Using Path Spectra to Direct Function Cloning

Tom Way   Lori Pollock

Department of Computer and Information Sciences
University of Delaware, Newark, DE 19716
(302) 831-1953   (302) 831-8458 (Fax)
{way,pollock}@cis.udel.edu

Abstract

While function cloning can improve the precision of interprocedural analysis and thus the opportunity for optimization by changing the structure of the call graph, its successful application relies on the cloning decisions. This paper explores the use of program spectra comparisons for guiding cloning decisions. Our hypothesis is that this approach provides a good heuristic for determining which calls contribute different dynamic interprocedural information and thus suggest good candidates for cloning for the purpose of improving optimization.

Keywords: function cloning, path spectra, profile-guided optimization

1 Introduction

As modular and object-oriented programming are becoming the norm, and architectures offer more available fine-grain parallelism, the importance of analysis and optimization across function boundaries continues to increase. While function inlining allows each function to be optimized within the separate context of each call site, its well known limitation is the potentially exponential code growth and associated increased compile time. The most common alternative is interprocedural data flow analysis, which avoids the code growth, but is limited by the program's original calling structure, in particular, making conservative assumptions at the convergence of paths in the call graph. Function cloning seeks to play an intermediary role by partitioning the calls in the program and creating multiple copies of the function body, one for each set of "closely related" calls. This graph restructuring can create opportunity to compute more precise information during interprocedural analysis because each node representing a copy of the function has fewer incoming edges on which information must be merged during analysis; cloning also seeks to avoid the potential exponential code growth of inlining.

This paper explores the use of path spectra to guide cloning decisions. By instrumenting the program, a profiler can report the frequency of execution of individual paths executed during a program run [2][1]. For a particular program run, the path spectrum is the set of paths executed along with their execution frequencies [14]. As an indication of program behavior, path spectra have been used successfully for program optimization [1, 8, 9, 10] as well as software maintenance and testing [14].

Our approach to making cloning decisions is based on comparing the program spectra for the execution of the same function by different calls to that function. Our hypothesis is that this approach provides a good heuristic for determining which calls contribute different dynamic interprocedural information and thus suggest good candidates for cloning. Path profiles indicate run-time control flow, which will have an effect on the actual dynamic data flow and data dependencies within the callee; path profiles for different calls to the same function indicate differences in data values flowing into the function through parameters or via global variables. In addition, profile-guided optimization and path-qualified data flow analysis within the cloned functions may benefit from creating clones based on similar run-time paths through the function. The technique of using program spectra comparison for cloning decisions is different from, but could be used in combination with, the use of program profiling to uncover the most frequently executed calls to guide inlining decisions.

The objective of cloning decisions is threefold: (1) increase program performance, (2) manage compile time, and (3) control code growth. We have been evaluating our technique in terms of its effectiveness in making intelligent cloning decisions that will lead to improved optimization as well as its associated overhead.

[1]Actually, the length of the paths that are profiled must be limited, as there can be an unbounded number of paths for general flow graphs and an exponential number for directed acyclic graphs (i.e., flow graphs without loop backedges included).

Our preliminary experiments have focused on comparing this approach to goal-directed cloning [11] which is based on static interprocedural data flow analysis, directed toward particular optimizations. Thus far in our limited study, we have found that cloning based on path spectra comparisons can produce results comparable to that of goal-directed cloning for constant propagation, with compile-time and run-time analysis that can be reused for cloning that is performed for multiple optimization goals.

We first describe the related work on cloning, profiling, and path spectra comparisons. A simple motivating example for our approach is presented, followed by our profile-directed cloning algorithm. We describe our path spectra comparison technique which is used to partition call sites for cloning. Finally, this method is compared to goal-directed cloning decisions on an example code, and we make some concluding remarks and summarize our future work.

