Feature-Specific Profiling

Vincent St-Amour, Leif Andersen, and Matthias Felleisen

PLT @ Northeastern University
{stamourv,leif,matthias}@ccs.neu.edu

Abstract. High-level languages come with significant readability and maintainability benefits. Their performance costs, however, are usually not predictable, at least not easily. Programmers may accidentally use high-level features in ways that compiler writers could not anticipate, and they may thus produce underperforming programs as a result.

This paper introduces *feature-specific profiling*, a profiling technique that reports performance costs in terms of linguistic constructs. With a feature-specific profiler, a programmer can identify specific instances of language features that are responsible for performance problems. After explaining the architecture of our feature-specific profiler, the paper presents the evidence in support of adding feature-specific profiling to the programmer's toolset.

1 Weighing Language Features

Many linguistic features,[1] come with difficult-to-predict performance costs. First, the cost of a specific use of a feature depends on its context. For instance, use of reflection may not observably impact the execution time of some programs but may have disastrous effects on others. Second, the cost of a feature also depends on its mode of use; a higher-order type coercion tends to be more expensive than a first-order coercion (see section 2).

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

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

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

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

In this paper, we

- introduce the idea of feature-specific profiling,
- explain the architecture of our prototype and its API for feature plug-ins,
- and present an evaluation of our prototype covering both the actionability of its results and the effort required to implement plug-ins.

The rest of this paper is organized as follows. In section 2 we describe the features that we chose to support in our prototype. In section 3 we outline the architecture of our framework and provide background on its instrumentation technique. In sections 4 and 5 we describe the implementation in detail. We present evaluation results in section 6, then explain the limitations of our architecture, relate to existing work, and conclude.

2 Feature Corpus

In principle, a feature-specific profiler should support all the features that a language offers or that the author of a library may create. This section presents the Racket (Flatt and PLT 2010) features that our prototype feature-specific profiler supports, which includes features from the standard library, and from three third-party libraries. The choice is partially dictated by the underlying technology; put differently, the chosen technology can deal with linguistic features whose dynamic extent obeys a stack-like behavior.

The list introduces each feature and outlines the information the profiler provides about each. We provide additional background for three features in particular—contracts, Marketplace processes (Garnock-Jones et al. 2014), and parser backtracking—which are key to the evaluation case studies presented in section 6.1.

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

Output Our tool traces time programs spend in Racket's output subsystem back to individual console, file or network output function call sites.

Generic sequence dispatch Racket's iteration forms can iterate over any sequence datatype, which includes built-in types such as lists and vectors as well as user-defined types. Operating generically requires dynamic dispatch and imposes a run-time cost. Our profiler reports which iteration forms spend significant time in dispatch and thus suggests which ones to replace with specialized iteration forms.

Type casts and assertions Typed Racket, like other typed languages, provides type casts to help programmers get around the constraints of the type system. Like Java's casts, Typed Racket's casts are safe and involve runtime checks, which can have a negative impact on performance. Casts to higher-order types wrap values with proxies and are therefore especially expensive. Our tool reports time spent in each cast and assertion.

Shill security policies The Shill scripting language (Moore et al. 2014) restricts how scripts can use system resources according to user-defined security policies. Shill enforces policies dynamically, which incurs overhead on every restricted operation. Because Shill is implemented as a Racket extension, it is an ideal test case for our feature-specific profiler. Our tool succeeds in reporting time spent enforcing each policy.

Pattern matching Racket comes with an expressive pattern matching construct. Our profiler reports time spent in individual patterns matching forms, excluding time spent in form bodies.

Optional and keyword argument functions Racket's functions support optional as well as keyword-based arguments. To this end, the compiler provides a special function-call protocol, distinct from, and less efficient than, the regular protocol. Our tool reports time spent on this protocol per function.

Method Dispatch On top of its functional core, Racket supports class-based object-oriented programming. Method calls have a reputation for being more expensive than function calls. Our tool profiles the time spent performing method dispatch for each method call site, reporting the rare cases where dispatch imposes significant costs.

