Visualizing the Performance of Higher-Order Programs*

Oscar Waddell
Indiana University
owaddell@cs.indiana.edu

J. Michael Ashley
University of Kansas
jashley@eecs.ukans.edu

Abstract

Profiling can provide the information needed to identify performance bottlenecks in a program, but the programmer must understand its relation to the program source in order to use this information. This is difficult due to the tremendous volume of data collected. Moreover, program transformations such as macro expansion and procedure inlining can obscure the relationship between the source and object code. Higher-order programs present additional challenges due to complex control flow and because they often consist of many small, often anonymous, procedures whose individual performance properties may be less interesting than their characteristics as a group.

To address these challenges we have implemented a profiler and interactive profile visualizer and integrated them into an optimizing Scheme compiler. The profiler instruments target code and maintains correlation with the original source despite compiler optimizations that can eliminate, duplicate, or move code. The visualizer operates as a source-code browser with features to examine execution counts and execution times from several perspectives. It supports the programmer in identifying program hot spots as well as code regions responsible for or affected by those hotspots. It also supports profile differencing which permits the programmer to study program behavior in different execution contexts.

Our experience suggests that visualization tools can help to present raw profile data in a meaningful way. The tool can synthesize a high-level picture of program performance while still giving the programmer the ability to explore the details in interesting regions of code.

*This material is based on work supported in part by the National Science Foundation under grants CDA-9312614, CDA-9401021, and CCR-9623753.

1 Introduction

It is difficult to tune the performance of applications written in higher-order programming languages such as Scheme and ML because these languages provide a high degree of abstraction from stock hardware. This abstraction permits the compiler great freedom in mapping programs to stock hardware, but it also makes it difficult for the programmer to understand an application's performance properties.

To assist the programmer, a compiler may instrument an application for profiling. Correlating the collected information back to the programmer is a nontrivial task, however. In languages with sophisticated hygienic macro systems [7], code is subject to significant source-to-source rewrites before it even reaches the compiler. Subsequent compiler optimization may further transform the code, making its relationship to the original source even less clear. When instrumenting the code for profiling, the compiler must preserve sufficient information to correlate the instrumented code with the original source.

Presenting profile metrics to the user is also a challenge when working with high-level languages and higher-order languages in particular. Higher-order languages encourage the use of many small procedures, so it is essential that the presentation environment have the ability to synthesize interprocedural metrics based on intraprocedural information. Due to the volume of profiling information collected, it must also be possible to present the information to the programmer in a manageable format that permits easy browsing of the data and quick identification of performance problems.

To address these issues we have developed techniques for profiling higher-order programs and have implemented a profiler based on these techniques. The profiler is part of an optimizing Scheme compiler that may duplicate, eliminate, and move code. The compiler generates instrumented code to collect raw execution counts and timing information, and the raw information is correlated with the original source program. A static control-flow graph is also used to synthesize interproce-