2 Related Work

Goal-directed cloning [6, 11] first solves a forward interprocedural data flow problem with slight modification in order to compute a set of cloning vectors for the particular data flow problem of interest at each call graph node. Cloning vectors which produce equivalent effects on the optimization of interest are merged, and finally the cloning is performed until the program size reaches some threshold. Cooper et al. [6] presented an experiment on the `matrix300` code from release one of the SPEC benchmark suite, in which they showed that significant improvement in code quality could be attained by using this method to expose sufficient information to perform inlining and unroll and jam. The approach requires either knowledge of what optimizations would most likely benefit from cloning in order to focus on that forward data flow problem, or a separate clone decision-making phase for each optimization of interest, and then some heuristic to select the clones suggested by each phase. Only goal-directed cloning for constant propagation has been investigated. It is not clear how easy it would be to construct a goal-directed partitioning algorithm for other optimizations or how much goal-directed cloning could improve the opportunity for other optimizations, or even more challenging, for multiple simultaneous optimization goals.

A general framework for *selective specialization*, the equivalent of cloning for object-oriented languages, combines static analysis and profile data to identify the most profitable specializations [7]. This goal-directed technique helps to reduce the number of expensive dynamic dispatches, providing significant improvements in performance and reduced code growth over *customization* [5], the previous state-of-the-art specialization technique. In dynamic compilation environments, cloning is sometimes performed on the fly as a statement is executed the first time [13]. To our knowledge, none of these techniques has used path spectra comparison in their cloning decisions.

Path profiling has been used successfully in compiler optimization [1, 9, 10]. Other basic types of control flow profiling are edge profiling, which measures the execution frequency of each individual flow graph edge, and basic block profiling, which measures how many times each basic block is executed. Edge profiles are good predictors of frequently executed paths (hot paths) for programs with a large amount of definite flow relative to total flow, while path profiles are better when there is less definite flow [3]. Path profiles can be collected efficiently and provide more accurate information than edge profiling [2]. In particular, different path profiles can result in the same edge profile, making it impossible to accurately compute the execution frequency of paths based on edge profile information. In this paper, we use path profiling rather than edge profiling as we are interested in the differences, or actually, the similarities in the run-time behavior of different invocations of a given function. Differences in path spectra obtained from two different calls to a function with different parameter values indicate differences in the execution states, and therefore the function's behavior, due to differences in the parameter values. Edge or basic block profiling could be used, but comparisons of spectra created from path profiles generally will give better predictions of differences in run-time behavior.

When variables cannot be conservatively identified as constants at compile time, value profiling [4] can be used to determine whether they exhibit a high degree of invariant behavior at run-time. Value profiling records information about the invariance of a variable, typically the top N values for an instruction and the number of occurrences for each of those values. In addition to other compilation and optimization decisions, this information can be used for duplicating and specializing code conditioned on particular high frequency values. While value profiling will certainly aid in cloning decisions with the goal of specializing on constant values, we believe that cloning that utilizes path spectra differences as proposed in this paper may provide a more general, or at least complementary, approach to judging the run-time behavioral differences between separate calls to the same function, and enable cloning decisions that help in multiple different optimizations of the called function.

3 A Motivating Example

Consider the control flow graph in Figure 1 for a procedure *P* called from two call sites, *S1* and *S2*, and the following facts about *P*, collected through static program analysis and from runtime profiling:

1) The decision to take path 1-2-4 or path 1-3-4 is determined by a comparison in node 1, using the value of one of the procedure arguments.

2) Path profiling reveals that 1-2-4 is executed 100 times and path 1-3-4 is executed 10 times.

3) Constant propagation analysis discovers that one of only two constant values are passed in and used for the branch decision in node 1. It is also discovered that each call site passes a different one of these two values, causing a different path from the branch to be executed.

By basing cloning decisions on a goal-directed approach with constant propagation [6], two clones of *P* are produced. One clone is produced for each call site, since an "important constant" can be discovered and propagated.

If cloning decisions are based instead on path profiles, or path execution frequency, or both, the cloning decision will be the same as that produced by goal-directed cloning. The instrumentation and collection of profiling information is performed only once, with the data being reused as needed for subsequent optimizations and decision-making.

