Profile-guided meta-program optimization

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

Swaha Miller
swamille@cisco.com
Cisco Systems, Inc

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

Abstract

Many contemporary compilers allow the use of profile information to guide various low-level optimizations. This is not the case for contemporary meta-programming systems, although profile information can have an even greater impact on the high-level optimizations performed by meta-programs. For example, a meta-program sometimes has control over the data structures and algorithms used by the generated code, and use of profiling information to select appropriate data structures and algorithms can potentially lead even to asymptotic improvements in performance.

This paper describes a general-purpose mechanism for supporting arbitrary profile-guided meta-program optimization. It makes profile information available at the granularity of arbitrary source points identified by the meta-program, while making use of standard and efficient block-level profile-instrumentation techniques. We have implemented the mechanism as part of Chez Scheme, with profile information made available via the syntactic abstraction facility through which Scheme supports meta-programming. Our mechanism can be adapted to most meta-programming systems with compilers that support profiling.

1. Introduction

Meta-programs, or programs that write other programs, are often used to implement high-level abstractions ranging from simple syntactic abstractions, to compiler generators, to domain-specific languages (DSLs). To name a few, C, C++, Haskell, Scheme, ML, and Scala have support for meta-programming [3, 6, 9, 10, 19, 20]. Ideally, meta-programs would not be concerned with generating optimized code but instead leave that to the target-language compiler. However, information is sometimes lost or obscured during the translation into the target-language program. For example, constraints on types, ranges, and effects can be lost, as can the lack of constraints on data representation, algorithms, and evaluation order. Optimizations that depend on the lost information cannot be performed by the target-language compiler and thus must be performed by the meta-program, if at all.

Profile-guided optimization (PGO) is a compiler technique that uses data gathered at run-time on representative inputs to recompile and generate optimized code. The code generated by this recompilation usually exhibits improved performance on that class of inputs than the code generated with static optimization heuristics. For instance, a compiler can decide which loops to unroll based on which loops are executed more frequently. Many compilers such as .NET, GCC, and LLVM use profile-guided optimizations. The profile information used by these compilers, such as execution counts of basic blocks or control flow graph nodes, is low-level compared to the source-language operated on by meta-programs. So the optimizations that use the profile information are also performed on low-level constructs. Common optimizations include reordering basic blocks, inlining decisions, conditional branch optimization, and function layout decisions.

Profile information can have an even greater impact on meta-program optimizations. For example, a meta-program might select data structures and algorithms based on the frequency with which certain operations are performed, potentially even leading to improvements in asymptotic performance.

Existing techniques that use profile information for these kinds of meta-program optimizations introduce a custom toolchain, or expect the programmer to optimize code by hand. Chen et. al. implement their own profile and meta-program toolchain to provide a profile-guided meta-program for performing process placement for SMP clusters [11]. Liu and Rus provide a toolset that uses profile information find suboptimial usage of the C++ STL, but leaves it up to the programmer to make these changes [13]. Hawkins et. al. implement a compiler for a language that generates C++ implementations of data structures based on high-level specifications [16, 17]. These works implement highly specific meta-programming or profiling systems to provide very advanced optimizations. Yet no general-purpose mechanism has been proposed to date that makes profile information available to meta-programs arbitrary optimizations.

This paper describes such a general-purpose mechanism. Our mechanism makes profile information available at the granularity of arbitrary source points identified by the meta-program. In the case of a meta-program implementing an embedded DSL, these could correspond to source expressions already present in the source-language program. In a manner similar to standard profile-guided optimization mechanisms, making use of this mechanism involves running the meta-program and compiler once to instrument the code, running the resulting executable one or more times on representative data to gather profile data, and running the meta-program and compiler a second time to generate the optimized code. During the second run of the meta-program, the meta-program retrieves the profile information associated with source points. The profile information is also available to the target-language compiler to support the optimizations it performs.

