Feature-Specific Profiling

Vincent St-Amour Matthias Felleisen
PLT @ Northeastern University
{stamourv,matthias}@ccs.neu.edu

Abstract

High-level language and library features provide significant readability and maintainability benefits. Their performance costs, however, are usually not predictable, at least not easily. Programmers may accidentally misuse features and produce underperforming programs as a result.

This paper introduces *feature-specific profiling*, a profiling technique that reports performance costs in terms of language and library features. With a feature-specific profiler, a programmer can identify specific feature instances that are responsible for performance problems. In addition the paper presents the architecture of our feature-specific profiler. At a high level, the profiler consists of two pieces: a generic kernel and feature-specific plug-ins. The API between the two relies on a lightweight form of stack inspection.

1. Weighing Language Features

Many language and library features come with difficult-to-predict performance costs. First, the cost of a feature highly depends on how heavily a program uses it. 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 sometimes depends on its mode of use; higher-order functions that escape their defining scope can be significantly more expensive than those that remain local. In cases where performance matters, programmers must exert care in both regards.

When problems emerge, programmers tend to turn to performance tools such as profilers. Profilers report costs, e.g., time or space costs, in terms of location, which helps programmers focus on the relevant 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 profiling technique that reports performance costs in terms of the time spent in language or library features. Where a traditional program may break down execution time across modules, functions, or lines of code, a feature-specific profiler assigns costs to instances of features—a specific type coercion, a particular contract, or an individual pattern matching form—whose actual costs may be spread across multiple program locations.

Feature-specific profiling provides an additional, complementary view of program performance. In many cases, it makes profiling information truly actionable. Our evaluation points to specific advantages of feature-specific profiling. Because these profilers report costs in terms of specific features, they point programmers towards potential solutions. In many cases, the solution is to use a feature differently or to avoid it in a particular context. Because they assign costs to individual feature instances, feature-specific profilers direct programmers' attention in a manner that is orthogonal to traditional profilers.

In this paper, we

- introduce the first feature-specific profiler,
- explain its architecture and API for library authors,
- and present an evaluation of the actionability of our feature-specific profiler's measurements. We also report on the effort required to implement a feature plug-in.

The rest of this paper is organized as follows. In section 2 we describe the corpus of features our profiler supports. In section 3 we outline the architecture of our framework and provide background on the instrumentation technique we use. In sections 4 and 5 we describe the implementation in more detail. We present results from evaluating our profiler in section 6, and then explain the limitations of our architecture, relate to existing work and conclude.

2. Feature Corpus

Because programmers may not be aware of the costs of various features, and because they may not know what to profile, feature-specific profilers must support a variety of features. This section presents the corpus of feature-specific extensions for our profiler that support features from the Racket language (Flatt and PLT 2010), its standard library and one third party library. We briefly describe each feature and its uses, and outline the information our profiler provides about each of them.

We have identified the first four features below as occasional causes of performance issues in existing Racket programs. The fifth, Marketplace (Garnock-Jones et al. 2014) processes, is a feature that hinders reasoning about performance while not being expensive itself. The remaining features are considered expensive, and are often first on the chopping block when optimizing programs, but our feature-specific profiler does not discover a significant impact on performance in ordinary cases. For these features, a feature-specific profiler can dispel the myths surrounding these features and provide programmers with actionable evidence.

2.1 Contracts

Behavioral software contracts are a linguistic mechanism for expressing and dynamically enforcing specifications. Contracts were introduced in Eiffel (Meyer 1992) and have since spread to a number of platforms including Python, Ruby, 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 contract. This is enforced by checking the relevant contract whenever a value flows from one party to another. If the value satisfies the specification, program execution continues normally. If, on the other hand, the value does not satisfy the contract, an exception is raised.

Programmers can write contracts using the full power of the host programming language, which allows for rich and detailed specifications. This flexibility, however, comes at a cost. Because contracts are checked dynamically, detailed, computation-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), contract checking costs are often non-local. The contract system must defer checking until methods are called or objects are accessed, which may happen at an arbitrary time after crossing the contract boundary.

The non-local nature of these costs complicates reasoning for programmers and tools alike. To predict how often a given contract is going to be checked, programmers must know where the contracted value may flow. Traditional profilers also fall short; they attribute contract costs to the location where contracts are checked, leaving it to programmers to infer that the cost is due to contracts.

Figure 1 shows an excerpt from an HTTP client library. The library provides the `make-fetcher` function, which takes a user agent as argument, and returns a function that performs requests using that user agent. The HTTP client allows requests for URLs only if they are on a whitelist, which it enforces using the underlined contract. The `driver` module creates a crawler that uses a fetching function from the `http-client` module. The crawler then calls this fetching function to access web pages, triggering the contract each time. Because contract checking happens while executing crawler code, a traditional profiler attributes contract checking costs to the `crawl` function. The contract between the `http-client` and `driver` modules is the point where any performance issues must be addressed.