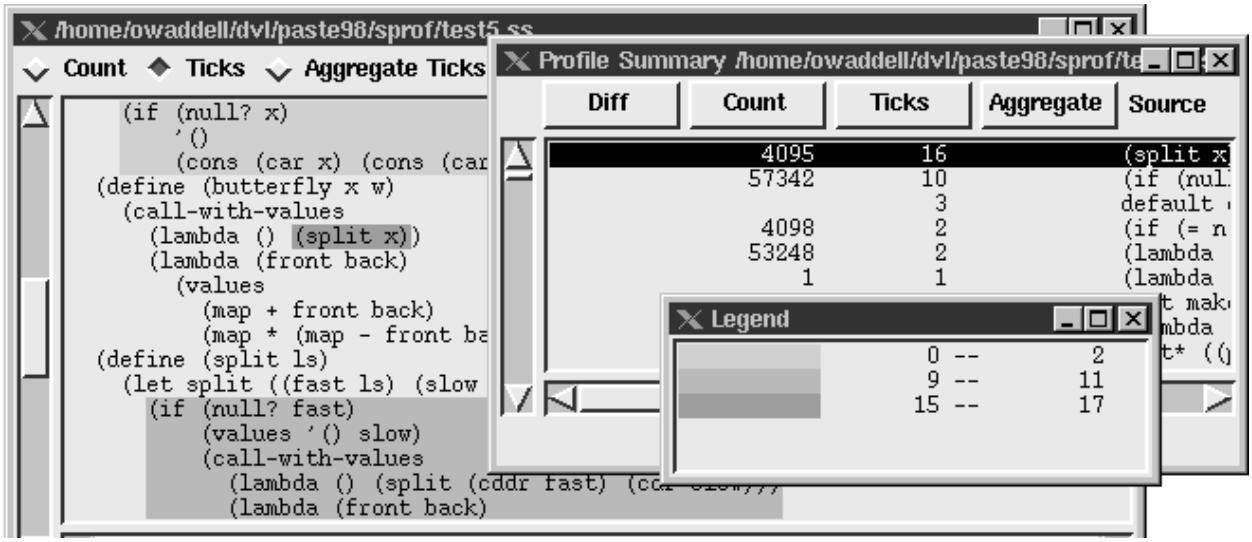

Figure 1: Profile results for a fast Fourier transform are tabulated in the profile summary window sorted by execution time. Expressions in the program source are colored on the basis of execution time. The user has selected the first row in the summary window and the corresponding expression, `(split x)`, has been scrolled into view.

dural timing information, but the tool may nevertheless be used in situations in which control-flow information is imperfect, *e.g.*, when the whole program is unavailable. Our tool provides considerably more information than conventional "gprof-style" profilers which simply tabulate execution counts and time spent in each procedure. In particular we provide source-level tools that enable the programmer to

- inspect profile metrics on a per-expression or per-procedure basis,
- color the program based on a selected metric to visualize global behavior,
- browse program control flow to understand in what context and how often control arrived at interesting program points, and
- compare different profile runs to compare program performance when run with different inputs, or compiled with different optimizations, etc.

The rest of the paper is outlined as follow. Section 2 describes the visualization tool and its use. Sections 3 and 4 describe the collection of raw profile data and the way this data is combined with static control-flow information to construct the performance database used by the visualization tool. Section 5 gives related work. Section 6 gives our conclusions.

2 Visualizing profile information

When tuning the performance of programs we have two goals. First, we want to obtain good absolute performance. Second, we want to ensure that performance scales gracefully with problem size. To meet these goals, we must first locate and then eliminate the cause of the performance problems. To assist the programmer in these tasks, we have constructed tools that integrate the results of static flow analysis and dynamic execution profiles in a common database and provide high-level and source-level views of this information.

2.1 Obtaining good absolute performance

To identify areas of interest quickly, the programmer usually consults the profile summary window first. For each expression the profile summary window displays execution frequency, execution time, and abbreviated source code, as well as the aggregate time attributed to that expression. The display can be sorted on the basis of different metrics by clicking on an appropriate column. Selecting a row from the table opens the corresponding source file in another window and highlights the appropriate expression. In this way the programmer can quickly survey regions of the program that have the highest cost according to some metric, say execution time, as in Figure 1.

The source-code display provides a graphical interface to an unified database containing the results of static and dynamic program analysis. Because the programmer is most familiar with the source code, this is an intuitive way to present program properties in context. Color is used to display values for particular metrics at different points in the program. For example, each expression in the program can be colored according to its execution frequency.

The source-code display also provides a convenient query interface. By clicking on an expression the user

can inspect the database entry containing all the static and dynamic information we have collected for that expression. For example, clicking on a call site lists the procedures called from that site sorted by execution frequency. Clicking on a procedure lists the call sites from which that procedure was called, again sorted by execution frequency. Selecting an item in one of these lists highlights and scrolls into view the corresponding call site or procedure. This interface makes it easy to navigate the control flow graph, guided by execution frequency, when trying to understand why an expression is evaluated so often. By integrating information from static and dynamic analyses, the source-level query interface supports programmer efforts to understand why a particular performance problem exists and how to correct it.

