 2pi i)))))
```

```
Feature Report
(Feature times may sum to more or less than 100% of the total running time)

1649 samples

Output : 71.4% of run time
  1813 ms   : example.rkt:8:5
  1423.5 ms : example.rkt:6:5
  1227.5 ms : example.rkt:7:5

Contracts : 26.86% of run time
(-> Number Number any)        3610 ms
    arc-area                1823.5 ms
    arc-length              1786.5 ms
```

Figure 8: Feature Profiler Results for Circle Properties

Report composition. Finally, after generating individual feature reports, the FSP combines them into a unified report. Constructs absent from the program and those inexpensive enough to never be sampled are pruned to avoid clutter. The report lists features in descending order of cost. Likewise, each feature instance is listed in descending order grouped by their associated feature.

Figure 8 shows a program that uses the `utils.rkt` library shown in figure 2. Specifically, the program prints the radius, area, and circumference for 1,000,000 circles of increasing size. The right half of the figure also gives a profile report for this program. Most of the execution time is spent printing the circles' properties (lines 7-11), and thus appears first in the feature list. Specifically, printing the circle's circumference (line 9) takes the most time (18 s). Finally, the second item, contract verification, has a relatively small cost compared to output for this program (4 s).

5 PROFILING COMPLEX FEATURES

The feature-specific protocol in the preceding section assumes that there is a one-to-one correspondence from the placement of a feature to the location where it incurs a run-time cost. This process, however, does not apply to features whose instances have costs appear either in multiple places or in different places than than their syntactic location suggests. These are features with *non-local costs*, because a feature instance and its cost are separated. Higher-order contracts illustrate this idea particularly well because they are specified in one place yet incur costs at many others. In other cases, several different instances of a feature contribute to a single cost center, such as a concurrent program that wants to attribute a cost to the program as a whole as well as the particular thread or actor running associated with it. These features have *conflated costs*.

While the creator of features with non-local or conflated costs can use the FSP protocol to measure some aspects of their costs, adopting a better protocol produces better results when evaluating such features. This section shows both how to extend the FSP's analysis component

with feature-specific plug-ins and how to adapt the communication protocol appropriately. It is divided into two parts. First, we discuss custom payloads, values that the authors of features use to describe their non-local or conflated costs (section 5.1). Using custom payloads, an analysis plug-in may convert the information into a form that programmers can digest and act on (section 5.2). We use three running examples to demonstrate non-local and conflated features and their payloads: contracts, actor-based concurrency, and parser backtracking.

5.1 Custom Payloads

The instrumentation for features with complex-cost accounting, non-local or conflated, makes use of arbitrary values to mark payloads instead of source locations. These payloads must contain enough information to identify a feature's cost center and to distinguish specific instances. Contracts, actor-based concurrency and parser backtracking are three cases where features benefit from having such custom payloads.

Although storing precise and detailed data in payloads is attractive, developers must also avoid excessive computation or allocation when constructing their payloads. After all, payloads are constructed every time feature code is executed, whether or not the sampler observes it.

Contracts. As discussed in section 3, higher-order behavioral contracts have non-local costs. Rather than using source locations as cost-centers, a contract uses *blame objects*. The latter tracks the parties to a contract so that its possible to poinpoint the faulty party in case of a violation. Every time an object traverses a higher-order contract boundary, the contract system attaches a blame object. This blame object holds enough information to reconstruct a complete picture of contract checking events—the contract to check, the name of the contracted value, and the names of the components that agreed to the contract.

Actor-Based Concurrency. Marketplace is a DSL for writing programs in terms of actor-based (Hewitt et al. 1973) concurrency (Garnock-Jones et al. 2014). Programs that use Marketplace features have conflated costs. The cost-centers of these programs are attributed in terms of the processes the language uses, rather than the functions that an individual process runs. To handle this, Marketplace uses process identifiers as payloads. Since `current-continuation-marks` gathers all the marks currently on the stack, the sampling thread can gather *core samples*.[5] 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.

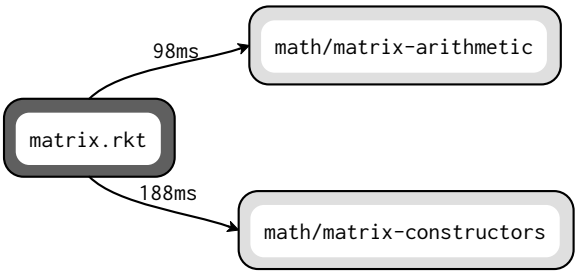
Parser backtracking. The Racket ecosystem includes a parser generator named Parsack. A parser's cost-centers are the particular parse path that it follows, rather than any particular production rule that the parser happens to be using. In particular, a feature-specific approach shines when determining on which paths the parser eventually backtracks. This allows a programmer to improve a program's performance by reordering production rules when possible. To accommodate this, payloads for Parsack combine three values into a payload: the source location of the current production rule disjunction, the index of the active branch within the disjunction, and the offset in the input where the parser is currently matching. Because parsing a term may require recursively parsing sub-terms, a Parsack payload includes core samples that allow the plugin to to attribute time to all active non-terminals.

5.2 Analyzing Complex-Cost Features

Even if payloads contain enough information to uniquely identify a feature instance's cost-center, programmers usually cannot directly digest the complex information in the corresponding payloads.

[5] In analogy to geology, a core sample includes marks from the entire stack, rather than the top most mark.

```
1 (define (random-matrix)
2   (build-matrix 200 200
3                 (lambda (i j)
4                   (random))))
5
6 (feature-profile
7  (matrix* (random-matrix)
8           (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 [...]))
10 ms : make-matrix-multiply  (-> Int Int Int (-> any any any) Array)
```

Figure 9: Module graph and by-value views of a contract boundary

When a feature uses such payloads, its creator is encouraged to implement an analysis plug-in 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 this checking costs. A programmer can act only after identifying the relevant components. Hence, the analysis aims to provide an at-a-glance overview of the cost of each contract and boundary.

To this end, the contract 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 DARK GRAY) from untyped modules (in LIGHT GRAY). To generate this view, the analysis extracts component names from blame objects. It then groups payloads that share pairs of parties and computes costs as discussed in section 4.3. The top-right part of figure 9 shows the module graph for a program that constructs two random matrices and multiplies them. This latter 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, an FSP can provides other views as well. For example, the bottom portion of figure 9 shows the *by-value* view, which provides fine-grained information about the cost of individual contracted values.

Actor-Based Concurrency. The goal of the Marketplace analysis plug-in is to assign costs to individual Marketplace processes and VMs, as opposed to the code they execute. Marketplace feature marks use the names of processes and VMs as payloads, which allows the plug-in to distinguish separate processes executing the same functions.

The plug-in 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 the function calls that led to a certain point in the execution. We exploit that similarity and reuse standard edge profiling techniques[6] to attribute costs to the entire ancestry of a process. To

[6]VM cost assignment is simpler than edge profiling because VM/process graphs are in fact trees. Edge profiling techniques still apply, though, which allows us to reuse part of the Racket edge profiler's implementation.

