Profile-Guided Meta-Programming

William J. Bowman
Northeastern University
wilbowma@ccs.neu.edu

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 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. General-purpose meta-programming systems should provide access to profile information so meta-programmers can write their own, potentially domain-specific optimizations. Existing profile-guided meta-programs come with their own special-purpose toolkits, specific to a single optimization, limiting use and development of new profile-guided meta-programs.

We propose an approach for supporting profile-guided meta-programs in a general-purpose meta-programming system. Our approach is parametric over the particular profiling technique and meta-programming system. We demonstrate our approach by implementing it in two different meta-programming system: the syntactic extensions systems of Chez Scheme and Racket.

1. Introduction

Profile-guided optimization (PGO) is a compiler technique in which a compiler uses profile information, e.g., counts of how often each expression is executed, gathered from test runs on representative sets of inputs to generate more efficient code. The profile information acts as an oracle for runtime behavior, and informs low-level optimization decisions, e.g., decisions about reordering basic blocks, inlining, reordering conditional branches, and function layout in memory (Gupta et al. 2002). The resulting code usually exhibits improved performance, at least on the represented class of inputs, compared to code generated with static optimization heuristics. For example, using profiling information to inform inlining decisions in Java resulted in up to 59% improvement over static heuristics (Arnold et al. 2000). Modern compilers that support PGO include .NET, GCC, and LLVM (Lattner 2002).

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

Existing general-purpose meta-programming systems do not provide profile information about the input programs on which meta-programs operate. Therefore, existing profile-guided meta-programs introduce new special-purpose toolkits for profiling and meta-programming. Instead, general-purpose meta-programming systems should provide access to profile information. Meta-programmers could then implement profile-guided meta-programs while reusing the meta-

* Author's current affiliation: VMware, Inc

programming and profiling tools of an existing, familiar, system. Programmers could then take advantage of all the meta-programs implemented in that system.

We propose an approach for supporting profile-guided meta-programming in a general-purpose meta-programming system. The approach provides a simple API through which meta-programs can access fine-grain source-level profile information, and does not interfere with traditional, i.e., "low-level" PGOs.

We implement this approach in Chez Scheme using standard and efficient block-level profiling techniques (Ball and R. Larus 1994; G Burger and Kent Dybvig 1998). We also implement this approach in Racket (Flatt and PLT 2010) purely as a library, using pre-existing profiling and meta-programming tools.

The remainder of the paper is organized as follows. In section 2, we present a running example and introduce Scheme-style meta-programming. In section 3, we present the design of our approach and an example of an API provided by a meta-programming system using our approach. In section 4, we present two instantiations of our approach, one in Chez Scheme and one in Racket. In section 5, we demonstrate that our approach is general enough to implement and extend existing PGOs and profile-guided meta-programs. In section 6, we related to existing work on PGOs and profile-guided meta-programming.

The main contributions of the paper are:

- A general approach for profile-guided meta-programming.
- Two instantiations of our approach
- An evaluation of our approach based on implementing three existing profile-guided meta-programs.

2. A running example

We first introduce a Scheme syntax extension which we use as a running example and to familiarize readers with Scheme and Racket style meta-programming.

```
(define-syntax (if-r stx)
  (syntax-case stx ()
    [(if-r test t-branch f-branch)
     ; Rewrites to
     (let ([t-prof (profile-query #'t-branch)]
           [f-prof (profile-query #'f-branch)])
       (if (< t-prof f-prof)
           ; if false branch is more frequent:
           #'(if (not test) f-branch t-branch)
           ; if true branch is more frequent:
           #'(if test t-branch f-branch)))]))
```

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. For example, in figure 2 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.

```
; Source code
(define (classify email)
  (if-r (subject-contains email "PLDI")
        (flag email 'important) ; Run 5 times
        (flag email 'spam)))    ; Run 10 times

; Code generated from expanding syntax extension
(define (classify email)
  (if (not (subject-contains email "PLDI"))
      (flag email 'spam)
      (flag email 'important)))
```

Figure 2: Using `if-r`

At compile time, `if-r` looks up the profile information attached to each branch, using `profile-query`, and generates an `if` expression with the branch of the `if` ordered based on which branch is more likely to be executed. This transformation is not a meaninful optimizaion and is only used for illustrative purposes. The syntax extension is expanded at compile-time, while the resulting `if` is run at runtime.

