Profile-Guided Meta-Programming

William J. Bowman *
Northeastern University and
Cisco Systems, Inc
wjb@williamjbowman.com

Swaha Miller †
Cisco Systems, Inc
swaham@vmware.com

Vincent St-Amour
Northeastern University
stamourv@ccs.neu.edu

R. Kent Dybvig
Cisco Systems, Inc
dyb@cisco.com

Abstract

Modern compilers systems such as GCC, .NET, and LLVM incorporate profile-guided optimizations (PGOs) on low-level intermediate code and basic blocks, with impressive results over purely static heuristics. Recent work shows that profile information is also useful for performing source-to-source optimizations via meta-programming. For example, using profiling information to inform decisions about data structures and algorithms can potentially lead to asymptotic improvements in performance.

We present a design for supporting profile-guided meta-programs in a general-purpose meta-programming system. Our design is parametric over the particular profiling and meta-programming systems. We implement this design in two different meta-programming systems—the syntactic extensions systems of Chez Scheme and Racket—and provide several profile-guided meta-programs as usability case studies.

1. Introduction

Profile-guided optimization (PGO) is an optimization technique in which a compiler uses profile information gathered at runtime to improve the performance of the generated code. The profile information acts as an oracle for runtime behavior. For example, a profiler might gather information about how many times each function in a program is called to inform decisions about function inlining. Compilers use profile information to guide decisions about reordering basic blocks, function inlining, reordering conditional branches, and function layout in memory (Gupta et al. 2002). Modern compilers systems that support PGO include .NET, GCC, and LLVM (Lattner 2002). Code generated using PGOs usually exhibits improved performance, at least on the represented class of inputs, compared to code generated with static optimization heuristics. For example, Arnold et al. (2000) show that using profiling information to guide inlining decisions in Java resulted in up to 59% improvement over static heuristics.

Profile information has also proven useful to implement profile-guided meta-programs. Meta-programs are programs that operate on programs. Languages with general-purpose meta-programming systems include C, C++, Java (Erdweg et al. 2011), ML (Taha and Sheard 2000), OCaml (Kiselyov 2014), Racket (Flatt and PLT 2010), Scheme (Dybvig et al. 1993), Scala (Burmako 2013), and Template Haskell (Sheard and Jones 2002). Meta-programming is used to implement high-level yet efficient abstractions. Boost libraries (Dawes and Abrahams 2009) make heavy use of C++ meta-programming. Sujeeth et al. (2014) and Rompf and Odersky (2010) implement high-performance domain specific languages using staged meta-programming in Scala. Chen et al. (2006a) implement process placement for SMP clusters using profile-guided meta-programming. Liu and Rus (2009) provide tools that use profile information to identify suboptimal usage of the STL in C++ source code.

Current meta-programming systems do not provide profile information about the input programs on which meta-programs operate. Therefore, profile-guided meta-programs introduce new special-purpose toolkits for profiling and meta-programming. Instead, meta-programming systems should provide access to profile information from existing profilers. Meta-programmers could then implement profile-guided meta-programs while reusing the meta-programming and profiling tools of a familiar system. We present a design for supporting profile-guided meta-programming in general-purpose meta-programming systems. To demonstrate the generality of our design, we implement it in two languages. Both implementations reuse existing meta-programming and profiling infrastructure.

The remainder of the paper is organized as follows. In Section 2, we introduce a running example and Scheme-style meta-programming. In Section 3, we describe our requirements on the underlying profiling system and an API for supporting profile-guided meta-programming. In Section 4, we present two implementations of the specification in Section 3: one in Chez Scheme and one in Racket. In Section 5, we sketch implementations for other general-purpose meta-programming systems. In Section 6, we demonstrate that our design is general enough to implement and extend existing PGOs and profile-guided meta-programs. In Section 7, we relate to existing work on PGOs and profile-guided meta-programming.

* Author's current affiliation: Northeastern University

† Author's current affiliation: VMware, Inc

2. A Running Example

We first introduce a syntax extension to familiarize readers with Scheme and Racket style meta-programming and to provide a running example. This transformation presented is not a meaningful optimization and is only used for illustrative purposes.

```
(define-syntax (if-r stx)
  (syntax-case stx ()
    [(if-r test t-branch f-branch)
     ; This let expression runs at compile-time
     (let ([t-prof (profile-query #'t-branch)]
           [f-prof (profile-query #'f-branch)])
       ; This cond expression runs at
       ; compile-time, and conditionally
       ; generates runtime code based on profile
       ; information.
       (cond
         [(< t-prof f-prof)
         ; This if expression would run at
         ; runtime when generated.
          #'(if (not test) f-branch t-branch)]
         [(>= t-prof f-prof)
         ; So would this if expression.
          #'(if test t-branch f-branch)]))]))

; Example use of if-r
(define (classify email)
  (if-r (subject-contains email "PLDI")
        (flag email 'important)
        (flag email 'spam)))

; Assuming profile information tells us:
;    (flag email 'important) runs 5 times
;    (flag email 'spam)      runs 10 times
; Then the above use of if-r is rewritten to:
(define (classify email)
  (if (not (subject-contains email "PLDI"))
      (flag email 'spam)
      (flag email 'important)))
```

Figure 1: Example syntax extension

In Figure 1, `define-syntax` introduces a new syntax extension `if-r`. Any uses of `if-r` in the source will be rewritten using the code in the body of the extension. The syntax extension can be thought of as a function from source expressions to source expression.