Our mechanism uses standard and efficient block-level profiling techniques and is potentially suitable for dynamic optimization of a running program in systems that support dynamic recompilation [2]. It enables using data sets from multiple executions of the instrumented program, and does not interfere with traditional ("low-level") PGO. We implement this mechanism as part of a high performance Scheme system, with profile information made available via an API accessible from the high-level syntactic abstraction

facility through which Scheme supports meta-programming, and even accessible at run-time. It should be straightforward to adapt to most meta-programming systems with compilers that already support profiling. Our implementation is used in an internal corporate project.[1]

The remainder of the paper is organized as follows. Section 2 presents the design of our system at a high level. Section 3 demonstrates how to use our mechanism to implement several optimizations as meta-programs. These examples demonstrate how our work can be used to implement and build on past work in a single, general system. In particular, we show our work could be used to automate the recommendataions produced by Liu and Rus by automatically specialize an abstract sequence datatype [13]. We also demonstrate how to implement profile-guided receiver class [7] prediction using our mechanism. Section 4 discusses our implementation and how it works with traditional PGOs. Section 5 discusses PGO and meta-programming in more detail. We conclude in section 6 with a discussion of how our work could be implemented in other meta-programming systems.

2. Design

This section presents the essential points of our system. We first discuss how source points are identified and manufactured. We then discuss how we store profile information and handle multiple data sets. We elide implementation particulars until section 4.

2.1 Source objcets

To perform arbitrary meta-program optimizations, we require profile information from arbitrary points in the source program. We model this using source objects, which act as unique keys to how often a particular point in the code is reached. Each source expression of a program is annotated with a unique source object. We can create new (fresh) source objects using `(make-source-object)`, and can create a new profile point by using `(profile src)`, where `src` is some source object. We access the profile information through the function `profile-query-weight`, which takes a source object and returns a number representing the execution frequency.

2.2 Profile weight

Instead of exact counts, we store execution counts relative to the most frequently executed expression in the program. This provides a single value that represents the relative importance of an expression and supports using multiple profile data sets. These profile weights are associated with each source object, and returned by `profile-query-weight`.

We considered comparing to the total number of expressions executed and the average number of times an expression is executed. In both cases, the results are distorted when there are a large number of expressions that are executed infrequently. In that case, a main loop might look infrequently executed if there are many start up or shut down steps. By comparing to the most expensive expression, we have a relatively stable comparison of how expensive some expression is, even in cases with many unused expressions or a few very expensive expressions.

To understand how we handle profile weights, consider a program with two loops, `A` and `B`. If `A` is executed 5 times, and `B` is executed 10 times, we store `(profile-query-weight A) = 5/10 = 0.5` and `(profile-query-weight B) = 10/10 = 1`. To support multiple data sets, we simple compute the average of these weights. For instance, if in a second data set `A` is executed 100 times and `B` is executed 10 times, then `(profile-query-weight A) = ((5/10) + (100/100))/2 = 0.75` and `(profile-query-weight B) = ((10/10) + (10/100))/2 = 0.55`.[2] Multiple data sets enable reuse and help the developer collect representative profile data. This is important to ensure our PGOs optimize for the class of inputs we expect in production.

3. Examples

This section demonstrates how our to use our mechanism, and how it generalizes and advances past work on profile-guided meta-programs. The first example demonstrates profile-guided receiver class prediction for a object-oriented DSL based on profile information. The final example demonstrates specializing a data structure based on profile information.

3.1 Scheme macro primer

Our system and examples are implemented in Scheme, so we give a quick intro to Scheme meta-programming and its syntax.

```
; Defines a macro (meta-program) 'do-n-times'
; Example:
; (do-n-times 3 (display "*")) expands into
; (begin (display "*")
;          (display "*")
;        (display "*"))
(define-syntax (do-n-times stx)
  ; pattern matches on the inputs syntax
  (syntax-case stx ()
    [(do-n-times n body)
     ; Start generating code
     #`(begin
         ; Runs at compile time then
         ; splices the result into the
         ; generated code
         #,@(let loop [(i (syntax->datum n))]
              ; Loops from n to 0
              (if (zero? i)
                  '()
                   ; Create a list #'body
copies
                   (cons #'body (loop (sub1 i)))))))]))
```

Figure 1: Sample macro

The meta-program in figure 1 expects a number `n` and an expression `body` and duplicates the expression `n` times. Each meta-program, created by `define-syntax`, takes a single piece of syntax as its argument. We use `syntax-case` to pattern matches on the syntax. `#'`, `#`` , and `#,` implement Lisp's quote, quasiquote, and unquote but on syntax instead of lists. In the example, we run a loop at compile-time that generates a list with `n` copies of the syntax `body`, and then unquote-splice (`#,@`) the list of syntax into the program at compile-time.

