Using Path-spectra-based Cloning in Region-based Optimization for Instruction-Level Parallelism*

Tom Way, Ben Breech, Wei Du, Veselin Stoyanov and Lori Pollock
Department of Computer and Information Sciences
University of Delaware, Newark, DE 19716
{way,breech,wei,stoyanov,pollock}@cis.udel.edu

Abstract

Profile-driven, region-based compilation presents an instruction-level parallelism (ILP) optimizer with compilation units that reflect the program's dynamic behavior. Interprocedural code motion, particularly instruction scheduling, is enabled without creating large optimization units or requiring interprocedural data flow analysis. Sophisticated optimization analysis can be applied on high execution frequency regions, while little to no optimization effort is expended on low execution frequency regions. However, the quality of the generated code depends greatly upon the quality of the region formation. In this paper, we describe our techniques to intensify both the role and the sophistication of profiling within a region-based ILP compiler to achieve the goal of improving the quality of formed regions for optimization. We have designed and implemented a path-spectra-based cloner within the Trimaran compiler to increase the accuracy of profiling information available for region formation. We present and discuss our experimental results from comparing the impacts of path-spectra-based cloning in region formation, in conjunction with aggressive and demand-driven inlining.

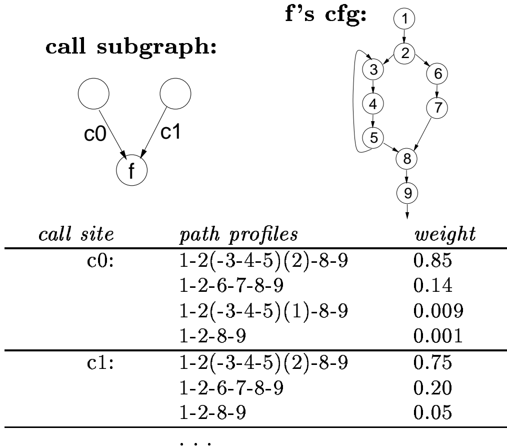
1 Introduction

Program analysis and optimization that span across function boundaries have become increasingly critical to exploit the higher degrees of available instruction-level parallelism in modern architectures particularly in the context of growing emphasis on application modularity and object-oriented program design. Region-based compilation [12] offers an approach that enables aggressive interprocedural code motion without the expense of performing optimization on large function bodies. By applying function inlining techniques and constructing new regions to serve as the unit of compiler optimization, region-based compilation obtains more freedom to move code between functions than interprocedural analysis techniques, while maintaining tight control over the compilation unit size and content. In addition to exposing opportunities for interprocedural instruction scheduling and optimization, the size of the resulting regions (i.e., compilation units) has been shown to be smaller than functions, and can be limited to any desired size, in order to reduce the impact of the quadratic algorithmic complexity of the applied optimization transformations [12].

The key component of a region-based compiler is the region formation phase which partitions the program into regions using heuristics with the intent that the optimizer will be invoked with a scope that is limited to a single region at a time. Thus, the quality of the generated code depends greatly upon the ability of the region formation phase to create regions that a global optimizer can effectively transform in isolation for improved instruction-level parallelism. Function inlining plays an important role in enabling optimizations that cross function boundaries by making it possible for the region formation phase to easily analyze across function boundaries and form interprocedural regions, which consist of instructions from more than one function of the original program.

The use of execution profile information in region formation allows the compiler to create regions that more accurately reflect the dynamic behavior of the program. With a partitioning based on profile information, different optimization effort can be devoted to different regions, applying expensive, sophisticated optimization analysis to high execution frequency regions, and little to no optimization on low execution frequency regions. Unlike profile-guided data flow analysis which seeks to compute more precise data flow analysis for hot paths [1], and profile-guided optimization which trades more aggressive optimiza-

*Supported in part by the National Science Foundation Grant EIA-9806525.



call site	*path profiles*	*weight*
c0:	1-2(-3-4-5)(2)-8-9	0.85
	1-2-6-7-8-9	0.14
	1-2(-3-4-5)(1)-8-9	0.009
	1-2-8-9	0.001
c1:	1-2(-3-4-5)(2)-8-9	0.75
	1-2-6-7-8-9	0.20
	1-2-8-9	0.05
	...	