When used in Figure 1, the syntax extension `if-r` receives the argument:

```
#'(if-r (subject-contains-ci email "PLDI")
        (flag email 'important)
        (flag email 'spam))
```

This is a data representation of a term called a syntax object. The forms `#'`, `` #` ``, and `#,` provide a templating system for syntax objects,[1] and `syntax-case` performs pattern matching on syntax objects.

The syntax extension `if-r` is expanded at compile-time, while the resulting `if` expression is run at runtime. At compile time, `if-r` looks up the profile information attached to each branch, using `profile-query`. Using profile information, `if-r` conditionally generates an `if` expression whose branches are ordered by how likely they are to be executed. When the false branch is executed more frequently than the true branch, `if-r` expands into an `if` expression by negating the test and swapping the branches. Otherwise, `if-r` expands into an `if` expression by keeping the original test and branches. While this transformation is not meaningful, it strongly resembles the optimization we present in Section 6.1.

[1] Specifically, these forms implement Lisp's quote, quasiquote, and unquote on syntax objects instead of lists.

3. Design

Profile-guided meta-programming requires that the underlying language comes with a profiling system, and that the meta-programming system can access the profile information. This section presents the high-level requirements of our design, and sketches an API that suffices to provide profile-guided meta-programming. Our design is not specific to a particular profiling technique, but for simplicity our explanations refer to counter-based profile information.

3.1 Profile Points

As the profiling system may not understand source expressions, our design introduces *profile points* as an abstraction of source expressions for the profiler. Each profile point uniquely identifies a counter. Any expression can be associated with at most one profile point. Associating a profile point with an expression indicates which counter to increment when profiling the expression. For instance, if two expressions are associated with the same profile point, then they both increment the same counter when executed. Conversely, if two expressions are associated with different profile points, then they increment different profile counters when executed. The profiling system uses profile points when a program is instrumented to collect profile information. When the program is not instrumented to collect profile information, profile points need not introduce any runtime overhead.

For fine-grained profiling, each input expression and subexpression can be associated with a unique profile point. In the case of our running example, the ASTs for `if`, `subject-contains`, `email`, `"PLDI"`, etc, are each associated with separate profile points. Note that `flag` and `email` appear multiple times, but each occurrence is associated with separate profile point.

A profiler may implicitly insert profile points on certain nodes in the AST, but it is also important that meta-programs can manufacture new profile points. Meta-programmers may want to generate expressions that are profiled separately from any other expression in the source program.

Meta-programmers can access profile information by passing a profile point, or an object with an associated profile point, to an API call, such as the function `profile-query` in our running example.

3.2 Profile Information

Multiple data sets are important to ensure PGOs can optimize for multiple classes of inputs expected in production. However, absolute profile information is incomparable across different data sets. Instead, our design considers *profile weights*. The profile weight is represented as a number in the range [0,1]. The profile weight of a profile point is the ratio of the counter for that profile point to the counter of the most executed profile point in the same data set. This provides a single value identifying the relative importance of an expression and simplifies the combination of multiple profile data sets.

```
;; After loading data from data set 1
(flag email 'important)→ 5/10             ;; 0.5
(flag email 'spam)      → 10/10            ;; 1

;; After loading data from data sets 1 and 2
(flag email 'important)→ (.5 + 100/100)/2 ;; 0.75
(flag email 'spam)      → (1 + 10/100)/2   ;; 0.55
```

Figure 2: Sample profile weight computations

To demonstrate profile weights, consider the running example from Figure 1. Suppose in the first data set, `(flag email 'important)` runs 5 times and `(flag email 'spam)` runs 10 times. In the second data set, `(flag email 'important)` runs 100 times and `(flag email 'spam)` run 10 times. Figure 2 shows the profile weights computed after each data set.

3.3 API

This section presents an example of an API provided by a meta-programming system that implements our design. We assume a single object, `(current-profile-information)`, exists in the meta-programming system. Figure 3 documents the methods of this object. The type `SyntaxObject` is provided by the meta-programming system. This API assumes that the underlying implementation has some way to profile expressions that are associated with profile points. The API is only concerned with providing meta-programs with access to that profile information and the ability to manipulate profile points.

```
type ProfilePoint
type ProfileWeight
type ProfileInformation
```

```
(make-profile-point) → ProfilePoint
```

Generates a profile point deterministically so meta-programs can access the profile information of the generated profile point across multiple runs.

```
(annotate-expr e pp) → SyntaxObject
  e : SyntaxObject
  pp : ProfilePoint
```

Associates the expression *e* with the profile point *pp*. The underlying profiling system will increment the counter for *pp* anytime *e* is executed.

```
(profile-query e) → ProfileWeight
  e : SyntaxObject
```

Retrieves the profile weight associated with the profile point for the expression *e*.

```
(store-profile f) → Null
  f : Filename
```

Stores the current profile information from the underlying profile system in the file with the filename *f*.

```
(load-profile f) → ProfileInformation
  f : Filename
```

Loads the profile information stored in the file with the filename *f*.

Figure 3: API Sketch

4. Implementations

To validate the design principles from Section 3, we provide two implementations. This section describes implementations in Chez Scheme and Racket. While both languages belong to the Lisp family, they differ in their meta-programming and profiling facilities.

4.1 Chez Scheme Implementation

Chez Scheme implements precise counter-based profiling, using standard and efficient block-level profiling techniques (Ball and Larus 1994; Burger and Dybvig 1998). The Chez Scheme profiler separately profiles every source expression, and provides profiles in terms of source code locations.