An advantage of the goal-directed method is that it fine-tunes the cloning decisions to a specific optimization, enabling potentially better code improvement than the more general path-profile-guided method. A disadvantage of the goal-directed approach is that it may produce unnecessary clones in the event that the particular optimization goal identifies an unimportant difference.

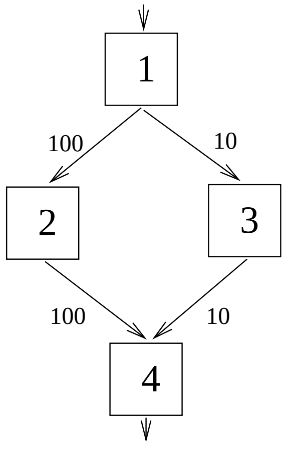

Figure 1: Procedure P's control flow graph with path frequencies.

For example, if the two propagated constants mentioned above caused less orthogonal branch behavior, clones may be produced where not needed, leading to additional code growth. Suppose procedure *P'* replaces *P*, and produces two distinct path profiles, *p1* and *p2* depending on some constant value passed at each of *S1* and *S2*. Based on constant information, it appears that there are four unique cloning vectors. Ideally, the portion of calls to *P'* that produce path profile *p1* should be merged, as should those that produce path profile *p2*. However, using constant information alone, it is possible that four clones may be produced, rather than two. This is not a problem in smaller program codes, but could significantly impact larger programs.

Dynamic path information offers the advantage of guiding optimization decisions based on actual program performance characteristics. Repeated goal-directed static analysis at compile time can be costly, particularly for very large program sources. Alternatively, a single execution profile can provide guidance for numerous optimization decisions while incurring only a one-time overhead.

The advantages and disadvantages of one method versus the other are not clear-cut. Our research is motivated by the hypothesis that using path profiles and path execution frequency information can potentially enable equal or better optimization decision-making in general while reducing compilation overhead.

4 Cloning Algorithm

Profile-guided cloning consists of the following steps:

1. For each potentially cloned function f, create a path profile for each call *c* to f by instrumenting and running the program. For each call *c*, the path profile consists of an acyclic path profile starting at the call site, through f's entry, and ending at the exit of f. We profile only acyclic paths in order to limit the length and number of paths profiled. Acyclic paths can be created by ignoring nodes and edges within loop bodies [10], by restructuring the control flow graph to replace each backedge by an entry-to-loop header edge and a backedge source-to-function exit edge [14], or by only recording the first visit to each node during a path profile.

2. For each site s where f is called, merge the path profiles for each call made from s into a path spectrum, including frequency information per path through f from this call site s. This gives a path-count spectrum [12] per call site s.
3. Perform path spectra comparison among the path spectra of the different call sites where f is called. Various comparison algorithms could be used for this step. Based on the comparison results, apply a heuristic to perform the partitioning of call sites for f into sets that are similar in the dynamic behavior of their function invocation.
4. Perform the cloning of f indicated by the partitioning.

The next subsection concentrates on the details of step 3, the method for comparing two path spectra and how this comparison is used to partition call sites for cloning. The subsequent subsection compares the overhead of profile-guided cloning with goal-directed cloning.

4.1 Partitioning Path Profiles

For two path spectra resulting from two different calls to the same function to be judged similar for purposes of grouping in the same partition, the similarities must be quantified in a manner that captures the dynamic behavior of the associated function invocations. The path spectrum is a set of ordered sequences of nodes representing acyclic portions of the program execution through a function, while the frequency with which these paths were taken expresses the relative importance of the paths.

The techniques for comparing path spectra for purposes of software maintenance and testing [12, 14] focus on locating the potential causes of dynamic behavioral changes in program execution due to changes in program input. In [14], they identify paths that occur in the spectrum of one run, but not the other run, and use the shortest prefix of these paths that does not occur in paths of the other run as starting points for locating potential behavioral changes. In our context, we are interested instead in quantifying the *diff*, or *similarity*, of the path spectra of different invocations of the same function through different call sites.

