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). While meta-programs can be written for virtually any language, some languages have built-in support for meta-programming, including C, C++, Haskell, Scheme, ML, and Scala. [3, 6, 10, 11, 25, 27]. Ideally, meta-programs would not be concerned with generating optimized code but instead leave that to the target-language compiler. However, information is sometimes unavoidably 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 in which profile information, e.g., execution counts, from test runs on representative sets of inputs is fed into the compiler to enable it to generate more efficient code. The resulting code usually exhibits improved performance, at least on the represented class of inputs, than code generated with static optimization heuristics. Compilers that support PGO include .NET, GCC, and LLVM [18, 21, 23] 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 [26 Chapter 4].

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 tools to provide a profile-guided meta-program for performing process placement for SMP clusters [5]. Liu and Rus provide a tools that uses profile information to identify suboptimial usage of the C++ STL, but leaves it up to the programmer to take corrective action [19]. Hawkins et. al. implement a compiler for a language that generates C++ implementations of data structures based on high-level specifications [15, 16]. These works implement highly specific meta-programming or profiling systems to provide advanced optimizations. Yet no general-purpose mechanism has been proposed to date that makes profile information available to meta-programing systems for 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 our 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 avail-

able 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.

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 how our work could be used to automate the recommendataions produced by Liu and Rus by automatically specialize an abstract sequence datatype [19]. We also demonstrate how to implement profile-guided receiver class [14] 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 what profile information we use and how we handle multiple data sets. We delay giving implementation details until section 4.

In a typical meta-programming situation, a meta-program takes as input a *source program* in a high-level domain-specific language (DSL) and produces a *target program* in some other language, e.g., C, Haskell, or Scheme. To perform arbitrary meta-program optimizations, we might require profile information for arbitrary points in the source program, arbitrary points in the target program, or both. We use *source objects* [10] to uniquely identify these points, and the compiler maintains a database associating source objects with profile information, whenever profile information from earlier program runs has been supplied.

2.1 Source objects

Source objects are typically introduced by the lexer and parser for a source language and maintained throughout the compiler to correlate source with intermediate or object code, enabling both compile-time source-error messages and source-level debugging, among other things. While the source objects created by the lexer and parser encapsulate a source file descriptor and character range for a specific source expression, source objects can contain other or different information. Meta programs can make use of this to manufacture new source objects representing unique points in the target program, perhaps based on corresponding points in the source program.

2.2 Profile weight

Instead of storing exact counts in the profile database, we store *weights* instead. The weight of a source point in a given dataset is the ratio of the exact count for the source point and the maximum count for any source point, represented as a floating-point number in the range [0,1]. This provides a single value identifying the relative importance of an expression and simplifies the combination of multiple profile data sets.

We considered recording absolute counts, but this compilicates the combination of multiple datasets, since absolute counts from one run to the next are not directly comparable. We also considered recording ratios of individual counts to total or average counts. In both cases, the results are distorted when there are a few heavily executed expressions, potentially leading to difficulty distinguishing profile weights for two less frequently executed expressions. We also considered using fixed-precision rather than floating point, but the floating-point representation makes it easy to determine the importance of a particular expression overall while still providing substantial precision when comparing the counts for source points with similar importance.

To understand how we compute 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 `A` $\rightarrow$ `5/10 = 0.5` and `B` $\rightarrow$ `10/10 = 1`. To support multiple data sets, we simply 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 `A` $\rightarrow$ `((5/10) + (100/100))/2 = 0.75` and `B` $\rightarrow$ `((10/10) + (10/100))/2 = 0.55`. [1] Multiple data sets enable reuse and help the developer collect representative profile data. This is important to ensure our PGOs can optimize for multiple classes of inputs expected in production.

3. Examples

This section demonstrates how 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 [14] for an object-oriented DSL based on profile information. We then reuse part of that meta-program to optimize a tokenizer. The final example demonstrates specializing a data structure based on profile information.

3.1 Scheme macro example

Our system and examples are implemented in Scheme, so we give below a simple example to briefly introduce 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
                  (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 perform 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`

[1] TODO: Diagram

copies of the syntax `body`, and then splice (`#,@`) the copies into the generated program.