```
============================================================
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 10: Marketplace process accounting (excerpt)

```
1 (define $a (compose $b (char #\a)))
2 (define $b (<or> (compose (char #\b) $b) (nothing)))
3 (define $s (<or> (try $a) $b))
4
5 (feature-profile (parse $s input))
```

```
Parsack Backtracking
=====================================================
Time (ms)       Time (%)       Disjunction       Branch
=====================================================
2076            46%            ab.rkt:3:12       1
```

Figure 11: An example Parsack-based parser and its backtracking profile

disambiguate between similar processes in its reports, the plug-in uses a process's full ancestry as an identity.

Figure 10 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.

The plug-in also reports the overhead of the Marketplace library itself. Any time attributed directly to a VM; i.e., not to any of its children—is overhead from the library. In our echo server example, 32.3% of the total execution time is reported as the ground VM's *self time*, which corresponds to the library's overhead.[7]

Parser backtracking. The feature-specific analysis for Parsack determines how much time is spent backtracking for each branch of each production rule 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.

The plug-in detects backtracking as follows. Because disjunctions are ordered, the parser must backtrack from early branches in the disjuction before it reaches a production rule that parses. Therefore, whenever the analysis observes a sample from the matching branch at a given input location, it attributes backtracking cost 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,

[7]The echo server performs no actual work which, by comparison, increases the library's relative overhead.

the Parsack plug-in uses core samples and edge profiling to handle the recursive structure of the process.

Figure 11 shows a simple parser that first attempts to parse a sequence of bs followed by an a, and in case of failure, backtracks in order to parse a sequence of bs. The right portion of figure 11 shows the output of the FSP 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.

6 CONTROLLING PROFILER COSTS

Features that implement the feature-specific protocol insert continuation marks regardless of whether a programmer wishes to profile the program. For features where individual instances perform a significant amount of work, such as contracts, the overhead of marks is usually not observable as shown in section 7.3. For other features, such as fine-grained console output, where the aggregate cost of individually inexpensive instance annotations are significant, the overhead of marks can be problematic. In such cases, programmers want to choose when marks are applied on a by-execution basis.

In addition, programmers may also want to control when mark insertions take place to avoid reporting costs in code that they wish to ignore or cannot modify. For instance, reporting that the plot library heavily relies on pattern-matching in its implementation is useless to most programmers; they cannot fix it. It makes sense only if they are prepared to replace the plotting library altogether.

To establish control over when and where continuation marks are added, a profiler must support two kinds of marks: active and latent. We refer to the marks described in the previous sections as active marks A latent mark is an annotation that can be turned into an active mark as needed. An implementation may employ a preprocessor for this purpose. We distinguish between *syntactically latent marks* for use with compile-time meta-programming and *functional latent marks* for use with library or run-time functions.

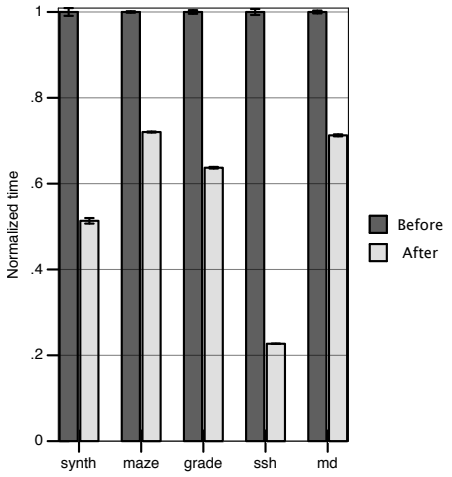
6.1 Syntactically Latent Marks

Syntactically latent marks exist as annotations on the intermediate representation (IR) of a program. To add a latent mark, the feature implementation leaves tags[8] on the residual program's IR instead of directly inserting feature marks and antimarks. These tags are discarded after compilation and thus have no run-time effect on the program execution. Other meta-programs or the compiler can observe latent marks and turn them into active marks.

A feature-specific profiler can rely on a dedicated compiler pass to convert syntactic latent marks into active ones. Many compilers have some mechanism to modify a program's pre-compiled source. Racket, for example, uses the language's *compilation handler* mechanism to interpose this activation pass. The pass traverses the input program, replacing every relevant syntactic latent mark it finds with an active mark. As this mechanism relies on the compiler, a programmer using latent marks must recompile the user's code. The library code, however, does not need to be re-compiled, which make syntactic latent marks practical for large environments.

This implementation method applies only to features implemented using meta-programming such as the sntactic extensions used in many Racket or R programs. Thus many of these features use syntactically latent marks. Languages without any meta-programming facilities can still support latent marks with external tools that emulate meta-programming.

[8]Many compilers have means to attach information to nodes in the IR. Our Racket prototype uses syntax properties (Dybvig et al. 1993).



Program	Problem features(s)	Negative Information
synth	Contracts	Generic sequences, output
maze	Output	Casts
grade	Security policies	-
ssh	Processes, contracts	Pattern matching, generic sequences
markdown	Backtracking	Patern matching

Results are the mean of 30 executions on a 6-core 64-bit Debian GNU/Linux system with 12GB of RAM.
Because Shill supports only FreeBSD, results for *grade* are from a 6-core FreeBSD system with 6GB of RAM.
Error bars are one standard deviation on either side.

Figure 12: Execution time after profiling and improvements (lower is better)

6.2 Functional Latent Marks

Functional latent marks offer an alternative to syntactically latent marks. Instead of tagging the programmer's code, a preprocessor recognizes calls to feature-related functions and rewrites the program's code to wrap such calls with active marks. Like syntactic latent marks, functional latent marks require recompilation of code that uses the relevant functions. Also like syntactic latent marks, they do not require recompiling libraries that *provide* feature-related functions, which makes them appropriate for functions provided as runtime primitives.

As an example, Racket's output feature uses functional latent marks instead of active marks. Functional latent marks are appropriate here because a program may contain many instances of the output feature, each having little overhead. The output feature includes a list of runtime and standard library functions that emit output and adds feature marks around all calls to those functions, as well as antimarks around their arguments to avoid measuring their evaluation.

7 EVALUATION: PROFILER RESULTS

Our evaluation of the Racket feature-specific profiler addresses three promises: that measuring in a feature-specific way supplies useful insights into performance problems; that it is easy to add support for new features; and that the run-time overhead of profiling manageable. This section first presents case studies that demonstrate how feature-specific profiling improves the performance of programs. Then it reports on the effort required to mark features and implement plug-ins. Finally, it discusses the run-time overhead imposed by the profiler.

7.1 Case Studies

To be useful, a profiler must accurately identify feature use costs and provide *actionable* information to programmers. Ideally, it identifies specific feature uses that are responsible for significant performance costs in a given program. When it finds such instances, the profiler must point

programmers towards solutions. Additionally, it must also provide *negative* information, i.e., confirm that some uses of language constructs need not be investigated.

Here we present five case studies. Each one describes a program, summarizes the profiler's feedback, and explains the changes that directly follow from the report. Figure 12 displays a concise overview of the performance after incorporating this feedback. These case-studies range in size from 1 to 15 modules, the difference in size did not affect the effectiveness of the project.

Sound Synthesis Engine This case study concerns a sound synthesis engine written by St-Amour. The engine uses the `math` library's arrays to represent sound signals. It consists of a `mixer` module that handles most of 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. Unlike the engine, the `math` library is written in Typed Racket. To ensure a sound interaction between the languages, a contract boundary separates it from the untyped synthesis engine. For scale, the synthesis engine spans 452 lines of code, and we profile it with ten seconds of music.[9]

Racket's traditional statistical profiler reports that around 40% of total execution time is spent in two functions from the `math` library:

```
=====================================================
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
[...]
```

Such profiling results suggest a problem with the `math` library. Rewriting or avoiding it altogether would be a significant undertaking.

