Abstract
Contemporary compiler 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 profile-guided meta-programming in a general-purpose meta-programming system. Our design is parametric over the particular profiler and meta-programming system. We implement this design in two different meta-programming systems—the syntactic extension systems of Chez Scheme and Racket—and provide several profile-guided meta-programs as usability case studies.

Categories and Subject Descriptors D.3.3 [Programming Languages]: Language Constructs and Features; D.3.4 [Programming Languages]: Processors

General Terms Design, Performance, Languages

Keywords Optimization, profiling, profile-guided optimization, PGO, meta-programming

1. Introduction
Profile-guided optimization (PGO) is an optimization technique in which a compiler uses profile information gathered at run time to improve the performance of the generated code. The profile information acts as an oracle for run-time behavior. For example, a profiler might count how many times a program calls each function 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 [18]. Contemporary compiler systems that support PGO include GCC, .NET, and LLVM [26]. 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. [1] 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, i.e., PGOs on source programs. Meta-programs are programs that operate on programs. Languages with general-purpose meta-programming systems include C, C++, Haskell [32], Java [13], ML [35], OCaml [24], Racket [15], Scala [5], and Scheme [12]. Meta-programming is used to implement high-level yet efficient abstractions. Boost libraries [9] make heavy use of C++ meta-programming. Sujeeth et al. [33] and Rompf and Odersky [31] implement high-performance domain-specific languages using staged meta-programming in Scala. Chen et al. [7] implement process placement for SMP clusters using profile-guided meta-programming. Liu and Rus [27] provide tools that use profile information to identify suboptimal usage of the STL in C++ source code.

Current meta-programming systems do provide profile information to meta-programs. Therefore, to implement a profile-guided meta-program, programmers must introduce new specialized toolkits for profiling and meta-programming. Instead, meta-programming systems should provide access to profile information from existing profilers. Then programmers could reuse existing and familiar meta-programming and profiling tools.

This paper presents a design for supporting profile-guided meta-programming in general-purpose meta-programming systems. To demonstrate the generality of our design, we implement it for both Racket and Scheme. Both implementations reuse existing meta-programming and profiling infrastructure.

The rest 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.

The source code for our case studies and Racket implementation is available online.

2. A Running Example

We first introduce a simple syntax extension to familiarize readers with Scheme and Racket style meta-programming and to provide a running example. The transformation we present is not a meaningful optimization and is used only for illustrative purposes. The structure of this transformation strongly resembles the optimization we present in Section 6.1.

We first introduce a simple syntax extension to familiarize readers with Scheme and Racket style meta-programming and to provide a running example. The transformation we present is not a meaningful optimization and is used only for illustrative purposes. The structure of this transformation strongly resembles the optimization we present in Section 6.1.

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

    Figure 1: Example syntax extension

In Figure 1 define-syntax introduces a new syntax extension if-r (for reordering if). A syntax extension can be thought of as a function from source expressions to source expressions. The compiler rewrites any uses of if-r using the code in the body of the extension.

When used at the bottom of 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 and syntax-case performs pattern matching on syntax objects.

The syntax extension if-r expands at compile time, while the resulting if expression runs at run time. At compile time, the if-r expression uses profile-query to look up the profile information attached to each branch. Using this profile information, the if-r expression 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, the if-r expression generates an if expression by negating the test

3. Design

Profile-guided meta-programming requires that the underlying language comes with a profiling system and that the meta-programming system can associate profile information with source expressions. This section presents the abstractions introduced by our design and sketches an API that suffices to support profile-guided metaprogramming. For simplicity, our explanations refer to counter-based profiling. Our design should work for other point profiling systems, but does not extend to path profiling.

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 overhead.

For fine-grained profiling, each node in the AST of a program can be associated with a unique profile point. In the case of our running example, the AST nodes 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 different 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-programs 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 Weights

Our design introduces profile weights as an abstraction of the profile information provided by the underlying profiling system. Profile weights serve two purposes.