3.2 Profile-guided receiver class prediction

In this example we demonstrate how to implement profile-guided receiver class prediction [7] for a hypothetical object-oriented DSL with virtual methods, similar to C++. We perform this optimization through a general meta-program called `exclusive-cond`, a branching construct that can automatically reorder its clauses based on which is most likely to be executed.

`cond` is a Scheme branching construct analogous to a series of if/else if statements. The clauses of `cond` are executed in order

[1] TODO: I want to say something about our lack of benchmarks here.

[2] TODO: Diagram

until the left-hand side of a clause is true. If there is an `else` clause, the right-hand side of the `else` clause is taken only if no other clause's left-hand side is true.

Figure 2 shows an example of a `cond` generated by our hypothetical OO DSL. We assume the DSL compiler simply expands every virtual method call into a conditional branch for known instances of objects and relies on another meta-program to reorder branches and throw out uncommon cases.

We borrow the following example from Grove et. al. [7]. Consider a class `Shape` with a virtual method `area`. `Square` and `Circle` inherit `Shape` and implement the virtual `area`. We will use `exclusive-cond` to reorder inlined virtual method calls to optimize the common case, and fall back to dynamic virtual method dispatch.

```
(cond
 [(class-equal? obj Square)
  (* (field obj length) (field obj width))]
 [(class-equal? obj Circle)
  (* pi (sqr (field obj r)))]
 [else (method obj "area")])
```

Figure 2: An example of `cond`

By profiling the branches of the `cond`, we can sort the clauses in order of most likely to succeed, or even drop clauses that occur too infrequently inline. However, `cond` is order dependent. While the programmer can see the clauses are mutually exclusive, the compiler cannot prove this in general and cannot reorder the clauses.

Instead of wishing our compiler was more clever, we use meta-programming to take advantage of this high-level knowledge. We define `exclusive-cond`, figure 3, with the same syntax and semantics of `cond` [3], but with the restriction that clause order is not guaranteed. We then use profile information to reorder the clauses.

The `exclusive-cond` macro rearranges clauses based on the profiling information of the right-hand sides. Since the left-hand sides are executed depending on the order of the clauses, profiling information from the left-hand side is not enough to determine which clause is executed most often. The clause structure stores the original syntax for the clause and the weighted profile count for that clause. Since a valid `exclusive-cond` clause is also a valid `cond` clause, the syntax is simply copied, and a new `cond` is generated with the clauses sorted according to profile weights. If an `else` clause exists then it is emitted as the final clause.

Figure 4 shows how our receiver class prediction example is optimized through `exclusive-cond`. The generated `cond` will test for `Circle` (the common case) first.

3.3 Fast Path Lexical Analyzer

In this example we demonstrate how to use the general meta-program, `exclusive-cond`, presented in the previous example to optimize a lexical analyzer. A lexical analyzer in Scheme can be written naturally using `cond` or `case`, a pattern matching construct similar to C's `switch`. Such a lexical analyzer can be easily optimized be instead using the `exclusive-cond` macro we saw earlier.

`case` 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 `eqv?` to some element of the left-hand. If `key-expr` is not `eqv?` to any element

[3] Schemers: we omit the alternative cond syntaxes for brevity.

```
(exclusive-cond
 [(class-equal? obj Square)
  ; executed 2 times
  (* (field obj length) (field obj width))]
 [(class-equal? obj Circle)
  ; executed 5 times
  (* pi (sqr (field obj r)))]
 [else (method obj "area")])


(cond
 [(class-equal? obj Circle)
  ; executed 5 times
  (* pi (sqr (field obj r)))]
 [(class-equal? obj Square)
  ; executed 2 times
  (* (field obj length) (field obj width))]
 [else (method obj "area")])
```

Figure 4: An example of `exclusive-cond` and its expansion

of any left-hand side and an `else` clause exists then the right-hand side of the `else` clause is executed.

```
(case (read-token)
      [(#\space) e1]
      [(#\)) e2]
      [(#\( #\)) e3]
      ...
      [else e-else])
```

Figure 5: An example tokenizer using `case`

Figure 5 shows an example `case` expression. If the token is a space, `e1` is executed. If the token is a right paren then `e2` is executed. If the token is a left paren then `e3` is executed. If no other clauses match, then `e-else` is executed. Note that the third clause has an extra right paren character that can never be reached, since it would first match teh second clause.