Figure 13 shows the FSP's take of the same program. According to its report, almost 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 those contracts. Since the `math` library's contracts are automatically generated by Typed Racket, improving their performance directly is not practical. Reducing the use of contracts is more likely to be profitable. Because contract generation happens only at the boundary of typed and untyped code, modifying a few modules that create this boundary may lower the imposed cost. In order to determine how to move a boundary, the programmer turns to the module graph view in the lower portion of figure 13. This graph is provided by our feature-specific analysis for contracts. Almost half the total execution time lies between the untyped interface to the `math` library used by the `mixer` module (in LIGHT GRAY) and the typed portions of the library (in DARK GRAY). This suggests converting the `mixer` module to Typed Racket; a 15-minute effort that improves performance by∼48%.

Figure 13 also shows that generic sequence operations, while often expensive, do not impose a significant cost in this program, despite their pervasive use. 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 would also be a waste of time.

[9]The synthesized song is "Funky Town", by Lipps Inc.

```
Contracts : 73.77% of run 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 : 0.04% of run time (10 / 23816 ms)
  10 ms : wav-encode.rkt:51:16
```

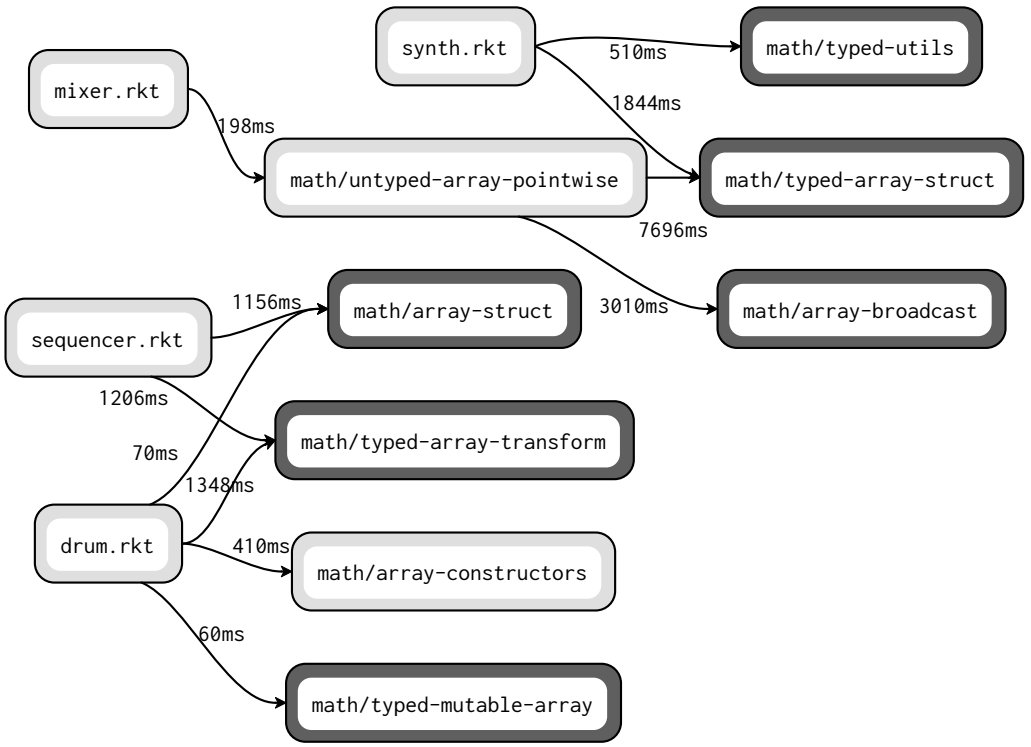


Figure 13: Feature profile (excerpt) and module graph view for the synthesizer

Maze Generator The second case study employs a version of a maze generator written by Olin Shivers. The program is 758 lines of Racket; it generates a maze on a hexagonal grid, ensures that it is solvable, and prints it.

The top portion of the output of an FSP shows 55% of the execution time is spent on output:

```
Output accounts for 55.31% of running time (1646 / 2976 ms)
  386 ms : maze.rkt:2:2
  366 ms : maze.rkt:3:2
  290 ms : maze.rkt:4:2
  [...]
```

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. The result of this reorganization is shown in figure 14. Following this advice results in a 1.39× speedup.

```
;; BEFORE
(display (if sw #\\ #\space))
(display (if s  #\_ #\space))
(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 14: Fusing output operations in the maze generator

The profiler reports that a dynamic cast inside an inner loop has no effect on performance. This result deviates from the more intuitive thought that such a cast would be costly. Programmers can use this information to keep the benefits of the cast.

Shill-Based Grading Script Our third case study involves a grading script, written by Scott Moore, that tests students' OCaml code. The script is 330 lines of Shill (Moore et al. 2014) code; Shill is a least-privilege shell scripting language written in Racket.

According to the FSP, contracts for security permissions account for more than 66% of execution time:

```
Shill Language account(s) for 98.33% of total running time (13809/2 / 7022 ms)
Cost Breakdown
  13809/2 ms : Capability Language

Contracts account(s) for 66.93% of total running time (9399/2 / 7022 ms)
Cost Breakdown
  4095 ms : pkg-native
  843/2 ms : grade
  141/2 ms : make
  [...]
```

Overhead from calling external programs causes the most slowdown. Unlike the sound synthesis example, Shill uses contracts and a kernel extension to ensure external programs do not violate Shill's security properties. The script contains three external programs, one being OCaml and the other two being text manipulation utilities. Reimplementing the two text manipulation utilities in Shill reduces the time spent in permission checking, resulting in a 32% improvement in the script's performance.

The results of this profile also contain useful negative information. Shill uses an ambient language to interface between traditional operating system permission models and Shill's capability language. The FSP shows that capability code accounts for 98% of the time spent inside of the Racket environment. This demonstrates that the transition layer imposed by the ambient language has little overhead.