2.2 Obtaining scalable performance

To address the goal of scalable performance we provide a convenient interface for comparing execution profiles from different runs of the same program. The programmer can specify an arbitrary function to be used in comparing the profiles. This function is applied to corresponding entries in the profiles and these results are recorded in the database where they may be included in the sorted tabular display or used to color the source program text. For example, to identify areas of the program with non-linear time complexity, we might compare the profiles generated when a program is run on one problem of size x and another of size $2x$.

Figure 2 compares two runs of a program that computes Fibonacci numbers using the doubly-recursive algorithm and a more efficient iterative algorithm. An input of ten was used for the first run and twenty for the second run. The first column in the profile summary window shows the difference for each expression between the execution count obtained in the second run and the execution count predicted by scaling the results from the first run to account for the change in input size. The line highlighted in the summary window corresponds to the body of the iterative loop. In the first run of the program this body was executed ten times. In the second run the loop body was executed twenty times, as predicted by scaling, so the difference is zero. A significant difference is evident for the doubly-recursive version which stands out in the color-coded source display.

3 Collecting intraprocedural profile data

To enable profiling a compiler switch is set to instruct the code generator to instrument generated code. Execution counts are collected for each basic block, but counters are not placed on each edge in the intraprocedural flow graph. Instead, counters are placed only where necessary, and compile-time heuristics are used to minimize counter placement [5, 6]. A counter increment requires a load, a register increment, and a write. Minimizing counter placement reduces perturbations in code size and execution time.

Execution time is collected using *cost centers* [3, 11] to correlate the source and object code. The current cost center is maintained in a dedicated memory location and its count incremented by a timer interrupt handler. A cost center is associated with the code for each procedure in the program. On procedure entry the code's cost center is installed as the current cost center. At call sites the current cost center is untouched by the caller. This allows the cost of unprofiled code to be charged to the caller. On return from nontail calls, the current cost center is reloaded with the caller's cost center.

No special action is taken for tail calls, although this introduces an inconsistency between Scheme's semantics and profile data collected. Scheme implementations are required to handle tail calls as jumps. For example, if `f` calls `g` and `g` tail calls `h`, then the execution stack will record `f` as `h`'s caller. Suppose that `h` is unprofiled. Then by taking no special action for tail calls the profiler will attribute the cost of `h` to `g` instead of `f`. Despite the inconsistency, we have found that handling tail calls in this fashion is what the programmer expects.

Profile data collected at runtime is correlated with the original source program. The compiler arranges this by associating with each parse tree node the file name and position corresponding to the original source expression. These source records are preserved through macro expansion [7] and the rest of compilation. The code generator associates each cost center with the source record identifying the source procedure and each execution counter with a source record identifying the basic block.

4 Synthesizing interprocedural profile data

The raw profile information collected is used to synthesize aggregate execution times and interprocedural call counts. The aggregate execution time of a procedure is the execution time of the procedure itself plus the execution time of its callees. An interprocedural call count is the number of times a procedure is called from a given call site. Because procedure bodies in higher-order languages are often small, aggregate execution time and interprocedural call counts are useful metrics to report to the programmer.

Some form of control-flow graph is needed in order to compute interprocedural call counts and aggregate execution times. Our tool uses a compile-time flow analysis [4] to compute a control-flow graph. The graph contains nodes for procedures and call sites. There is

