Profile directed meta-programming

William J. Bowman <wilbowma@ccs.neu.edu>,
R. Kent Dybvig <dyb@cisco.com>,
and Swaha Miller <swamille@cisco.com>

Abstract

Profile directed optimization is a compiler technique that uses sample data gathered at run-time to recompile and further optimize a program. The profile data can be more accurate than heuristics normally used in a compiler and thus can lead to more optimized code. Modern compilers like .NET, LLVM, and GCC use profile directed optimization by profiling the low level code and performing low level optimizations, such as reordering basic blocks.

Modern languages such as Haskell, C++, and Scheme provide powerful meta-programming facilities that help programmers create generic libraries, new language constructs, or even domain specific languages. Meta-programs can manipulate source programs in way reminiscent of a compiler. This paper presents a system for using profile information to optimize programs in the meta-programming language. The system is implemented and used in a high-performance implementation of Scheme.

1. Introduction

Profile directed optimization is a standard compiler technique that uses sample data gathered at run-time to recompile and further optimize a program. If the sample data is representative of how the program is used in practice then this data can be more accurate than static heuristics and can lead to a more optimized program.

cite

Current compilers such as .NET, GCC, and LLVM do low level profile directed optimization such as reordering blocks, and optimizing switch statements.

Say something about missing high level information;

These optimizations are often fragile because of their dependence on low level, compiler internal structures. At this low level, much information about the program is lost, limiting which optimizations the compiler can do. By bringing profile directed optimizations up to the source level, programmers can use specific knowledge of the problem domain, program, and the high-level code to optimize programs in ways a compiler can not.

Rewrite this, rephrase what is below. And cite

A simple example is a conditional branching construct like Scheme's `cond`. `cond` takes an arbitrary number of clauses of the form `(lhs rhs)`, executing the first right-hand side whose left-hand side is true, or executing a final `else` clause if no left-hand side is true. This construct essentially expands into a sequence of if/else expressions. However, if the programmer knows that each clause is mutually exclusive, it is beneficial to sort the clauses from most to least likely to succeed. In general, the compiler cannot prove such a property and must emit the clauses in the original order, so even traditional profile directed optimization cannot optimize `cond`.

This paper presents a system for doing profile directed meta-programming, including a workflow for using it with traditional low level profile directed optimizations. Section 2 presents the API, section 3 presents several examples of Scheme macros that use this system, and section 4 discuses how this profiling system is implemented.

While Scheme is used in this paper, the same techniques should work in any language with sufficient meta-programming capabilities, such as Template Haskell, C++ Templates, or MacroML.

2. API

This section presents the API to the profile system used in later examples.

The system requires the following four primitives. We present the primitives here so the reader can understand the examples in the next section, but delay discussion of the implementation until section 4.

- `profile-query-weight`
- `profile-load-data`
- `profile-dump-data`
- `compile-profile`

`compile-profile` is a parameter used to enable profiling. `compile-profile` is `#f` by default and can be set to `'source` or `'block`. When `compile-profile` is `'source`, compiled programs are instrumented to collect source level profile information that can be used in macros and in the compiler front-end. When `compile-profile` is `'block`, compiled programs are instrumented to collect block-level profile information that can be used in the compiler back-end.

`profile-dump-data` is used to dump any profile information that has been collected to a file.

`profile-load-data` is used to load previously dumped data.

`profile-query-weight` is used to retrieve the weighted profile count of an syntax or source object. Profile counts must have been loaded from a file via `profile-load-data`. When writing Scheme macros, we primarily use syntax objects since the macro system manipulates Scheme syntax objects, but manually constructing source objects from a source file and expression position can be useful when manipulating generated Scheme code that should correspond to some higher-level source code. `profile-query-weight` returns the weighted profile information or `#f` if there is no profile information associated with the syntax or source object.

3. Examples

This section presents several macros that use profiling information to eptimize the expanded code. The first example is `exclusive-cond`, which was mentioned in section 1. The second example is a profile directed loop unrolling macro. While loop unrolling can be done with block-level profiling, it is simple to do as a macro and avoids the problem of reconstructing loops from basic-blocks. The final example is a sequence datatype that is conditionally represented using a linked-list or a vector, depending on profile information.