```
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)
  [...]
```

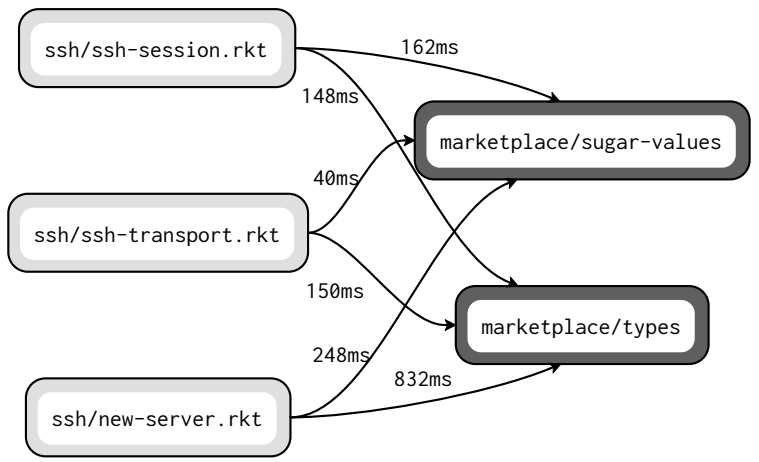


Figure 15: Profiling results for the SSH server (excerpt, top) module graph view of SSH server (bottom)

Marketplace-Based SSH Server The fourth case study involves an SSH server[10] in Marketplace. The SSH server is 3,762 lines of untyped Marketplace code and Marketplace itself is 4,801 lines of Typed Racket code. To exercise it, a driver script starts the server, connects to it, launches a Racket

[10]https://github.com/tonyg/marketplace-ssh

read-eval-print-loop on the local host, evaluates the expression `(+ 1 2 3 4 5 6)`, disconnects and terminates the server.

As figure 15 shows, the 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 25% of execution time. In Marketplace, spies are processes that observe other processes 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, shown in figure 15, shows 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 results in a speedup of around 4.41×. In addition, the report provides negative information. First, pattern matching again shows to have little cost despite its pervasive use. Additionally, Racket data structures can be implicitly coerced to a sequence that a program is capable of iterating over. This coercion has a runtime cost, but we show it is small.

Markdown Parser Our last case study involves a Parsack-based Markdown parser[11] written by Greg Hendershott. The Markdown parser is 4,058 lines of Racket code that we run on 1,000 lines of sample text.[12]

The FSP's feedback shows one interesting result. Specifically, backtracking from three branches takes noticeable time and accounts for 34%, 2%, and 2% of total execution time, respectively:

```
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
```

Based on the tool's report, moving the problematic branches further down in their enclosing disjunction is the appropriate action. Making this change leads to 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, and hand-written, instrumentation. His application specific instrumentation leads to a speed up of 1.37×. With no knowledge of the parser's internals, we were able to achieve a similar speedup in only a few minutes of work.

7.2 Plug-in Implementation Effort

Getting a Racket library ready for feature-specific profiling requires little effort, both in terms of the profilier's protocol and the creation of an optional analysis plug-in. It is easily within reach for library authors, especially because it does not require advanced profiling knowledge. To support this claim, we report anecdotal evidence and the lines of code for adding marks to other features, as well as their plug-ins.

For illustrative purposes, the instrumentation for Marketplace is shown in figure 16 with the added code highlighted. Unlike other examples, which use symbols as continuation mark keys, this code creates a fresh key using `make-continuation-mark-key` to avoid key collisions.

[11] https://github.com/greghendershott/markdown

[12] The sample text is "The Time Machine", by H. G. Wells. http://www.gutenberg.org/ebooks/35

```
1  (define marketplace-continuation-mark-key
2    (make-continuation-mark-key 'marketplace))
3
4  [...]
5
6  (marketplace-log 'debug "Entering process ~v(~v)" debug-name pid)
7  (define result (with-continuation-mark
8                   marketplace-continuation-mark-key (or debug-name pid)
9                   enclosed-expr))
10 (marketplace-log 'debug "Leaving  process ~v(~v)" debug-name pid)
```

Figure 16: Instrumentation for Marketplace (excerpt)

Feature	Cont. Marks Instrumentation LOC	Optional Analysis LOC	Feature LOC
Output	11	N/A	3685
Generic sequences	18	N/A	2225
Type casts and assertions	37	N/A	2479
Shill security policies	23	N/A	4501
Pattern matching	18	N/A	1834
Optional and keyword arguments	50	N/A	2041
Method dispatch	12	N/A	6823
Contracts	183	627	18840
Marketplace processes	7	9	5279
Parser non-terminals	18	60	3386

Figure 17: Instrumentation and analysis LOC per feature

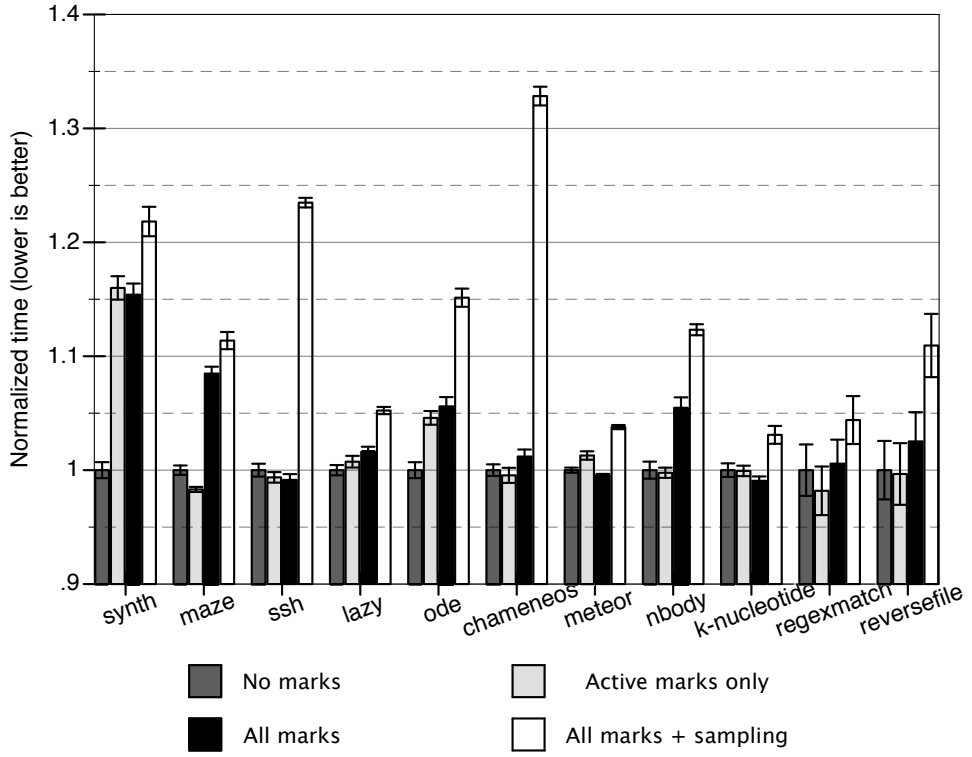
We report the number of lines of code for each remaining features' plug-in in figure 17. The second column reports the number of lines that are required to instrument the feature with marks. The third column reports the number of lines of plug-in analysis code. Finally, the fourth column reports the feature's implementation size in lines of code. The line counts for Marketplace and Parsack do not include the roughly 500 lines of Racket's edge profiler, which are re-linked into the plug-ins. With the exception of contract instrumentation—which covers multiple kinds of contracts and is spread across about 16,000 lines of the contract system—instrumentation is local and non-intrusive.

7.3 Overhead

Our prototype imposes an acceptable overhead on program execution. figure 18 summarizes our measurements. The results are the mean of 30 executions with 95% confidence error bars. The machine for these tests is a 64-bit Debian GNU/Linux system with 12 core Intel Xeon CPU clocked at 2.4 GHz and 11 GB of 1333 MHz DDR3 ram.

We use the programs listed in figure 18 as benchmarks. They include three of the case studies from section 7.1, two programs that make heavy use of contracts (lazy and ode), and six programs from the Computer Language Benchmarks Game[13] that use the features supported by our prototype. The first column of figure 18 corresponds to programs executing without any feature marks and serves as our baseline. The second 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 reports results for a program

[13] http://benchmarksgame.alioth.debian.org



Benchmark	Description	Features
synth	Sound synthesizer	contracts, output, generic sequences, keyword protocol
maze	Maze generator	output, assertions
ssh	SSH server	contracts, output, generic sequences, assertions, 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	assertions
k-nucleotide	K-nucleotide frequencies	generic sequences
regexmatch	Matching phone numbers	assertions, pattern matching
reversefile	Reverse lines of a file	output

Figure 18: Instrumentation and sampling overhead

that is run with all marks activated. The fourth column includes all of the above as well as the overhead from the sampling thread; 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%.