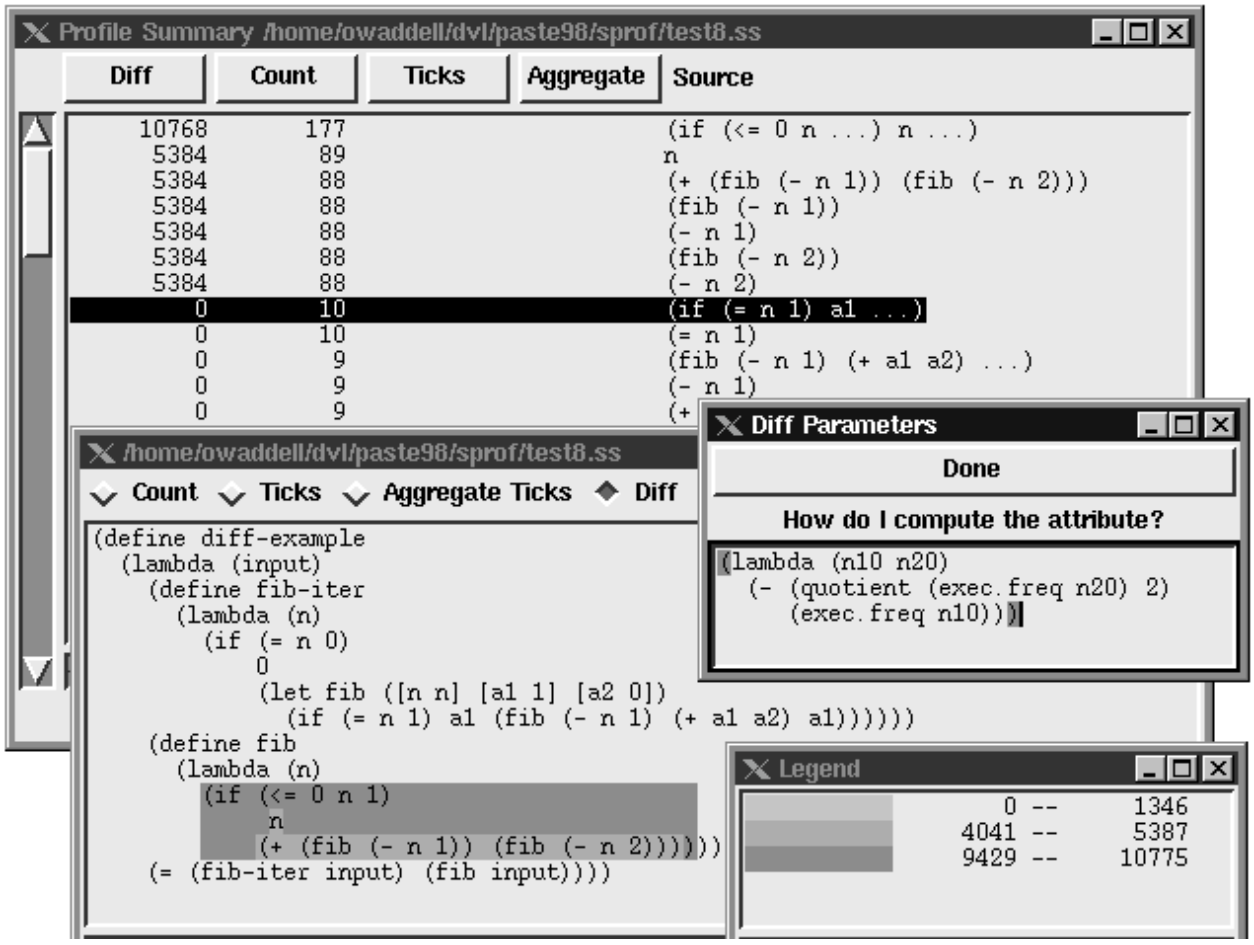

Figure 2: Execution frequencies for two runs of the program are compared to suggest areas with non-linear time complexity. Note that the iterative function is not colored in this view.

an arc from each procedure node to each of the call sites in the procedure's body and an arc from each call site to each procedure that may be called from that call site at run time. Anonymous call sites and procedure nodes are introduced into the control-flow graph when the flow analysis is unable to determine all callers of the procedure or all procedures called from the call site.