3. Approach and API

This section presents the design of our approach and presents an example of an API provided to meta-programmers by a meta-programming system using our approach. Our approach is not specific to a particular profiling technique, but for simplicity our explanations refer to counter-based profile information.

3.1 Profile points

To store and access profile information, we need to associate profile information with the source expressions on which meta-programs operate. *Profile points* uniquely identify expressions that should be profiled by the underlying profiling system. For instance, if two expressions are associated with the same profile point, then they both increment the same profile 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

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

not instrumented to collect profile information, profile points do not introduce any runtime overhead.

For fine-grained profiling, each input expression and subexpression can have a unique profile point. In the case of our running example, separate profile points are associated with `#'(if ...)`, `#'(subject-contains email "PLDI")`, `#'subject-contains`, `#'email`, `#'"PLDI"`, `#'(flag email 'spam)`, and so on. Note that `#'flag` and `#'email` appear multiple times, but each occurrence can have a unique profile point.

In addition to profile points associated with input expressions, meta-programs can also manufacture new profile points. Meta-programmers may want to generate expressions that are profiled separately from any other expression in the source program.

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

3.2 Profile Information

Absolute profile information, such as the exact execution count of an expression, is incomparable across different data sets. Multiple data sets are important to ensure PGOs can optimize for multiple classes of inputs expected in production. Instead, our approach considers *profile weights*. The profile weight of a profile point in a given data set is the ratio of the absolute profile information for that profile point to the maximum of all other profile points. The profile weight is represented as a number in the range [0,1]. In the case of counter-based profiling, the profile weight for a given profile point is execution count for that point divided by the the execution count for the most frequently executed point in the data set. This provides a single value identifying the relative importance of an expression and simplifies the combination of multiple profile data sets.

