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 mechanism for supporting profile-guided meta-program optimization. It makes profile information available at the granularity of arbitrary target-language 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. The mechamism 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 some support for meta-programming [2, 5, 6, 7, 15, 16]. In the ideal, 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 generate with static optimization heuristics. For instance, a compiler can decide which loops to focus on unrolling 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.

Many compiler optimizations can benefit from the availability of profile information and many contemporary compilers provide support for gathering and using profile information for this purpose. 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 [8]. 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 [9]. Hawkins et. al. implement a compiler that generates C++ implementations of data structures based on high-level specifications [12, 13]. 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 for these kinds of optimizations.

This paper describes such a mechanism. The mechanism makes profile information available at the granularity of arbitrary target-language 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. The 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 [1]. It enables using data sets from multiple executions of the instrumented program, and works with traditional ("low-level") PGO.

This mechanism has been implemented 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. It would be straightforward to adapt to most meta-programming systems with compilers that already support profiling.

The reminder 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. Section 4 discusses our implementation and how it works with traditional PGOs.

2. Design

This section presents the design of our profile system. We discuss the system at a high-level and sketch implementations for other meta-programming systems. We discuss implementation details in section 4

2.1 Source and syntax objcets

In Scheme, macros (Scheme meta-programs) operate on *syntax objects*, direct representations of Scheme syntax, and can run arbitrary Scheme code. Each syntax object has an associated *source object*—a filename and a beginning and ending character position for the expression. To abstract away from the particular syntax, we associate profile information with these source objects. To access profile information, all we require is a function that allows retrieving profile information from a syntax or source object. We added the function `profile-query-weight` to our Scheme implementation. Given a syntax object, `profile-query-weight` returns a number between 0 and 1, or false if there is no profile information associate with that piece of syntax.

2.2 Profile weight

We represent profile information as a floating point number between 0 and 1. Profile information is not stored as exact counts, but as execution frequency with respect to the most executed expression (refered to as 'percent of max'). If an expression `e1` is executed 1 time, and the most frequently executed expression `e10` is executed 10 times, then `(profile-query-weight e1)` returns .1, while `(profile-query-weight e10)` returns 1.

We use percent of max count in part because an exact execution count can be meaningless in some contexts. Consider an expression that is executed 5 times. We cannot know if this statement is executed frequently or not without some comparison.

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 distored 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.

This relative information is not perfect. Loop unrolling can benefit from exact counts more than a weight. If we know a loop is executed exactly 5 times, unrolling it 5 times might make sense. If we know a loop is executed 20% of the max, we do not know if the loop is executed 1 or 1,000,000 times.

Ideally we would track both relative and exact information, but this doubles profiling overhead. One of our design goals is to enable 'always on' profiling, so even release builds of software can have profiling enabled without too much performance impact. Using previous work to decrease profiling overhead [1], running a set of benchmarks with profiling enables gives only 10% slowdown [1].

2.3 Source + block profiling

When designing our source level profiling system, we aimed to take advantage of prior work on low level profile directed optimizations [2]. However, optimizations based on source-level profile information may result in a different set of blocks than the blocks generated for the profiled run of a program. If blocks are profiled naively, for instance, by assigning each block a number in the order in which the blocks are generated, then the block numbers will not be consistent after optimizing with source information. Therefore optimization using source profile information and those using block profile information cannot be done after a single profiled run of a program.

We take the naive approach to block profiling and use the following workflow to take advantage of both source and block leve profile directed optimizations. 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.

While the workflow seems to significantly complicate the compilation process, the different between using only block-level profiling and using both source-level and block-level profiling is small. To use any kind of profile directed optimizations requires a 300% increase in the number of steps (from compile to compile-profile-compile). To use both source-level and block-level profile directed optimizations requires only an additional 66% increase in number of steps (compile-profile-compile to compile-profile-compile-profile-compile).

2.4 Comparison to other meta-programming languages

We take advantage of Scheme's powerful meta-programming facilities that allows running full-fledged Scheme programs at compile time. While many programming langauges have meta-programming systems, their expressiveness and support for manipulating syntax varies.