```
1 Rprof(filename="log.out", marks.profiling=TRUE);
2 for(i in 1:1e+6)
3   print(i)
4
5 for(i in 1:1e+5)
6   print(i)
7
8 Rprof(NULL);
9 feature.profile(filename="log.out")
```

```
Feature Report
(Feature times may sum to more or less than 100% of the total running time)
samples               558
time                  11.16s

feature: for, accounts for 0% of running time
0 : N/A
```

Figure 19: Looping Constructs

Based on this experiment, we conclude that instrumentation overhead is 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 (Mytkowicz et al. 2010).

This evaluation has one threat to validity. Because instrumentation is localized to feature code, its overhead is also localized. That is to say, the act of profiling a feature makes that feature slightly slower compared to the rest of the program. This may cause feature execution time to be overestimated. However, we conjecture that this is not a problem in practice because these overheads are low in general. In contrast, sampling overhead is uniformily[14] distributed across a program's execution and should not introduce such biases.

8 BROADER APPLICABILITY: PROFILING R

The applicability of feature-specific profiling is not limited to a particular language. Clearly linguistic features with complex costs are not unique to Racket, and many languages support some sort of user-defined features. Specifically, languages with first-class functions, macros, or facilities for embedding DSLs tend to come with complex-cost features and can therefore benefit from our idea.

This section demonstrates the feasibility of implementing a feature-specific profiler for the R programming language. For a straightforward adaptation of the Racket prototype, a language must have a sampling profiler and a stack annotation mechanism. While sampling profilers have been implemented for many languages, stack annotations are less commonly supported. In particular, R lacks them. Fortunately, adding continuation marks to a language such as R takes only a few lines of code.

8.1 A Sample Feature in R

Like most programming languages, R provides looping and mapping constructs such as `for`, `while`, and `lapply`.[15] Unfortunately, R implementers and users have different opinions on the performance

[14] Assuming random sampling, which we did not verify.

[15] `lapply` is similar to `map` in functional languages.

```
1  SEXP attribute_hidden do_for(SEXP call, SEXP op, SEXP args, SEXP rho) {
2      [...]
3      R_AddMark(FOR, call, TRUE);
4      for (i = 0; i < n; i++) {
5          switch (val_type) {  ... }
6          [...]
7          R_AddMark(FOR, ANTIMARK, TRUE);
8          eval(body, rho);
9          R_AddMark(FOR, call, TRUE);
10     }
11     [...]
12     return R_NilValue;
13 }
```

Figure 20: For-loop implementation with marks (excerpt)

of loops. “Tribal knowledge” in the R community suggests that looping constructs are slow and should be avoided in favor of vectorized operations. By contrast, R implementers claim that loops run reasonably fast and are slow only because of *secondary effects*. That is, loops are slow because of effects that are a by-product of using a feature but are not caused by using the feature directly. A profiler can help decide which of the common beliefs matters.

The left-hand side of figure 19 shows two `for` loop instances, the first on line 2 and the second on line 5. These loops have an accumulator whose costs must be attributed to the feature and a body of user code whose costs must *not* be attributed to the feature.

The right-hand side of figure 19 shows a run of these loops with a feature-specific profiler. As with the Racket prototype, a sampling profiler collects marks and antimarks, and an analyzer converts the data into information for programmers. The resulting display shows that no time is spent on the looping constructs. That is, the output (figure 19) shows no samples collected during code associated with looping constructs. While this one run is not conclusive evidence, it supports the R implementers’ claim that the direct overhead of looping constructs is not significant. R code that uses loops may still be slow, but the slowdown is not directly caused by the loop construct.

8.2 Implementation

Only a few modifications to R’s implementation were required to support feature-specific profiling. We implemented continuation marks in 134 lines of C. The extension to Rprof to inspect the new continuation marks accounted for 105 lines of code. Finally, we created a library to implement the analysis tool in 136 lines of R code. The implementation was created over a week with no prior experience with the R language or its internals. These results suggest that implementing feature-specific profiling may be possible even when the host language does support continuation marks or stack annotations.

Continuation marks. Although R does not support continuation marking directly, R programs can inspect and manipulate the call stack. It is possible to extend the frames in the call stack to support continuation marks with modifications to the R’s engine, namely, by extending frames to store marks in a hash map with unique keys and multiple payloads; by teaching the garbage collector how to track these maps; and by adding primitives to add and inspect continuation marks.

The capability to add marks to the stack must be accessible from both R and C, as R features are written in both languages. While supporting continuation marks does add to the complexity of the

```
# Extend %in% to operate over lists of nodes
setGeneric("%in%")
setMethod("%in%", c(x="Node", table="list"),
          function(x, table) {
            if(length(table) == 0)   return(FALSE)
            else if(table[[1]] == x) return(TRUE)
            else                     return(x %in% table[-1])
          })
```

```
Feature Report
(Feature times may sum to more or less than 100% of the total running time)
samples             3676
time                72.26s

feature: dynamic dispatch, accounts for 77% of running time
55.02s : %in%
0.26s : step
[...]
```

Figure 21: Dynamic Dispatch (top) and profile output (excerpt, bottom)

R code base, that complexity is localized. Marks also do not affect the performance of programs when they are disabled.[16]

The API for continuation marks in R is similar to its Racket variant:

- `add.mark(key, value)`, which imperatively adds (`key`,`value`) to the call stack.
- `marks(key)`, which walks the call stack and retrieves all marks that match `key`.

The API for Racket and R differ in primarily one aspect. The function to add a mark in Racket takes an expression, which is missing in the R variant. Unlike in Racket, `add.mark` places the continuation mark on the stack; the mark is implicitly removed when the current stack frame is popped.

R features that are implemented in C use the `R_AddMark` and `R_Marks` functions to manipulate continuation marks. These functions behave identically to their R equivalents. As an example, figure 20 shows the marks in R's implementation of `for`. The modified implementation places a mark at the beginning of the loop and replaces it with an antimark when the call to `eval` begins executing the loop's body. Once finished, the run-time removes the frame for `do-for` from the call stack, which also removes the mark.

Sampling profiler. Our prototype profiler uses Rprof, which is R's built-in sampling profiler. This profiler uses Unix interrupts to sample the call stack during execution. These samples are written to a file for post-processing. We modified Rprof to capture marks in addition to local variables To enables continuation marks, one must set `marks.profiling`, as shown in figure 19. Modifying Rprof to track continuation marks rather than using R's native stack inspection mechanism allows programmers to use other Rprof features, such as disabling the profiler during portions of the computation.

Analysis pass. Similar to the analysis pass in Racket, the R analysis pass shows four pieces of information: (1) the execution time; (2) number of samples collected; (3) a detailed list of every feature under analysis; (4) as well as the time spent in that feature and its instances. Programmers

[16]With our modifications, R can be compiled with and without continuation marks. While this may seem like a questionable design, it is actually a standard practice for many R tools (Morandat et al. 2012).