`driver.rkt`

```
(require "http-client.rkt" "crawler.rkt")
(define fetch (make-fetcher "fetcher/1.0"))
(define crawl (make-crawler fetch))
... (crawl "splashcon.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 altogether or use them lightly. Avoiding contracts completely, however, is not always possible. First, the Racket standard library uses contracts pervasively, both to preserve its internal invariants and to 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, the Typed Racket implementation automatically inserts contracts at boundaries between typed and untyped code (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 feature-specific profiler provides several contract-related reports and visualizations; see section 4.2. Two of our case studies demonstrate that programmers can benefit significantly from the contract plug-in.

2.2 Output

Programs that interact heavily with files, the console or the network may spend a significant portion of their running time in the input/output subsystem. In these cases, optimizing the program's computation is of limited usefulness. It is therefore important for programmers to be aware of input/output costs.

Our tool profiles the time programs spend in Racket's output subsystem and traces it back to individual output function call sites. Section 6.1.2 presents the results from the analysis of a maze generation program; output-specific profiling induces a 27% speedup.

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

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

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

Figure 2: Generic and specialized sequence operations

2.3 Generic Sequence Dispatch

Racket's `for` loops operate on any sequence datatype. This includes built-in types such as lists, vectors, hash-tables and strings as well as user-defined datatypes that implement the `sequence` interface. For example, the first loop in figure 2 checks for the existence of all files contained in `source-files`, regardless of which type of sequence it is.

While this genericity encourages code reuse, it introduces extra runtime dispatch. For loops whose body does not perform much work, the overhead from dispatch can dwarf the cost of the loop's actual work.

To alleviate this cost, programmers can manually specialize their code for a specific type of sequence by providing type hints to iterations forms. The second loop in figure 2 uses the `in-list` type hint and is only valid if `source-files` is a list; it throws an exception if `source-files` is, e.g., a vector. This eliminates dispatch overhead, but is not always desirable from a software engineering perspective. Nevertheless, specializing sequences is often one of the first changes experienced Racket programmers perform when optimizing programs.

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

2.4 Type Casts and Assertions

Typed Racket, like other typed languages, provides type casts as an "escape hatch". It can help programmers get around the type system. Figure 3 shows one case where such an escape hatch is necessary. The program reads (what it expects to be) a list of numbers from standard input. Because the user can type in anything they want, the call to `read` is not guaranteed to return a list of numbers, meaning its result must be cast to get around the typechecker.

Like Java's casts, Typed Racket's casts are safe. Runtime checks ensure that the casts's target is of a compatible type, otherwise the cast throws an exception. As a result, casts can have a negative impact on performance. It can be especially problematic for casts to higher-order types that must wrap

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

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

Figure 3: Taking the mean of a user-provided list of numbers

their target, causing extra indirections and imposing an overhead similar to that of higher-order contracts.

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

Our feature-specific profiler reports time spent in each cast and assertion site in the program. None of our case studies reveal any problems with casts and assertions.

2.5 Marketplace Processes

The Marketplace library[1] allows programmers to express concurrent systems functionally as sets of processes grouped within task-specific virtual machines (VMs)[2] that communicate via publish/subscribe. Marketplace is especially suitable for building network services and is used as the basis of an SSH server (discussed in section 6.1.3) and a DNS server.

While organizing processes in a hierarchy of VMs has clear software engineering benefits, deep VM nesting can hinder reasoning about performance. Multiple processes can also execute the same code, but because these processes do not map to either OS or Racket threads, existing profilers may conflate different processes and mis-attribute costs.

Our feature-specific profiler addresses both these issues and provides process accounting for Marketplace VMs and processes. It 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. Our case study of a Marketplace program suggests a reorganization that improves performance.

2.6 Pattern Matching

Like Scala and Haskell, Racket supports pattern matching, which allows for concise conditional and destructuring idioms. Racket's pattern matcher (Tobin-Hochstadt 2011) is also extensible; users can write pattern-matching plug-ins to implement, e.g., custom views on data structures or new types of binding patterns.

[1] `https://github.com/tonyg/marketplace`

[2] Despite the name, these virtual machines are process containers running within a Racket OS-level process. Their relationship with their more heavy-weight cousins such as VirtualBox, or the JVM, is one of analogy only.

Racket programmers often worry that pattern matching introduces more predicate tests and control transfers than handwritten code would, resulting in less efficient programs. In reality, Racket's pattern matching library uses a sophisticated pattern compiler that normally generates highly efficient code. However, some features of the pattern matching library, such as manual backtracking, carry additional costs and must be used with caution.

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

2.7 Method Dispatch

While Racket is the descendant of a mostly-functional language, it fully supports class-based object-oriented programming with first-class classes (Flatt et al. 2006). In comparison to function calls, though, method calls have a reputation for being expensive.

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

Our tool profiles the time spent performing method dispatch for each method call site, reporting the rare cases where dispatch costs are significant. Our evaluation does not cover generic dispatch. We only include it as an example.

2.8 Optional and Keyword Argument Functions

Like many languages, Racket offers functions that support optional positional arguments as well as keyword-based non-positional arguments. These features allow for transparently extending APIs without breaking backwards compatibility, and for writing highly configurable functions.[3]

Figure 4 provides two examples of API extension. The `member?` function—which checks whether a value is contained in a list—optionally takes an equality function for list elements. Racket's `sort` function accepts an optional `#:key` keyword argument, which specifies the part of the list elements that should be used for comparison.

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

The reality is less bleak than imagined. The special protocol is only necessary when calling "unknown"[4] functions with optional or keyword argument. In all other cases, uses of the protocol can be eliminated statically. Unknown uses account for only a small portion of all uses of optional and keyword argument functions, and the special protocol is therefore rarely a problem in practice.

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


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

Figure 4: Example use of optional and keyword arguments

To inform programmers about the true cost of optional and keyword argument functions, our feature-specific profiler reports time spent performing the special function call protocol for individual functions. While our case studies use functions with keyword arguments, the profiler reports no problem with their use.

3. The Profiler's Architecture

Our feature-specific profiler consists of two parts: the core and feature-specific plug-ins. The core is a sampling profiler with an API that empowers the implementors of languages and the authors of libraries to create plug-ins for specific features. Client programmers can then benefit from feature-specific profiling.

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 processes the collected stack information, looks for feature-specific stack markers, and produces programmer-facing reports.

By implication, the feature-specific plug-ins must place relatively unique markers on the control stack. 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 API for lightweight stack inspection. Concretely, it uses continuation marks (Clements et al. 2001). Since this API differs from stack inspection protocols in other languages, the first subsection briefly introduces the necessary technical background. The second subsection explains how the implementor of a language or the author of a library uses continuation marks to interact with the profiler

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

[4] Unknown functions are typically functions used in a higher-order way, or that are bound to mutated variables.

framework for structurally simple features. The last subsection presents the off-line analysis. The following sections deal with profiling structure-rich features and tailoring the instrumentation to specific needs.

3.1 Inspecting the Stack with Continuation Marks

Continuation marks allow user code to programmatically attach key-value pairs to stack frames and retrieve them at a later time. Concretely, Racket's continuation marks come with two main operations. With