Figure 3 shows a small example program that solves the n-queens problem. The corresponding control-flow graph is in Figure 4. A procedure is returned as the value of the program. Consequently the analysis must assume that it escapes, and this is represented in the control-flow graph with the call site marked a. The call site h has two exit arcs since two higher-order procedures may arrive at that site at run time.

The profiler uses the graph to compute interprocedural call counts by labeling each arc in the graph with the number of times it was crossed during execution. This is done by solving a system of linear equations derived from the control-flow graph. Intuitively, the count on a procedure node is the sum of the counts on the arcs into that node. The count on a call-site is the sum of the counts on the arcs out of that node.

More formally, the equation system is generated as follows. For each call site node C_i, let $\text{count}(C_i)$ be the number of times the call was executed and let $\{P_0, \ldots, P_m\}$ be the procedure nodes that are children of C_i. Then for $0 \leq i \leq n$ add

$$\text{count}(C_i) = \sum_{j=0}^{m} \text{arc}(C_i, P_j)$$

to the equation set.

For each procedure node P_j, let $\text{count}(P_j)$ be the number of times that procedure was entered and let $\{C_0 \ldots, C_n\}$ be the call site nodes that are parents of P_j. Then for $0 \leq j \leq m$ add

$$\text{count}(P_j) = \sum_{i=0}^{n} \text{arc}(C_i, P_j)$$

```
(lambda1 (n)
  (define queens
    (lambda2 (board sk fk)
      (if (= (length board) n)
          (sk board fk) c
          (place-queen board (lambda7 (board fk) (queens board sk fk) j) fk) d)))
  (define place-queen
    (lambda4 (board sk fk)
      (letrec ([f (lambda5 (i)
                    (cond
                      [(= i n) (fk) h]
                      [(or (memq i board) (on-diag i board) i) (f (+ i 1))]
                      [else (sk (cons i board) (lambda6 () (f (+ i 1)) f)) g]))) e])
        (f 0) e)))
  (define on-diag
    (lambda9 (i board)
      (letrec ([loop (lambda (board up down)
                       (and (not (null? board))
                            (or (= (car board) up)
                                (= (car board) down)
                                (loop (cdr board) (- up 1) (+ down 1)))))])
        (loop board (- i 1) (+ i 1)))))
  (queens '() (lambda3 (board fk) board) (lambda8 () ``no solution'')) b)
```

Figure 3: A program that solves the n-queens problem using a backtracking algorithm.

to the equation set. Figure 5 gives the equation set for the control-flow graph in Figure 4.

Because the static control-flow graph approximates the actual dynamic control flow, the resulting system of equations may be underconstrained. If so, the resulting system has many solutions and additional constraints must be added until a unique solution is obtained. In particular, if $\text{arc}(C_i, P_j)$ is unconstrained, an estimate $c = \text{arc}(C_i, P_j)$ is added to the set of equations where $0 \leq c \leq \min(\text{count}(C_i), \text{count}(P_j))$. The count c is constrained in this way since a count can neither be negative nor can it exceed the minimum count of the nodes it connects. Suppose $\text{count}(C_i) < \text{count}(P_j)$. Our current implementation chooses c to be $\text{count}(C_i)$ divided by the number of arcs out of C_i, assuming that each is equally likely on average.

The solved equation set is used to decorate the edges of the interprocedural call graph. For the equation set in Figure 5 the solver is able to determine a unique solution, and that solution has been used to decorate the arcs in Figure 4.

Once decorated, the graph is used to compute aggregate execution times. A procedure's aggregate time is conceptually its own execution time plus the aggregate execution time of its call sites. A call site's aggregate execution time is more subtle. Since each procedure at a call site C may be called from multiple points, the execution time attributed to C should be just the sum of a fraction of each callee's execution time. The fraction is the ratio of the number of calls from C to the callee over the total number of calls to the callee.

The aggregate execution time of a node N is computed by traversing the graph recursively starting from N. Aggregate execution time for each node is accumulated as described above. If an edge leads to a node previously encountered, the edge is eliminated from the set of edges used to compute the total number of times a procedure is entered. This ensures that recursive procedures, including loops, are attributed properly. The edge removal in effect coalesces the execution time of recursive procedures at the point execution entered the cyclic subgraph.