3.2 Profile-guided receiver class prediction

In this example we demonstrate how to implement profile-guided receiver class prediction [14] 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 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. [14]. 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. 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 cursing Rice's theorem, we use meta-programming to encode and take advantage of this high-level knowledge. We define `exclusive-cond`, figure 3, with the same syntax and semantics of `cond` [2], but without the specific order of execution. 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 `exclusive-cond` clause and the weighted profile count for that clause. Since a valid `exclusive-cond` clause is also a valid `cond` clause, we copy the syntax and generate a new `cond` with the clauses sorted according to profile weights. Of course we do not include the `else` clause when reordering other clauses; it is always last.

We use the function `profile-query-weight` to access the profile information. Given a source object or source expression, it returns the associated profile weight.

[2] 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

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 Tokenizer

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

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

```
(define-syntax (exclusive-cond 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`

clauses in the original order. After removing overlapping keys, we generate an `exclusive-cond`.

```
(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

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-guided 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 [19], 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 defaults 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-rest` 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. Here we assume we can create fresh source objects via the function `make-fresh-source-obj!`. We discuss its implementation in section 4.

4. Implementation

This section describes the details of how we represent profile information, how we instrument code, and how we ensure source-level and block-level profile-guided optimizations work together in our system.

4.1 Source objects

In the previous sections we elided what exactly 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, starting file position, and ending file 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 number and character positions in the source language [9 Chapter 11].

```
(define make-fresh-source-obj!
  (let ([x 0])
    (lambda ()
      (let ([src (make-source-object
                   "sequence-src" x x)])
        (set! x (add1 x))
        src))))
...
(define list-src (make-fresh-source-obj!))
(define vector-src (make-fresh-source-obj!))
...
```

Figure 9: Creating custom source objects

To create custom source objects for fresh profile counters, we can use arbitrary filenames and 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. We store `#f` (false) when there is no profile information, and 0 when the counter was never executed. 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 filenames to hash tables. Each second level hash table maps the starting character 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 expression (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 expressions 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 a source object attached is expanded internally to `(begin (profile src) e)`, where `src` is the source object attached to `e`. The profile form is considered an effectful expression 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 [30].

```
(define-syntax (define-sequence-datatype 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-fresh-source-obj!))
  (define vector-src (make-fresh-source-obj!))
  ; Defines all the sequences operations, giving implementations for
  ; lists and vectors.
  (define op*
    '((make-seq ,#'list ,#'vector)
      (seq? ,#'list? ,#'vector?)
      (seq-map ,#'map ,#'vector-map)
      (seq-first ,#'first ,#'(lambda (x) (vector-ref x 0)))
      ; Wrap the operations we care about with a profile form
      (seq-rest ,#`(lambda (ls) (profile #,list-src) (rest ls))
                ,#`(lambda (v)
                     (profile #,list-src)
                     (let ([i 1]
                           [v-new (make-vector (sub1 (vector-length v)))])
                       (vector-for-each
                         (lambda (x)
                           (vector-set! v-new i x)
                           (set! i (add1 i)))
                         v))))
      (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 abstract sequence
(define-sequence-datatype seq1 (0 3 2 5))
```

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

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 minimize the number of counters incremented at runtime. After generating basic blocks and attaching the counters to blocks, we analyze the blocks to determine which counters can be calculated in terms of other 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]. We generate at most one increment per block, and fewer in practice.

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 filename. We give each block a source object using the fake filename 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 wanted to continue using prior work on low level profile-guided optimizations [22, 31][3]. 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. From this point on, source-level optimizations should run and the blocks should remain stable. 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

5.1 Low-level PGO

Modern systems such as GCC, .NET, and LLVM use profile directed optimizations [18, 21, 23]. These systems uses profile information to guide decisions about code positioning, register allocation, inlining, and branch optimizations.

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 [4]. 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 3.3.

Our system supports all these optimizations and has several advantages. While .NET extends their profiling system to get additional optimizations, we can support all the above optimizations in a single general-purpose system. By using profile information associated with source expressions, we reduce reliance specific internal compiler decisions and make profile information more reusable. When there is no substitute for block-level information, such as when reordering basic blocks, we support both source and block profiling in the same system.

5.2 Dynamic Recompilation

The standard model for PGO requires the instrument-profile-optimize workflow. 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 optimiztions on the LLVM bytecode.

While not currently enabled, our mechanism supports this kind of reoptimization. We build on the work of Burger and Dybvig, who present an infrastructure for profile-directed dynamic reoptimization [2]. Their work shows just 14% run-time overhead for instrumented code, but they express concerns that dynamic recompilation will not overcome this cost. Our internal benchmarks show similar overhead. To enable dynamic PGO, we would need to modify our mechanism to automatically reload profile information, such as whenever `profile-query-weight` is called, instead of manually loading information from a file. This is a trivial change to our system, but we have not optimizations in mind that make use of profile-guided at runtime. It may also increase overhead, since we compute profile weights and many counters when loading new profile data.