```
1 # Extend %in% to operate over lists of nodes
2 setGeneric("%in%")
3 setMethod("%in%", c(x="Node", table="list"), static.in)
4
5 static.in <- function(x,table) {
6   if(length(table) == 0) return(FALSE)
7   else if(table[[1]] == x) return(TRUE)
8   else return(static.in(x, table[-1]))
9 }
```

```
Feature Report
(Feature times may sum to more or less than 100% of the total running time)
samples     1758
time        35.16s

feature: s4-dispatch, accounts for 1% of running time
0.18s : %in%
0.12s : step
0.02s : show
[...]
```

Figure 22: Dynamic Dispatch (fixed, top) and profile output (excerpt, bottom)

run the analysis pass by giving the Rprof trace to the `feature.profile` function, as shown in figure 19 line 9. Processing each feature happens again in the same three steps that the Racket analysis performs. Figure 21 shows a report. It presents the cost dynamic dispatch for one of R's object systems. The analysis lists feature instances by method name rather than the source location. The data is particularly interesting because, like behavioral contracts, dynamic dispatch has dispersed costs. The source of dynamic dispatch is where the method definition is, but the cost manifests itself at the method's call sites. Because the continuation mark payloads store the name of the method, we can attribute the cost of dynamic dispatch to the proper source.

8.3 Use Cases

Next we present four small case studies of features that demonstrate how our profiler can help programmers. The case studies range over a wide spectrum of features: dynamic dispatch, parameter-naming function applications, copy-on-write parameter passing, and vector subsetting (Wickham 2014).[17]

Dynamic Dispatch. R's S4 object system supports multiple dispatch. Any R function, including primitives, can be transformed into the default implementation of an S4 method. When a method is called, it executes the implementation whose arguments best match the parameter types. The run-time system calls the default version of the function if no arguments match the required input types.

Figure 21 depicts the method `%in%`, used here as a part of Kruskal's algorithm to find a minimum spanning tree of a graph. This version uses dynamic dispatch recursively until it finds the desired node or the list is empty. The variant of this code in figure 22 uses dynamic dispatch *once* and thereafter calls a static function. Both variants of this method have equivalent behavior when the list is a homogeneous list of nodes. The recursive use of dynamic dispatch causes the first definition to be slower than the second. Conventional profilers identify the use of dynamic dispatch as having

[17] Called slicing in other languages.

```
# Set up a server servlet
# Boolean Boolean Boolean Boolean Boolean Boolean String Positive-Integer String
#   String Boolean Stuffer Manager Namespace String List<String> Responder String
#   String String String Boolean -> Boolean
serve <- function(command.line = FALSE, connection.close = FALSE,
                  launcher.browser = FALSE,  quiet = FALSE, banner = FALSE,
                  listen.ip = FALSE, port = "127.0.0.1", max.waiting = 511,
                  servlet.path = "", servlet.regexp = "", stateless = FALSE,
                  stuffer = NULL, manager = NULL, mime.type = FALSE,
                  servlet.namespace = NULL, servlet.root.path = "",
                  extra.file.paths = NULL, ssl.cert = "", log.file = "",
                  file.not.found.responder = NULL, ssl = FALSE, log.format = "") {
    FALSE
}

# Prepares the server's response
# String -> List<String>
respond <- function(x) {
    paste(paste(x,"hello"),sample(20));
}

attr(paste,"source") <- "paste"
attr(sample,"source") <- "sample"
attr(serve,"source") <- "serve"
attr(respond,"source") <- "respond"

for(i in 1:500000) {
  serve(port = "0.0.0.0", ssl = TRUE, qui = TRUE, max.w = 1023, log.file = "LOG",
        connection = TRUE, stuffer = function(x) x, banner = FALSE, command = TRUE)
  respond("somelongstring");
}
```

```
Feature Report
(Feature times may sum to more or less than 100% of the total running time)
samples             536
time                10.72s

feature: apply, accounts for 34% of running time
2.16s : serve
0.64s : paste
0.36s : respond
0.26s : sample
0.24s : generic
```

Figure 23: Function Application (top) and Profile Output (bottom)

a major performance impact in the program. Unfortunately, they cannot identify which specific use of dynamic dispatch is causing the performance problems, as they point to the S4 implementation but do not trace the costs back to calls. A feature-specific profile, as shown in figure 21, not only identifies dynamic dispatch as a major problem in the program, but it also points to the `%in%` method as the culprit

Function Application. Function calls in R may use named arguments in addition to traditional positional arguments. Named arguments at call sites are matched with named parameters. When a function is called and an argument is passed with a name, the argument is bound to the parameter whose name has the longest matching prefix of the name given for the argument. Thus, every

function used with named arguments must perform run-time string comparisons. Additionally, such a function application succeeds even if the number of arguments does not coincide with the number of parameters. Execution halts only when a parameter without a value is evaluated. As a result, function calls are difficult to optimize, and thus programmers consider them to be slow. An profiler can help identify which function calls cause the most runtime overhead and which are not cause for concern.

Figure 23 shows the skeleton of two functions: `serve` and `respond`. The former has a computationally simple and fast function body compared with a complicated slow calling interface. The latter has a complicated and slow function body but fast and simple calling interface. Traditional profilers find similar execution times for each function, because the combined running time of both the function body and calling interface are the same. While both timings are similar, `serve` spends more time in the calling interface than required. As shown in figure 23, our profiler identifies the primary bottleneck for `serve`'s calling interface. Thus, the program's performance can be improved by inlining `serve` or simplifiying its interface, which programmers can do in response to the FSP's actionable report.

Copy-on-Write. Conceptually, the semantics of R requires a deep copy of every argument passed into a function. In reality, the implementation only duplicates objects when absolutely necessary. Operations such as mutation force the duplication, creating copies. If no such operation occurs, then objects are never duplicated. This so-called copy-on-write policy can lead to unpredictable performance effects.

The `array.duplicate` function in figure 24 illustrates the surprising impact of copy-on-write. It duplicates the vector only if the second parameter is true. The program has two loops: a slow loop that causes the duplication of the array and a fast loop that does not duplicate the array. Traditional profilers correctly identify `array.duplicate` as a bottleneck. Our profiler identifies array duplication as the problem and furthermore identifies the duplication of a specific vector.

Vector Subset. Vectors are the basic data structures in R. Even a number such as 42 is a vector, which allows functions to operate over both vectors and other objects seamlessly. The vector-subset feature retrieves elements from a vector based on a vector of indices. Subset occurs frequently and some of their uses are more expensive than others. The syntax for subset uses square brackets, similar to array indexing. Traditional indexing is a special case of subsetting where the argument is a singleton vector. For example, the expression `c(2,4,6)[2]`, which uses the function `c` to create a vector, evaluates to 4.

Figure 25 shows a code snippet with two subset operations. The first retrieves every second element from the given vector. The other retrieves every third element; it occurs roughly one fourth as often as the first. Traditional profilers identify vector subsetting as the primary bottleneck in the program. Unfortunately, these profilers point to the implementation of subset, which is not enough information to identify which subset operation is costly. Our profiler instead indicates that the first subset operation is the primary cost center.

8.4 Profiling Overhead

Figure 26 reports the overhead our prototype imposes on several benchmarks. These results are the mean of 30 executions on a machine running OS X Yosemite with a 4 core Intel Core i7 clocked at 2.5 GHz and 16 GB of 1600 MHz DDR3 ram. The error bars show the 95% confidence interval. The samples are collected with R build r69166,[18] and the sampling interval is 20ms.

[18] https://github.com/LeifAndersen/R