2.1 Contracts

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

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

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

Figure 1 shows an excerpt from an HTTP client library. It provides `make-fetcher`, which accepts a user agent and returns a function that performs requests using that user

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

driver.rkt

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

http-client.rkt

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

Figure 1: Contract for an HTTP client

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

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

2.2 Marketplace Processes

The Marketplace library allows programmers to express concurrent systems functionally as trees of sets of processes grouped within task-specific virtual machines (VMs)[2] that communicate via publish/subscribe. Marketplace is especially suitable for building network services; it has been used as the basis of an SSH server (see section 6.1.2) and a DNS server. While organizing processes in a hierarchy of VMs has clear software engineering benefits, deep VM nesting hinders reasoning about performance. Worse, different processes often execute the same code, but because these processes do not map to threads, traditional profilers may attribute all the costs to one location.

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

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

2.3 Parser Backtracking

The Parsack parsing library[3] provides a disjunction operator that attempts to parse alternative non-terminals in sequence. The operator backtracks in each case unless the non-terminal successfully matches. When the parser backtracks, however, any work it did for matching that non-terminal does not contribute to the final result and is wasted.

For this reason, ordering non-terminals within disjunctions to minimize backtracking, e.g., by putting infrequently matched non-terminals at the end, can significantly improve parser performance. Our feature-specific profiler reports time spent on each disjunction branch from which the parser ultimately backtracks.

3 The Profiler's Architecture

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

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

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

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

3.1 Inspecting the Stack with Continuation Marks

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

[3] `https://github.com/stchang/parsack`

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

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

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

Outside of these operations, continuation marks do not affect a program's semantics.[4] Figure 2 illustrates the working of continuation marks with a function that traverses binary trees and records paths from roots to leaves. Whenever the function reaches an internal node, it leaves a continuation mark recording that node's value. When it reaches a leaf, it collects those marks, adds the leaf to the path and returns the completed path.

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


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

Figure 2: Recording paths in a tree with continuation marks

Continuation marks are extensively used in the Racket ecosystem, notably for 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, and for exception handling. Serializable continuations in the PLT web server (McCarthy 2010) are also implemented using continuation marks.

Beyond Racket, continuation marks have also been implemented on top of 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.

3.2 Feature-specific Data Gathering

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

[4] Continuation marks also preserve proper tail call behavior.

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

Figure 3 shows an excerpt from the instrumentation of Typed Racket assertions. The underlined conditional is responsible for performing the actual assertion. The feature mark's key should uniquely identify the construct. In this case, we use the symbol `'TR-assertion` as key. Unique choices avoid false reports and interference by distinct plug-ins. As a consequence, our feature-specific profiler can present a unified report to users; it also implies that users need not select in advance the constructs they deem problematic.

The mark value—or *payload*—can be anything that identifies the instance of the feature to which the cost should be assigned. In figure 3, the payload is the source location of a specific assertion in the program, which allows the profiler to compute the cost of individual assertions.