Our current heuristic for partioning call sites uses edit-distance and frequency-distance measurements for pairs of paths. Two paths are judged to be similar if each distance falls within some appropriately determined threshold range. For the set of path profiles corresponding to all call sites for a given function, the path profiles are pairwise compared for both edit- and frequency-distance similarity. If found to be similar, paths are merged into the same partition. If found dissimilar, the paths are put into different partitions. This process is performed for each of the functions in the program.

The *edit-distance* between two paths, or path profiles, is described as follows: A path P is said to be of distance k to a path Q if we can transform P to be equal to Q with a sequence of k insertions, deletions and subsitutions of single path nodes in P.

The *frequency-distance* is the difference between the execution frequencies of two path profiles, expressed as the ratio of the lesser frequency over the greater frequency. This expresses the distance as closer to 1.0 for similar frequencies, and closer to 0.0 for less similar frequencies.

Comparison of path profiles is accomplished using an adaptation of an *edit-distance* or *Levenshtein measure* algorithm. This algorithm is widely used for approximate string matching [15, 16], and other approximate pattern matching tasks. Our method includes a *frequency-distance* measure as a second point of comparison.

Experimentation is needed to determine the quality of this comparison technique for our purposes, and to discover appropriate threshold values for quantifying similarity of edit-distance and frequency-distance. Investigation of other heuristics, including using only edit-distance or frequency-distance, or incorporating other static or dynamic information, is needed. Other basic path profile comparison techniques that we are considering include:

1) Measure the length of the common sub-path prefix shared by the two paths, with longer prefixes indicating greater similarity.

2) Apply a set-theoretic size ratio computation to express the difference between two paths as the ratio of the intersection of nodes in the two paths divided by the union of all the nodes in the two paths. Similar path spectra will produce a ratio closer to 1.0, while a ratio approaching 0.0 will result from dissimilar spectra.

4.2 Overhead

The overhead incurred for cloning based on path spectra comparisons includes a single code instrumentation, profiled program execution, partitioning of call sites based on path spectra, and the actual cloning. The partitioning step involves the path spectra comparisons. A single path spectra comparison takes O(length of path A * length of path B) to compare paths A and B. For a given function, there will be O(C^2) comparisons where C is the number of call sites for that function. This step is bounded by O(E^2*A^2) where E = the number of call sites in the program and A = the length of the longest acyclic path in a function.

In goal-directed cloning [6], the most expensive step is the forward data flow analysis step to create cloning vectors. It takes O((N + E)*V^L) time, where N = the number of functions in the program, L = the maximum number of elements in the cloning vectors, and V = the maximum number of values for each element in the cloning vector. The partitioning of cloning vectors can be done by hashing strings, with an expected time linear in the number of cloning vectors.

Once the partition of call sites is created, the actual cloning step takes the same time for both methods. We trade the cost of profiling and instrumented code execution for a simple path comparison analysis for partitioning the call sites rather than data flow analysis to compute cloning vectors for each separate optimization goal. While both profiled-guided and goal-directed cloning will include the static data flow analysis and optimization phases performed after cloning to exploit the cloning and the initial “Is it safe to clone?” pass for ensuring correctness in cloning, profile-guided cloning will not require multiple data flow analyses phases for determining profitability of cloning for different optimizations.

5 Initial Experimental Comparison Study

We compared our profile-directed cloning with goal-directed cloning in an experiment performed on the C version of the `linpackd` benchmark. `Linpackd` is a well-known benchmark collection of common matrix operations. Figure 2 shows the call graph for the computationally significant functions in `linpackd`.

In our figures, we annotate each call graph edge with the number of call sites represented by that edge, rather than separate edges, for presentation purposes only.

The original code was hand-instrumented to generate runtime path profile and path frequency data, as well as constant values of procedure arguments. This data was used to produce two versions of the original code, using our path spectra comparison method and the goal-directed method. Cloning was performed by hand, and the results of a number of test runs were collected and averaged (see Table 1). For comparison, the benchmark code was compiled using *gcc* with all optimizations turned off, and run under Solaris on a SPARCstation-20.