Figure 1: Call subgraph, callee control flow graph (cfg) and example path spectra.

tion along heavily executed paths for potentially increasing the execution time along less frequently executed paths [10], no specialized data flow analysis is required to gain the same capability with profile-guided region formation. However, the goodness of the formed regions depends on the quality of the profiling information.

In this paper, we present a path-spectra-based function cloning algorithm and evaluate its impact on the quality of profile-based region formation. Function cloning partitions the calls to a given function, say f, based on a heuristic such that multiple copies of f's body are created, one copy for each set of closely related call sites where f is called. The primary motivation for the use of cloning in region-based compilation is the key insight that creating clones based on path spectra will result in more accurate profiling information being used to direct region formation within the clones than the profiling information in the original function f which was generalized to all of its callers. Thus, more refined regions could be formed, and global optimization opportunities within regions could increase without undue code growth of an inline-only approach to region formation.

The *path spectra* for function f is the set that contains a path spectrum for each call site where function f is called [15]. The *path spectrum* for a given call site c to f is the set of weighted execution path profiles through f from the call site c. A weighted path profile from call site c is an execution trace through the function f from c, weighted by the percentage of time this particular path was taken through f compared with other paths taken from the same call site c. In Figure 1, a portion of an example path spectra for function f is shown. Each path is represented by a sequence of basic block numbers, connected by hyphens. Loops are enclosed in parentheses and are followed by a loop count in a second pair of parentheses. Weights are expressed such that a weight of 1.00 for path p indicates that 100% of the runs through f from call site c executed path p.

Our approach to making cloning decisions for a function f is to compare the program spectra for the execution of f over all call sites to f, and create clone groups based on a similarity threshold, which can be experimentally determined to limit the number of created clones according to code growth concerns. This heuristic for cloning seeks to determine which calls contribute different dynamic interprocedural information and thus suggests good candidates for cloning and then separate region formation decisions.

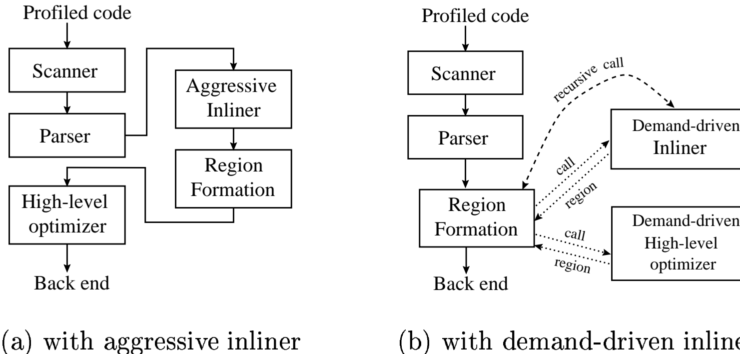
After presenting an overview of region-based compilation and region formation, we present our path-spectra-based cloning algorithm. This algorithm is a new algorithm that significantly improves upon a previous path-spectra comparison algorithm which we developed for cloning directed towards optimization in a function-oriented compiler [17]. In addition, we have implemented path spectra gathering, the path spectra comparison algorithm, and cloning within the Trimaran compiler [14], which embodies region-based compilation. We conclude with a report on our experimental findings.

2 Background and Motivation

2.1 Region-based compilation

Region-based compilation is an approach developed by Hank, Hwu, and Rau for ILP compilers [12], in which the compiler repartitions the program into a new, *more desirable*, set of compilation units prior to program analysis and optimization. Profile-based region formation heuristics determine which basic blocks to add to a region based on each block's execution frequency (e.g., to be included in the same region, a block must be executed at least 50% as frequently as its predecessor block). In current region-based compilers, the profiling information used to guide region formation is generalized to each function, rather than targeted to each call site to the function.