To demonstrate profile weights, consider the running example from figure 2. 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 3 shows the profile weights computed after each data set.

```
;; 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 3: Sample profile weight computations

3.3 Complete API sketch

Here we sketch an example of an API provided by a meta-programming system using our approach. The API assumes the underlying language implementation has some way to profile expressions that are associated with profile points. The API is only concerned with providing the meta-program with access that profile information and the ability to manipulate profile points.

To create profile points,

`(make-profile-point)`

generates profile points. This function must generate profile points deterministically so the meta-program can access the profile information of a generated profile point across multiple runs.

To attach profile points to expressions,

`(annotation-expr expr profile-point)`

takes an expression, such as a syntax object, and a profile point, and associates the expression with the profile point. The underlying profiling system should then profile that expression separately from any other expression with a different associated profile point.

To access profile information,

`(profile-query expr)`

takes an expression associated with a profile point, and returns the profile information associated with that profile point.

To store profile information from a sample run,

`(store-profile-info filename)`

takes a filename and stores the current profile information from the underlying profiling system to the file.

To load profile information from a previous run,

`(load-profile-info filename)`

takes a filename and loads the profile information from the file, making the profile information available via `profile-query`.

4. Implementations

In this section we describe the instantiations of our approach in Chez Scheme and in Racket, and briefly describe other meta-programming systems in which our approach should apply.

4.1 Chez Scheme implementation

Chez Scheme implements counter-based profiling. Adding a profile point for every single source expression requires care to instrument correctly and efficiently. As optimizations duplicate or throw out expressions, the profile points must not be duplicated or lost.

For every profile point, Chez Scheme generates an expression `(profile e)`, where `e` is an object representing that profile point. Chez Scheme treats these `profile` expressions as effectful, like an assignment to an external variable, and never duplicates or removes them. To instrument profiling efficiently, `profile` expressions are preserved until generating basic blocks. While generating basic blocks, all the profile points from `profile` expressions are attached to the basic block in which they appear. [2] Using techniques from G Burger and Kent Dybvig (1998), Chez Scheme generate at most one counter per block, and fewer in practice.

Chez Scheme implements profile points using *source objects* (Dybvig et al. 1993). Chez Scheme source objects contains a filename and starting and ending character positions. Source objects uniquely identify every source expression, providing fine-grain profile information. The Chez Scheme reader automatically creates and attaches these to each syntax object read from a file, using them to give error messages in terms of source locations, among other things. Chez Scheme also provides an API to programmatically manipulate source objects and attach them to syntax objects (Dybvig 2011 Chapter 11).

We generate a new source object by adding a suffix to the filename of a base source object. By basing generated source objects on those from the original source program, errors in generated code are easier to debug as the generated code contain source information from the file in which the code was generated. The meta-programming system maintains an associative map of source objects to profile weights, which is updated by API calls. The API provided by Chez Scheme is nearly identical to the one sketched in section 3.3.

4.1.1 Source and Block PGO

One goal of our approach is to complement rather than interfere 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 will change after meta-programs perform optimizations. Therefore, we need to instrument and perform source-level and basic block-level optimizations separately. We describe a workflow for using both source-level and basic block-level PGOs via the running example from figure 2.

To get both source-level and block-level optimizations, we first instrument profiling for source expressions. After running it on representative inputs, we get the profile weights such as in figure 3. Next, we recompile using that source profile information, and instrument profiling for basic blocks. The generated source code, figure 2, will remain stable as long as we continue to optimize using the source profile information. Since the generated source code remains stable, so do the generated basic blocks. Now we profile the basic blocks generated from the optimized source program. Finally, we use both the source profile information and the basic block profile information to do both profile-guided meta-programming and low-level PGOs.

[2] Chez Scheme reuses this infrastructure to profile basic blocks by generating a new profile point for each basic block.

4.2 Racket implementation

The Racket implementation uses a pre-existing Racket profiling library called `errortrace`. The `errortrace` library provides counter-based profiling, like the Chez Scheme profiler.

The Racket implementation uses source information attached to each syntax object to implement profile points. Racket attaches the filename, line number, etc to every syntax object, and provides functions for attaching source information when building a new syntax object. Our implementation provides wrappers to extract source information into separate source objects, and to merge source objects into Racket syntax objects. We then generate source objects in essentially the same way as in Chez Scheme.

We implement a library which provides a similar API to the one sketched in section 3.3. This library maintains the map from source objects to profile information and computes profile weights. This library is implemented as a standard Racket library that can be called by meta-programs, and requires no changes to either the Racket implementation or the `errortrace` library.

4.3 Instantiations in other meta-programming systems

Both of our instantiations are in similar Scheme-style meta-programming systems, but the approach can work in any sufficiently expressive meta-programming system.

Languages such as Template Haskell (Sheard and Peyton Jones 2002), MetaML (Taha and Sheard 2000), and Scala (Burmako 2013) feature powerful meta-programming facilities similar to that of Scheme (Dybvig et al. 1993). They allow executing expressive programs at compile-time, provide direct access to input expressions, and provide template-style meta-programming facilities. C++ template meta-programming is more restricted than the above systems, so it is not clear how to instantiate our approach for C++ templates.

5. Case Studies

In this section we evaluate the generality of our approach by implementing existing PGOs and profile-guided meta-programs in the Racket instantiation of our approach. We first demonstrate optimizing Scheme's `case`, a multi-way branching construct similar to C's `switch`, as a meta-program. Then we then implement profile-guided receiver class prediction (Grove et al. 1995; Hölzle and Ungar 1994) for an object system implemented as a syntax extension. Finally we implement a sequence datatype that specializes each instance to a `list` or `vector`, based on profiling information, automating the recommendations performed by tools like Perflint (Liu and Rus 2009).

5.1 Profile-guided conditional branch optimization

As our first case study, we perform profile-guided optimization of Scheme's `case` construct, which is similar to C's `switch` statement. 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. This case study shows that our approach is general enough to implement this optimization. The entire implementation is 65-line.

The `case` construct takes an expression `key-expr` and an arbitrary number of clauses, followed by an optional `else` clause. The left-hand side of each clause is a list of constants. `case` executes the right-hand side of the first clause in which `key-expr` is `equal?` to some element of the left-hand side. For simplicity, we present a version that assumes the left-hand sides are mutually exclusive and ignore the `else` clause[3]. Figure 4 shows an example `case` expression.