Since `case` permits clauses to have overlapping elements and uses order to determine which branch to take, we must remove overlapping elements before clauses can be reordered. We parse each clause is parsed into the set of left-hand side keys and right-hand side bodies. We remove overlapping keys are removed by keeping only the first instance of each key when processing the clauses in the original order. After removing overlapping keys, an `exclusive-cond` is generated. Figure 6 shows the full implementation of case. The majority of the work is in `trim-keys!`, which removes duplicate keys.

```
(let ([x (read-token)])
  (exclusive-cond
    [(memv x '(#\space)) e1]
    [(memv x '(#\)) e2]
    [(memv x '(#\() e3]
    ...
    [else e-else]))
```

Figure 7: The expansion of figure 5

```
(define-syntax exclusive-cond
  (lambda (x)
    (define-record-type clause (fields syn weight))
    (define (parse-clause clause)
      (syntax-case clause ()
        [(e0 e1 e2 ...) (make-clause clause (or (profile-query-weight #'e1) 0))]
        [_ (syntax-error clause "invalid clause")]))
    (define (sort-clauses clause*)
      (sort (lambda (cl1 cl2)
              (> (clause-weight cl1) (clause-weight cl2)))
       (map parse-clause clause*)))
    (define (reorder-cond clause* els?)
      #`(cond
          #,@(map clause-syn (sort-clauses clause*)) . #,els?))
    (syntax-case x (else)
      [(_ m1 ... (else e1 e2 ...)) (reorder-cond #'(m1 ...) #'([else e1 e2 ...]))]
      [(_ m1 ...) (reorder-cond #'(m1 ...) #'())])))
```

Figure 3: Implementation of `exclusive-cond`

```
(define-syntax (case x)
  (define (helper key-expr clause* els?)
    (define-record-type clause (fields (mutable keys) body))
    (define (parse-clause clause)
      (syntax-case clause ()
        [((k ...) e1 e2 ...) (make-clause #'(k ...) #'(e1 e2 ...))]
        [_ (syntax-error "invalid case clause" clause)]))
    (define (emit clause*)
      #`(let ([t #,key-expr])
          (exclusive-cond
            #,@(map (lambda (clause)
                      #`[(memv t '#,(clause-keys clause))
                         #,@(clause-body clause)])
                    clause*)
            . #,els?)))
    (let ([clause* (map parse-clause clause*)])
      (define ht (make-hashtable equal-hash equal?))
      (define (trim-keys! clause)
        (clause-keys-set! clause
          (let f ([keys (clause-keys clause)])
            (if (null? keys)
                '()
                (let* ([key (car keys)]
                       [datum-key (syntax->datum key)])
                  (if (hashtable-ref ht datum-key #f)
                      (f (cdr keys))
                      (begin
                        (hashtable-set! ht datum-key #t)
                        (cons key (f (cdr keys))))))))))
      (for-each trim-keys! clause*)
      (emit clause*)))
    (syntax-case x (else)
      [(_ e clause ... [else e1 e2 ...])
       (helper #'e #'(clause ...) #'([else e1 e2 ...]))]
      [(_ e clause ...)
       (helper #'e #'(clause ...) #'())]))
```

Figure 6: Implementation of `case` using `exclusive-cond`

Figure 7 shows how the example `case` expression from figure 5 expands into `exclusive-cond`. Note the duplicate right paren in the third clause is dropped to preserve ordering constraints from `case`.

3.4 Data Structure Specialization

The example in section 3.2 shows that we can easily bring well-known optimizations up to the meta-level, enabling the DSL writer to take advantage of traditional profile directed optimizations. While profile-guided meta-programming enables such traditional optimizations, it also enables higher level decisions normally done by the programmer.

Past work has used profile information to give programmer feedback when they make suboptimal use of algorithms and data structures provided by standard libraries [13], but left it up to the programmer to change the code. Our mechanism enables automating these changes. By giving meta-programs access to profile information we can automatically generate optimized code.

In this example, we provide an abstract sequence data structure that changes its implementation based on profile information. This simple example default to a list, but specializes to a vector (array) when vector operations are more common than list operations. While simplified, this example shows that our mechanism support making high-level decisions normally left to the programmer.

The example in figure 8 chooses between a list and a vector using profile information. If the program uses `seq-set!` and `seq-ref` operations more often than `seq-map` and `seq-cons`, then the sequence is implemented using a `vector`, otherwise using a `list`.