Hank et al.'s [12] region formation phase is run on a form of the whole program which has been aggressively inlined in order to remove function call barriers during region formation, thereby allowing global optimizations to gather some of the benefits of interprocedural optimizations. Since not all functions can be



(a) with aggressive inliner (b) with demand-driven inliner

Figure 2: Compiler phases of two region-based compiler frameworks.

inlined, any remaining function boundaries are treated as region boundaries. Figure 2(a) depicts the organization of a compiler based on this region-based framework. In a previous paper, we described how the potentially costly aggressive inlining step can be replaced by a recursive region formation algorithm that incorporates a demand-driven inliner [16]. In this demand-driven scheme, region formation is performed recursively within the callee prior to inlining. Optimization is integrated into region formation and is performed on regions rather than functions. Figure 2(b) shows the region-based compilation framework assuming demand-driven inlining. We also presented experimental results that showed that region formation with demand-driven inlining reduces compile-time memory requirements without degrading runtime performance.

Regardless of the approach to region formation, forming a region is followed by encapsulating the region to appear like a function; the encapsulated region is then passed to the backend of the compiler, which includes unaltered, function-based optimization phases. Hank, Hwu, and Rau [12] demonstrated solid experimental evidence of the positive impact of region-based compilation, with one of the more significant results being that the size of each of the resulting compilation units is typically smaller than functions, which reduces the effect of the quadratic algorithmic complexity of many optimizing analyses and transformations. Additionally, the homogeneous execution frequency of the blocks within a given region allows for prioritization of optimization efforts based on the importance of a region, measured by its execution frequency.

2.2 Path-spectra-based Cloning

Function cloning is a goal-directed compiler technique used to reveal particularly important details about the calling context of a function by call site [7]. When there are at least two different sets of call sites calling a given function, say `f`, where call sites in one set share the same context which is different from call sites in the other set, the function `f` is cloned. A copy of the function is made, and the call sites in the same set (with similar context information) are redirected from the original function to the copy. Cloning is a good complement to inlining, allowing code to be optimized for different call sites to the same function while producing less code growth, with a trade-off being fewer new opportunities for interprocedural analyses and optimization [2].

Within a region-based compiler, function cloning could be very beneficial by allowing inlining and region formation decision making to be fine-tuned. A function could be cloned, with some clones being inlined in more frequently executed sections of code, while less frequently executed clones could be analyzed in place. This could further improve region formation by reducing some of the code growth in an inline-only framework, allowing more inlining to be performed where it might not have been due to static region size and code growth limits. In addition, as stated earlier, the profiling information within clones would be more accurate to the context of the clone than the profiling information in the original function which was generalized to all call sites leading directly to it. This could lead to more refined regions being formed within the clones.

As part of our region-based compilation project, we have developed a *path-spectra-based* cloning technique. Path profiling has been used successfully in compiler optimization [1, 10, 18], can be collected efficiently and can provide more accurate information than the competing technique of edge profiling [3]. Path spectra have been used successfully as an indication of program behavior for program optimization [1, 9, 10] as well as software maintenance and testing [15].

3 Cloning Algorithm

Our path-spectra-based cloning is performed as an early phase before region formation, and in the case of Hank's method, also before aggressive inlining. Figure 3 presents our algorithm for cloning guided by path-spectra comparisons. There are four phases in the algorithm, as indicated in `main`: (1) gathering the path profiles for each function, (2) creating a path spectrum for each call site by manipulating and organizing the path profiles of the called function, (3) calculating clone groups for each function, and (4) performing the actual cloning operation on the code.