Writing such plug-ins, while simple and non-intrusive, requires access to the implementation of the feature of interest. Because it does not require any specialized profiling knowledge, however, it is well within the reach of the authors of linguistic constructs.

```
(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 3: Instrumentation of assertions (excerpt)

Antimarking Features are seldom "leaves" in a program; feature code usually runs user code whose execution time may not have to count towards the time spent in the feature. For example the profiler must not count the time spent in function bodies towards the function call protocol for keyword arguments.

To solve this problem, a feature-specific profiler expects *antimarks* on the stack. Such antimarks are continuation marks with a distinguished value that delimit a feature's code. Our protocol dictates that the continuation mark key used by an antimark is the same as that of the feature it delimits and that they use the `'antimark` symbol as payload. Figure 4 illustrates the idea with code that instruments a simplified version of Racket's optional and keyword argument protocol. In contrast, assertions do not require antimarks because user code evaluation happens outside the marked region.

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

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

Figure 4: Use of antimarks in instrumentation

Sampling During program execution, our profiler's sampling thread periodically collects and stores continuation marks from the main thread. The sampling thread has knowledge of the keys used by feature marks and collects marks for all features at once.

3.3 Analyzing Feature-specific Data

After the program execution terminates, the core profiler analyzes the data collected by the sampling thread to produce a feature cost 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 As explained in section 3.2, payloads identify individual feature instances. Our accounting algorithm groups samples by payload and adds up the cost of each sample; the sums correspond to the cost of each feature instance. Our profiler then generates reports for each feature, using payloads as keys and time costs as values.

Report composition Finally, after generating individual feature reports, our profiler combines them into a unified report. Constructs absent from the program or those inexpensive enough to never be sampled are pruned to avoid clutter. The report lists features in descending order of cost, and does likewise for instances within feature reports.

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

[5] `http://imranontech.com/2007/01/24/`

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

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

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

Figure 5: Feature profile for FizzBuzz

4 Profiling Rich Features

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

While the creator of a structurally rich feature can use a basic plug-in to measure some aspects of its cost, it is best to adapt a different strategy for evaluating such features. This section shows how to go about such an adaptation. Section 6.2 illustrates with an example how to migrate from a basic plug-in to one appropriate for a structurally rich feature.

4.1 Custom Payloads

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

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

Marketplace processes The Marketplace plug-in uses process names as payloads. Since `current-continuation-marks` gathers all the marks currently on the stack, the sampling thread can gather *core samples*.[6] 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.

[6] In analogy to geology, a core sample includes marks from the entire stack.

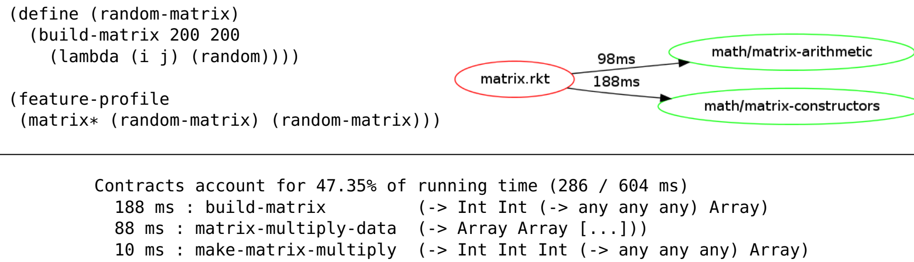


Figure 6: Module graph and by-value views of a contract boundary

Parser backtracking The Parsack plug-in combines three values into a payload: the source location of the current disjunction, the index of the active branch within the disjunction, and the offset in the input where the parser is currently matching. Because parsing a term may require recursively parsing sub-terms, the Parsack plug-in gathers core samples that allow it to attribute time to all active non-terminals.

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

4.2 Analyzing Rich Features

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

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

To this end, our analysis generates a *module graph* view of contract boundaries. This graph shows modules as nodes, contract boundaries as edges and contract costs as labels on edges. Because typed-untyped boundaries are an important source of contracts, the module graph distinguishes typed modules (in green) from untyped modules (in red). To generate this view, our analysis extracts component names from blame objects. It then groups payloads that share pairs of parties and computes costs as discussed in section 3.3. The top-right part of figure 6 shows the module graph for a program that constructs two random matrices and multiplies them. This code resides in an untyped module, but the matrix functions of the `math` library reside in a typed module. Hence linking the client and the library introduces a contract boundary between them.

In addition to the module graph, our feature-specific profiler provides other views as well. For example, the bottom portion of figure 6 shows the *by-value* view, which provides fine-grained information about the cost of individual contracted values.

Marketplace Processes The goal of our feature-specific analysis for Marketplace processes is to assign costs to individual 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 code.

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

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

Figure 7 shows the accounting from a Marketplace-based echo server. The first entry of the profile shows the ground VM, which spawns all other VMs and processes. The rightmost column shows how execution time is split across the ground VM's children. Of note are the processes handling requests from two clients. As reflected in the profile, the client on port 53588 is sending ten times as much input as the one on port 53587.

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

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

Figure 8 shows a simple parser that first attempts to parse a sequence of `b`s followed by an `a`, and in case of failure, backtracks in order to parse a sequence of `b`s. The right portion of figure 8 shows the output of the feature-specific profiler when running the parser on a sequence of 9,000,000 `b`s. 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.