[3]

C++

limits inspecting syntax and does not allow IO at compile-time. Without compile-time IO, it seem that template meta-programming in C++ cannot currently support profile directed optimization through meta-programming.

Template Haskell

[4] supports generating Haskell code via quotes and splicing, similar to Scheme and MetaOCaml, but also provides constructors to create Haskell ASTs directly. Template Haskell allows IO and running ordinary Haskell functions at compile-time. This suggests compilers DSLs written in Haskell can easily incorporate a source expression profiler and allow profile directed optimizations at compile time. Because Template Haskell can manipulate Haskell syntax, it

[1] TODO: Run these on some reproducable benchmarks

[2] TODO: cite

[3] TODO: figure out if (profile-query-weight) could be run in C++, MetaOCaml at compile time

[4] TODO: cite

should be simple to even write an optimization for Haskell in Template Haskell.

MetaOCaml [5] supports generating OCaml code through quotes and splicing, similar to Scheme and Template Haskell. MetaOCaml allows IO and running ordinary OCaml functions at compile-time, however, it discourages inspecting OCaml syntax. This suggest compilers for DSLs written as OCaml data can easily incorporate a source expression profiler and allow profile directed optimizations at compile time.

[6] [7] for more in-depth comparison of those three.

3. Examples

This section presents several macros that use profiling information to optimize the expanded code. The first example demonstrates unrolling loops based on profile information. While loop unrolling can be done with low level profile information, we discuss when it can be useful or even necessary to do at the meta-programming level. The second example demonstrates call site optimization for a object-oriented DSL by reordering the clauses of a conditional branching structure, called `exclusive-cond`, based on profile information. The final example demonstrates specializing a data structure based on profile information.

3.1 Scheme macro primer

[8]

3.2 Loop Unrolling

Loop unrolling is a standard compiler optimization. However, striking a balance between code growth and execution speed when unrolling loops is tricky. Profile information can help the compiler focus on the most executed loops.

Profile directed loop unrolling can be done using low-level profile information. However, loop unrolling at a low-level requires associating loops with the low level profiled structures, such internal nodes or even basic blocks, and cannot easily handle arbitrary recursive functions. More importantly, with the rise in interest DSLs, implementing loop unrolling via meta-programming may be necessary to get high performance loops in a DSL.

This loop example unrolls Scheme's named let [9], as seen in figure 1. This defines a loop that runs for `i=5` to `i=0` computing factorial of `5`. This named let might normally be implemented via a recursive function, as seen in figure 3. With a high-performance compiler, this named let is equivalent to the C implementation in figure 2 The example in figure 1 would produce a recursive function `fact`, and immediately call it on `5`.

```
(let fact ([i 5])
  (if (zero? i)
      1
      (* n (fact (sub1 n)))))
```

Figure 1: The most executed program in all of computer science

```
int i = 5;
int n = 1;
fact: if(i == 0){
  n;
} else {
  n = n * --i;
  goto fact;
}
```

Figure 2: And in C

```
(define-syntax let
  (syntax-rules ()
    [(_ name ([x e] ...) body1 body2 ...)
     ((letrec
        ([name (lambda (x ...)
                 body1 body2 ...)])) e ...)]))
```

Figure 3: a simple definition of a named let

Figure 4 defines a macro, `named-let`, that create a loop and unrolls it between 1 and 3 times, depending on profile information. At compile time, the compiler runs `(or (profile-query-weight #'b1) 0)`. This looks up the profile information associated with `b1`, the first expression in the body of the loop. If the profile weight is 1, meaning the expression is executed more than any other expression during the profiled run, `unroll-limit` is 3. If the weight is 0, meaning the expression is never executed during the profiled run, `unroll-limit` is 0. Finally, `named-let` generates another macro called `name`, where name is the identifier labeling the loop in the source code, which inlines the body of the loop according to `unroll-limit` and `profile-weight`.

In fact, a named let defines a recursive function and immediately calls it. While this can be used for simple loops, a named let may have non-tail calls or even multiple recursive calls along different branches. This macro does more than loop unrolling–it does recursive function lining. A more clever macro could unroll each call site a different number of times, depending on how many times that particular call is executed. This would allow more fine grain control over code growth. For brevity, we restrict the example and assume `named-let` is used as a simple loop. Each call site is unrolled the same number of times.