3.1 exclusive-cond

`cond` is a Scheme branching construct, described briefly in in section 1. Figure 1 shows the various forms of a `cond` clause. 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.

```
(cond
  [ls (car ls)]
  [ls => car]
  [(or ls (car ls))]
  [else '()])
```

Figure 1: An example of `cond`'s clause syntaxes

The first clause has a test on the left-hand side and some expression on the right-hand side. If the left-hand side evaluates to a true value, then the right-hand side is executed. The second form passes the value of the left-hand side to the function on the right-hand side only if the left-hand side evaluates to a true value. In Scheme, any value that is not `#f` is true, so this can be used to post-process non-boolean true values. The third form simply returns the value of the left-hand side if it evaluates to a true value. The last form is equivalent to the clause `(e => (lambda (x) x))`.

The `exclusive-cond` macro, figure 2, shows an implementation of `cond` that 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. Unfortunately, this means we cannot [1] implement the third syntax in figure 1 which has only a left-hand side.

In order to sort the clauses, all clauses are parsed before the code is generated. `exclusive-cond` first parses each clause into a clause record. 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.

Maybe why we pick an expression from the body of each clause here, instead of up there

After parsing each clause, the clause records are sorted by the profile weight. Once sorted, a `cond` expression is generated by emitting each clause in sorted order. If an `else` clause exists then it is emitted as the final clause.

Figure 3 shows an example of `exclusive-cond` and the code to which it expands. In this example, we assume `e1` is executed 3 times, `e2` is executed 8 times, and `e3` is executed 5 times.

3.1.1 case

`case` is a pattern matching construct 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.

Figure 4 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

```
(define-syntax exclusive-cond
  (lambda (x)
    (define-record-type clause
      (nongenerative)
      (fields (immutable clause) (immutable count))
      (protocol
        (lambda (new)
          (lambda (e1 e2)
            (new e1 (or (profile-query-
weight e2) 0))))))
    (define parse-clause
      (lambda (clause)
        (syntax-case clause (=>)
          ; [(e0) (make-clause clause
???)]
          [(e0 => e1) (make-
clause clause #'e1)]
          [(e0 e1 e2 ...) (make-
clause clause #'e1)]
          [_ (syntax-
error clause "invalid clause")])))
    (define (helper clause* els)
      (define (sort-em clause*)
        (sort (lambda (cl1 cl2)
                (> (clause-
count cl1) (clause-count cl2)))
          (map parse-clause clause*)))
      #`(cond
          #,@(map clause-clause (sort-
em clause*))
          #,@(if els #`(,els) #'())))
    (syntax-case x (else)
      [(_ m1 ... (else e1 e2 ...)) (helper #'(m1 ...)
      [(_ m1 ...) (helper #'(m1 ...) #f)])))
```

Figure 2: Implementation of `exclusive-cond`

```
(exclusive-cond
  [(fixnum? n) e1] ; e1 executed 3 times
  [(flonum? n) e2] ; e2 executed 8 times
  [(bignum? n) e3] ; e3 executed 5 times
  [else e4])

(cond
  [(flonum? n) e2] ; e2 executed 8 times
  [(bignum? n) e3] ; e3 executed 5 times
  [(fixnum? n) e1] ; e1 executed 3 times
  [else e4])
```

Figure 3: An example of `exclusive-cond` and its expansion

[1] By manually hacking source objects, it may be possible but would not be pretty.

```
(case x
  [(1 2 3) e1]
  [(3 4 5) e2]
  [else e3])
```

Figure 4: An example of a `case` expression

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 5: The expansion of figure 4

Figure 5 shows how the example `case` expression from figure 4 expands into `exclusive-cond`. Note the duplicate 3 in the second clause is dropped to preserve ordering constraints from `case`.

3.2 Loop Unrolling

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

Profile directed loop unrolling could be done using block-level profile information. However, loop unrolling at the block-level requires associating loops with basic blocks and cannot easily handle arbitrary recursive functions. As this example shows, doing loop unrolling as a macro is simple and can easily handle recursive functions.

Note that in our implementation we wait until after macro expansion to unroll loops. We pass the source-level profile information associated with function calls through the compiler until more loops can be exposed than only those created by a single macro.

This paragraph seems out of place.

A loop can be written using a named let in Scheme, as shown in figure 6. This defines a recursive function `fact` and calls it with the argument `5`. This named let might normally be implemented using `letrec` as seen in figure 7.