First, a profile weight provides a single value identifying the relative importance a profile point. The profile weight is represented as a number in the range [0,1]. The profile weight of a profile point

1 http://dx.doi.org/10.5281/zenodo.16784
2 Specifically, these forms implement Lisp’s quote, quasiquote, and unquote on syntax objects instead of lists.
is the ratio of the counter for that profile point to the counter of the most executed profile point in the same data set. Second, profile weights simplify merging multiple profile data sets. Multiple data sets are important to ensure PGOs can optimize for multiple classes of inputs expected in production. However, absolute profile information is generally incomparable across different data sets. On the other hand, merging the profile weights computed from multiple data sets is straightforward—the computation is essentially a weighted average across the data sets.

\[
\text{profile weight} = \frac{\text{counter for profile point}}{\text{counter for most executed profile point}}
\]

\[
\text{profile weight} = \frac{5/10}{10/10} = 0.5
\]

\[
\text{profile weight} = \frac{0.5 + 100/100}{2} = 0.75
\]

\[
\text{profile weight} = (1 + 10/10)/2 = 0.55
\]

Figure 3: Example profile weight computations

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, while in the second data set, (flag email 'important) runs 100 times and (flag email 'spam) runs 10 times. Figure 3 shows the resulting profile weights and how to merge the profile weights of these two data sets.

3.3 API

This section presents an example of an API that implements our design. We assume an object, (current-profile-information), exists in the meta-programming system. Figure 4 documents the methods of this object. The API assumes that the underlying profiler has some way to profile expressions that are associated with profile points. The API is concerned only with interfacing meta-programs and the profiler. The type SyntaxObject stands for the type of source expressions on which meta-programs operate.

4. Implementations

To validate the design principles from Section 3, we provide two implementations in Chez Scheme and Racket and discusses some implementation concerns. 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 [2, 4]. The Chez Scheme profiler effectively profiles every source expression and provides profiles in terms of source-code locations.

In Chez Scheme, we implement profile points using source objects [12] which can be attached to syntax objects. Chez Scheme uses source objects to report errors at their precise source location. 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 provides an API to programmatically manipulate source objects and attach them to syntax objects [11, 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 meta-programs.

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 profiles only 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. Racket provides an API for attaching source information when building a new syntax object. A separate library exists that 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 profiles only 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 different performance characteristics while profiling, it does not change the counters used to calculate profile weights.
We implement a library that maintains the associative map from source locations to profile weights. 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 a user-level library due to Racket’s advanced meta-programming facilities and the extensive API provided by the errortrace 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. The different low-level code would invalidate the low-level profile information. To solve this problem, the source code is compiled three times in a specific order, instead of the usual two times. Doing so ensures that 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. After running this instrumented program, we get the profile weights for the basic blocks generated from the optimized source program. These block-level profile weights will not be invalidated as long as we continue to optimize using the source-level profile weights. That is, the high-level code generated by the meta-program optimizations, e.g., Figure 2 will remain stable as long as we continue to optimize using the source profile weights. Because the generated high-level code remains stable, the generated low-level code also remains stable. Third (the final compilation), 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 run-time 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 amortized constant-time. Since they run at compile time, a profile-guided meta-program might slow down or speed up compilation, depending on the complexity of the metaprogram and whether it produces more or less code as a result of the optimization.

The API does not directly introduce run-time overhead; however, a meta-programming system using our technique inherits overhead from the profiler used in the implementation. Previous work measured about 9% run-time overhead introduced by the Chez Scheme profiler [4]. According to the errortrace documentation, the profiler introduces a factor of 4 to 12 slowdown. This does not include the additional instrumentation our implementation of annotate-expr performs, i.e., wrapping each annotated expression in a function call. Typically, profiling is disabled for production runs of a program, so this overhead affects only profiled runs.

### 5. Beyond Scheme and Racket

Our design should work in most meta-programming systems. Languages such as Template Haskell [32], MetaOCaml [24], and Scala [29] 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 [32] adds general-purpose meta-programming to Haskell, and comes with the current version of the Glasgow Haskell Compiler (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:

```haskell
{-# 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 [24] 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 is stored to a file outside of the language.

#### 5.3 Scala

Scala features powerful general-purpose meta-programming [5], multi-stage programming [31], 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 [17, 21]. In 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 [27].

#### 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 branches according to which branch is most likely to succeed.
In this section, we describe a similar optimization for Scheme and Racket case expressions. The implementation is straightforward and just 81 lines long. More importantly, it is not baked into the compiler and can be adapted to other forms of conditional expressions without changes to the underlying compiler. 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 does not support an else clause and assumes no constant appears in the left-hand side of more than one clause. Figure 5 shows an example case expression. Since ... is a literal expression used by syntax-case and syntax templates to indicate a sequence of elements, we use ... to indicate elided code.

Figure 6 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 implementation of case in Racket is 50 lines long, not including the implementation of exclusive-cond.

Figure 7 shows the implementation of the exclusive-cond expression. This is a multi-way conditional branch similar to Lisp’s 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 8 shows the code generated after expanding case and then after expanding exclusive-cond in the example case expression in Figure 5. 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.

Figure 5: An example using case