```
(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`

Figure 5 shows the profile-guided implementation of `case` that reorders branches according to which clause is most likely to succeed. First it parses each clause into a struct containing the list of keys from the left-hand side, and the expressions from the right-hand side. Then it generates an invocation of another meta-program, `exclusive-cond`, which reorders its branches based on profile information. Each clause of the generated `exclusive-cond` tests if the key expression is `equal?` to any key in the list of keys from the `case` clause. The full implementation of `case` is 41-line.

Figure 6, introduces the `exclusive-cond` construct, a multi-way conditional branch, like Lisp's `cond`[4], that expects all branches to be mutually exclusive. Because the branches are mutually exclusive, `exclusive-cond` can safely reorder them. `exclusive-cond` simplifies the implementation of `case`, and demonstrates an important feature of profile-guided meta-programming—meta-programming allows the programmer to encode their knowledge, e.g., that the branches of this conditional are mutually exclusive, in their program and take advantage of optimizations that would have otherwise been impossible.

The implementation of `exclusive-cond` parses each clause into a struct that contains the original clause and the profile information about how many times that branch was taken. Then it sorts the clauses by profile weight and generates a regular `cond`. The full implementation of `exclusive-cond` is 24-line.

Figure 7 shows the code generated from the example `case` expression from figure 4

```
; 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

This case study has demonstrated that our approach is general enough to implement a well-known PGO via meta-programming.

5.2 Profile-guided receiver class prediction

In this case study we provide a meta-program that implements profile-guided receiver class prediction (Grove et al. 1995; Hölzle and Ungar 1994) for a simplified object system implemented as a syntax extension. This demonstrates that our mechanism is both general enough to implement well-known PGOs, and powerful enough to provide domain-specific languages with PGOs not available in the host language. The full implementation of profile-guided receiver class prediction is 44-line, while the implementation of the entire object system (including receiver class prediction) is 129-line.

Figure 8 shows the implementation of profile-guided receiver class prediction. A method call such as `(method shape area)` is actually a meta-program that generates code as follows. First, it generates a new profile point for each class in the system. Then it attaches each profile point to a call to the dynamic dispatch routine. When profile data is not available, the implementation generates a `cond` expression which tests the class of the object and calls the dynamic dispatch routine, but *with a different profile point*

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

[4] The full implementation handles other `cond` syntaxes and `else` clauses.

```
(define-syntax (case syn)
 (struct case-clause (keys body))
 (define (parse-clause clause)
  (syntax-case clause ()
   [((k ...) e1 e2 ...)
    (make-case-clause #'(k ...) #'(e1 e2 ...))]))
 (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
        ; Parse clauses and splice a list of generated tests
        ; into exclusive-cond
        #,@(for/list ([clause (map parse-clause #'(clause ...))])
           ; Take this branch if the key expression is equal? to some
           ; key in the list of keys for this branch.
           #`((key-in? t '#,(case-clause-keys clause))
              #,@(case-clause-body clause)))))]))
```

Figure 5: Implementation of `case`

```
(define-syntax (exclusive-cond syn)
  (struct clause (syn weight))
  (define (parse-clause clause)
    (syntax-case clause ()
      [(test e1 e2 ...) (make-clause clause (profile-query #'e1))]))
  (define (sort-clauses clause*)
    (sort (map parse-clause clause*) > #:key clause-weight))
  (syntax-case x (else)
    [(_ clause ...)
     #`(cond #,@(map clause-syn (sort-clauses #'(clause ...))))]))
```

Figure 6: Implementation of `exclusive-cond`

for each branch.[5] 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 that tests for the most frequently used classes at this method call site, and inlines those methods—that is, it performs polymorphic inline caching using the profile information. The full implementation of profile-guided receiver class prediction is 44-line. The rest of the object system implementation is an additional 87-line.

Figure 9 shows an example method call, 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.

[5] 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.

This case study has demonstrated that our approach is general enough to implement a well-known PGO as a meta-program, and provide domain-specific languages with PGOs not available in host language.

5.3 Data Structure Specialization

In this case study we show that our approach is general enough to implement 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 then go beyond recommendation and provide the programmer with a library that automatically specializes each instance of a data structure according to profile information. The full implementation of the list library is 80-line, the vector library is 88-line, and the sequence library is 111-line.

We provide implementations of lists and vectors that warn the programmer when a different representation may lead to asymptotic performance gains. The implementations provide wrappers around the standard list and vector functions, using newly generated profile point to separately profile each instance of the data structures. Finally, we provide an im-