3.3 Call site optimization

In this section we present a branching construct called `exclusive-cond` that can automatically reorder the clauses based on which is mostly likely to be executed. This optimization is analogous to basic block reordering, but operates at a much higher level.

We consider this construct in the context of an object-oriented DSL with classes, inheratence, and virtual methods, similar to C++. Consider a class with a virtual method `get_x`, called `Point`. `CartesianPoint` and `PolarPoint` inherit `Point` and implement the virtual `get_x`. We will use `exclusive-cond` to inline virtual method calls.

[10]

[5] TODO: cite

[6] TODO: Look at some other systems

[7] TODO: cite Czarnecki04

[8] TODO: See languages as libraries intro to macros and add something here.

[9] Strictly speaking, we do not implement named let, since in loop unrolling macro, the name is not assignable.

[10] TODO: borrowed from http://courses.engr.illinois.edu/cs421/sp2011/project/self-type-feedback.pdf

```
(define-syntax named-let
  (lambda (x)
    (syntax-case x ()
      [(_ name ([x e] ...) b1 b2 ...)
       #`((letrec ([tmp (lambda (x ...)
            #,(let* ([profile-weight
                       (or (profile-query-weight #'b1) 0)]
                     [unroll-limit
                       (floor (* 3 profile-weight))])
                #`(define-syntax name
                    (let ([count #,unroll-limit]
                          [weight #,profile-weight])
                      (lambda (q)
                        (syntax-case q ()
                          [(_ enew (... ...))
                           (if (or (= count 0)
                                   (< weight 0.1))
                               #'(tmp enew (... ...))
                               (begin
                                 (set! count (- count 1))
                                 #'((lambda (x ...) b1 b2 ...)
                                    enew (... ...))))])))))
             b1 b2 ...)])
           tmp)
         e ...)])))
```

Figure 4: a macro that does profile directed loop unrolling

[11] `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 5 shows an example of a `cond` generated by our hypothetical OO DSL. The DSL compiler simply expands every virtual method call into a conditional branch for known instances of an object.

```
(cond
 [(class-equal? obj CartesianPoint)
  (field obj x)]
 [(class-equal? obj PolarPoint)
  (* (field obj rho) (cos (field obj theta)))]
 [else (method obj "get_x")])
```

Figure 5: 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 6, with the same syntax and semantics of `cond` [12], but with the restriction that clause order is not guaranteed. We then use profile information to reorder the clauses.

The `exclusive-cond` macro will rearrange clauses based on the profiling information of the right-hand sides. Since the left-hand sides will be executed depending on the order of the clauses, profiling information from the left-hand side is not enough to determine which clause is true most often.[13] The clause record 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 7 shows an example of `exclusive-cond` and the code to which it expands. In this example, we assume the object is a `PolarPoint` most of the time.

3.3.1 `case`: Another use of exclusive-cond

`case` is a pattern matching construct, similar to C's `switch`, that is easily given profile directed optimization by implementing it in terms of `exclusive-cond`. `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 x
  [(1 2 3) e1]
  [(3 4 5) e2]
  [else e3])
```

Figure 8: An example of a `case` expression

Figure 8 shows an example `case` expression. If `x` is 1, 2, or 3, then `e1` is executed. If `x` is 4 or 5, then `e2` is executed. Note

[11] TODO: This optimization is straight out of http://dl.acm.org/citation.cfm?id=217848

[12] We omit the alternative cond syntaxes for brevity.

[13] Schemers will note this means we cannot handle the single expression cond clause syntax.