```scheme
(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 6: Implementation of case

```scheme
(define-syntax (exclusive-cond syn)
  ; Internal definitions
  (define (rewrite-clause key-expr clause)
    (syntax-case clause ()
      [(key-in? #,key-expr)
        (map (curry rewrite-clause #\key-expr)
             #\(clause ...)))]
      [(_ clause ...)]
      [(_ clause ...)])
      [
        (cond #,@(sort-clause #\key-expr)
              #\(clause ...)))])
      [(_ clause ...)])
      [(_ clause ...)])
  )
  ; into an exclusive-cond clause
  #,@ (map (curry rewrite-clause #\key-expr)
            #\(clause ...)))])
)
```

Figure 7: Implementation of exclusive-cond

```scheme
(define (parse stream)
  (exclusive-cond
    [(key-in? #,key-expr)
      (map (curry rewrite-clause #\key-expr)
           #\(clause ...)))]
    [(_ clause ...)])

  ; after case expands
  (let ([(t (peek-char stream))])
    (exclusive-cond
      [(key-in? #\space #\tab) (white-space stream)]
      [(key-in? t '\0123456789) (digit stream)]
      [(key-in? t '\#\{}) (start-paren stream)]
      [(key-in? t '\#\})] (end-paren stream)]
      ...)))
      ...
    )
)
```

Figure 8: Generated code from Figure 5

---

3 The implementation available online handles the full generality of Scheme’s case
6.2 Profile-Guided Receiver Class Prediction

Profile-guided receiver class prediction [17, 21] 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 prediction. 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.

Figure 9 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. That is, each method call site is instrumented by generating a multi-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 10 shows an example code snippet using this object system. Figure 11 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 most likely order.

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.

Figure 10: Example of profile-guided receiver class prediction

Figure 11: Generated code from Figure 10

Figure 12: 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 [27], 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 wrappers use generated profile point to separately profile each instance of the data structures. Finally, we develop a sequence...
The key difference is we conditionally generate wrapped versions of the list constructor. The code follows the same pattern as the profiled list. At compile-time, based on profile information, that will automatically specialize each instance to a list or vector, depending on the profile information.

Figure 9: Implementation of profile-guided receiver class prediction

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 13 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.

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 14 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

In Section 1, we briefly discussed some related work in the areas of profile-guided optimization and profile-guided meta-programming. In this section, we relate in more detail to work on PGO and meta-programming.

7.1 Profile-Guided Optimizations

Compiler systems such as GCC, .NET, and LLVM [26] 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 [6]. 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 to optimize switch statement, similar to our optimization of case expressions in Section 6.1. Our design can express this optimization at the user-level via the same profiler machinery used in our other case studies.

LLVM takes a different approach to PGO. LLVM uses a run-time reoptimizer that monitors the running program. The run-time system can profile the program “in the field”. This run-time system can perform simple optimizations on the machine code during program execution. More complex optimization require running an offline optimizer on the LLVM bytecode [25]. Burger and Dybvig [4] develop a similar run-time recompilation mechanism that allows simple optimizations to be performed at run time (during garbage collection) but does not support source-level PGO.


7.2 Meta-Program Optimizations

Meta-programming combines the ability to provide high-levels of abstraction while producing efficient code. For instance, meta-programming has been widely used to implement high performance domain-specific languages [31, 33, 34]. Others use meta-

(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 9: 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 13: Implementation of profiled list

(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 14: Implementation of profiled sequence

programming to implement whole general-purpose languages [3, 30, 36]. Keep and Dybvig [23] develop a production-quality compiler generator via Scheme meta-programming. Tobin-Hochstadt et al. [37] implement the optimizer for the Typed Racket language as a meta-program. Farmer et al. [14] develop the HERMIT toolkit, which provides an API for performing program transformations on Haskell intermediate code before compiling and even allows interactive experimentation. Hawkins et al. [19, 20] 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. [7] perform process placement for SMP clusters via profile-guided meta-programming. Šimunić et al. [38] optimize source code using energy profiles, although the bulk of the optimization requires programmer intervention. Karuri et al. [22] 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 with 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 ex-
pressivity of this design by using it to implement several representative profile-guided meta-programs.

Acknowledgments
We thank our anonymous reviewers for their thoughtful questions and feedback. The first author is grateful to Matthias Felleisen, Benjamin Greenman, and Max New, who provided valuable feedback on the presentation of this work.

References