The raw profile information, execution counts, confidence factors, and aggregate execution times are used to construct a database of analysis results. The database is indexed by source file positions to support efficient source-level queries. Each entry includes information about the source file name and start and end positions for a particular expression together with the analysis results. The start and end positions are used when highlighting or coloring an expression. When installing profile information we accumulate results for duplicate source records since source records may be duplicated in the compiler output as a result of macro expansion and procedure integration.

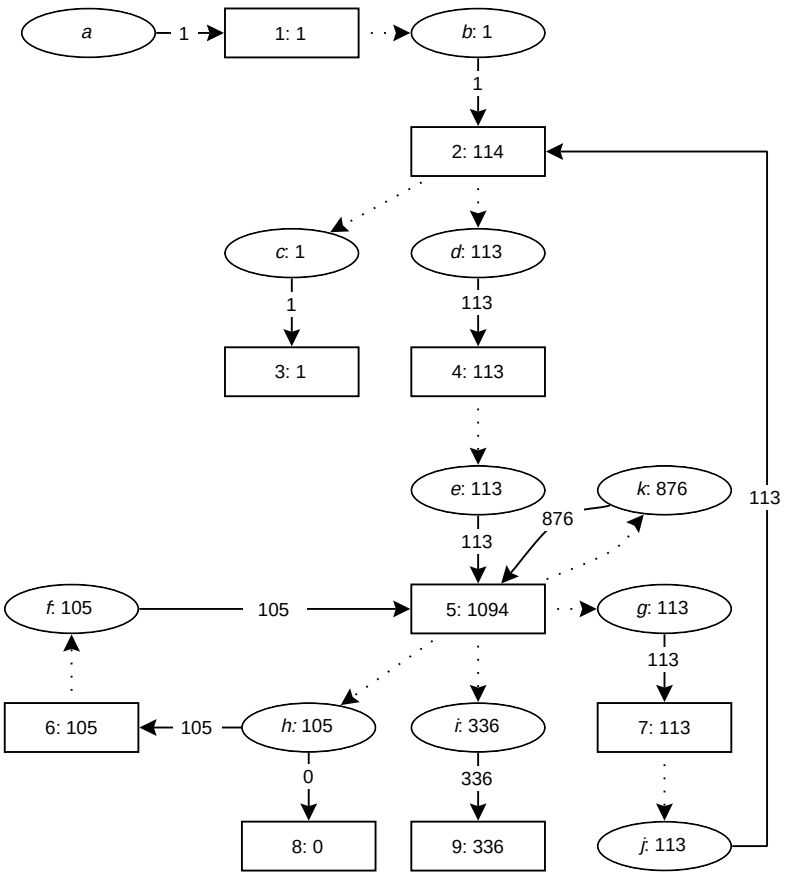

Figure 4: A control-flow graph for the n-queens program; the subgraph for the loop in `on-diag` has been omitted. Boxed nodes indicate procedures and elliptical nodes indicate call sites. Nodes are labeled with the labels embedded in Figure 3 and execution counts from a sample profile run. The arcs have also been labeled with execution counts synthesized from the equations in Figure 5.