1. Gather Path Profiles. In order to gather path profiles for a program P, function call sites and basic blocks are instrumented to output an execution trace, consisting of the callee name and unique call site number at each call site, and basic block ID at each basic block. We are currently gathering intraprocedural path profiles. We call this the raw path profile data.

2. Create Path Spectra. The raw path profile data is processed to collapse loops and create path spectra (Figure 1). Loops are collapsed in order to reduce the computational time for performing path spectra comparisons. First, each loop profile is reduced to a single iteration, with loops below a *loop count threshold* differentiated from those with a count equal to or above the threshold. We use a threshold of 1, so loops that are executed just once (in effect a section of straight-line code) are considered behaviorally distinct from those that execute 2 or more times. We differentiate the two kinds of loops in the reduced profile by including an iteration count of either (1) or (2) after the loop profile information.

Next, a path spectrum is created for each call site `c`, by identifying paths executed in the called function when the function is invoked from the call site `c`. Each path in a path spectrum is assigned a weight which is computed as the ratio of execution frequency for that path to the sum of execution frequencies for all paths in the path spectrum. The call sites to each function are ordered by decreasing dynamic call count from each call site. The path profiles within each spectrum are ordered by decreasing execution frequency count. These orderings are done to make path comparisions more efficient.

3. Calculate Clones. Clone groups for function `f` are calculated using an ordered pair-wise comparison of the path spectra for each of the call sites where function `f` is being called. The comparison consists of comparing the paths and their associated weights. First, the spectrum c for the most frequent call site of all call sites S calling `f` is selected. This spectrum is compared for *similarity* with all other spectra ($\forall s \in S, s \neq c$) for the function `f`.

Two spectra are considered to be *similar* if all matching paths in the two spectra have weights (i.e., $w1$ and $w2$) that are within a fixed execution *weight_threshold* of each other (i.e., $|w1 - w2| <$ *weight_threshold*). Varying the *weight_threshold* can dramatically affect the number of clones created, with a threshold of 0% leading to a clone for each call site and 100% leading to no cloning. We selected a value of 50% to limit code growth while allowing a significant amount of cloning. If a path is present in one spectrum but not the other spectrum being compared to it, then the missing path is assumed to have a weight of 0% for comparison purposes.

Two paths are considered to *match* if they are identical for the first k blocks in the path. We experimentally determined that the number of clones created using our algorithm stabilizes when the length of path prefixes being compared exceeds about 25-30 blocks. We found many times that initial paths were identical up to this length, diverging beginning after 15-35 blocks or so. There was variation among benchmarks, and among functions within each benchmark, but generally speaking, paths that compared uniquely did so by 30 blocks if they were going to at all. Thus, we use a conservative comparison *k_limit* of 50 to guarantee to be as accurate as if we compared entire paths, which often contained thousands of blocks, while significantly reducing the amount of actual comparison performed.

Additionally, a call site is not considered for cloning if it has an overall basic block execution count lower than a fixed *basic block limit.* This eliminates creation of clones for infrequently executed function calls. We selected a *basic block limit* of 25 to eliminate the class of small functions that might be better inlined because of their small size and minor impact on code growth as a result.

4. Perform Function Cloning. For each function, a clone copy is created for each clone group C in that function's calculated list of clone groups $Clist$. The clone is created by duplicating the function code and renaming the new function. Each call site c in C is redirected to the newly created clone by replacing the target name of the call with the clone name. The original copy of the function is used for the first clone group.