In Chez Scheme, we implement profile points using *source objects* (Dybvig et al. 1993) which can be attached to syntax objects. Chez Scheme source objects contain a filename and starting and ending character positions. The Chez Scheme reader automatically creates and attaches source objects to each syntax object it reads from a file. Chez Scheme uses source objects to report errors at their precise source location.

Chez Scheme provides an API to programmatically manipulate source objects and attach them to syntax objects (Dybvig 2011, Chapter 11). We use this API to implement `make-profile-point` and `annotate-expr`. The former deterministically generates fresh source objects by adding a suffix to the filename of a base source object. This scheme has the added benefit of preserving source locations for error messages when errors occur in the output of a profile-guided optimization.

We modify the meta-programming system to maintain an associative map of source objects to profile weight, which implements `(current-profile-information)`. The function `profile-query` simply queries this map. The function `load-profile` updates this map from a file and the function `store-profile` stores it to a file.

4.2 Racket Implementation

Racket includes an `errortrace` profiling library. The `errortrace` library provides counter-based profiling and returns profiles in terms of source code locations, similar to the Chez Scheme profiler. Note that in contrast to the Chez Scheme profiler, the `errortrace` library only profiles function calls.

In Racket, we implement profile points in essentially the same way as in Chez Scheme—by using source information attached to each syntax object. The Racket reader automatically attaches the filename, line number, etc to every syntax object it reads from a file. These source locations are used to report errors at their precise location.

Racket provides an API for attaching source information when building a new syntax object. A separate library exists which provides a more extensive API for manipulating source information. We use this library to implement `make-profile-point` and `annotate-expr` in essentially the same way as in Chez Scheme. There is one key difference because the `errortrace` library only profiles functions calls. When annotating an expression `e` with profile point `p`, we generate a new function `f` whose body is `e`. The result of `annotate-expr` is a call to the generated function `f`. This call to `f` is annotated with the profile point `p`. While this results in difference performance characteristics while profiling, it does not change the counters used to calculate profile weights.

We implement a library which maintains the associative map from source locations to profile weight. The library provides our API as simple Racket functions that can be called by meta-programs. We are able to implement the entire API as user-level a library due to Racket's advanced meta-programming facilities and the extensive API provided by the existing Racket profiler.

4.3 Source and Block-level PGO

One goal of our approach is to avoid interfering with traditional, e.g., basic block-level PGO, which Chez Scheme also supports. However, since meta-programs may generate different source code after optimization, the low-level representation would have to change when meta-programs perform optimizations. To solve this problem, the source code is compiled three times in a specific order, instead of the usual two times. Doing so ensure profile information remains consistent at both the source-level and the block-level. First, we compile while instrumenting the code to profile source expressions. After running the instrumented program on representative inputs, we get the profile weights as in Figure 2. Second, we recompile, using those profile weights to perform profile-guided meta-program optimizations, while instrumenting the code to profile basic blocks. The generated source code, Figure 1, will remain stable as long as we continue to optimize using the source profile weights. Because the generated source code remains stable, so do the generated basic blocks. After running the instrumented program, we get the profile weights for the basic blocks generated from the optimized source program. Third, we recompile using both the profile weights for the source expressions and for the basic blocks to do both profile-guided meta-programming and low-level PGOs.

4.4 Compile-Time and Profiling Overhead

As with any technique for performing profile-guided optimizations, our approach introduces compile-time overhead for optimizations and runtime overhead when profiling.

The compile-time overhead of our API is small. In our implementations, loading profile information is linear in the number of profile points, and querying the weight of a particular profile point is constant-time. The API does not introduce slowdown at runtime, however.

Profile-guided meta-programs may also slow down (or speed up) compilation, as they run at compile-time. The slowdown will depend on the complexity of the meta-program.

A meta-programming system using our technique inherits overhead from the profiler used in the implementation. Previous work measured about 9% runtime overhead introduced by the Chez Scheme profiler (Burger and Dybvig 1998). According to the **`errortrace`** documentation, profiling introduces a factor of 4 to 12 slowdown. Typically, profiling is disabled for production runs of a program, so this overhead affects only for profiled runs.

5. Beyond Scheme and Racket

Our design should work in most meta-programming systems. Languages such as Template Haskell (Sheard and Jones 2002), MetaOCaml (Kiselyov 2014), and Scala (Odersky et al. 2004) feature powerful meta-programming facilities. They allow executing expressive programs at compile-time, support direct access to input expressions, and provide templating systems for manipulating expressions. In this section, we briefly sketch implementation strategies for these meta-programming systems to validate the generality of our design.

5.1 Template Haskell

Template Haskell (Sheard and Jones 2002) adds general-purpose meta-programming to Haskell, and comes with the current version of the Glasgow Haskell Compilers (GHC).

GHC's profiler attributes costs to *cost-centers*. By default, each function defines a cost-center, but users can define new cost-centers by adding an *annotation* to the source code:

```
{#- SCC "cost-centre-name" #-}
```

Cost-centers map easily to profile points.

Implementing our API using Template Haskell would be simple. Template Haskell, as of GHC 7.7, supports generating and querying annotations. Since cost-centers are defined via annotations, implementing **`make-profile-point`**, **`annotate-expr`**, and **`profile-query`** would be straightforward. Implementing **`load-profile`** is a simple matter of parsing profile files. The GHC profiler is called via a system call, and not inside the language as in Chez Scheme and Racket. Therefore, it would be useful to implement **`store-profile`**, which stores profile information to a file. Instead, profile information is stored to a file by the GHC profiler.

5.2 MetaOCaml

MetaOCaml (Kiselyov 2014) provides general-purpose meta-programming based on multi-stage programming for OCaml.