The last line of figure 8 demonstrates the usage of the `define-sequence-datatype` macro. In this example, a sequence named `seq1` is defined and initialized to contain elements `0`, `3`, `2`, and `5`.

The macro defines new profiled version of the sequence operations and defines a new instance of sequence. The profiled operations are redefined for *each* new sequence, creating fresh source objects, for each seperate sequence. This ensures each instance of a sequence is profiled and specialized seperately.

In this section, we presented several examples of profile-guided meta-program optimizations. These examples are simplified versions of past work, but demonstrate that a single general mechanism can be used to implement and improve optimizations language implementors develop whole toolchains to implement. [4]

4. Implementation

This section describes details of how profile information is represented, how code is instrumented, and how source-level and block-level profile directed optimizations can work together in our system. First we present how we represent source objects and profile information. Next we describe how code is instrumented to collect profile information. Then we present how profile information is stored and accessed. Finally we present how we use both source-level and block-level profile directed optimizations in the same system.

4.1 Source objects

In the previous sections we elided what exact a source object is, assuming that we can use them as keys, create fresh ones, and attach them to syntax. Chez Scheme implements source objects to use in error messages. A source object contains a filename, line number, and starting and ending character positions. The Chez Scheme reader automatically creates and attaches these to each piece of syntax read from a file, but Chez Scheme also provides an API to programmatically manipulate source objects. This is useful when using Chez Scheme as a target language. Custom source objects can be attached to target syntax to provide error messages with line and character positions in the source language [8 Chapter 11].

[4] TODO: This paragraph might be unnecessary

```
...
(define list-src (make-source-
object "sequence-generate-src" 0 0 0))
(define vector-src (make-source-
object "sequence-generate-src" 1 0 0))
...
```

Figure 9: Creating custom source objects

To create custom source objects for for fresh profile counter, we can use arbitrary filenames, lines numbers, and character positions. For instance, in section 3.4 we create custom source objects to profile list and vector operations. In our implementation, these might be created as seen in figure 9.

4.2 Profile weights

We represent profile information as a floating point number between 0 and 1. As mentioned in section 2, profile information is not stored as exact counts, but as a weighted relative count. We considered using Scheme fixnums (integers) for additional speed, but fixnums quickly loose precision, particularly when working with multiple data sets.

We store profile weights by creating a hash table from source file names to hash tables. Each second level hash table maps the starting file position to a profile weight. These tables are not updated in real time, only when a new data set is manually loaded via `profile-load-data`.

4.3 Instrumenting code

The naive method for instrumenting code to collect source profile information is to attach the source information to each AST node internally. At an appropriately low level, that source information can be used to generate code that increments profile counters. However this method can easily distort the profile counts. As nodes are duplicated or thrown out during optimizations, the source information is also duplicated or lost.

Instead we create a separate profile form that is created after macro expansion. Each expression `e` that has source information attached is expanded internally to `(begin (profile src) e)`, where `src` is the source object attached to `e`. The profile form is consider an effectful expression internally and should never be thrown out or duplicated, even if `e` is. This has the side-effect of allowing profile information to be used for checking code-coverage of test suites. While the separate profile form has benefits, it can interfere with optimizations based on pattern-matching on the structure of expressions, such as those implemented in a nanopass framework [1].

We keep profile forms until generating basic blocks. While generating basic blocks, the source objects from the profile forms are gathered up and attached to the basic block in which they appear. When a basic-block is entered, every instruction in that block will be executed, so any profile counters in the block must be incremented. Since all the profile counters must be incremented, it is safe to increment them all at the top of the block.

