Profile-Guided Meta-Programming

Anonymous

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-program come with their own special-purpose toolkits, specific to a single optimization, creating barriers to 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 run-time 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).

TODO: Not sure if I should cite .net and gcc documentation or not.

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 (Sheard and Jones 2002; 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-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.

TODO: Say something about profiling information being available for run-time decisions, in later sections

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 flavor 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. For example, the function in figure 2 uses `if-r`. The extension will receive 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 `#,` implement Lisp's quote, quasiquote, and unquote but on syntax instead of lists. Together they give us a templating system for syntax. `syntax-case` performs pattern matching on syntax objects.

At compile time, `if-r` it looks up the profile information attached to each branch, reorders the branches based on which is more likely to be executed. It uses `profile-query` to access the profile information for each branch and generate an `if` expression. 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` will be run at run-time.

```
; 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`

3. Approach and API

This section presents the high-level design of our approach and presents a sketch of the API provided to meta-programmers. Our approach is not specific to a particular profile technique, but for simplicity our explinations 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 profiler. 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 source objects, then they increment different profile counters when executed.

For fine-grain 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 occurance 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 profile generated expressions separately from any other expression in the source program.

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

3.2 Profile Information

Absolute profile information such as exact counts are 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 API provides *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 point, represented as a number in the range [0,1]. In the case of counters, the profile weight for a given profile point is count for that point divided by the the count for the most frequently executed profile 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.

TODO: Still want to say something about percent-of-max vs percent-of-total and percent-of-average

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 the API.

To create profile points,
`(make-profile-point)`
generates profile points deterministically, to ensure profile information of generated profile points from previous runs is accessible.

To attach profile points to generated 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.

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 a run,
`(store-profile-info filename)`
takes a filename and stores profile information from the profiling system to file.

To load profile information a run,
`(load-profile-info filename)`
takes a filename and loads profile information from the file so the its accessible by `profile-query`.

4. Implementations

In this section we describe the instantiations of our approach in Chez Scheme and in Racket, and briefly desc

4.1 Chez Scheme implementation

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

For every profile point Chez Scheme generates an expression `(profile src)`, where `src` is the source object for that profile point, treats this `profile` expression as an effectful expression that must not be removed or duplicated. To instrument profiling efficiently, `profile` expressions are preserved until generating basic blocks. While generating basic blocks, all the source objects from the profile expressions can be attached to the basic block in which they appear. [1] Using techniques from Burger and Dybvig (1998), we generate at most one counter per block, and fewer in practice.

Chez Scheme implements profile points using *source objects* (Dybvig et al. 1993) to uniquely identify profile points. Chez Scheme source objects contains a file name 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, e.g., to give error messages in terms of source locations. Chez Scheme also provides an API to programmatically manipulate source objects and attach them to syntax (Dybvig 2011 Chapter 11).

We generate a new source objects by adding a suffix to the file name of a base source object. By basing generated source objects on source objects from the original source program, errors in generated code are easier to debug as the generated code contains source file information from the meta-program that generated the code. The meta-programming system's runtime maintains an associative map of source objects to profile weights which is updated by API calls. The API provide by Chez Scheme 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

[1] We reuse this infrastructure to profile basic blocks by generating a new profile point for each basic block.

low-level representation will change after meta-programs perform optimizations. Therefore we need to instrument and perform source and basic block-level optimizations separately. We describe a workflow for our approach via the running example from figure 2.

TODO: Lots of 'we'

To get both source-level and block-level optimizations, we first instrument the program for source profiling. 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 profile-guided optimizations via meta-programming and traditional low-level optimizations

4.2 Racket implementation

The Racket implementation uses a pre-existing Racket profiling implementation call `errortrace`. The `errortrace` library provides exact profile counters, like the Chez Scheme profiler.

The Racket implemention implements profile points by using source information attached to each syntax object. Racket attaches the file name, 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 computers 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 instantiations of our approach are in similar Scheme-style meta-programming systems, but the approach can work in any sufficiently expression meta-programming system.

Template Haskell (Sheard and Jones 2002), MetaML (Taha and Sheard 2000), MetaOCaml (Czarnecki et al. 2004), and Scala (Burmako 2013) all 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 similar to Scheme. 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 profile-guided meta-programs in each of our instantiations 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-oriented DSL. Finally we implement a sequence datatype that specializaes each instannce 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 conditional branch optimization for Scheme's `case` construct, which is similar 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 profile-guided reordering `switch` statements to check the most common cases first. This case study shows that our approach can be used to easily implement this optimization.

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. For simplicity, we assume the left-hand sides are mutually exclusive and ignore the `else` clause[2]. 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 tries the most executed clauses first. 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. The full implementation of `case` is 41-line.

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