OCaml features a counter-based profiler that associates counts with the locations of certain source expressions. To implement **`make-profile-point`** and **`annotate-expr`**, MetaOCaml would require the ability to manipulate source locations and attach them to source expressions. Then implementing **`profile-query`** should be straightforward. Like in Haskell, implementing **`load-profile`** simply requires parsing profile files, and profile information disk is stored to a file outside of the language.

5.3 Scala

Scala features powerful general-purpose meta-programming (Burmako 2013), multi-stage programming (Rompf and Odersky 2010), and various reflection libraries.

Existing profilers for Scala work at the level of the JVM. However, it should be possible to map the profiling information at the JVM level back to Scala source code. With such a mapping, a Scala implementation of our API should be similar to the implementation sketches for Haskell and MetaOCaml.

6. Case Studies

To evaluate the expressive power and usability of our design, we carry out three case studies. In the first study, we demonstrate an implementation of **`case`** expressions, which are analogous to C's **`switch`** statements, that performs a well-known PGO. In the second study, we equip an embedded object system with profile-guided receiver class prediction (Grove et al. 1995; Hölzle and Ungar 1994). In the the third and final study, we present libraries that recommend and automate high-level changes to data structures, similar to the recommendations given by tools like Perflint (Liu and Rus 2009).

6.1 Profile-guided conditional branch optimization

In C#, **`switch`** statements must be mutually exclusive and do not allow fall through—each case must end in a jump such as **`break`**. The .NET compiler features a profile-guided optimization of **`switch`** statements that uses profile information to reorder the branches according to which branch is most likely to succeed. The entire implementation is 81 lines long.

The **`case`** expression takes an expression **`key-expr`** and an arbitrary number of clauses, followed by an optional **`else`** clause. Each clause consists of a list of constants on the left-hand side and a body expression on the right-hand side. A **`case`** expression executes the body of the first clause in which **`key-expr`** is **`equal?`** to some element of the left-hand side. For simplicity, we present a version of **`case`** that assumes that a constant does not appear in the left-hand side of more than one clause and does not support an **`else`** clause[2]. Figure 4 shows an example **`case`** expression.

Figure 5 shows the profile-guided implementation of **`case`** that reorders branches according to which clause is most likely to succeed. It creates an invocation of another meta-program, **`exclusive-cond`**, which reorders its branches based on profile information. The implementation rewrites each **`case`** clause into an **`exclusive-cond`** clause. The form **`#,@`** splices the list of rewritten clauses into the template for the **`exclusive-cond`** expression. An **`exclusive-cond`** clause consists of a boolean expression on the left-hand side and a body expression on the right-hand side. Each **`case`** clause is transformed by converting the left-hand side into an explicit membership test for **`key-expr`**, while leaving the body unchanged. The full implementation of **`case`** in Racket, which also removes duplicate constants from successive clauses and supports an optional **`else`** clause that is never reordered, is 50 lines long.

Figure 6 shows the implementation of the **`exclusive-cond`** expression. This is a multi-way conditional branch similar to Lisp's

[2] The full implementation handles the full generality of Scheme's **`case`**

```
(define (parse stream)
 (case (peek-char stream)
  [(#\space #\tab) (white-space stream)]
  [(0 1 2 3 4 5 6 7 8 9) (digit stream)]
  [(#\() (start-paren stream)]
  [(#\)) (end-paren stream)]
  ....))
```

Figure 4: An example using `case`

```
(define-syntax (case syn)
 ; Internal definition
 (define (rewrite-clause key-expr clause)
  (syntax-case clause ()
   [((k ...) body)
    ; Take this branch if the key expression
    ; is a equal? to some element of the list
    ; of constants
    #`((key-in? #,key-expr '(k ...)) body)]))
 ; Start of code transformation.
 (syntax-case syn ()
  [(_ key-expr clause ...)
   ; Evaluate the key-expr only once, instead of
   ; copying the entire expression in the
   ; template.
   #`(let ([t key-expr])
       (exclusive-cond
        ; transform each case clauses
        ; into an exclusive-cond clause
        #,@(map (curry rewrite-clause #'key-expr)
                #'(clause ...))))]))
```

Figure 5: Implementation of `case`

`cond`, except that all branches must be mutually exclusive. Because the branches are mutually exclusive, `exclusive-cond` can safely reorder them. The implementation of `exclusive-cond` simply sorts each clause by profile weight and generates a regular `cond`. Since each `exclusive-cond` clause is also a `cond` clause, the clauses do not need to be transformed. Figure 7 shows the code generated after expanding `case` and then after expanding `exclusive-cond` in the example `case` expression in Figure 4. The full implementation of `exclusive-cond` in Racket, which also handles additional `cond` syntaxes and an optional `else` clause that is never reordered, is 31 lines long.

Separating the implementation of `exclusive-cond` and `case` in this way simplifies the implementation of `case`. The `exclusive-cond` expression also demonstrates an important feature of profile-guided meta-programming—meta-programming allows the programmer to encode their domain-specific knowledge, e.g., that the branches of this conditional are mutually exclusive, in order to take advantage of optimizations that would have otherwise been impossible.