Conceptually, our algorithm for comparing two path spectra is mathematically reminiscent of a curve-

```
Main()
    GatherPathProfiles( P )                          // Instrument and run
    CreatePathSpectra( P )                           // Pre-process the profiles
    foreach function f in program
        Clist = CalculateClones( f )                 // Group similar path-profiles
        Clone( Clist )                               // Duplicate callees, rename call sites

CalculateClones( f )
    SortAndFilter( call sites )                      // Order by thresh., filter by bb count
    group = 1                                        // Initialize clone group counter
    S = all call sites to f                          // Initialize S to all call sites in f
    Clist = null                                     // List of clone groups starts empty
    while ( call sites remain in S )
        c = most frequent call site in S             // Find call site with max frequency
        C = {c}                                      // Seed clone group with call site
        freq = Frequency(c)                          // Initialize group frequency
        S = S − {c}                                  // Delete call site from list
        foreach call site s in S
            if Similar(s,c) && Similar(c,s)          // Are paths similar?
                C = C ∪ {s}                          // Yes, so add to clone group
                freq = freq + Frequency(s)           // and add to group frequency
                S = S − {s}                          // and delete from list
        C→group = group                              // Assign the clone group number
        group = group + 1                            // and increment the counter
        C→weight = freq / Frequency(f)               // and assign proportional weight
        Clist = Clist ∪ C                            // Then add to list of clone groups
    return Clist

Similar( s1, s2 )                                    // Compares paths in two spectra
    foreach profile p1 in spectrum for s1
        w1 = p1→weight                               // Get weight of path to compare
        if ( p1 ∈ s2 )                               // If s2 contains patching path
            p2 = matching path-profile to p1 in s2   // then get the path
            w2 = p2→weight                           // and get its weight
        else                                         // otherwise, no matching path
            w2 = 0                                   // implies a zero weight
        diff = abs( w1 −w2 )                         // calculate the difference
        if ( diff > weight_threshold )               // if not within a threshold
            return false                             // paths (and spectra) are not similar
    return true                                      // All paths were similar
```

Figure 3: Path-spectra-based cloning algorithm.

fitting algorithm. If the path profiles in a spectrum are along the x-axis and the execution weights for each path profile are plotted on the y-axis, a curve is created for each spectrum. The fit of the curves for two spectra determines the similarity of the two spectra being compared.

4 Experimental Study

Implementation of our techniques and our experiments have been conducted in the context of the Trimaran compiler system [14]. The Trimaran System is an integrated compilation and performance monitoring infrastructure developed through the combined efforts of the Compiler and Architecture Research Group at Hewlett Packard, the IMPACT Group at the University of Illinois and the ReaCT-ILP Laboratory at New York University. The inclusion of a version of Hank et al.'s region formation was a prime consideration in our infrastructure choice.

We modified Trimaran in a number of ways for this research. The code instrumentation module for edge profiling was extended to gather path profiles, a cloning module was added, and the existing region formation phase was enhanced to encapsulate regions into functions so that region-based optimization could be performed. In addition, we had previously integrated our demand-driven region formation algorithm, along with an inliner that works at a lower level of the code than previously implemented.

To evaluate and compare the impact of our cloning technique on region-based optimization, we performed a set of experiments with four different compilation strategies, on a suite of seven C benchmarks, four of which are from SPEC (124.m88ksim, 130.li, 023.eqn-

Benchmark	Lines of source code	Num. of func.	Trimaran Lcode instructions: Total prog. size	Max. func. size	Avg. func. size	Num. clones created	Num. call sites cloned
124.m88ksim	19092	252	55783	1537	192.7	27	36
130.li	7597	357	31552	526	80.9	32	74
023.eqntott	3628	62	11738	1507	168.7	16	19
026.compress	1503	16	2601	884	151.8	3	3
paraffins	388	10	1115	324	104.3	0	0
clinpack	313	9	1294	585	182.9	2	5
bmm	106	7	538	97	72.3	0	0

Table 1: Code size, function and cloning characteristics of benchmarks.

tott, and 026.compress), one from Netlib (clinpack) and two that were included in the Trimaran distribution (paraffins and bmm). These benchmarks were selected from well-known sources to include a variety of code sizes, program characteristics, and computational behaviors.