```
(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 ...) (profile-query #'e1))]))
 (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 ...))])
            ; Transform each case clause into branch that tests if the
            ; key expression, t, is in the list of keys for the clause
            #`((in-list? t '#,(case-clause-keys clause))
               #,@(case-clause-body clause)))))]))
```

Figure 5: Implementation of `case`

Figure 6, introduces the `exclusive-cond` construct, a multi-way conditional branch, like Lisp's `cond`[3], but expects all branches to be mutually exclusive. Because the branches are mutually exclusive we can safely reorder the clauses to execute the most likely clauses first. `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.

First the implementation of `exclusive-cond` parses each clause into a struct that contains the original clause and the profile information. Then it sorts the clauses by 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
     [(in-list? t '(#\space #\tab))
      (white-space stream)]
     [(in-list? t (0 1 2 3 4 5 6 7 8 9)) (digit stream)]
     [(in-list? t (#\()) (start-paren stream)]
     [(in-list? t (#\))) (end-paren stream)])))

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

Figure 7: Generated code from figure 4

5.2 Profile-guided receiver class prediction

We provide a meta-program that implements profile-guided receiver class prediction (Grove et al. 1995; Hölzle and Ungar 1994) for a simplfied object system implemented as a syntax extension. This case study demonstrates that our mechanism is both general enough to implement well-known profile-guided optimizations, and powerful enough to provide DSL writers with standard PGOs.

Figure 8 shows the key parts of the implementation of recevier 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 source object for each class in the system. Then for each of these newly generated source objects, it instruments a call to the dynamic dispatch routine. When profile data is not available, the implemation generates a `cond` expression which tests the class of the object and calls the dynamic dispatch routine, but *with a different profile point for each branch.*[4] That is, method call site is instrumented by generating a multi-way branch to the same dynamic dispatch routine, but with separate profile points in each branch. When profiling information is

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

[4] A production implementation would create a table of instrumented dynamic dispatch calls and dynamically dispatch through this table, instead of instrumenting code with `cond`, but this complicates vizualizing ths instrumentation.

```
(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 > #:key clause-weight (map parse-clause clause*)))
  (syntax-case x (else)
    [(_ clause ...)
     #'(cond #,@(map clause-syn (sort-clauses #'(clause ...))))]))
```

Figure 6: Implementation of `exclusive-cond`

available, we expand into a `cond` expression that tests for the most frequently used classes at this method call site, and inlines thost methods—that is, we perform polymorphic inline caching using the profile information. Otherwise we fall back to dynamic dispatch. The full implementation of profile-guided receiver class prediction is 44-line of code. The rest of the DSL implementation is an additional 82-line.

TODO: Maybe implement the instrumented hash table later

Figure 9 shows an example method call, the resulting code after instrumentation, and the resulting code after optimization. Note that the each occurences of `(instrumented-dispatch x area)` has a different source objects, so they are each profiled separately.

We have demonstrated that our approach can easily implement a well known profile-guided optimization as a meta-program. As a further improvement, we could reuse `exclusive-cond` to test for the most likely class first.

5.3 Data Structure Specialization

So far we've demonstrated that our technique support traditional profile-guided optmizations via meta-programming. However, meta-programs have access to high-level information such as different algorithms or data structures might be cause an asymptotic speed up (Liu and Rus 2009). In this case study we show our approach is general enough to implement this kind of profiling tool, and even go beyond it by automating the recommendations.

We provide implementations of lists and vectors[5] 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 source objects to separately profile the uses of each instance of the data structures. Finally, we provide an implementation 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.

[5] Vectors are essentially arrays in Scheme.

```
(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
    [(class-equal? x 'Square)    ; Run 1 time
     (instrumented-dispatch x area)]
    [(class-equal? x 'Circle)    ; Run 3 times
     (instrumented-dispatch x area)]
    [(class-equal? x 'Triangle)  ; Run 0 times
     (instrumented-dispatch x area)]
    [else (dynamic-dispatch x area)]))