```
(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*))
          #,@(if els #`(,els) #'())))
    (syntax-case x (else)
      [(_ m1 ... (else e1 e2 ...)) (reorder-cond #'(m1 ...) #'(else e1 e2 ...))]
      [(_ m1 ...) (reorder-cond #'(m1 ...) #f)])))
```

Figure 6: Implementation of `exclusive-cond`

```
(exclusive-cond
  [(class-equal? obj CartesianPoint) (field obj x)] ; executed 2 times
  [(class-equal? obj PolarPoint)
   (* (field obj rho) (cos (field obj theta)))] ; executed 5 times
  [else (method obj "get_x")]) ; executed 8 times
```

```
(cond
  [(class-equal? obj PolarPoint) (* (field obj rho) (cos (field obj theta)))]
  [(class-equal? obj CartesianPoint) (field obj x)]
  [else (method obj "get_x")]) ; executed 8 times.
```

Figure 7: An example of `exclusive-cond` and its expansion

that while 3 appears in the second clause, if `x` is 3 then `e1` will be evaluated. The first occurrence always take precedence.

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. Each clause is parsed into the set of left-hand side keys and right-hand side bodies. 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.

```
(exclusive-cond x
  [(memv x (1 2 3)) e1]
  [(memv x (4 5)) e2]
  [else e3])
```

Figure 9: The expansion of figure 8

Figure 9 shows how the example `case` expression from figure 8 expands into `exclusive-cond`. Note the duplicate 3 in the second clause is dropped to preserve ordering constraints from `case`.

3.4 Data type Selection

The previous examples show 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 directed meta-programming enables such traditional optimizations, it also enables higher level decisions normally done by the programmer.

In this example we present a library that provides a sequence datatype. We consider this in the context of a DSL or library writer whose users are domain experts, but not computer scientists. While a domain expert writing a program my know they need a sequence for their program, they may not have the knowledge to figure out if they should use a tree, or a list, or a vector. Past work has bridge this gap in knowledge by providing tools that can recommend changes and provide feedback [14]. We take this a step further and provide a library that will automatically specialize the data structure based on usage.

The example in figure 10 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-first`, then the sequence is implemented using a `vector`, otherwise using a `list`.

[15] Figure 11 demonstrates the usage of the `define-sequence-datatype` macro. In this example, a sequence named `seq1` is de-

[14] TODO: http://dx.doi.org/10.1109/CGO.2009.36

[15] TODO: To hell with this example. We need to break it up and make it slightly more sensible to use. I hate to make it OO, but that would make it scoping issues easier. Maybe move `choose` and nonsense to an appendix and just focus on the macro here.

```
(define-syntax define-sequence-datatype
  (let ([ht (make-eq-hashtable)])
    (define args
      `((seq? . #'(x))
        (seq-map . #'(f s))
        (seq-first . #'(s))
        (seq-ref . #'(s n))
        (seq-set! . #'(s i obj))))
    (define defs
      `((make-seq    ,#'list . ,#'vector)
        (seq?    ,#'list? . ,#'vector?)
        (seq-map    ,#'map . ,#'for-each)
        (seq-first    ,#'car . ,#'(lambda (x) (vector-ref x 0)))
        (seq-ref    ,#'list-ref . ,#'vector-ref)
        (seq-set!    ,#'(lambda (ls n obj) (set-car! (list-tail ls n) obj)) . ,#'vector-set!)))
    (define (choose-args name)
      (cond
        [(assq name args) => cdr]
        [else (syntax-error name "invalid method:")]))
    (define (choose name)
      (let ([seq-set!-count (hashtable-ref ht 'seq-set! 0)]
            [seq-ref-count (hashtable-ref ht 'seq-ref 0)]
            [seq-first-count (hashtable-ref ht 'seq-first 0)]
            [seq-map-count (hashtable-ref ht 'seq-map 0)])
      (cond
        [(assq name defs) =>
         (lambda (x)
           (let ([x (cdr x)])
             (if (> (+ seq-set!-count seq-ref-count)
                    (+ seq-first-count seq-map-count))
                 (cdr x)
                 (car x))))]
        [else (syntax-error name "invalid method:")])))
    (lambda (x)
      (syntax-case x ()
        [(_ var (init* ...) name* ...)
         (for-each
           (lambda (name)
             (hashtable-set! ht name
               (or (profile-query-weight name) 0)))
           (map syntax->datum #'(name* ...)))
         (with-syntax ([(body* ...) (map (lambda (name) (choose (syntax->datum name))) #'(name* ...))]
                       [(args* ...) (map (lambda (args) (choose-args (syntax->datum name))) #'(name* ...))])
           #'(begin (define (name* args* ...) (begin name* (body* args* ...))) ...
                    (define var (#,(choose 'make-seq) init* ...))))]))))
```

Figure 10: a macro that defines a sequence datatype based on profile information