6.2 Profile-guided receiver class prediction

Profile-guided receiver class prediction (Grove et al. 1995; Hölzle and Ungar 1994) is a well-known PGO for object-oriented languages. However, when an object-oriented language is implemented via meta-programming as a domain-specific language (DSL), the host language may not be able to implement this PGO. In this second case study, we implement a simplified object system as a syntax extension. Using our design, we easily equip this object system with profile-guided receiver class predication. This demonstrates that our design is both expressive enough to implement well-known PGOs and powerful enough to provide DSLs with PGOs not available in the host language. The full implementation of profile-guided receiver class prediction is 44 lines long, while the implementation of the entire object system (including the PGO) is 129 lines long.

```
(define-syntax (exclusive-cond syn)
  ; Internal definitions
  (define (clause-weight clause)
    (syntax-case clause ()
      [(test e1 e2 ...) (profile-query #'e1)]))
  (define (sort-clauses clause*)
    ; Sort clauses greatest-to-least by weight
    (sort clause* > #:key clause-weight))
  ; Start of code transformation
  (syntax-case x ()
    [(_ clause ...)
     ; Splice sorted clauses into a cond
     ; expression
     #`(cond #,@(sort-clause #'(clause ...)))]))
```

Figure 6: Implementation of `exclusive-cond`

```
; After case expands
(define (parse stream)
 (let ([t (peek-char stream)])
   (exclusive-cond
     [(key-in? t '(#\space #\tab))
      (white-space stream)]
     [(key-in? t '(0 1 2 3 4 5 6 7 8 9))
      (digit stream)]
     [(key-in? t '(#\()) (start-paren stream)]
     [(key-in? t '(#\))) (end-paren stream)]
     ....)))

; After exclusive-cond expands
(define (parse stream)
 (let ([t (peek-char stream)])
   (cond
     [(key-in? t '(#\space #\tab))
      (white-space stream)] ; Run 55 times
     [(key-in? t '(#\())
      (start-paren stream)] ; Run 23 times
     [(key-in? t '(#\)))
      (end-paren stream)]    ; Run 23 times
     [(key-in? t '(0 1 2 3 4 5 6 7 8 9))
      (digit stream)]        ; Run 10 times
     ....)))
```

Figure 7: Generated code from Figure 4

Figure 8 shows the implementation of profile-guided receiver class prediction. A method call such as `(method s area)` is actually a meta-program that generates code as follows. First, it generates a new profile point for each class in the system. When profile information is not available, the method call generates a `cond` expression with a clause for each class in the system. Each clause tests if `s` is an instance of a specific class, ignores the result, and uses normal dynamic dispatch to call the `area` method of `s`. However, *a different profile point is associated with each branch.*[3] That is, each method call site is instrumented by generating a multi-

[3] A production implementation would create a table of instrumented dynamic dispatch calls and dynamically dispatch through this table, instead of instrumenting code with `cond`. However, using `cond` simplifies visualizing the instrumentation.

```
(class Square
  ((length 0))
  (define-method (area this)
    (sqr (field this length))))
(class Circle
  ((radius 0))
  (define-method (area this)
    (* pi (sqr (field this radius)))))
(class Triangle
  ((base 0) (height 0))
  (define-method (area this)
    (* 1/2 base height)))
....
(for/list ([s (list cir1 cir2 cir3 sqr1)])
  (method s area))
```

Figure 9: Example of profile-guided receiver class prediction

```
; ------------------
; Generated code after instrumentation
....
(for/list ([s (list cir1 cir2 cir3 sqr1)])
  (let* ([x s])
    (cond
      [(instance-of? x 'Square)    ; Run 1 time
       (instrumented-dispatch x area)]
      [(instance-of? x 'Circle)    ; Run 3 times
       (instrumented-dispatch x area)]
      [(instance-of? x 'Triangle)  ; Run 0 times
       (instrumented-dispatch x area)]
      [else (dynamic-dispatch x area)])))

; ------------------
; Generated code after optimization
....
(for/list ([s (list cir1 cir2 cir3 sqr1)])
  (let* ([x s])
    (cond
      [(instance-of? x 'Square)  ; Run 1 time
       (sqr (field x length))]
      [(instance-of? x 'Circle)  ; Run 3 times
       (* pi (sqr (field x radius)))]
      [else (dynamic-dispatch x area)])))
```

Figure 10: Generated code from Figure 9

way branch to the standard dynamic dispatch routine, but with a separate profile point in each branch. When profile information *is* available, the method call generates a `cond` expression with clauses for the most frequently used classes at this method call site. Each clause again tests if `s` is an instance of a specific class, but the body of the clause is generated by inlining the method for that class—that is, it performs polymorphic inline caching for the most frequently used classes based on profile information. The full implementation of profile-guided receiver class prediction is 44 lines long. The rest of the object system implementation is an additional 87 lines long.

Figure 9 shows an example code snippet using this object system. Figure 10 demonstrates the resulting code after instrumentation, and the resulting code after optimization. Note that each occurrence of `(instrumented-dispatch x area)` has a different profile point, so each occurrence is profiled separately.

As a further improvement, we could reuse `exclusive-cond` to test for classes in the the most likely order.

```
; ------------------
; After optimization
....
(for/list ([s (list cir1 cir2 cir3 sqr1)])
  (let* ([x s])
    (exclusive-cond
      [(instance-of? x 'Square)  ; Run 1 time
       (sqr (field x length))]
      [(instance-of? x 'Circle)  ; Run 3 times
       (* pi (sqr (field x radius)))]
      [else (dynamic-dispatch x area)])))

; ------------------
; After more optimization
....
(for/list ([s (list cir1 cir2 cir3 sqr1)])
  (let* ([x s])
    (cond
      [(instance-of? x 'Circle)  ; Run 3 times
       (* pi (sqr (field x radius)))]
      [(instance-of? x 'Square)  ; Run 1 time
       (sqr (field x length))]
      [else (dynamic-dispatch x area)])))
```

Figure 11: Profile-guided receiver class prediction, sorted.