```
26 (define $a (compose $b (char #\a)))
27 (define $b (<or> (compose (char #\b) $b)
28                  (nothing)))
29 (define $s (<or> (try $a) $b))
30
31 (feature-profile (parse $s input))
```

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

Figure 8: An example Parsack-based parser and its backtracking profile

5 Instrumentation Control

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

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

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

5.1 Syntactic Latent Marks

Syntactic latent marks exist as annotations on the intermediate representation (IR) of user code. To add a latent mark, the implementation of a feature leaves tags[7] on the residual program's IR instead of directly inserting feature marks. These tags are discarded after compilation and thus have no run-time effect on the program. Other meta-programs or the compiler can observe latent marks and turn them into active marks.

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

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

[7] We use Racket's syntax property mechanism, but any IR tagging mechanism would apply.

5.2 Functional Latent Marks

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

6 Evaluation

Our evaluation addresses two promises concerning feature-specific profiling: that measuring in a feature-specific way supplies useful insights into performance problems, and that it is easy to implement new plug-ins. This section first presents case studies that demonstrate how feature-specific profiling improves the performance of programs. Then it reports on the amount of effort required to implement plug-ins. The online version of this paper[8] includes an appendix that discusses run-time overhead.

6.1 Case Studies

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

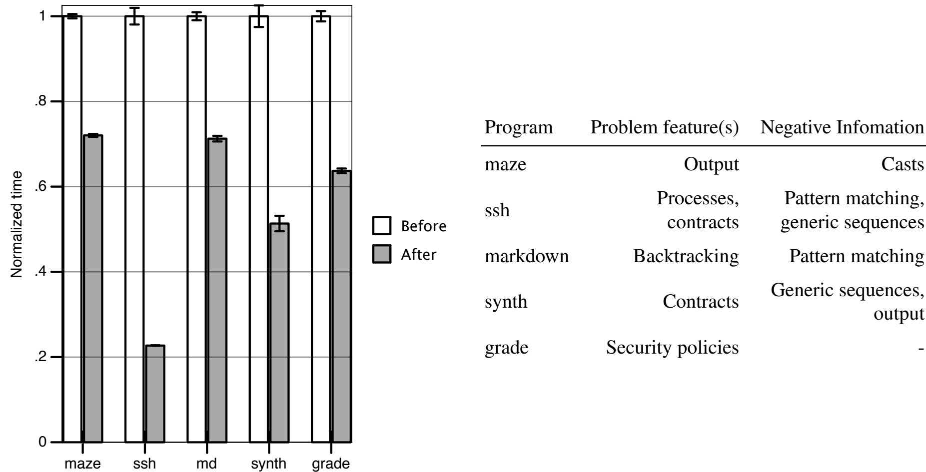
We present three case studies suffering from the overhead of specific features. Each subsection describes a program, summarizes the feature-specific profiler's feedback, and explains the changes that directly follow from the report. Figure 9 presents the results of comparing execution times before and after the changes. It also includes results from two additional programs—a sound synthesis engine and a Shill-based automatic grading script—which we do not discuss due to a lack of space.

Maze Generator Our first case study employs a Typed Racket version of a maze generator, due to Olin Shivers. For scale, the maze generator is 758 lines of code. The program generates a maze on a hexagonal grid, ensures that it is solvable, and prints it.

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

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

[8] `http://www.ccs.neu.edu/racket/pubs/#cc15-saf`



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

Results are the mean of 30 executions on a 6-core 64-bit Debian GNU/Linux system with 12GB of RAM.
Because Shill only supports 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 9: Execution time after profiling and improvements (lower is better)

Marketplace-Based SSH Server Our second case study involves an SSH server[9] written using the Marketplace library. For scale, the SSH server is 3,762 lines of code. To exercise it, a driver script starts the server, connects to it, launches a Racket read-eval-print-loop on the host, evaluates the expression `(+ 1 2 3 4 5 6)`, disconnects and terminates the server.

As figure 10 shows, our feature-specific profiler brings out two useful facts. First, two *spy* processes—the `tcp-spy` process and the boot process of the `ssh-session` VM—account for over 25% of the total 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, of which figure 11 shows an excerpt, reports that the majority of these contracts lie at the boundary between the typed Marketplace library and the untyped SSH server. We can selectively remove these contracts in one of two ways: by adding types to the SSH server or by disabling typechecking in Marketplace.

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

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

```
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 10: Profiling results for the SSH server (excerpt)

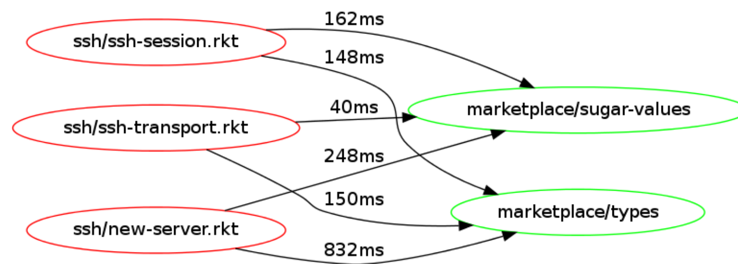


Figure 11: Module graph view for the SSH server (excerpt)

Markdown Parser Our last case study involves a Parsack-based Markdown parser,[10] due to Greg Hendershott. For scale, the Markdown parser is 4,058 lines of Racket code. To profile the parser, we ran it on 1,000 lines of sample text.[11]

As figure 12 shows, backtracking from three branches took noticeable time and accounted for 34%, 2%, and 2% of total execution time, respectively. Based on the tool's report, we moved the problematic branches further down in their enclosing disjunction, which produced a speedup of $1.40\times$.