5.3 Meta-program optimizations

Meta-programming has proven successful at providing high-levels of abstraction while still producing efficient code. Meta-programming has been used to implement abstract libraries [7][4], domain specific languages [12, 17], and even whole general purpose languages [1, 24, 28, 29]. These meta-programs can lose or obscure information during the translation into target-language code.

We're not the first to realize this. Many meta-program optimizations exist. Tobin-Hochstadt et. al. implement the optimizer for Typed Racket as a meta-program [29]. Sujeeth et. al. provide a framework for generated optimized code from DSLs [17]. Hawkins et. al. implement a compiler for a language that generates C++ implementations of data structures based on high-level specifications [15, 16].

Even using profile information to perform optimizations in meta-programs is not new. Chen et. al. implement their own profile and meta-program tools to provide a profile-guided meta-program for performing process placement for SMP clusters [5]. Liu and Rus provide a tools that uses profile information to identify suboptimial usage of the C++ STL.

We support these works by providing 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.

5.4 More PGO

We have previously presented some past work on both low-level PGOs and profile-guided meta-programs. But the use of profile information is still an active area of research. Furr et. al. present a system for inferring types in dynamic languages to assist in debugging [13]. Chen et. al. use profile information to reorganize the heap and optimize garbage collection [5]. Luk et. al. use profile information to guide data prefetching [20]. Debray and Evans use profile information to compress infrequently executed code on memory constrained systems [8].

[3] TODO: Fix auto-bib

[4] TODO: STL?

With so many profile-guided optimizations, we need a general-purpose mechanism in which to implement them without reimplementing profiling, compiling, and meta-programing tools.

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, 10, 25, 27]. 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 an 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 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 optimizations previously implemented in seperate, specialized toolchains.

Bibliography

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

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

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

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

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

[10] 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

[11] 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

[12] Matthew Flatt, Eli Barzilay, and Robert Bruce Findler. Scribble: Closing the Book on Ad Hoc Documentation Tools. In *Proc. Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, 2009. http://doi.acm.org/10.1145/1596550.1596569

[13] Michael Furr, Jong-hoon (David) An, and Jeffrey S. Foster. Profile-guided Static Typing for Dynamic Scripting Languages. In *Proc. Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, 2009. http://doi.acm.org/10.1145/1640089.1640110

[14] David Grove, Jeffrey Dean, Charles Garrett, and Craig Chambers. 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

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

[16] 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

[17] 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. Proceedings of the 12th International Conference on Generative Programming: Concepts \&\#38; Experiences*, 2013. http://doi.acm.org/10.1145/2517208.2517220

[18] Chris Authors Lattner. LLVM: An infrastructure for multi-stage optimization. Master dissertation, University of Illinois, 2002.

[19] 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.

[20] Chi-Keung Luk, Robert Muth, Harish Patil, Richard Weiss, P. Geoffrey Lowney, and Robert Cohn. Profile-guided Post-link Stride Prefetching. In *Proc. Proceedings of the 16th International Conference on Supercomputing*, 2002. http://doi.acm.org/10.1145/514191.514217

[21] Optimize Options - Using the GNU Compiler Collection. 2013. http://gcc.gnu.org/onlinedocs/gcc-4.7.2/gcc/Optimize-Options.html#index-fprofile_002duse-867

[22] Karl Pettis and Robert C. Hansen. 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

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

[24] Jon Rafkind and Matthew Flatt. Honu: Syntactic Extension for Algebraic Notation Through Enforestation. In *Proc. Proceedings of the 11th International Conference on Generative Programming and Component Engineering*, 2012. http://doi.acm.org/10.1145/2371401.2371420

[25] 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

[26] YN Srikant and Priti Shankar. The compiler design handbook: optimizations and machine code generation. 1 edition. CRC Press, 2002.

[27] 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

[28] Sam Tobin-Hochstadt and Matthias Felleisen. The Design and Implementation of Typed Scheme. In *Proc. Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2008. http://doi.acm.org/10.1145/1328438.1328486

[29] Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. Languages As Libraries. In *Proc. Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2011. http://doi.acm.org/10.1145/1993498.1993514

[30] 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.

[31] W. W. Hwu and P. P. Chang. 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