```
(with-continuation-mark key value expr)
```

a program attaches the pair (*key*, *value*) to the current stack frame before it evaluates *expr*. With

```
(continuation-marks thread)
```

a program walks the stack and retrieves all key-value pairs. Optionally, `continuation-marks` accepts a *thread* argument that specifies which thread's stack should be walked. This option allows one thread to query the continuation marks of another. Programs can also filter marks to consider only those with relevant keys:

```
(continuation-mark-set->list marks key)
```

This utility function takes a continuation mark set returned by `continuation-marks` and a key, and returns the list of values from marks with that key. Outside of these operations, continuation marks do not affect programs' semantics.[5]

Figure 5 illustrates the working of continuation marks with a function that traverses binary trees and records paths from the roots to the 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.

Continuation marks are extensively used in the Racket ecosystem, notably for the generation of error messages in the DrRacket IDE (Findler et al. 2002), a stepping debugger (Clements et al. 2001) as well as the standard DrRacket debugger, for thread-local dynamic binding (Dybvig 2009), and for exception handling. Continuation marks have also been used to implement serializable continuations in the PLT web server (McCarthy 2010).

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

[5] Continuation marks also preserve proper tail call behavior, though this property is only peripherally relevant in this context.

```
; a tree is a number or a list (number tree tree)
(define (paths tree)
  (if (number? tree)
      (cons tree
            (continuation-mark-set->list
             (continuation-marks (current-thread))
             'paths))
      (with-continuation-mark
        'paths
        (first tree)
        (list (paths (second tree))
              (paths (third  tree))))))


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

Figure 5: Recording paths in a tree with continuation marks

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.

Marking To implement profiling support for a feature, a plug-in writer must change its implementation so that specific instances mark themselves with *feature marks*. Roughly speaking, it suffices to wrap the relevant code in `with-continuation-mark`. Once placed, feature marks allow the profiler to observe whether a thread is currently executing code related to a given feature.

Figure 6 shows an excerpt from the instrumentation of Typed Racket assertions. The (underlined) conditional inside the `with-continuation-mark` form is responsible for performing the actual assertion. The feature mark's key should uniquely identify the feature, to avoid false positives when retrieving the marks. In this case, we use the symbol `'TR-assertion` as key.

By using distinct keys for each feature, plug-ins compose naturally. This allows our feature-specific profiler to present a unified report to users, as well as not requiring them to select in advance the feature they expect to be 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 6, the payload is the source location of a specific assertion in the program, which allows the profiler to compute the cost of each assertion individually.

Writing such plug-ins, while simple and non-intrusive, requires access to the implementation of the feature of interest to find the relevant code and instrument it. Because doing so does not require any specialized profiling knowledge, it is well within the reach of language and library authors. For details see section 6.2.

Antimarks Features are seldom "leaves" in a program; feature code usually contains or calls code from the use site. The execution time of that user-provided code should not

```
(define-syntax (assert stx)
  (syntax-case stx ()
    [(assert v p)
     ; the compiler rewrites the above to:
     (quasisyntax
       ; evaluate use-site code once
       (let ([val  v]
             [pred p])
         ; mark feature code
         (with-continuation-mark
           'TR-assertion
           (unsyntax (source-location stx))
           (if (pred val)
               val
               (error "Assertion failed."))))])))
```

Figure 6: Instrumentation of assertions (excerpt)

```
(define-syntax (lambda/keyword stx)
  (syntax-case stx ()
    [(lambda/keyword formals body)
     ; the compiler rewrites the above 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 7: Use of antimarks in instrumentation

count towards the time spent in the feature itself. For example, when profiling the function call protocol for keyword arguments, the profiler must not count the time spent in the function body towards the protocol.

To solve this problem, our profiler expects *antimarks* on the stack. Such antimarks are continuation marks with a distinguished value that delimit the end of feature code. Our protocol dictates that the continuation mark key used by an antimark is the same as that of the feature it delimits. Figure 7 shows the use of `'antimark` symbols as values on a continuation mark. This code instruments a simplified version of the 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 the most recent mark is an antimark, the program is currently executing user code, which should not be counted.

Sampling During program execution, our profiler spawns a sampling thread that runs concurrently alongside the program. The sampling thread periodically collects continuation marks from the main thread using `continuation-marks` and collects them. 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 profiler processes the data collected by the sampling thread to produce a feature cost report.

Cost assignment The profiler assigns a time cost to each sample based on the elapsed time between it and its predecessor and its successor. We use a standard sliding window technique for the cost assignment algorithm.

Antimark filtering Only samples with a feature mark as the most recent mark contribute time towards features. For each feature, our accounting scheme ignores samples that begin with a corresponding antimark.

Payload grouping As explained in section 3.2, payloads identify individual feature instances. Our accounting algorithm groups samples according to 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 each individual feature report, our feature-specific profiler combines them into a unified report for programmer consumption. Features not present in the program or those inexpensive enough to never be sampled are pruned from the profile to avoid clutter. To help users focus on high-priority targets, features are sorted in descending order of cost, as are feature instances within each feature report.

Figure 8 shows a feature profile for a Racket implementation of the FizzBuzz[6] program with an input of 10,000,000. Predictably, most of the execution time is spent printing numbers that are not divisible by either 3 or 5 (line 15), since that includes most numbers. Just under a second is spent in generic sequence dispatch; the `range` function produces a list, but the `for` iteration is not specialized for lists.

4. Profiling Rich Features

The preceding section presents a basic architecture for feature-specific profilers. Structure-rich features, however, come with more information than the source location of instances. For the programmer's benefit, a feature-specific profiler should exploit that structure to generate detailed and actionable reports.

This section shows how to adapt our basic architecture to profile structure-rich features fruitfully. It uses two example features to demonstrate the idea: contracts and Mar-

[6] `http://imranontech.com/2007/01/24/using-fizzbuzz-to-find-developers-who-grok-coding/`

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