```
# x and y are large vectors
modified <- 1:1000000
constant <- 1:1000000

# Annotate x and y with "source" attribute,
# copy-on-write uses to distinguish
# individual instances
attr(x,"source") <- "modified object";
attr(y,"source") <- "constant object";

# Mutate first element in copied vector
# Vector<Any> Boolean -> Vector<Any>
array.modify <- function(x, copy) {
    z <- x
    if(copy) z[1] <- 42
    x
}

# Slow Loop
for(i in 1:1000)
  array.modify(x, TRUE);

# Fast Loop
for(i in 1:1000)
  array.modify(y, FALSE);
```

```
Feature Report
(Feature times may sum to more or less than 100% of the total running time)
samples                    3430
time                       68.6s

feature: duplicate, accounts or 28% of running time
11.82s : modified object
6.46s : generic
0.64s : constant object
```

Figure 24: Copy-on-Write (top) and profile output (bottom)

The benchmark programs are described in figure 26. They include two benchmarks from the Computer Language Benchmark Game that use features our prototypes supports, the five feature samples used earlier in the paper, and Oliver Keyes's "GoingPostel", a program that aggregates information about IETF RFCs.

We report runs of each program in three configurations:

- The first configuration corresponds to the program executing without continuation marks or profiler in a build of R with all required packages installed.
- The second configuration corresponds to the program executing in a build of R with continuation marks. All of features that our profiler supports annotate the stack with continuation marks, but the sampling is turned off.
- The third configuration is like the second, but with profiling turned on.

With continuation marks and profiling, the overhead is lower than 20% for half of the programs and larger for the other half (85%, 100%, 42%, and 59%). The latter four programs, however, are feature samples, which essentially perform no work except exercise the relevant feature, and therefore represent pathological worst cases. In all cases the cost of sampling is less than 2%. The

```
x <- sample(1000000)

for(i in 1:1000) {
  x[seq(1,length(x), 2)]
  if(sample(4)[1] == 1) x[seq(1,length(x), 3)]
}
```

```
Feature Report
(Feature times may sum to more or less than 100% of the total running time)
samples                    1279
time                       25.58s