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

Figure 6: The most executed program in all of computer science

Figure 8 defines a macro, `named-let`, that unrolls the body of the loop between 1 and 3 times, depending on profile information.

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

Figure 7: a simple definition of a named let

```
(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
                         (+ 1 (* 3 (/ profile-
weight 1000)))])
                  #`(define-syntax name
                      (let ([count #,unroll-
limit]
                            [weight #,profile-
weight])
                        (lambda (q)
                          (syntax-
case q ()
                            [(_ enew (... ...))
                              (if (or (= count 0)
                                      (< weight 100))
                                  #'(tmp enew (... ...
                                  (begin
                                    (set! count (- cou
                                    #'((lambda (x ...)
                                         enew (... ...))
             b1 b2 ...)])
            tmp)
          e ...)])))
```

Figure 8: a macro that does profile directed loop unrolling

The macro uses profile information associated with the body of the loop to determine how frequently the loop is executed. Loops that are executed less than a certain threshold are not unrolled at all. If a loop is executed more often than any other expression, then it may be unrolled 3 times. Note that in `named-let` the name of the loop is not assignable, as it is in the standard Scheme named let.

Reword that last sentence

A named let may have multiple recursive calls, some of which may be more frequently used than others. 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. This example unrolls all call sites the same number of times.

Similar macros are easy to write for `do` loops, and even `letrec` to unroll general recursive functions. Even in the `named-let` example, calls to the loop do not need to be tail calls, so `named-let` can unroll some recursive functions and not just loops.

I don't like that paragraph

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

Figure 10: Use of the define-sequence-datatype macro

3.3 Data type Selection

The previous optimizations focus on low level changes that can improve code performance. Reordering clauses of a `cond` can improve speed by maximizing straight-line code emitted later in the compiler. Loop unrolling can reduce overhead associate with loops and maximize straight-line code emitted later in the compiler. While profile directed meta-programming enables more of such low level optimizations, it also enables higher level decisions normally done by the programmer

Consider a program in which a sequence type is required but the it is not obvious what should be used to implement the sequence. The example in figure 9 chooses between a list and a vector using profile information. If `seq-set!` and `seq-ref` operations are used more often than `seq-map` and `seq-first`, then a `vector` is used, otherwise a `list` is used.

I don't like the first sentence.

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.

Definitely going to need Kent to check this section.

Figure 10 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 requires the function names The unique source information attached to each function name is used to profile the operations of that *particular* sequence. The definitions of each operation evaluate 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.

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.

Make mention of how this affects pattern-matching optimizations, i.e. a compiler that uses nanopass.

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.

Mention how profile info can be used for

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.

This explanation is probably wrong

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.

Maybe an example of creating fake sources

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.

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

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.

Percent of Max

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

We choose percent of max because this compares each statement to the most frequently executed statement. We considered comparing to the total number of statements executed, but this can skew results when a large number of statements are executed infrequently. In that case, a main loop might look infrequently executed if there are many start up or shut down steps.

This weighted average is not perfect. Loop unrolling can benefit from exact counts. 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.

4.3 Source + block profiling

Not sure where this section belongs

Optimizations based on source-level profile information may result in a different set of blocks than the blocks generated on a previous run of a program. If blocks are profiled naively, for instance,

```
(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 defs) => 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 9: a macro that defines a sequence datatype based on profile information

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. We use the following workflow to take advantage of both kinds of optimizations. First a program is compiled and instrumented to collect source-level information. A profiled run collects only the source-level information. The program is recompiled and optimized using that source-level information, and instrumented to collect block-level information. A profiled run collects only the block-level information. The program is finally recompiled and optimized using both the source-level information and the block-level 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).

5. Related and Future Work

I'm not sure what I'm doing with this section yet.

Modern systems such as GCC, .NET, and LLVM use profile directed optimizations [3, 4, 5]. 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.

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 [1]. 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 [5]. By probing the values used in a switch statement, the compiler can attempt to reorder the cases of the `switch` statement.

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.

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 [1, 2]. However, our techniques for reducing the number of counters and our careful representation of profile forms allows accurate source profiling with little overhead .

measure overhead on a standard set of benchmarks. The benchmarks I

Bibliography

[1] 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`

[2] 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`

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

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

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