```
Output
accounts for 68.22% of total running time
5580 / 8180 ms

Cost Breakdown
  4628 ms : examples/fizzbuzz.rkt:15:28
  564 ms : examples/fizzbuzz.rkt:14:28
  232 ms : examples/fizzbuzz.rkt:13:28
  156 ms : examples/fizzbuzz.rkt:12:28


Generic Sequences
account for 11.78% of total running time
964 / 8180 ms

Cost Breakdown
  964 ms : examples/fizzbuzz.rkt:11:11
```

Figure 8: Feature profile for the infamous FizzBuzz program

ketplace processes. These adapted strategies require a customized analysis for each feature. Naturally, library authors can implement basic plug-ins for structure-rich features and evolve them into rich plug-ins if and when needed. We discuss an instance of this in section 6.2.

4.1 Custom Payloads

The instrumentation for structure-rich features takes advantage of the full power of continuation marks. Specifically, it exploits the power of associating arbitrary values with keys.

Contracts Our contract plug-in uses *blame objects* as payloads. Roughly speaking, a blame object holds enough information—the contract to check, the name of the contracted value and the names of the components that agreed to the contract—to reconstruct a complete picture of contract checking events. The contract system uses blame objects to explain contract violations and to pinpoint the faulty party.

Marketplace processes The Marketplace plug-in uses process names as payloads. Since `continuation-marks` collects all the marks currently on the stack, the sampling thread can gather "core samples".[7] Because Marketplace VMs are spawned and transfer control using regular function calls, these core samples include not only the current process but also all its ancestors—its parent VM, that VM's parent, etc.

While storing arbitrary data as payload is attractive, plug-in writers must avoid excessive computation or allocation when constructing payloads. Even though the profiler uses sampling to record marks, payloads must still be constructed every time feature code is executed, whether or not the sampler observes it.

To reduce overhead, we restrict payloads used by our plug-ins to constants (such as source locations) or values already computed by the feature (blame objects and process names). This guideline is not enforced by the framework.

[7] We borrow the "core sample" terminology from geology. Here it denotes a sample that includes continuation marks from the entire stack.

4.2 Analyzing Rich Features

Programmers usually cannot directly digest profiling information generated via custom payloads. A further analysis pass, separate from the one described in section 3.3, is needed to generate user-facing reports.

Contracts Profiling contracts is all about measuring which pairs of parties impose contract checking, and how much these cost. A programmer can act only after identifying the relevant components. Contract-specific analysis therefore aims to provide an at-a-glance overview of the cost of each contract boundary, in addition to reporting the costs of individual contracts.

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 for each group as discussed in section 3.3.

Figure 9 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 Racket's `math` library reside in a typed module. Hence linking these modules introduces a contract boundary between the client and the library.

In addition to the module graph, our feature-specific profiler provides other views as well. For example, figure 10 shows the *by-function* view, which provides fine-grained information about the cost of individual contracted functions, including which contract was applied to the function.

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 exe-

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

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

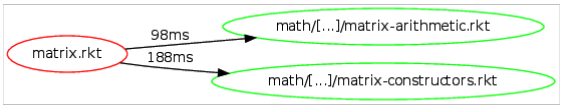

Figure 9: Module graph view of a contract boundary

```
Contracts
account for 47.35% of total running time
286 / 604 ms

Cost Breakdown
  188 ms : build-matrix
    (-> Integer Integer (-> any/c any/c any/c) Array/c)
  88 ms : matrix-multiply-data
    (-> Array/c Array/c
        (values any/c any/c any/c any/c any/c
                (-> any)))
  10 ms : make-matrix-multiply
    (-> Index Index Index (-> any/c any/c any/c)
        Array/c)
```

Figure 10: By-function view for matrix multiplication