6.3 Data Structure Specialization

In this final case study, we show that our approach is expressive enough to implement and improve upon state-of-the-art profile-guided tools such as Perflint (Liu and Rus 2009), which provides high-level recommendations for changes in data structures and algorithms that may result in asymptotic improvements. We describe implementations of list and vector libraries that warn the programmer when a different representation may lead to asymptotic performance gains. The new libraries wrap the standard list and vector functions. These wrapper use generated profile point to separately profile each instance of the data structures. Finally, we develop a sequence datatype that will automatically specialize to a list or vector based on profiling information. As this is done via a library, programmers can easily opt-in to such automated high-level changes without many changes to their code. The full implementation of the list library is 80 lines long, the vector library is 88 lines long, and the sequence library is 111 lines long.

Figure 12 shows the implementation of the profiled list constructor. This constructor has the same interface as the standard Scheme list constructor—it takes an arbitrary number of elements and returns a representation of a linked list. The representation of a `profiled-list` is a pair of the underlying linked list and a hash table of profiled operations. That is, each instance of a `profiled-list` contains a table of instrumented calls to the underlying list operations. The profiled list constructor generates these instrumented operations by wrapping the underlying list operations with the appropriate profile point. The constructor generates two profile points for each profiled list. One is used to profile operations that are asymptotically fast on lists and the other is used to profile operations that are asymptotically fast on vectors. Finally, the library exports new versions of the list operations that work on the profiled list representation. For instance, it exports `car`, which takes a `profiled-list`, and uses the instrumented call to `car` from the hash table of the profiled list on the underlying linked list. When profiling information already exists, for instance, after a profiled run, this list constructor emits a warning (at compile time) if fast vector operations were more common than fast list operations. We provide an analogous implementation of vectors. This approach would scale to the other data structures analyzed by Perflint.

```
(define-syntax (method syn)
  (syntax-case syn ()
    [(_ obj m val* ...)
     ....
     ; Don't copy the object expression throughout the template.
     #`(let* ([x obj])
         (cond
           #,@(if no-profile-data?
                  ; If no profile data, instrument!
                  (for/list ([d instr-dispatch-calls] [class all-classes])
                    #`((instance-of? x #,class) (#,d x m val* ...)))
                  ; If profile data, inline up to the top inline-limit classes
                  ; with non-zero weights
                  (for/list ([class (take sorted-classes inline-limit)])
                    #`((instance-of? x #,class)
                       #,(inline-method class #'x #'m #'(val* ...)))))
           ; Fall back to dynamic dispatch
           [else (dynamic-dispatch x m val* ...)]))]))
```

Figure 8: Implementation of profile-guided receiver class prediction

```
(struct list-rep (instr-op-table ls))
....
(define-syntax (profiled-list syn)
  ; Create fresh profile points.
  ; Use list-src to profile operations that are asymptotically fast on lists
  ; Use vector-src profile operations that are asymptotically fast on vectors
  (define list-src (make-profile-point))
  (define vector-src (make-profile-point))
  ....
  (syntax-case syn ()
    [(_ init-vals ...)
     (unless (>= (profile-query list-src) (profile-query vector-src))
       ; Prints at compile time.
       (printf "WARNING: You should probably reimplement this list as a vector: ~a\n" syn))
     #`(make-list-rep
         ; Build a hash table of instrumented calls to list operations
         ; The table maps the operation name to a profiled call to the
         ; built-in operation.
         (let ([ht (make-eq-hashtable)])
           (hashtable-set! ht 'car #,(instrument-call car list-src))
           ...
           ht)
         (list init* ...))]))
```

Figure 12: Implementation of profiled `list`

Our approach enables us to go beyond just providing recommendations. Because our meta-programs are integrated into the language, rather than separate tools outside the language, we can easily provide libraries that automatically follow these recommendations rather than asking programmers to change their code. To demonstrate this point, we implement a profiled sequence datatype that will automatically specialize each instance to a list or vector, at compile-time, based on profile information.

Figure 13 shows the implementation of the profiled sequence constructor. The code follows the same pattern as the profiled list. The key difference is we conditionally generate wrapped versions of the list *or* vector operations, and represent the underlying data using a list *or* vector, depending on the profile information.

7. Related Work

7.1 Profile-Guided Optimizations

Modern compiler systems such as GCC, .NET, and LLVM (Lattner 2002) use profile-guided optimizations. These systems use profile information to guide decisions about code positioning, register allocation, inlining, and conditional branch ordering.

GCC profiles at the level of an internal control-flow graph (CFG). To maintain consistent CFGs across instrumented and optimized builds, GCC requires similar optimization decisions across builds (Chen et al. 2010). This requirement is similar to how we ensure consistency when using both source and block-level PGOs in Chez Scheme.

In addition to the common optimizations noted previously, the .NET profiler features special support for `switch` statements called *value probes*. The .NET compilers use value probes optimize `switch` statement common values, similar to our optimization of `case` expressions in Section 6.1. Our design can express

```
(struct seq-rep (instr-op-table s))
....
(define-syntax (seq syn)
   (define list-src (make-profile-point))
   (define vector-src (make-profile-point))
   (define previous-list-usage (profile-query list-src))
   (define previous-vector-usage (profile-query vector-src))
   (define list>=vector (>= previous-list-usage previous-vector-usage))
   ....
   (syntax-case syn ()
     [(_ init* ...)
      #`(let ()
          (make-seq-rep
            (let ([ht (make-eq-hashtable)])
                #`(hashtable-set! ht 'car #,(pick-op list>=vector 'car))
                ...
                ht)
            (#,(if list>=vector #'list #'vector) init* ...)))]))
```

Figure 13: Implementation of profiled sequence

this optimization via the same profiler machinery used in our other case studies.