$$
\begin{aligned}
\text{count}(1) &= \text{arc}(a,1) \\
\text{count}(2) &= \text{arc}(b,2) + \text{arc}(j,2) \\
\text{count}(3) &= \text{arc}(c,3) \\
\text{count}(4) &= \text{arc}(d,4) \\
\text{count}(5) &= \text{arc}(e,5) + \text{arc}(f,5) + \text{arc}(k,5) \\
\text{count}(6) &= \text{arc}(h,6) \\
\text{count}(7) &= \text{arc}(g,7) \\
\text{count}(8) &= \text{arc}(h,8)
\end{aligned}
\qquad
\begin{aligned}
\text{count}(b) &= \text{arc}(b,2) \\
\text{count}(c) &= \text{arc}(c,3) \\
\text{count}(d) &= \text{arc}(d,4) \\
\text{count}(e) &= \text{arc}(e,5) \\
\text{count}(f) &= \text{arc}(f,5) \\
\text{count}(g) &= \text{arc}(g,7) \\
\text{count}(h) &= \text{arc}(h,6) + \text{arc}(h,8) \\
\text{count}(i) &= \text{arc}(i,9) \\
\text{count}(j) &= \text{arc}(j,2) \\
\text{count}(k) &= \text{arc}(k,5)
\end{aligned}
$$

Figure 5: Equation set derived from the control-flow graph in Figure 4.

5 Related Work

In the realm of higher-order languages there have been few tools for visualizing program properties. For source-level browsing we are aware of only Mr. Spidey, an interactive tool for using static dataflow information to debug Scheme programs [8]. Our environment is similar to theirs in that both require information be collected in terms of macro-expanded code yet be correlated with the original, unexpanded source.

There has been some work on profiling higher-order languages. Appel *et al.* [3] describe a profiler for ML. Our approach to instrumenting code is similar to theirs. They introduce a counter and cost center for each procedure. The counter is incremented on procedure entry, and the cost center is reset upon return from nontail calls. Unlike our environment, however, their profiler does not compute any control-flow information to aggregate profile information. Instead, the programmer must manually uninstrument interesting program points in order to charge metrics to callers. Finally, information collected by their profiler is displayed in tabular form, and no attempt is made to correlate the information with the original source.

Sansom and Peyton-Jones [11] describe a time and space profiler for Haskell. Their profiling methodology requires the programmer to manually instrument the code with named cost centers. Execution times attributed to each cost center are then presented in tabular form. There is also no cost aggregation; as in the Appel approach, cost centers must be removed from interesting program points in order to apportion costs to the callers. Also, no attempt is made to correlate the profile information with the original source,

Our algorithm for aggregating execution times is similar to the one used by `gprof` [9]. Like `gprof`, our algorithm can misattribute time because the static control-flow graph must attribute time to callees based on execution counts only. For example, a procedure `f` may call `g` ten times, and `h` may call `g` one hundred times, but because of the data passed by `f`, most of the execution time spent in `g` should be attributed to `f`. Instead, our algorithm will attribute most of the time to `h`. The only general solution to this data dependence problem is to compute an exact dynamic call tree at run-time, but this is impractical. We mitigate the problem, however, by providing enough information to permit the programmer to understand how time is being (mis)aggregated.

Outside of higher-order languages, programming environments provide better support for displaying profile information. In such environments correlating profile information with the original source is trivial unless the compiler performs aggressive code optimizations. One such prototype environment is a combination of the Pablo performance environment with the Rice University Fortran D compiler [1]. The Fortran D compiler is an aggressive, automatically parallelizing compiler. As in our compiler, the Fortran D compiler must maintain enough source-level information so that instrumentation code can generate information that is ultimately reported to the programmer at source level. Their preliminary experience supports our conclusion that it is crucial that profile information be reported at the source level in order for the programmer to be able to tune the code.

Profile differencing has many applications. For example, by comparing execution profiles collected for different datasets we may discover which parts of a program are affected by the differences in the inputs. This technique was recently developed to help identify portions of legacy code that might be vulnerable to Year 2000 problems [10]. It may also prove interesting to compare profiles generated on different architectures or with different compiler optimizations enabled.

6 Conclusions

Performance tuning is a crucial component of the software lifecycle. A profiler can provide the information needed to identify the code that needs improvement, but that information is useless if the programmer is unable to correlate it with the original source or is overwhelmed by the volume of profile data. For high-level languages, macro expansion and compiler optimizations require the programming environment to provide assistance in correlating the information with the original source. The programming environment should also provide ways of filtering and organizing profile data to enable the programmer to more quickly isolate inefficient code.