```
=======================================================
                Caller
 Total  Self     Name                            Local%
 Time   Time     Callee
=======================================================
100.0%  32.3%  ground
                (tcp-listener 5999 ::1 53588)    33.7%
                (tcp-listener 5999 incoming)     10.5%
                tcp-driver                        9.6%
                6                                 9.0%
                (tcp-listener 5999 ::1 53587)     2.6%
                #:boot-process                    2.2%
-------------------------------------------------------
                ground
33.7%   33.7%  (tcp-listener 5999 ::1 53588)
-------------------------------------------------------
[...]
-------------------------------------------------------
                ground
2.6%    2.6%   (tcp-listener 5999 ::1 53587)
-------------------------------------------------------
[...]
```

Process identifiers are cut to their stem for space reasons.

Figure 11: Marketplace process accounting (excerpt)

cute. Marketplace feature marks use process and VM names as payloads, which allows our profiler 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. 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 take advantage of that similarity and reuse standard edge profiling techniques[8] to attribute costs to the entire ancestry of a process. To disambiguate between similar processes in its reports, the profiler uses processes' full ancestry as identifers.

Figure 11 shows the accounting from a Marketplace-based echo server. The first line of the profile shows the ground VM, which is the VM that 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: one on port 53587 and the other on port 53588. The latter is sending ten times as much input as the former, as reflected in the profile.

Our feature-specific profiler 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.[9]

5. Instrumentation Control

The plug-ins described so far insert feature marks and antimarks regardless of whether a programmer wishes to profile a program or not. We refer to marks that are present in programs as *active marks*. 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 executing individually inexpensive instances is significant, the overhead of marks can be problematic. For this reason, programmers want to control when marks are applied on an execution-by-execution basis.

In addition, programmers may also want control over where mark insertion takes place to avoid reporting costs in code that they cannot modify. For instance, reporting that some function in the standard library performs a lot of pat-

[8] 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.

[9] The echo server performs little actual work which, by comparison, increases the library's overhead. Marketplace's overhead is usually low.

tern matching is useless to most programmers. In general, the profiler should consider standard library functions as indivisible units; reports about their internals are a distraction.

To establish control over the *where* and *when* of continuation marks, our profiling framework introduces the notion of *latent marks*. A latent mark is an annotation that, on demand, can be turned into an actual 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 features implemented as library or runtime functions.

5.1 Syntactic Latent Marks

Syntactic latent marks exist as annotations on the intermediate representation of user code. When the meta-program for a feature processes user code, it leaves tags[10] on the residual program's intermediate representation instead of directly inserting feature marks and antimarks. These tags have no runtime effect on the program; they are discarded after compilation. Other meta-programs or the compiler's front end can observe latent marks and can then turn them into active ones.

Our implementation uses Racket's *compilation handler* mechanism to interpose the activation pass between macro-expansion and the compiler's front end. Programmers can use a command-line flag to enable this compilation handler when compiling their programs for profiling. The activation pass then traverses the input program, replacing any syntactic latent mark it finds with an active mark.

Because the latent marks are implicitly present in programmers' 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 of Racket's heavy reliance on syntactic extension, most of our feature-specific plug-ins use syntactic latent marks.

5.2 Functional Latent Marks

Functional latent marks offer an alternative to syntactic latent marks. Instead of tagging the programmer's code, a separate preprocessor or compiler pass recognizes calls to feature-related functions, and rewrites the programmer's code to wrap such calls in active marks. In our implementation, this marking pass is combined with the syntactic latent mark activation pass, but it is conceptually separate.

Because functional latent marks require the use of a separate activation pass, activating those marks requires recompiling user code when profiling. It does not, however, require recompiling libraries that provide feature-related functions, which makes functional latent marks appropriate for functions provided as runtime primitives.

Our plug-in for output profiling uses functional latent marks. It 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.

[10] We use Racket's syntax property mechanism, but any intermediate representation tagging mechanism would apply.

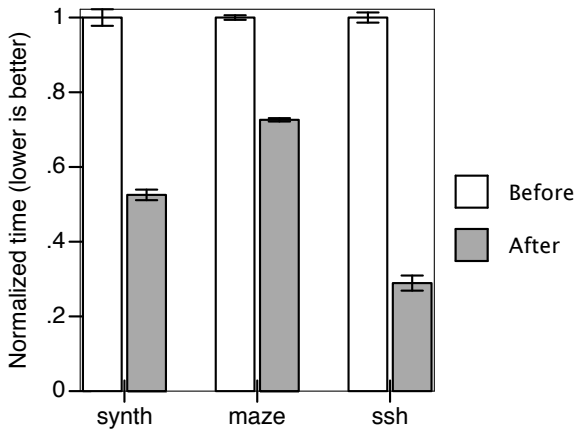

Figure 12: Speedups after profiling and improvements

6. Evaluation

Our evaluation of feature-specific profiling covers two criteria. First, in order to validate that feature-specific profilers accurately identify performance issues and point to potential solutions, we present case studies that demonstrate how feature-specific profiling improves the performance of programs. Second, to support our claims regarding the ease of implementing plug-ins for the profiler, we report on the amount of effort required to implement each of the plug-ins listed in section 2.

6.1 Case Studies

To be useful, a feature-specific profiler must accurately identify features and feature instances responsible for significant overhead in a given program. Furthermore, a feature-specific profiler must provide *actionable* information, that is, its reports must point programmers towards solutions. Conversely, it must also provide *negative* information, i.e., features that do not have a significant cost.

We present three case studies of programs suffering from the overhead of specific features. Each subsection briefly describes each of these programs and its architecture, then presents the feature-specific profiler's report. It then explains the changes we made to the program in response; the case studies include only changes that directly follow from the profiler's report. Finally, figure 12 presents the results of comparing execution times before and after the changes. These results are the mean of 30 executions on a 64-bit Debian GNU/Linux system with a 6-core Intel Xeon E5645 processor and 12GB of RAM, and include error bars one standard deviation above and below the mean.

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

Figure 13: Traditional profile for the synthesizer (excerpt)

```
Contracts
account for 73.77% of total running time
17568 / 23816 ms

Cost Breakdown
  6210 ms : Array-unsafe-proc
    (-> Array/c (-> (vectorof Index) any))
  3110 ms : array-append*
    (->* ((listof Array/c)) (Integer) Array/c)
  2776 ms : unsafe-build-array
    (-> (vectorof Index) (-> (vectorof Index) any/c)
        Array/c)
  2594 ms : array-broadcast
    (-> Array/c (vectorof Index) Array/c)
  1342 ms : build-array
    (-> (vectorof Integer) (-> (vectorof Index) any/c)
        Array/c)
  [...]