In our implementation, we attempt to minimize the number of counters executed at runtime. After generating basic blocks and attaching the source objects to their blocks, we analyze the blocks to determine which counters can be calculated in terms of other

```
(define-syntax define-sequence-datatype
   (lambda (x)
      ; Create fresh source object. list-src profiles operations that are
      ; fast on lists, and vector-src profiles operations that are fast on
      ; vectors.
      (define list-src (make-source-obj))
      (define vector-src (make-source-obj))
      ; Defines all the sequences operations, giving implementations for
      ; lists and vectors.
      (define op*
        `((make-seq ,#'list ,#'vector)
          (seq? ,#'list? ,#'vector?)
          ; Wrap the operations we care about with a profile form
          (seq-map ,#`(lambda (f ls) (profile #,list-src) (map f ls))
                   ,#`(lambda (f ls) (profile #,list-src) (vector-map f ls)))
          (seq-first ,#'first ,#'(lambda (x) (vector-ref x 0)))
          (seq-cons ,#`(lambda (x ls) (profile #,list-src) (cons x ls))
                    ,#`(lambda (x v)
                         (profile #,list-src)
                         (let ([i 0]
                               [v-new (make-vector (add1 (vector-length v)))])
                           (vector-for-each
                             (lambda (x)
                               (vector-set! v-new i x)
                               (set! i (add1 i)))
                             v))))
          (seq-ref ,#`(lambda (ls n) (profile #,vector-src) (list-ref ls n))
                   ,#`(lambda (v n) (profile #,vector-src (vector-ref v n))))
          (seq-set! ,#`(lambda (ls n obj)
                         (profile #,vector-src) (set-car! (list-tail ls n) obj)
                    ,#`(lambda (v n obj)
                         (profile #,vector-src) (vector-set! v n obj))))))
        ; Default to list; switch to vector when profile information
        ; suggests we should.
        (define (choose-op name)
          ((if (> (profile-query-weight vector-src)
                  (profile-query-weight list-src))
               third
               second)
           (assq name op*)))
      (syntax-case x ()
        [(_ var (init* ...))
         ; Create lists of syntax for operation names and definitions
         (with-syntax ([(name* ...) (map first op*)]
                       [(def* ...) (map choose (map first op*))])
           ; and generate them
           #`(begin (define name* def*) ...
           ; Finally, bind the sequence.
                    (define var (#,(choose 'make-seq) init* ...))))])))

; Define an abstrace sequence
(define-sequence-datatype seq1 (0 3 2 5))
```

Figure 8: Implementation of `define-sequence-datatype`

counters. If possible, a counter is computed as the sum of a list of other counters. This complicated the internal representation of counters and the generation of counters, but decreases the overhead of profiling. These techniques are based on the work of Burger and Dybvig [2]

To instrument block-level profiling, we reuse the above infrastructure by creating fake source objects. Before compiling a file, we reset global initial block number to 0, and create a fake source file based on the file name. We give each block a source object using the fake file name and using the blocks number as the starting and ending file position.

4.4 Source and block PGO

When designing our source level profiling system, we aimed to take advantage of prior work on low level profile-guided optimizations [12, 21]. However, optimizations based on source-level profile information may result in a different set of blocks, so the block-level profile information will be stale. Therefore optimization using source profile information and those using block profile information cannot be done after a single profiled run of a program.

To take advantage of both source and block-level PGO, first we compile and instrument a program to collect source-level information. We run this program and collect only source-level information. Next we recompile and optimize the program using the source-level information only, and instrument the program to collect block-level information. The profile directed meta-programs reoptimize at this point. We run this program and collect only the block-level information. Finally, we recompile the program with both source-level and block-level information. Since the source information has not changed, the meta-programs generate the same source code, and thus the compiler generates the same blocks. The blocks are then optimized with the correct profile information.

5. Related and Future Work

[5] [6] Modern systems such as GCC, .NET, and LLVM use profile directed optimizations [14, 15, 18]. However, these systems provide mostly low level optimizations, such as optimizations for block order and register allocation. In addition to limiting the kinds of optimizations the compiler can do, this low-level profile information is fragile.

Recently there has been work to give programmers advice on which data structure to use http://dx.doi.org/10.1109/CGO.2009.36, but with our techniques we can automagically optimize the generated code instead of just advice the programmer.

GCC profiles an internal control-flow graph (CFG). To maintain a consistent CFGs across instrumented and optimization builds, GCC requires similar optimization decisions across builds. By associating profile information with source expression we can more easily reuse profile information [4]. In our system, all profile information for a source file is usuable as long as the source file does not change.

.NET provides some higher level optimizations, such as function inlining and conditional branch optimization similar to `exclusive-cond` and `case` presented here. To optimize `switch` statements, .NET uses *value* profiling in addition to execution count profiling [18]. By probing the values used in a switch statement, the compiler can attempt to reorder the cases of the `switch` statement. [7]

The standard model for profile directed optimizations requires the instrument-profile-optimize workflow. LLVM has a different model for profile directed optimization. LLVM uses a runtime reoptimizer that monitors the running program. The runtime reoptimizer can profile the program as it runs "in the field" and perform simple optimizations to the machine code, or call off to an offline optimizer for more complex optimiztions on the LLVM bytecode.