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

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

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

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

Figure 12: Profiling results for the Markdown parser (excerpt)

6.2 Plug-in Implementation Effort

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

Mr. Garnock-Jones, an experienced Racket programmer, implemented the plug-in himself, with the first author acting as interactive documentation of the framework. Implementing the first version of the plug-in took about 35 minutes. At that point, Mr. Garnock-Jones had a working process profiler that performed the basic analysis described in section 3.3. Adding a feature-specific analysis took an additional 40 minutes. Less experienced library authors may require more time for a similar task. Nonetheless, we consider this amount of effort to be quite reasonable.

For the remaining features, we report the number of lines of code for each plug-in in figure 13. The third column reports the number of lines of domain-specific analysis code. The basic analysis is provided as part of the framework. The line counts for

[10] `https://github.com/greghendershott/markdown`
[11] An excerpt from "The Time Machine" by H.G. Wells.

Marketplace and Parsack do not include the portions of Racket's edge profiler that are re-linked into the plug-ins, which account for 506 lines. With the exception of contract instrumentation—which covers multiple kinds of contracts and is spread across the 16,421 lines of the contract system—instrumentation is local and non-intrusive.

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

Figure 13: Instrumentation and analysis LOC per feature

7 Limitations

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

Control features Because our instrumentation strategy relies on continuation marks, it does not support features that interfere with marks. This rules out non-local control features that unroll the stack, such as exception raising.

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

Diffuse features Some features, such as garbage collection, have costs that are diffused throughout the program. This renders mark-based instrumentation impractical. An event-based approach, such as Morandat et al.'s (2012), would fare better. The use of events would also make feature-specific profiling possible in languages that do not support stack inspection.

8 Related Work

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

8.1 Traditional Profiling

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

Each 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. Traditional profilers may not provide actionable information in such cases. Furthermore, by identifying costly features, feature-specific profilers point programmers towards potential solutions, namely correcting feature usage. In contrast, traditional profilers often report costs without helping find solutions. Conversely, traditional profilers may detect a broader range of issues than feature-specific profilers, such as inefficient algorithms, which are invisible to feature-specific profilers.

8.2 Vertical Profiling

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

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

8.3 Alternative Profiling Views

A number of profilers offer alternative views to the traditional attribution of time costs to program locations. Most of these views focus on particular aspects of program performance and are complementary to the view offered by a feature-specific profiler. 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.

8.4 Dynamic Instrumentation Frameworks

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., though they have not been used to measure the cost of linguistic features.

8.5 Optimization Coaching

Like a feature-specific profiler, an optimization coach (St-Amour et al. 2012) aims to help non-experts improve the performance of their programs. Where coaches focus on enabling compiler optimizations, feature-specific profilers focus on avoiding feature misuses. 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. On the other hand, by having access to run time data, feature-specific profilers can target actual program hot spots, while coaches must rely on static heuristics to prioritize reports.

9 Conclusion

This paper introduces feature-specific profiling, a technique that reports program costs in terms of linguistic features. It also presents an architecture for feature-specific profilers that allows the authors of libraries to implement plug-ins in their libraries.

The alternative view on program performance offered by feature-specific profilers allows easy diagnosis of performance issues due to feature misuses, especially those with distributed costs, which might go undetected using a traditional profiler. By pointing to the specific features responsible, feature-specific profilers provide programmers with actionable information that points them towards solutions.

Acknowledgements Tony Garnock-Jones implemented the Marketplace plug-in and helped us perform 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. We thank Eli Barzilay, Matthew Flatt, Asumu Takikawa, Sam Tobin-Hochstadt and Jan Vitek for helpful discussions.

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

Bibliography

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

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

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

Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *Proc. ICFP*, pp. 48–59, 2002.

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. ESOP*, pp. 473–492, 2014.

Matthias Hauswirth, Peter F. Sweeney, Amer Diwan, and Michael Hind. Vertical profiling. In *Proc. OOPSLA*, pp. 251–269, 2004.

Milan Jovic and Matthias Hauswirth. Listener latency profiling. *SCP* 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. OOPSLA*, pp. 153–168, 2006.