Generic Sequences
account for 0.04% of total running time
10 / 23816 ms

Cost Breakdown
  10 ms : wav-encode.rkt:51:16
```

Figure 14: Feature profile for the synthesizer (excerpt)

6.1.1 Sound Synthesis Engine

Our first case study is a sound synthesis engine due to the first author. The engine uses arrays provided by Racket's `math` library to represent sound signals. It consists of a `mixer` module that handles most of the interaction with the `math` library and is responsible for mixing multiple signals, as well as a number of specialized synthesis modules that interface with the mixer, such as function generators, sequencers, and a drum machine.

The `math` library uses Typed Racket, whereas the synthesis engine is in untyped Racket. Hence a contract boundary separates the two components. For scale, the synthesis engine spans 452 lines of Racket code. To profile the synthesis engine, the program is run to synthesize ten seconds of music[11] involving a lead track and a drum track.

[11] The opening of Lipps Inc.'s 1980 disco hit Funkytown, specifically.

Figure 13 shows an excerpt from the output of Racket's traditional statistical profiler, listing the functions in which the program spends most of its time. Two of those, accounting for around 40% of total execution time, are in the `math` library. Such profiling results suggest two potential improvements: optimizing the `math` library or avoiding it altogether. Either solution would be a significant undertaking.

Figure 14 shows a different view of the same execution, provided by our feature-specific profiler. According to the feature profile, almost three quarters of the program's execution time is spent checking contracts. The most expensive of these contracts are attached to the `math` library's array functions. Consequently, any significant performance improvements must come from reducing the use of those generated contracts. Optimizing the `math` library itself, as hinted at by the original profile, may not yield an improvement.

Since the `math` library's contracts are automatically generated by Typed Racket, optimizing them directly is not a practical solution. Reducing the use of contracts is more likely to be profitable. Because these contracts are checked at the boundary between typed and untyped code, moving code across the boundary to reduce the frequency of crossings may improve matters.

For this, a programmer turns to the module graph view in figure 15 provided by our feature-specific analysis for contracts. According to this view, almost half the total execution time lies between the untyped interface to the `math` library used by the `mixer` module (in red) and the typed portions of the library (in green). This suggests a conversion of the `mixer` module to Typed Racket. This conversion, a 15-minute effort, improves the performance by around 48%.

Figure 14 also shows that generic sequence operations, while often expensive, do not impose a significant cost in this program despite being used pervasively. Manually specializing sequences would be a waste of time. Similarly, since the report does not feature file output costs, optimizing how the generated signal is emitted as a WAVE file—which is currently done naively, one sample at a time—would also be a waste of time.

6.1.2 Maze Generator

Our second case study employs a Typed Racket version of a maze generator, originally written by Olin Shivers. For scale, the maze generator is 758 lines of Racket code. The program first generates a maze on a hexagonal grid, then checks whether it is solvable, and finally prints it.

Figure 16 shows the top portion of the output of our feature-specific profiler. Three locations, corresponding to three calls to `display`, stand out as especially expensive. Each of those is responsible for printing part of the bottom of hexagons. Printing each part separately results in a large number of output operations, each for a single character. Based on that report, fusing all three output operations into one should result in an improvement. And indeed, this

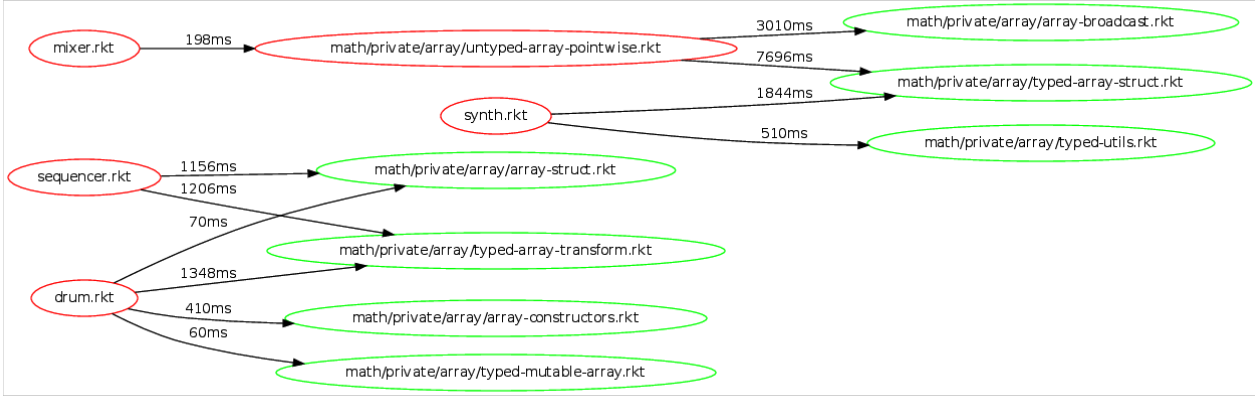

Figure 15: Module graph view of the synthesizer

```
Output
accounts for 55.31% of total running time
1646 / 2976 ms

Cost Breakdown
  386 ms : maze.rkt:730:2
  366 ms : maze.rkt:731:2
  290 ms : maze.rkt:732:2
  [...]
```

Figure 16: Feature profile for the maze generator (excerpt)

change reduces the number of calls to the runtime system and results in a 27% speedup.

As part of this programs's Typed Racket translation, a dynamic type assertion enforces an invariant that the type system cannot guarantee statically. This assertion ends up in the inner loop of one of the set operations used heavily during maze generation. Even though this might raise concerns with a cost-conscious programmer, the feature profile reports that the time spent in the cast is too small to observe.

6.1.3 Marketplace-Based SSH Server

Our last case study involves an SSH server[12] due to Tony Garnock-Jones based on the Marketplace library (Garnock-Jones et al. 2014). For scale, the SSH server itself is 3,762 lines of untyped Racket code, while the Marketplace library is 4,801 lines of Typed Racket code. To exercise the server, a driver script connects to the SSH server, 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 17 shows, our feature-specific profiler reports two interesting 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 do not contribute directly to protocols; they observe other processes for logging purposes. The SSH server spawns these two spy processes even when logging is ignored, resulting in unnecessary overhead.

The second observation is that contracts between the typed Marketplace library and the untyped SSH server account for close to 67% of the running time. This overhead can be removed in one of two ways: by (gradually) adding types to the SSH server or by disabling typechecking in the Marketplace library.

Disabling spy processes and type-induced contracts results in a speedup of around 71%. A post-mortem reveals that disabling contracts creates larger savings than disabling spy processes.

In addition to these two areas of improvement, the feature profile also provides two pieces of negative information: pattern matching and generic sequences, despite being used pervasively, account for only a small fraction of the server's running time.

6.2 Plug-in Implementation Effort

Writing feature-specific plug-ins is a low-effort endeavor. Indeed, it is easily within reach for library authors mostly because it does not require advanced profiling knowledge. To support this claim, we provide anecdotal evidence from observing Tony Garnock-Jones, the author of the Marketplace library, implement feature-specific profiling support for Marketplace.

Mr. Garnock-Jones implemented the plug-in himself, with the first author acting as interactive documentation of the framework. Implementing the first version of the Marketplace profiler, which performed no feature-specific analysis, took about 35 minutes. At that point, Mr. Garnock-Jones had a working Marketplace process profiler that performed

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