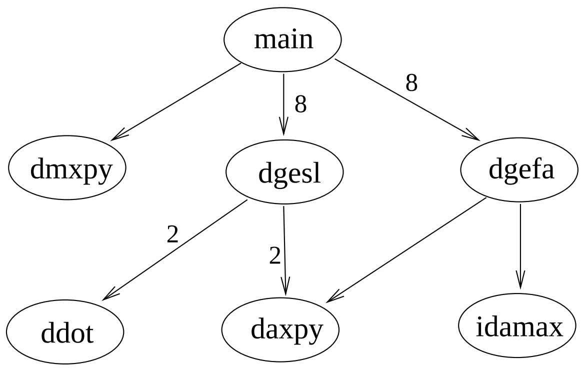

Figure 2: Call graph for subset of `linpackd`.

We did not implement the goal-directed method in a compiler, so comparison of compile times for the two methods is not possible. Since extra analysis is performed at compile time when the goal-directed method is used, we assume that compile time increases. This increase could be significant if cloning is performed for numerous goals, each requiring its own tuned analysis pass.

Path spectra comparison resulted in the partition of call sites of three of the original eleven functions (Table 2), resulting in the call graph shown in Figure 3. The function *daxpy*, which calculates a constant times a vector plus a vector and consumes nearly 50% of overall program execution time, had its three call sites partitioned into two clones based on path spectra similarities. The path profiles and execution frequencies for the first and third call sites were nearly identical, while the second site was significantly different.

Similarly, the call sites for *ddot*, a vector dot product function, and *dgesl*, which solves a basic system matrix equation, each were partitioned into two sets of similar path spectra.

Analysis based on the goal-directed technique for con-

Table 1: Comparison of cloning techniques applied to the `linpackd` benchmark.

	original	spectra	goal
source lines	420	530	629
procedures	11	14	16
compile time	2.6	3.0	n/a
binary size	29188	33036	37292

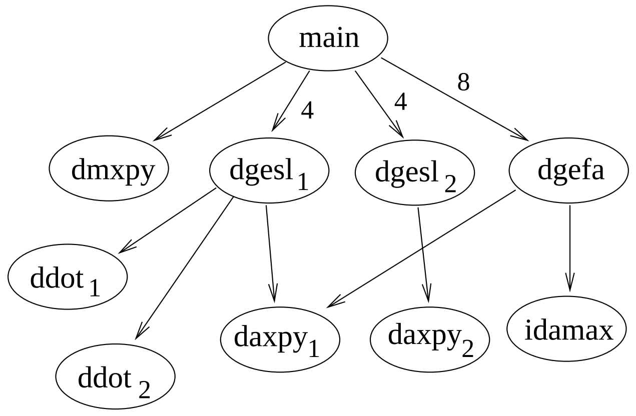

Figure 3: Call graph for `linpackd` after path spectra comparison directed function cloning.

stant propagation led to the identical partitioning of *daxpy* and *ddot*. In these two cases, interprocedural analysis for constant propagation identified "important constants" that determine control flow.

In *dgesl*, two important constants are identified, one appearing in a subscript expression and the other determining control flow. As a result, four unique *cloning vectors* are initially created. Typically with the goal-directed method, a subsequent merging step recognizes when propagated constants do not affect values of important expressions in a procedure, allowing cloning vectors to be merged. This merging does not occur in this case because the constant responsible for the increase in clones affects an expression that calculates an array index. As a result there are four clones created and a small amount of additional control code introduced at each call site.

The results of this experiment indicate that using comparison of program spectra can lead to intelligent cloning decisions that are comparable to goal-directed cloning. However, more experimentation and evaluation is needed to judge the effect of the different cloning decisions by these two techniques on actual optimization of the code. With profile-driven cloning, it is possible to avoid useless cloning efforts in less executed portions of the code, but path spectra differencing may miss information on optimization opportunities made possible through cloning that do not get reflected in differences in path spectra.