; ------------------
; After optimization
...
(let* ([x c])
  (cond
    [(class-equal? x 'Square)  ; Run 1 time
     (let ([this x]) (sqr (field x length)))]
    [(class-equal? 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

```
(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])
                    #'((class-equal? x #,class) (#,d x)))
                  ; If profile data, inline up the top inline-limit classes
                  ; with non-zero weights
                  (for/list ([class (take sorted-classes inline-limit)])
                    #'((class-equal? 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 source objects.
    ; Use list-src to profiles operations that are asymptotically fast on lists
    ; Use vector-src profiles operations that are asymptotically fast on vectors
    (define list-src (make-source-obj syn))
    (define vector-src (make-source-obj syn))
    ...
    (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" x))
      #'(make-list-rep
          ; Build 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`

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 simply wrapping the underlying list operations with the appropriate source object source objects. It generates two source objects for each list. One is used to profile operations that are asymptotically fast for lists and the other is used to profile operations that are asymptotically fast for vectors. Finally, the library exports new versions of the list operations that work on profiled list representation. For instance, it exports `car`, which a `profiled-list`, and uses the profiled call `car` from the hash table of the profiled list on the underlying list. When profiling information already exists, for instance, after a profiled run, this list constructor emits a warning (at compile time) if the list fast vector operations are more common than fast list operations.

We also provide an analogous implementation of vectors. Our technique would also scale to the other data structures analyzed by Perflint. Because our meta programs are integrated into the language, rather than existing as a separate

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

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

tool in front of the compiler, 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 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.

6. Related Work

TODO: Add links foisalhg fufstmaress to all references.

6.1 Profile-guided optimizations

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

GCC profiles at the level of an internal control-flow graph (CFG). To maintain a consistent CFGs across instrumented and optimizated 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 optimize `switch` branches, similar to the implementation of `case` we presented 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.

Recent work is still finding novel uses for profile information. Furr et al. (2009) present a system for inferring types in dynamic languages to assist in debugging. Chen 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 DSLs (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. (2012) implement a compiler for a language that generates C++ implementations of data structures based on high-level specifications.

Previous works 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-programing 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 introduce new toolkits for profiling and meta-programming. We provide a single, general-purpose mechanism in which we can implement new languages, DSLs, 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 being used to implement high-level optimizations, generate code from high-level specifications, and create DSLs. 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 toolchains designed for a specific meta-program or optimization. We have described a general mechanism for implementing arbitrary profile-guided meta-program optimizations, and demonstrated its use by implementing several

```
(struct seq-rep (instr-op-table s))
...
(define-syntax (seq syn)
   (define list-src (make-source-obj syn))
   (define vector-src (make-source-obj syn))
   (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 x ()
     [(_ 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

optimizations previously implemented in seperate, specialized toolchains.

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 Transactions on Programming Languages and Systems (TOPLAS)* 16(4), pp. 1319–1360, 1994.

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

Robert G. Burger and R. Kent Dybvig. An infrastructure for profile-driven dynamic recompilation. In *Proc. International Conference on Computer Languages, 1998.*, pp. 240–249, 1998. http://pdf.aminer.org/000/289/483/an_infrastructure_for_profile_driven_dynamic_recompilation.pdf

Eugene Burmako. Scala Macros: Let Our Powers Combine! In *Proc. of the 4th Annual Scala Workshop*, 2013.

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. Annual IEEE/ACM international symposium on Code generation and optimization*, 8, pp. 42–52, 2010. http://hpc.cs.tsinghua.edu.cn/research/cluster/papers_cwg/tamingsample.pdf

Hu Chen, Wenguang Chen, Jian Huang, Bob Robert , and Harold Kuhn. MPIPP: an automatic profile-guided parallel process placement toolset for SMP clusters and multiclusters. In *Proc. 20th Annual International Conference on Supercomputing*, 2006a.

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

Krzysztof Czarnecki, John T O'Donnell, Jörg Striegntiz, and Walid Taha. DSL implementation in MetaOCaml, Template Haskell, and C++. In *Proc. Domain-Specific Program Generation* volume Springer Berlin Heidelberg., pp. 51–72, 2004. http://camlunity.ru/swap/Library/ComputerScience/Metaprogramming/Domain-SpecificLanguages/DSLImplementationinMetaOCaml,TemplateHaskellandC++.pdf

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. of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pp. 391–406, 2011. http://www.informatik.uni-marburg.de/~seba/publications/sugarj.pdf

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. ACM SIGPLAN Notices*, 2012.

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

R. Gupta, E. Mehofer, and Y. Zhang. *Profile Guided Code Optimization*. 2002.

Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin Rinard,, and Mooly Sagiv. Concurrent data representation synthesis. In *Proc. 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. ACM SIGPLAN Notices*, 1994.

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 Authors 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. The 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2009.

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.

Time Sheard and Simon Peyton Jones. Template meta-programming for Haskell. In *Proc. ACM SIGPLAN workshop on Haskell*, 2002. http://research.microsoft.com/en-us/um/people/simonpj/Papers/meta-haskell/meta-haskell.pdf

Walid Taha and Time Sheard. MetaML and multi-stage programming with explicit annotations . *Theoretical Computer Science* 248((1 2)), pp. 211–242, 2000. http://www.cs.rice.edu/~taha/publications/journal/tcs00.pdf

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. The 18th ACM SIGPLAN international conference on Functional programming*, 2013.

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.