```
Marketplace Processes

=====================================================
                          Caller
 Total  Self               Name                Local%
 Time   Time              Callee
=====================================================
100.0%  3.8%   ground
                 ssh-session-vm                 51.2%
                 tcp-spy                        19.9%
                 (tcp-listener 2322 ::1 44523)  19.4%
                 ssh-tcp-listener                3.6%
                 tcp-driver                      2.2%
-----------------------------------------------------
                 ground
51.2%   1.0%   ssh-session-vm
                 ssh-reader                     35.1%
                 ssh-session                    31.0%
                 (#:boot-process ssh-session-vm) 14.1%
                 ssh-writer                     14.0%
                 ssh-application-vm              3.8%
-----------------------------------------------------
                 ground
19.9%   19.9%  tcp-spy
-----------------------------------------------------
[...]
-----------------------------------------------------
                 ssh-session-vm
7.2%    7.2%   (#:boot-process ssh-session-vm)
-----------------------------------------------------
[...]


Contracts
account for 66.93% of total running time
3874 / 5788 ms

Cost Breakdown
  1496 ms : add-endpoint
    (-> pre-eid? role? (-> any/c
                           (-> any/c transition/c))
        add-endpoint/c)
  1122 ms : process-spec
    (-> (-> any/c (parametric->/c (Result61) ...)) any)
  700 ms  : transition
    (->* (any/c) #:rest (listof g16901350273)
         transition/c)
  [...]


Pattern Matching
accounts for 0.76% of total running time
44 / 5788 ms

Cost Breakdown
  26 ms : marketplace/sugar-values.rkt:72:11
  10 ms : unknown location
  8 ms  : marketplace/process.rkt:62:2


Generic Sequences
account for 0.35% of total running time
20 / 5788 ms

Cost Breakdown
  20 ms : marketplace/struct-map.rkt:36:20
```

Figure 17: Profiling results for the SSH server (excerpt)

Feature	Instr. LOC	Analysis LOC
Contracts	183	672
Output	11	-
Generic sequences	18	-
Type casts and assertions	37	-
Marketplace	7	515
Pattern matching	18	-
Method dispatch	12	-
Opt. and keyword args	50	-

Figure 18: Instrumentation and analysis LOC per feature

```
(define marketplace-continuation-mark-key
  (make-continuation-mark-key 'marketplace))

[...]