Table 2: Breakdown of function calls by number of call sites, unique path spectra, path spectra partitions, and clones produced.

procedure	*call sites*	*unique spectra*	*partitions*	*clones*
daxpy	3	2	2	1
ddot	2	2	2	1
degsl	8	2	2	1
matgen	9	1	1	0
dgefa	8	1	1	0
idamax	1	2	1	0
dscal	1	1	1	0
dmxpy	1	1	1	0
epslon	1	1	1	0
printtime	1	1	1	0
second	1	1	1	0

In this study, we were interested in comparing our technique to goal-directed cloning, and thus focused on `linpackd` as a current benchmark that was closest to the `matrix300` benchmark used in evaluating goal-directed cloning [11]. However, more experiments are needed on larger, non-numeric codes to examine how well different flavors of path spectra directed function cloning help in optimizing large codes.

6 Concluding Remarks and Future Work

This paper has described an approach to function cloning decisions based on comparing the dynamic behavior, namely path profiles, of different calls to the same function. In our initial examples, the function's dynamic control flow paths have indeed indicated clones that caused different interprocedural information to be propagated to potential optimization sites within the function. However, we need to investigate further the relationship between a function's dynamic control flow and its optimization opportunities for a larger set of benchmarks in order to fully evaluate this approach. For this study, we focused on cloning for better interprocedural constant propagation in order to compare our approach to the static approach of goal-directed cloning, which has only been evaluated for constant propagation, to our knowledge. We are particularly interested in how profile-guided cloning could help in overall optimization of the code, and its role in region-based, path-sensitive optimizations.

Various heuristics for partitioning call sites based on dynamic information and possibly a combination of dynamic and static information are being studied. In particular, we plan to examine the combination of value profiling and path profiling in cloning decisions. Another planned experiment is the tradeoffs of using edge profiles and path profiles, and intraprocedural profiles for the callees versus more expensive interprocedural profiles, in the profile differencing for cloning decisions.

References

[1] G. Ammons and J. R. Larus. Improving data-flow analysis with path profiles. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 72–84, June 1998.

[2] T. Ball and J. R. Larus. Efficient path profiling. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Paris, France, Nov. 1996.

[3] T. Ball, P. Mataga, and M. Sagiv. Edge profiling versus path profiling: The showdown. In *ACM SIGPLAN Symposium on the Principles of Programming Languages*, pages 134–148, 1998.

[4] B. Calder, P. Feller, and A. Eustace. Value profiling. In *Micro-30*, December 1997.

[5] C. Chambers and D. Ungar. Customization: Optimizing compiler technology for Self, a dynamically-typed object-oriented programming language. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 146–160, June 1989.

[6] K. D. Cooper, M. W. Hall, and K. Kennedy. A methodology for procedure cloning. *Computer Languages*, pages 105–117, 1993.

[7] J. Dean, C. Chambers, and D. Grove. Selective specialization for object-oriented languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 93–102, June 1995.

[8] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, 30(7):478–490, 1981.

[9] R. Gupta, D. A. Berson, and J. Z. Fang. Path profile guided partial dead code elimination using predication. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 102–115, San Francisco, California, Nov. 1997.

[10] R. Gupta, D. A. Berson, and J. Z. Fang. Path profile guided partial redudancy elimination using speculation. In *IEEE International Conference on Computer Languages*, pages 230–239, May 1998.

[11] M. W. Hall. *Managing Interprocedural Optimization*. Ph.d. thesis, Rice University, Apr. 1991.

[12] M. J. Harrold, G. Rothermel, R. Wu, and L. Yi. An empirical investigation of program spectra. In *ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 83–90, June 1998.

[13] R. L. Johnston. The dynamic incremental compiler of APL/3000. In *APL'79 Conference*, pages 82–87, 1979.

[14] T. Reps, T. Ball, M. Das, and J. Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In *ESEC/FSE 97*. Springer-Verlag, 1997.

[15] S. Wu and U. Manber. AGREP – a fast approximate pattern-matching tool. In *Proceedings of the Winter 1992 USENIX Conference*, pages 152–162, Berkeley, California, Jan. 1992.

[16] S. Wu and U. Manber. Fast text searching allowing errors. *Communications of the ACM*, 35(10):83–91, Oct. 1992.