We have presented techniques for addressing these issues and a profiling tool for higher-order languages to test them. Profile data is correlated with the source code in a graphical, interactive browser, and mechanisms are provided to enable to programmer to quickly identify interesting program points. The tool is integrated with an optimizing Scheme compiler that is instrumented to collect execution counts and timing information.

Our preliminary experience with the profiler is encouraging. We have used it to quickly identify hot spots in several programs. We have also used it to identify nonlinear behavior in programs by comparing profile runs from datasets of different sizes. We plan to gain more experience with larger applications such as the compiler itself. We intend to use this experience to improve the profiler's code instrumentation strategy and the browser's reporting mechanisms.

The equation systems generated to synthesize interprocedural data are sometimes underconstrained, and this results in multiple solutions. Underconstrained equation systems are inevitable when an approximate

control-flow graph is used, and the current implementation arbitrarily picks a solution. A much better approach would be to provide a dial that permits the user to browse the solution space. When the dial is turned the synthesized profile information is adjusted, and the programmer can use knowledge about the program's behavior to find an appropriate solution. For example, the programmer might turn the dial to set execution counts on error cases to zero.

Another problem is the misattribution of execution time. We are considering using call context trees (CCTs) [2] to build more precise control path contexts. A CCT is constructed at runtime and more closely approximates the dynamic call tree than the control-flow graph computed by our analysis. It is unclear, however, if CCTs can be computed practically for Scheme programs in the presence of tail calls and first-class continuations. If CCTs can be used, we then intend to combine them with our dataflow analysis to estimate data dependencies on control-flow paths. This additional information should assist the programmer in identifying why particular paths are performance bottlenecks, and our visualization tool can be easily adapted to this improved profiling infrastructure if it can be realized.

Acknowledgements

We would like to thank Kent Dybvig, George Springer, and Brian Moore for helpful discussions about profiling and solving systems of linear equations.

References

[1] Vikram S. Adve, John Mellor-Crummey, Mark Anderson, Ken Kennedy, Jhy-Chun Wang, and Daniel A. Reed. Integrating compilation and performance analysis for data-parallel programs. In M. L. Simmons, A. H. Hayes, D. A. Reed, and J. Brown, editors, *Proceedings of the Workshop on Debugging and Performance Tuning for Parallel Computing Systems*. IEEE Computer Society Press, January 1996.

[2] Glenn Ammons, Thomas Ball, and James R. Laurus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, pages 85–96, June 1997.

[3] Andrew W. Appel, Bruce F. Duba, David B. MacQueen, and Andrew P. Tolmach. Profiling in the presence of optimization and garbage collection. Technical Report CS-TR-197-88, Princeton University, November 1988.

[4] J. Michael Ashley. A practical and flexible flow analysis for higher-order languages. In *Proceedings of the 23rd Annual ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 184–194, January 1996.

[5] Thomas Ball and James Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1319–1360, July 1994.

[6] Robert G. Burger. *Efficient Compilation and Profile-Driven Dynamic Recompilation in Scheme*. PhD thesis, Indiana University, March 1997.

[7] R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation*, 5(4):295–326, 1993.

[8] Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, and Stephanie Weirich. Catching bugs in the web of program invariants. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, pages 23–32, June 1996.

[9] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. gprof: a call graph execution profiler. In *Proceedings of the ACM SIGPLAN 1982 Symposium on Compiler Construction*, pages 120–126, 1982.

[10] Thomas Reps, Thomas Ball, Manuvir Das, and James Larus. The use of program profiling for software maintainence with applications to the year 2000 problem. In *Proceedings of ESEC/FSE 97*, volume 1301 of *Lecture Notes in Computer Science*, pages 423–449. Springer-Verlag, 1997.

[11] Patrick M. Sansom and Simon L. Peyton Jones. Formally based profiling for higher-order functional languages. *ACM Transactions on Programming Languages and Systems*, 19(2):334–385, March 1997.