LLVM takes a different approach to PGO. LLVM uses a runtime reoptimizer that monitors the running program. The runtime system can profile the program "in the field". This runtime system can perform simple optimizations on the machine code during program execution. More complex optimization require running an offline optimizer on the LLVM bytecode (Lattner and Adve 2004).

Recent work is still discovering novel applications for profile information. Furr et al. (2009) use profile information to infer types in dynamic languages to assist in debugging. Chen et al. (2006b) apply profile optimization to optimize garbage collection. Luk et al. (2002) perform data prefetching guided by profile information. Debray and Evans (2002) compress infrequently executed code based on profile information.

7.2 Meta-Program Optimizations

Meta-programming combines the ability to provide high-levels of abstraction while producing efficient code. Meta-programming has been widely used to implement high performance domain-specific languages (Rompf and Odersky 2010; Sujeeth et al. 2014; Sujeeth et al. 2013), whole general purpose languages (Barzilay and Clements 2005; Rafkind and Flatt 2012; Tobin-Hochstadt and Felleisen 2008), and production-quality compiler generators (Keep and Dybvig 2013). Tobin-Hochstadt et al. (2011) implement the optimizer for the Typed Racket language as a meta-program. The HERMIT toolkit provides an API for performing program transformations on Haskell intermediate code before compiling, even allowing interactive experimentation (Farmer et al. 2012). Hawkins et al. (2011; 2012) implement a compiler for a language that generates C++ implementations of data structures based on high-level specifications.

Previous work also integrates profiling to guide meta-program optimizations. Chen et al. (2006a) perform process placement for SMP clusters via profile-guided meta-programming. Šimunić et al. (2000) optimize source code using energy profiles, although the bulk of the optimization requires programmer intervention. Karuri et al. (2005) optimize ASIP designs with fine-grained source profile information.

In contrast, our own work introduces a single general-purpose approach in which we can implement new general-purpose languages, domain-specific languages, efficient abstract libraries, and arbitrary meta-programs, all of which can take advantage of profile-guided optimizations. Further, our approach reuses existing meta-programming and profiling facilities, rather than implementing new tools that interface the compiler in ad-hoc ways.

8. Conclusion

Meta-programming is used to implement high-level optimizations, generate code from high-level specifications, and create domain-specific languages. Each of these can take advantage of PGO to optimize before information is lost or constraints are imposed. Until now, such optimizations have been implemented via tools designed for a specific meta-program or a specific optimization. We described how to build a general mechanism for implementing arbitrary profile-guided meta-programs. We also demonstrated the expressivity of this design by by using it to implement several examples of profile-guided meta-programs.

Bibliography

Matthew Arnold, Stephen Fink, Vivek Sarkar, and Peter F. Sweeney. A Comparative Study of Static and Profile-based Heuristics for Inlining. In *Proc. of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization (DYNAMO)*, 2000. http://doi.acm.org/10.1145/351397.351416

Thomas Ball and James R. Larus. Optimally Profiling and Tracing Programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16(4), pp. 1319–1360, 1994. http://doi.acm.org/10.1145/183432.183527

Eli Barzilay and John Clements. Laziness Without All the Hard Work: Combining Lazy and Strict Languages for Teaching. In *Proc. of the Workshop on Functional and Declarative Programming in Education (FDPE)*, 2005. http://doi.acm.org/10.1145/1085114.1085118

Robert G. Burger and R. Kent Dybvig. An infrastructure for profile-driven dynamic recompilation. In *Proc. of the International Conference on Computer Languages (ICCL)*, 1998. http://dx.doi.org/10.1109/ICCL.1998.674174

Eugene Burmako. Scala Macros: Let Our Powers Combine!: On How Rich Syntax and Static Types Work with Metaprogramming. In *Proc. of the Workshop on Scala (SCALA)*, 2013. http://doi.acm.org/10.1145/2489837.2489840

Deheo Chen, Neil Vachharajani, Robert Hundt, Shih-wei Liao, Vinodha Ramasamy, Paul Yuan, Wenguang Chen , and Weimin Zheng. Taming Hardware Event Samples for FDO Compilation. In *Proc. of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2010. http://doi.acm.org/10.1145/1772954.1772963

Hu Chen, Wenguang Chen, Jian Huang, Bob Robert, and H. Kuhn. MPIPP: An Automatic Profile-guided Parallel Process Placement Toolset for SMP Clusters and Multiclusters. In *Proc. of the International Conference on Supercomputing (ICS)*, 2006a. http://doi.acm.org/10.1145/1183401.1183451

Wen-ke Chen, Sanjay Bhansali, Trishul Chilimbi, Xiaofeng Gao, and Weihaw Chuang. Profile-guided Proactive Garbage Collection for Locality Optimization. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2006b. http://doi.acm.org/10.1145/1133981.1134021

B. Dawes and D. Abrahams. Boost C++ Libraries. 2009. http://www.boost.org

Saumya Debray and William Evans. Profile-guided Code Compression. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2002. http://doi.acm.org/10.1145/512529.512542

R. Kent Dybvig. Chez Scheme Version 8 User's Guide. 8.4 edition. Cadence Research Systems, 2011. http://www.scheme.com/csug8