Meta-programs generate code at compile time, so the examples presented in section 3 require the standard instrument-profile-optimize workflow. However, because we expose an API to access profiling information, we could use this system to perform runtime decisions based on profile information. To truly be beneficial, this requires keeping the runtime overhead of profiling very low, which is not usually the case [4, 5]. However, our techniques for reducing the number of counters and our careful representation of profile forms allows accurate source profiling with little overhead [8].

6. Conclusion

We have presented a general mechanism for profile-guided meta-program optimizations implemented in Scheme. While our mechanism should easily extend to other meta-programming facilities, we conclude by discussing precisely how other common meta-programming facilities need to be extended to use our mechanism.

Template Haskell, MetaOcaml, and Scala all feature powerful meta-programming facilities similar to Scheme's [3, 6, 9, 19, 20]. They allow executing arbitrary code at compile-time, provide quoting and unquoting of syntax, and provide direct representations of the source AST. Source objects could be attached to the AST, and `profile-query-weight` could access the source objects given a AST. These languages all appear to lack source profilers, however.

C++ template meta-programming does not support running arbitrary programs at compile time. This might limit the kinds of optimizations that could be implemented using C++ template meta-programming as it exists today. Many source level profilers already exist for C++, so the challenge is in implementing source objects and `profile-query-weight`. C++ templates offers no way to directly access and manipulate syntax, so it is not clear where to attach source objects.

C preprocessor macros do support using syntax as input and output to macros, but are very limited in what can be done at compile time. Adding directives to create, instrument, and read source profile points might be enough to support limited profile-guided meta-programming using C preprocessor macros.

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 of 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.

Bibliography

[1] Keep, Andrew W and Dybvig, R Kent. A nanopass framework for commercial compiler development. In *Proc. Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*, 2013.

[5] TODO: felleisen04,tobin-hochstadt06

[6] TODO: I'm not sure what I'm doing with this section yet.

[7] TODO: Value probes seem like a pretty ad-hoc method to get a very specific optimization. I don't know if I want to say that.

[8] TODO: measure overhead on a standard set of benchmarks. The benchmarks I ran at cisco suggest ∼10% overhead, but those are not publically accessible. This sentence belongs in implementation

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

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

[4] 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

[5] Thomas M Conte, Kishore N Menezes, and Mary Ann Hirsch. Accurate and practical profile-driven compilation using the profile buffer. In *Proc. Annual ACM/IEEE international symposium on Microarchitecture*, 29, pp. 36–45, 1996. http://pdf.aminer.org/000/244/348/commercializing_profile_driven_optimization.pdf

[6] 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

[7] Grove, David, Dean, Jeffrey, Garrett, Charles, and Chambers, Craig. Profile-guided receiver class prediction. In *Proc. Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications*, 1995. http://doi.acm.org/10.1145/217838.217848

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

[9] 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

[10] Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. SugarJ: Library-based Syntactic Language Extensibility. In *Proc. Proceedings 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

[11] Chen, Hu, Chen, Wenguang, Huang, Jian, Robert, Bob , and Kuhn, Harold. 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*, 2006.

[12] Pettis, Karl and Hansen, Robert C. Profile Guided Code Positioning. In *Proc. Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, 1990. http://doi.acm.org/10.1145/93542.93550

[13] Liu, Lixia and Rus, Silvius. 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.

[14] LLVM: An infrastructure for multi-stage optimization. Master dissertation, University of Illinois, 2002.

[15] Optimize Options - Using the GNU Compiler Collection. 2013.

[16] Hawkins, Peter, Aiken, Alex, Fisher, Kathleen, Rinard, Martin, and Sagiv, Mooly. Data representation synthesis. In *Proc. ACM SIGPLAN Notices*, 2011.

[17] Hawkins, Peter, Aiken, Alex, Fisher, Kathleen, Rinard, Martin, and Sagiv, Mooly. 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

[18] Profile-Guided Optimizations. 2013. http://msdn.microsoft.com/en-us/library/e7k32f4k(v=vs.90).aspx

[19] 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

[20] 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

[21] Hwu, W. W. and Chang, P. P. Achieving High Instruction Cache Performance with an Optimizing Compiler. In *Proc. Proceedings of the 16th Annual International Symposium on Computer Architecture*, 1989. http://doi.acm.org/10.1145/74925.74953