```
(define-sequence-datatype seq1 (0 3 2 5)
  seq? seq-map seq-first seq-ref seq-set!)
```

Figure 11: Use of the define-sequence-datatype macro

fined and initialized to contain elements 0, 3, 2, and 5. The macro also takes the various sequence operations as arguments, though this is a hack. [16] To get unique per sequence source information, we simply use the source information from those extra arguments. A production example would omit this hack. [17]

The macro expands into a series of definitions for each sequence operations and a definition for the sequence datatype. This example redefines the operations for each new sequence and evaluates the name to ensure function inlining does not distort profile counts. A clever compiler might try to throw out the effect-free reference to `name` in the body of each operation, so this implementation is fragile.

4. Implementation

This section describes our implementation of the profiling system, and how source-level and block-level profile directed optimizations can work together in our system. First we present 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. [18]

[16] TODO: How can we fabricate the source information?

[17] TODO: I should omit this hack

[18] TODO: Definitely going to need Kent to check this section.

4.1 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 during 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. [19] [20]

These profile forms are retained until basic blocks are generated. 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 counters. If possible, a counter is computed as the sum of a list of counters (+counters) minus the sum of a list of counters (-counters). This complicated the internal representation of counters and the generation of counters, but decreases the overhead of profiling. [21]

To instrument block-level profiling, we reuse the above infrastructure by creating fake source objects. When a file is compiled, we reset global initial block number to 0, and create a fake source file descriptor based on the file name. When creating blocks, each block is given a source object using the fake file descriptor, and using the blocks number as the starting and ending file position. This fake source object is used when block-level profiling is enable. This fake source is ignored and the list of sources from the source code is used when source-level profiling is enable. [22]

4.2 Storing and Loading profile data

We store profile data by creating a hash table from source file names to hash tables. Each second level hash table maps the starting file position of the expression to the weighted count of the expression. This lookup table is only populated after loading profile data from a file and not from a current profiled run. After loading profile data, it is accessible through `profile-query-weight`.

Profile data is not immediately loaded into the lookup table after a profiled run of a program. Profile data must first be dumped via `profile-dump-data` and then loaded via `profile-load-data`.

To dump profile data, the run time gathers up all profile counters. Recall that some counters are computed indirectly in terms of other counters. The values for these indirect counters are computed. These values with their associated source objects are then written to a file. [23]

To support loading multiple data sets, we do not load execution counts directly into the lookup table. Instead we compute the percent of max for each counter. Before loading a new data set, we find the maximum counter value. Each weighted count is computed as a percent of the maximum counter value. If an entry for a source already exists in the lookup table then we compute the weighted average of the previous entry and the counter we're currently loading. We store the weighted count and the current weight in the lookup table, incrementing the weight by one with each new data set.

5. Related and Future Work

[24] [25] Modern systems such as GCC, .NET, and LLVM use profile directed optimizations [10, 11, 14]. 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 [3]. 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 [14]. By probing the values used in a switch statement, the compiler can attempt to reorder the cases of the `switch` statement. [26]

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 optimizations 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 [3, 4]. However, our techniques for reducing the number of counters and our careful representation of profile forms allows accurate source profiling with little overhead [27].

[19] TODO: Make mention of how this affects pattern-matching optimizations, i.e. a compiler that uses nanopass.

[20] TODO: Mention how profile info can be used for coverage checking?

[21] TODO: This explanation is probably wrong

[22] TODO: Maybe an example of creating fake sources

[23] TODO: I'm not 100% sure about how this works and I need to be. Some of the racket peoples were asking.

[24] TODO: felleisen04,tobin-hochstadt06

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

[26] 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.

[27] 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

Bibliography

[1] 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

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

[3] 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

[4] 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

[5] 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

[6] 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

[7] 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

[8] 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.

[9] 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.

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

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

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

[13] 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

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

[15] Time Sheard and Simon Peyton Jones. Template metaprogramming 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

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