R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic Abstraction in Scheme. *LISP and Symbolic Computation* 5(4), pp. 295–326, 1993. http://dx.doi.org/10.1007/BF01806308

Sebastian Erdweg, Tillmann Rendel, Christian Kästner , and Klaus Ostermann. SugarJ: Library-based Syntactic Language Extensibility. In *Proc. of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2011. http://doi.acm.org/10.1145/2048066.2048099

Andrew Farmer, Andy Gill, Ed Komp, and Neil Sculthorpe. The HERMIT in the Machine: A Plugin for the Interactive Transformation of GHC Core Language Programs. In *Proc. of the ACM SIGPLAN Haskell Symposium (Haskell)*, 2012. http://doi.acm.org/10.1145/2364506.2364508

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

Michael Furr, Jong-hoon (David) An, and Jeffrey S. Foster. Profile-guided Static Typing for Dynamic Scripting Languages. In *Proc. of the ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2009. http://doi.acm.org/10.1145/1640089.1640110

David Grove, Jeffrey Dean, Charles Garrett, and Craig Chambers. Profile-guided receiver class prediction. In *Proc. of the ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA)*, 1995. http://doi.acm.org/10.1145/217838.217848

Rajiv Gupta, Eduard Mehofer, and Youtao Zhang. *Profile Guided Code Optimization*. 2002.

Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin Rinard, and Mooly Sagiv. Data Representation Synthesis. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2011. http://doi.acm.org/10.1145/1993498.1993504

Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin Rinard, and Mooly Sagiv. Concurrent Data Representation Synthesis. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2012. http://doi.acm.org/10.1145/2254064.2254114

Urs Hölzle and David Ungar. Optimizing Dynamically-dispatched Calls with Run-time Type Feedback. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1994. http://doi.acm.org/10.1145/178243.178478

Kingshuk Karuri, Mohammad Abdullah Al Faruque, Stefan Kraemer, Rainer Leupers, Gerd Ascheid, and Heinrich Meyr. Fine-grained application source code profiling for ASIP design. In *Proc. of the Design Automation Conference (DAC)*, 2005. http://doi.acm.org/10.1145/1065579.1065666

Andrew W. Keep and R. Kent Dybvig. A Nanopass Framework for Commercial Compiler Development. In *Proc. of the ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2013. http://doi.acm.org/10.1145/2500365.2500618

Oleg Kiselyov. The Design and Implementation of BER MetaOCaml. *Functional and Logic Programming* 8475, pp. 86–102, 2014. http://dx.doi.org/10.1007/978-3-319-07151-0_6

Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proc. of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2004. http://dx.doi.org/10.1109/CGO.2004.1281665

Chris Arthur Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master dissertation, University of Illinois, 2002.

Lixia Liu and Silvius Rus. Perflint: A Context Sensitive Performance Advisor for C++ Programs. In *Proc. of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2009. http://dx.doi.org/10.1109/CGO.2009.36

Chi-Keung Luk, Robert Muth, Harish Patil, Richard Weiss, P. Geoffrey Lowney, and Robert Cohn. Profile-guided Post-link Stride Prefetching. In *Proc. of the International Conference on Supercomputing (ICS)*, 2002. http://doi.acm.org/10.1145/514191.514217

Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An Overview of the Scala Programming Language. EPFL Lausanne, IC/2004/64, 2004. http://infoscience.epfl.ch/record/52656?ln=en

Jon Rafkind and Matthew Flatt. Honu: Syntactic Extension for Algebraic Notation Through Enforestation. In *Proc. of the International Conference on Generative Programming and Component Engineering (GPCE)*, 2012. http://doi.acm.org/10.1145/2371401.2371420

Tiark Rompf and Martin Odersky. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. In *Proc. of the International Conference on Generative Programming and Component Engineering (GPCE)*, 2010. http://doi.acm.org/10.1145/1868294.1868314

Tim Sheard and Simon Peyton Jones. Template Meta-programming for Haskell. In *Proc. of the ACM SIGPLAN Workshop on Haskell (Haskell)*, 2002. http://doi.acm.org/10.1145/581690.581691

Arvind K. Sujeeth, Kevin J. Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. Delite: A Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages. *ACM Transactions on Embedded Computing Systems (TECS)* 13(4s), 2014. http://doi.acm.org/10.1145/2584665

Arvind K. Sujeeth, Austin Gibbons, Kevin J. Brown, HyoukJoong Lee, Tiark Rompf, Martin Odersky, and Kunle Olukotun. Forge: Generating a High Performance DSL Implementation from a Declarative Specification. In *Proc. of the International Conference on Generative Programming: Concepts & Experiences (GPCE)*, 2013. http://doi.acm.org/10.1145/2517208.2517220

Walid Taha and Tim Sheard. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science* 248(1–2), pp. 211–242, 2000. http://dx.doi.org/10.1016/S0304-3975(00)00053-0

Sam Tobin-Hochstadt and Matthias Felleisen. The Design and Implementation of Typed Scheme. In *Proc. of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2008. http://doi.acm.org/10.1145/1328438.1328486

Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper , Matthew Flatt, and Matthias Felleisen. Languages As Libraries. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2011. http://doi.acm.org/10.1145/1993498.1993514

Tajana Šimunić, Luca Benini, Giovanni De Micheli, and Mat Hans. Source Code Optimization and Profiling of Energy Consumption in Embedded Systems. In *Proc. of the International Symposium on System Synthesis (ISSS)*, 2000. http://dx.doi.org/10.1109/ISSS.2000.874049