feature: subset+body, accounts for 63% of running time
13.94s : x[seq(1, length(x), 2)]
2.14s : x[seq(1, length(x), 3)]
```

Figure 25: Vector Subset (top) and profile output (bottom)

primary cause of overhead comes from continuation marks rather than the modified sampling profiler. A threat to validity comes from the fact that continuation mark overhead is concentrated at feature annotations, which causes features to appear slower than they are, thus skewing results. Nevertheless, we consider this experiment to validate the viability of feature-specific profiling. While the overheads are greater than in Racket, performance of the R profiler remains acceptable. We conjecture that this prototype could be improved to match the performance of the Racket implementation with careful tuning of the implementation.

9 LIMITATIONS

Our approach to feature-specific profiling applies to some linguistic features. This section discusses limitations. We believe they are not fundamental to the idea of feature-specific profiling and that they could be addressed by different approaches to data gathering.

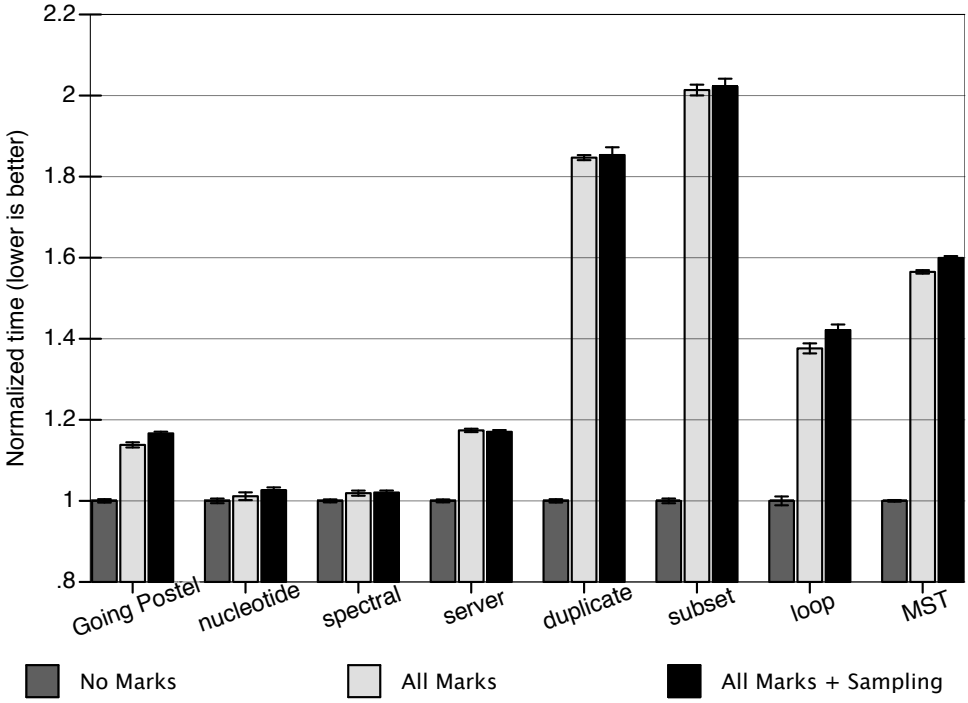
Because our instrumentation strategy relies on continuation marks, it does not support features that interfere with marks. This rules out non-local control features that unroll the stack, e.g. exception raising. This also prevents us from profiling continuation marks themselves.

The sampler must be able to observe a feature in order to profile it. This rules out uninterruptible features, e.g., 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 sampled reliably.

Some non-syntactic language features, such as garbage collection, have costs that cannot be attributed to a single source location in the program. Frequently, these features have costs that are small and spread out, and are thus difficult to capture with a sampling profiler. An event-based approach, such as Morandat et al.'s (2012), would fare better.

While our prototype profiles concurrent programs such as the Marketplace described in section 5, it cannot handle parallel programs. We conjecture that our approach could be extended to handle multi-threaded programs but we have not tried.

Features have both direct costs and indirect costs. Direct costs come from using a feature, while indirect costs are not imposed by the feature itself but by lost opportunities due to a feature's use. Profiliers only track direct costs.



Benchmark	Description	Features	Lines of Code
Going Postel	RFC Analyzer	Copy-on-Write, Function Calls	171
nucleotide	K Nucleotide Alternative	Copy-on-Write, Function Calls, Loops	51
spectral	Spectral Norm	Copy-on-Write, Function Calls, Loops	30
server	Server Creation/Response Calls	Function Calls	42
duplicate	Copy-on-Write	Copy-on-Write	19
subset	Vector Subset	Vector Subset	8
loop	loops	Loops, Function Calls	7
MST	Kruskal's Algorithm	Dynamic Dispatch, Function Calls, Loops	226

Figure 26: Instrumentation and Sampling Performance of the Going Postel (Left),Computer Language Benchmark Game Benchmarks (Center),and Feature Samples (Right)

Finally, it is up to the feature authors to work out the correctness of their annotations. While feature authors can clearly make mistakes when annotating their libraries, in our experience and that of our users, we have not found this to be an issue at all. Because authors are familiar with their libraries, they also tend to have a reasonable idea of where adding annotations will be *useful*.

10 RELATED WORK

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

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 definition—or dynamic—e.g.,

per HTTP request. Each of these views 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. In contrast, traditional profilers may detect a broader range of issues than feature-specific profilers, such as inefficient algorithms, which are invisible to feature-specific profilers.

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 performance. Vertical profiling focuses on helping programmers understand how the interaction between different layers of abstraction 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 some understanding of the compiler and run-time system. Hauswirth et al.'s work introduces the notion of *software performance monitors*, which are analogous to hardware performance monitors but record software-related performance events. These monitors could possibly be used to implement feature-specific profiling by tracking the execution of feature code.

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. Some recent examples include Singer and Kirkham's (2008) profiler, which assigns costs to programmer-annotated code regions, listener latency profiling (Jovic and Hauswirth 2011), which reports high-latency operations, and Tamayo et al.'s (2012) tool, which provides information about the cost of database operations.

Dynamic instrumentation frameworks such as Valgrind (Nethercote and Seward 2007) or Javana (Maebe et al. 2006) 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. They have not been used to measure the cost of individual linguistic features.

Like a feature-specific profiler, an optimization coach (St-Amour et al. 2012) focuses on enabling compiler optimizations through a feedback loop that involves the developer. The two are complementary. Optimization coaches operate at compile time whereas feature-specific profilers, like other profilers, operate at run time. Because of this, feature-specific profilers require representative program input to operate, whereas coaches do not. Then again, by having access to run time data, feature-specific profilers can target actual program hot spots, while existing optimization coaches must rely on static heuristics to prioritize reports.

An important tool for measuring R programs is `tracemem`. It is included with the R tool suite, but requires programmers to rebuild R. This tool serves to track uses of copy-on-write during the execution of R programs. It tracks the memory that is being copied, and the source location that is responsible for causing the copy. Also, it allows programmers to tag individual objects they care about tracking, while ignoring everything else.

11 CONCLUSION

Feature-specific profiling is a novel profiling technique that supplements traditional cost-centers with language-specific ones. These cost centers give a new perspective on program performance, enabling developers to tune their programs. Feature-specific profiling is especially useful when

programs use language features with dispersed or non-local costs. Additionally, feature-specific profiling is useful with languages that allow for the programmatic creation of new features such as Racket, R, or even C++. The implementation of a feature-specific profiler is straightforward. If the host language supports stack annotations and inspection, such as Racket, then implementing is as simple as that of a sampling profiler. Languages without this support, such as R, must be extended by adding stack annotations. This paper shows that modifications required are practical.

While using a feature-specific profiler requires little effort, it does require more setup than traditional profilers. Either library authors must add support for their code, or developers must modify the library's source. Fortunately, adding support is simple and generally requires only a few lines of code. The information provided by the profiler has the same limitations as that of stack-based sampling profilers. This means that language features that do not show up on the call stack cannot be measured. The sampling nature of our profiler also means that it can only profile interruptible features. Other profile designs, such as an event based profiler, trade these limitations for a different set. The idea of feature-specific profiling itself is not limited to the architecture designed in this paper. We conjecture that other architectures can also support feature-specific profiling.

Acknowledgements. Tony Garnock-Jones implemented the Marketplace plug-in and helped with the SSH case study. Stephen Chang assisted with the Parsack plug-in and the Markdown case study. Christos Dimoulas and Scott Moore collaborated on the Shill plug-in and the grading script experiment. Robby Findler provided assistance with the contract system. Oliver Keyes implemented Going Postel. We thank Eli Barzilay, Matthew Flatt, Asumu Takikawa, Sam Tobin-Hochstadt, Benjamin Chung, Helena Kotthaus, Tomas Kalibera, Oli Flückiger, Kyle Bemis, Olga Vitek, and Luke Tierney for helpful discussions. This work was partially supported by the National Science Foundation under Grant SHF 1544542 and the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement 695412). Any opinions, findings, and conclusions expressed in this material may be those of the authors and likely do not reflect the views of our funding agencies.

BIBLIOGRAPHY

Gene M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proc. Spring Joint Computer Conference*, 1967.

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

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

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

R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntax Abstracton in Scheme. In *Proc. Lisp and Symbolic Computation*, 1993.

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

Robert Bruce Findler and Matthias Felleisen. Contracts for Higher-order Functions. In *Proc. International Conference on Functional Programming*, 2002. https://doi.org/10.1145/581478.581484

Matthew Flatt and Eli Barzilay. Keyword and Optional Arguments in PLT Scheme. In *Proc. Workshop on Scheme and Functional Programming*, 2009.

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

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

Matthias Hauswirth, Peter F. Sweeney, Amer Diwan, and Michael Hind. Vertical profiling. In *Proc. Object-oriented Programming, Systems, Languages, and Applications*, pp. 251–269, 2004.

Carl Hewitt, Peter Bishop, and Richard Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proc. International Joint Conference on Artificial Intelligence*, 1973.

Milan Jovic and Matthias Hauswirth. Listener latency profiling. *Science of Computer Programming* 19(4), pp. 1054–1072, 2011.

Jonas Maebe, Dries Buytaert, Lieven Eeckhout, and Koen De Bosschere. Javana: A System for Building Customized Java Program Analysis Tools. In *Proc. Object-oriented Programming, Systems, Languages, and Applications*, 2006. https://doi.org/10.1145/1167515.1167487

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

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

Scott Moore, Christos Dimoulas, Dan King, and Stephen Chong. SHILL: a secure shell scripting language. In *Proc. USENIX Symposium on Operating Systems Design and Implementation*, 2014. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/moore

Floréal Morandat, Brandon Hill, Leo Osvald, and Jan Vitek. Evaluating the Design of the R Language. In *Proc. European Conference on Object-Oriented Programming*, 2012. https://doi.org/10.1007/978-3-642-31057-7_6

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

Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proc. Programming Langauges Design and Implementation*, 2007. https://doi.org/10.1145/1273442.1250746

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

R Development Core Team. R Language Definition. R Development Core Team, 3.3.1, 2016. http://web.mit.edu/~r/current/arch/amd64_linux26/lib/R/doc/manual/R-lang.pdf

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.

Vincent St-Amour, Leif Andersen, and Matthias Felleisen. Feature-specific Profiling. In *Proc. International Conference on Compiler Construction*, 2015. https://doi.org/10.1007/978-3-662-46663-6_3

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

Juan M. Tamayo, Alex Aiken, Nathan Bronson, and Mooly Sagiv. Understanding the behavior of database operations under program control. In *Proc. Object-oriented Programming, Systems, Languages, and Applications*, pp. 983–996, 2012.

Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of Typed Scheme. In *Proc. Principles of Programming Languages*, pp. 395–406, 2008.

Hadley Wickham. Advanced R. First edition. Chapman and Hall/CRC, 2014. http://adv-r.had.co.nz/