(marketplace-log 'debug "Entering process ~v(~v)"
                 debug-name pid)
(define result (with-continuation-mark
                 marketplace-continuation-mark-key
                 (or debug-name pid)
                 enclosed-expr))
(marketplace-log 'debug "Leaving  process ~v(~v)"
                 debug-name pid)
```

Figure 19: Instrumentation for Marketplace

basic analysis as described in section 3.3. Specializing the feature-specific analysis took an additional 40 minutes.

Mr. Garnock-Jones is an experienced Racket programmer and is the author of the Marketplace library. 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 18. The third column reports the number of lines of domain-specific analysis code. Basic analysis is provided as part of the profiler's core. The line count for Marketplace analysis includes the portions of Racket's edge profiler that are re-linked into the plug-in, which account for 506 of the 515 lines.

For illustrative purposes, the instrumentation for Marketplace is shown in figure 19 with the added code underlined. 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.

With the exception of contract instrumentation—which covers the implementation of multiple kinds of contracts, and is spread across the 16,421 lines of the contract system—instrumentation is always localized and mostly non-intrusive.

7. Limitations

Feature-specific profiling as presented here applies only to certain kinds of language constructs and library exports. This section describes cases where feature-specific profiling could be desirable but where our architecture does not apply.

Control features Because our instrumentation strategy relies on continuation marks, it does not support features that interfere with marks. This rules out control features that unroll part of the stack, because it discards continuation marks along the way. For example, we could not instrument exception raising. Instrumentation techniques that can operate in the presence of control effects are future work.

Uninterruptible features Because our core profiler relies on sampling, features must be interruptible. That is, a candidate feature must allow the sampling thread to be scheduled during its execution. Otherwise, the sampling thread cannot observe the feature's execution.

Library features do not typically suffer from this limitation, but some language features implemented as part of the runtime system do. For example, Racket's struct allocation and random-number generation are two such features, and both have a reputation for being expensive.

8. Related Work

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

8.1 Traditional Profilers

Profilers (Graham et al. 1982) are probably the most well known class of performance tools and have been used successfully to diagnose performance issues for decades. They most commonly report time costs but some also report space costs (Sansom and Peyton Jones 1995) and others report different kinds of information, such as I/O costs.

Traditional profilers group costs according to program organization. On occasion this means static organization, at others dynamic organization, e.g. per HTTP request. Feature-specific profilers, in contrast, group costs according to features and 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. As discussed in section 6.1.1, traditional profilers may not provide actionable information in such cases. Furthermore, by reporting costly features, feature-specific profilers point programmers towards potential solutions that involve correcting feature usage and modes of use. By comparison, traditional profilers often report costs without helping programmers discover solutions.

Traditional profilers can detect a broader range of performance issues than feature-specific profilers. In particular, bottlenecks due to inefficient algorithms would be reported by traditional profilers, while they are invisible to feature-specific profilers.

8.2 Vertical Profiling

Vertical profilers (Hauswirth et al. 2004) render the use of high-level language features transparent when profiling. To "see through" those features, a vertical profiler gathers information from multiple layers—hardware performance counters, the operating system, the virtual machine, libraries—and correlates them into a gestalt of program performance.

Vertical profiling focuses on helping programmers understand how the interaction of high levels and low levels in a system affects program performance. By comparison, feature-specific profiling focuses on helping programmers understand the cost of language and library features themselves. 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 in turn requires a higher degree of knowledge to understand.

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 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. We briefly mention recent pieces of relevant work that provide alternative profiling views.

Singer and Kirkham (2008) use the notion of concepts—programmer-annotated program portions that are responsible for a given task—to assign costs when profiling programs. Listener latency profiling (Jovic and Hauswirth 2011) helps programmers tune interactive applications by reporting high-latency operations, as opposed to operations with long execution times. Tamayo et al. (2012) present a tool that provide programmers with information on the cost of database operations in their programs and helps them optimize multi-tier applications.

8.4 Dynamic Instrumentation Frameworks

Dynamic instrumentation frameworks such as ATOM (Srivastava and Eustace 1994), 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. They could potentially be used to build feature-specific profilers in languages that do not directly support stack inspection. These frameworks are much

more heavy-weight than continuation marks, and in turn allow much more thorough instrumentation—memory hierarchy, hardware performance counters, etc.

Since these frameworks operate at the binary level, tools that use them provide potentially opaque information to non-experts. By comparison, feature-specific profilers specifically aim to provide information that is also accessible to non-experts.

9. Conclusion

This paper introduces feature-specific profiling, a profiling technique that reports program costs in terms of language and library features. In addition, it presents an architecture for feature-specific profilers that allows library authors to implement profiling plug-ins for features provided by their libraries with minimal effort.

The alternative view on program performance offered by feature-specific profilers allows easy diagnosis of performance issues due to feature misuses, including those with non-local costs that would 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 process profiler and helped us perform the SSH server case study. Robby Findler provided assistance with working on the contract system. Finally, we thank Eli Barzilay, Christos Dimoulas, Matthew Flatt, Asumu Takikawa and Sam Tobin-Hochstadt for helpful discussions about various aspects of this work.

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.

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

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, Robert Bruce Findler, and Matthias Felleisen. Scheme with classes, mixins, and traits. In *Proc. APLAS*, pp. 270–289, 2006.

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.

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

Matthias Hauswirth, Peter F. Sweeney, Amer Diwan, and Michael Hind. Vertical profiling: understanding the behavior of object-oriented applications. In *Proc. OOPSLA*, pp. 251–269, 2004.

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

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

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.

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

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.

Patrick M. Sansom and Simon L. Peyton Jones. Time and space profiling for non-strict, higher-order functional languages. In *Proc. POPL*, pp. 355–366, 1995.

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

Amitabh Srivastava and Alan Eustace. ATOM: a system for building customized program analysis tools. In *Proc. PLDI*, pp. 196–205, 1994.

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: run-time support for reasonable interposition. 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. Extensible pattern matching in an extensible language. `arXiv:1106.2578 [cs.PL]`, 2011.

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

A. Instrumentation Overhead

Our feature-specific profiler imposes an acceptably low overhead on program execution. For a summary, see figure 20 which reports preliminary overhead measurements. These results are the mean of 30 executions on a 64-bit Debian GNU/Linux system[13] and include error bars one standard deviation above and below the mean.

We use the programs listed on figure 21 as benchmarks. They include the three 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[14] that use the features supported by our prototype.

The first column of figure 20 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, and is the mode used 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 that 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[15] distributed across a program's execution and should not introduce such biases.

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

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

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

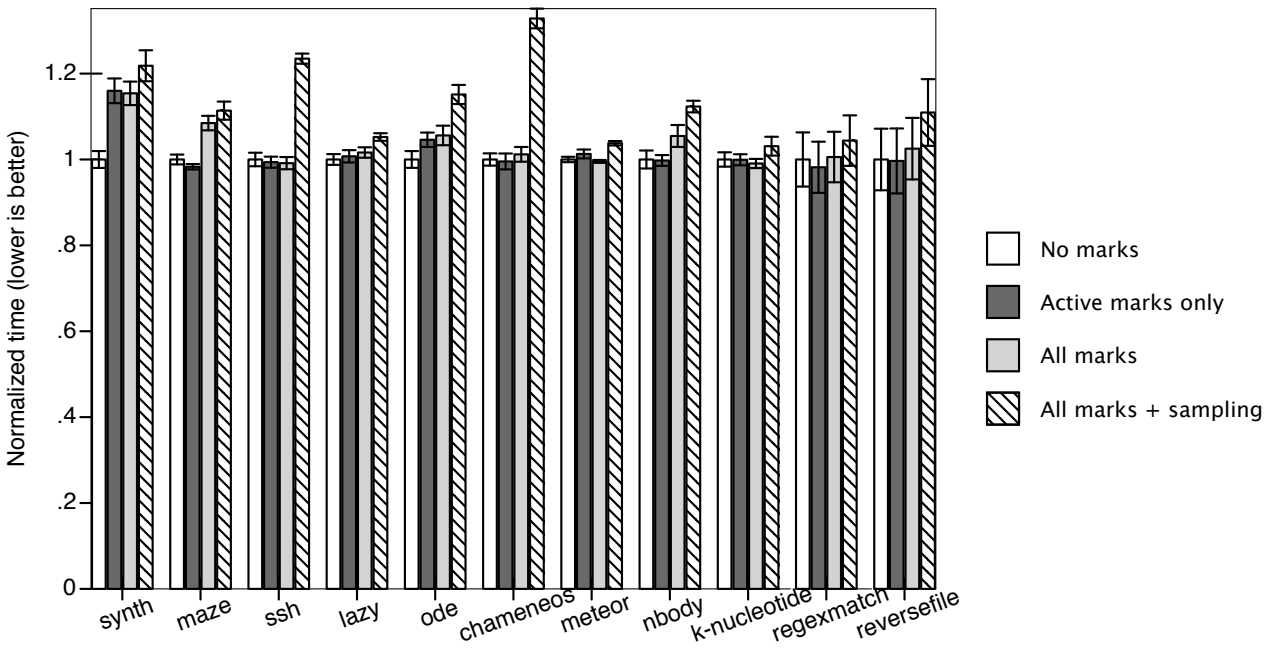

Figure 20: Instrumentation and sampling overhead

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 21: Benchmark descriptions