```
(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 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

```
(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))

; -------------------
; After instrumentation
...
(let* ([x c])
  (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)]))

; -------------------
; After optimization
...
(let* ([x c])
  (cond
    [(instance-of? x 'Square)  ; Run 1 time
     (let ([this x]) (sqr (field x length)))]
    [(instance-of? x 'Circle)  ; Run 3 times
     (let ([this x]) (* pi (sqr (field x radius))))]
    [else (dynamic-dispatch x area)]))
```

Figure 9: Example of profile-guided receiver class prediction

```
; -------------------
; After optimization
...
(let* ([x c])
  (exclusive-cond
    [(instance-of? x 'Square)  ; Run 1 time
     (let ([this x]) (sqr (field x length)))]
    [(instance-of? x 'Circle)  ; Run 3 times
     (let ([this x]) (* pi (sqr (field x radius))))]
    [else (dynamic-dispatch x area)]))

; -------------------
; After more optimization
...
(let* ([x c])
  (cond
    [(instance-of? x 'Circle)  ; Run 3 times
     (let ([this x]) (* pi (sqr (field x radius))))]
    [(instance-of? x 'Square)  ; Run 1 time
     (let ([this x]) (sqr (field x length)))]
    [else (dynamic-dispatch x area)]))
```

Figure 10: Profile-guided receiver class prediction, sorted.

plementation of a sequence datatype that will automatically specialize to a list or vector based on profiling information. As this is done via a library, the programmer can easily opt-in to such automated high-level changes without changing their code.

Figure 11 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 profiled calls to the underlying list operations. The profiled list constructor generates these profiled operations by

```
(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 11: Implementation of profiled `list`

wrapping the underlying list operations with the appropriate profile point. It 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 profiled 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 existing as a separate tool outside the language, we can provide libraries to the programmer that automatically follows these recommendations rather than asking the programmer to change their code. To demonstrate this, 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 12 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.

This case study has demonstrated that our approach is general enough to implement tools that provide recommendations that lead to asymptotic improvements to source code, and powerful enough to improve upon such tools by allowing the programmer to opt-in to automated changes without changing their source code.

6. Related Work

6.1 Profile-guided optimizations

Modern compilers 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). In addition to the common optimizations noted previously, .NET extends their profiling system to probe values in `switch` statements. They can use this value information to reorder the branches of a `switch` statement, as done for `case` in section 5.1.

LLVM has a different model for PGO. LLVM uses a runtime reoptimizer that monitors the running program. The runtime can profile the program as it runs "in the field" and perform simple optimizations to the machine code, or call to an offline optimizer for more complex optimizations on the LLVM bytecode (Lattner and Adve 2004).

Recent work is still finding novel uses for profile information. Furr et al. (2009) use profile information to infer types in dynamic languages to assist in debugging. Chen

```
(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 12: Implementation of profiled sequence

et al. (2006b) use profile information to reorganize the heap and optimize garbage collection. Luk et al. (2002) use profile information to guide data prefetching. Debray and Evans (2002) use profile information to compress infrequently executed code on memory constrained systems.