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

Jay McCarthy. The two-state solution: native and serializable continuations accord. In *Proc. OOPSLA*, pp. 567–582, 2010.

Scott Moore, Christos Dimoulas, Dan King, and Stephen Chong. SHILL: a secure shell scripting language. In *Proc. OSDI*, 2014.

Floréal Morandat, Brandon Hill, Leo Osvald, and Jan Vitek. Evaluating the Design of the R Language. In *Proc. ECOOP*, pp. 104–131, 2012.

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

Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proc. PLDI*, pp. 89–100, 2007.

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

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

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

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

T. Stephen Strickland, Sam Tobin-Hochstadt, Robert Bruce Findler, and Matthew Flatt. Chaperones and impersonators. In *Proc. OOPSLA*, pp. 943–962, 2012.

Juan M. Tamayo, Alex Aiken, Nathan Bronson, and Mooly Sagiv. Understanding the behavior of database operations under program control. In *Proc. OOPSLA*, pp. 983–996, 2012.

Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage refactoring: from scripts to programs. In *Proc. DLS*, pp. 964–974, 2006.

A. Instrumentation Overhead

Our feature-specific profiler imposes an acceptably low overhead on program execution. For a summary, see figure 14, which reports preliminary overhead measurements. These results are the mean of 30 executions on a 64-bit Debian GNU/Linux system[12] and include error bars one standard deviation above and below the mean.

We use the programs listed in figure 15 as benchmarks. They include three of the case studies from section 6.1, 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 14 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 also includes all activated latent marks. 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%.

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

We identify one threat to validity. Because instrumentation is localized to feature code, its overhead is also localized. This may cause feature execution time to be overestimated. Because these overheads are low in general, we conjecture this problem to be insignificant in practice. In contrast, sampling overhead is uniformily[14] distributed across a program's execution and should not introduce such biases.

[12] The same system used for the measurements of section 6.

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

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

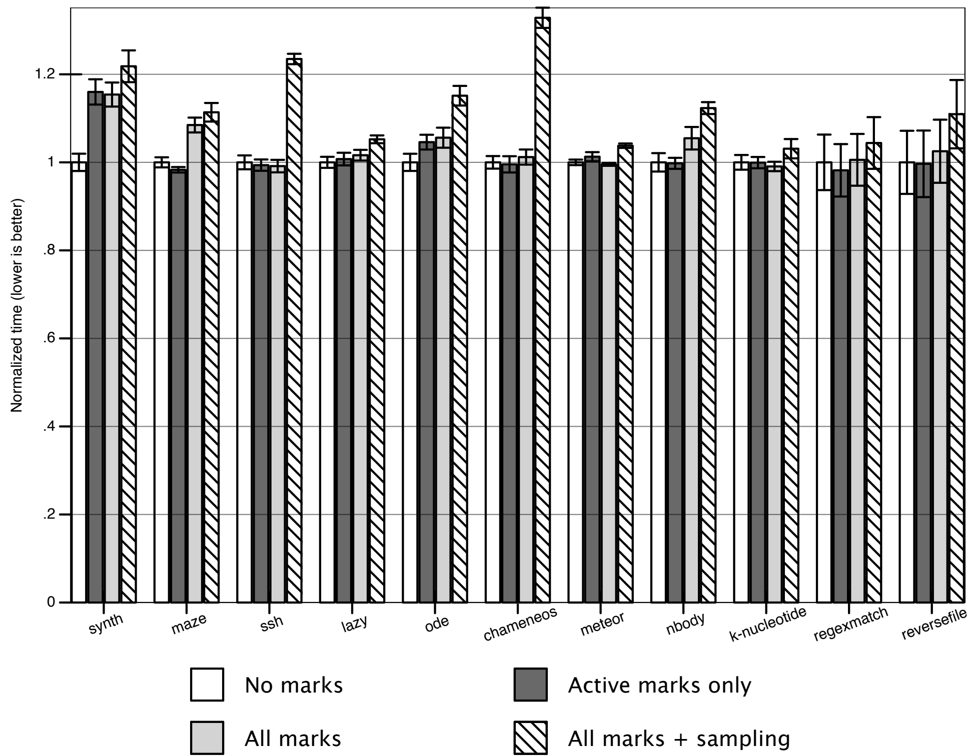


Figure 14: Instrumentation and sampling overhead

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 15: Benchmark descriptions