6.2 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 (K. Sujeeth et al. 2013; K. Sujeeth et al. 2014; Rompf and Odersky 2010), whole general purpose languages (Barzilay and Clements 2005; Rafkind and Flatt 2012; Tobin-Hochstadt and Felleisen 2008), and production-quality compiler generators (W. Keep and Kent 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 integrates profiling to guide meta-program optimizations. Chen et al. (2006a) use profile-guided meta-programming for performing process placement for SMP clusters. Šimunić et al. (2000) use profile-guided meta-programming to optimize the energy usage of embedded programs. Karuri et al. (2005) use fine-grained source profile information to optimize ASIP designs.

However, existing work introduces new toolkits for profiling and meta-programming. We provide a single, general-purpose approach in which we can implement new general-purpose languages, domain-specific languages abstract libraries, and arbitrary meta-programs, all taking 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.

7. 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 optimization. We have described a general mechanism for implementing arbitrary profile-guided meta-programs, and evaluated its use by implementing several PGOs and profile-guided meta-programs in a single general-purpose system.

Bibliography

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

Thomas Ball and James R. Larus. Optimally Profiling and Tracing Programs. *ACM Trans. Program. Lang. Syst.* 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. The 2005 Workshop on Functional and Declarative Programming in Education*, 2005. http://doi.acm.org/10.1145/1085114.1085118

Eugene Burmako. Scala Macros: Let Our Powers Combine!: On How Rich Syntax and Static Types Work with Metaprogramming. In *Proc. Proceedings of the 4th Workshop on 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. Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, 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. Proceedings of the 20th Annual International Conference on Supercomputing*, 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. The 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 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. The ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, 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://pdf.aminer.org/001/006/789/syntactic_abstraction_in_scheme.pdf`

Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. SugarJ: Library-based Syntactic Language Extensibility. In *Proc. Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, 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. Proceedings of the 2012 Haskell Symposium*, 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. The 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, 2009. `http://doi.acm.org/10.1145/1640089.1640110`

Robert G Burger and R Kent Dybvig. An infrastructure for profile-driven dynamic recompilation. In *Proc. Computer Languages, 1998. Proceedings. 1998 International Conference on*, 1998.

David Grove, Jeffrey Dean, Charles Garrett, and Craig Chambers. Profile-guided receiver class prediction. In *Proc. The tenth annual conference on Object-oriented programming systems, languages, and applications*, 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. Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 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. Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 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. Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, 1994. `http://doi.acm.org/10.1145/178243.178478`

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. The 12th International Conference on Generative Programming: Concepts \&\#38; Experiences*, 2013. `http://doi.acm.org/10.1145/2517208.2517220`

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 Trans. Embed. Comput. Syst.* 13(4s), 2014. `http://doi.acm.org/10.1145/2584665`

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. Design Automation Conference, 2005. Proceedings. 42nd*, 2005.

Chris Lattner and Vikram Adve. LLVM: a compilation framework for lifelong program analysis transformation. In *Proc. Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, 2004.

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. Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, 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. The 16th International Conference on Supercomputing*, 2002. `http://doi.acm.org/10.1145/514191.514217`

Jon Rafkind and Matthew Flatt. Honu: Syntactic Extension for Algebraic Notation Through Enforestation. In *Proc. The 11th International Conference on Generative Programming and Component Engineering*, 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. Proceedings of the Ninth International Conference on Generative Programming and Component Engineering*, 2010. http://doi.acm.org/10.1145/1868294.1868314

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

Walid Taha and Tim Sheard. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science* 248(1–2), pp. 211–242, 2000. http://www.sciencedirect.com/science/article/pii/S0304397500000530

Sam Tobin-Hochstadt and Matthias Felleisen. The Design and Implementation of Typed Scheme. In *Proc. The 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 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. The 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2011. http://doi.acm.org/10.1145/1993498.1993514

Andrew W. Keep and R. Kent Dybvig. A Nanopass Framework for Commercial Compiler Development. In *Proc. Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, 2013. http://doi.acm.org/10.1145/2500365.2500618

Tajana Šimunić, Luca Benini, Giovanni De Micheli, and Mat Hans. Source Code Optimization and Profiling of Energy Consumption in Embedded Systems. In *Proc. Proceedings of the 13th International Symposium on System Synthesis*, 2000. http://dx.doi.org/10.1145/501790.501831