Global optimization phases were executed on the individual regions created by the different region formation strategies. Execution statistics were gathered by running the resulting code through the Trimaran backend on a simulated 4-wide EPIC (**E**xplicitly **P**arallel **I**nstruction **C**omputer) machine with 128k I-cache (12 cycle miss latency), 128k D-cache, 1Mb L2-cache (4 cycle latency), and a 30 cycle memory latency.

Table 1 describes the benchmarks in terms most relevant to our study. The number of source code lines and function definitions for each benchmark are listed along with the number of clones created by our cloning algorithm based on the number of *Lcode instructions*. *Lcode* is a low-level intermediate code used in Trimaran roughly equivalent to *three-address code*. The *number of clones* column indicates the total number of function copies created during cloning. The *call sites cloned* column indicates the number of call sites that have been renamed to call a cloned function.

Results

We can see from the static measurements of cloning characteristics in the last two columns of Table 1 that clones do get created for these programs. The number of clones does not depend on either the program size or the number of functions in the program, but instead on the characteristics of the program at call sites. However, with larger programs, there is apt to be more functions and functions calls, leaving more room for cloning opportunities. In most of these programs, different call sites are causing clones to be created, indicated by the higher number of call sites cloned than clones created. We also gathered static measurements of region characteristics (not shown in the chart). Although the number of regions formed for each technique with and without cloning remained effectively the same on a per function basis, the average region size decreased slightly. This is indicative of the fact that cloning leads to more precise profiling. Fewer low frequency functions (clones) are inlined during region formation, producing slightly smaller and more behaviorly homogenous regions.

Table 2 presents the runtime results which compare the effect on the benchmark suite of the four compilation strategies, Hank's framework and the demand-driven inlining and region formation framework with and without cloning. We measured the code growth, ops/cycle, and speedup as calculated from the simulated execution times measured in cycles. One potential concern of cloning would be the code growth that would result, similar to code growth concerns due to function inlining. The *Code Growth* columns show the percentage increase in intermediate *Lcode* size, calculated as $(codesize(strategy) - codesize(original))/codesize(original)$ where $codesize$ is the number of *Lcode* instructions, $original$ is the original, unoptimized code, and $strategy$ is the code after one of the four compilation strategies has been applied. The *Ops/Cycle* columns show the percentage increase in the number of operations executed per cycle during the simulation, calculated in the same way as *Code Growth*. The *Speedup* columns show the percentage of improvement in simulated execution time over the original, unoptimized program, calculated as $(cycles(original) - cycles(strategy))/cycles(original)$.

In almost all cases, region formation with demand-driven inlining benefits in runtime performance when a path-spectra-based cloning phase in included, indicated by the higher speedups for CD over DD. Furthermore, this combination consistently outperforms region formation with aggressive inlining regardless of cloning. We believe that low gains in performance

Benchmark	Simulated Execution											
	Code Growth %				Ops/Cycle %				Speedup %			
	HA	CH	DD	CD	HA	CH	DD	CD	HA	CH	DD	CD
124.m88ksim	22	34	18	27	1	1	7	8	7	8	15	21
130.li	15	25	15	18	8	8	1	3	4	4	7	15
023.eqntott	30	7	14	5	2	2	1	1	2	3	7	8
026.compress	28	20	21	18	3	3	1	8	5	7	11	18
paraffins	45	45	38	38	9	9	15	15	11	11	16	16
clinpack	28	52	32	40	2	2	12	13	7	8	14	16
bmm	42	42	31	31	18	18	4	4	1	1	4	4

Table 2: Simulated execution statistics.

HA - Hank's method with aggr. inlining, no cloning
CH - Cloning, followed by HA
DD - Demand-Driven inlining, no cloning
CD - Cloning, followed by DD

are due to the interactions between the inlining and cloning. When cloning was performed, the better performance results are those cases with slightly higher code growth. This can be explained by the tradeoff between code growth and increased optimization opportunities. The code growth between region formation with demand-driven inlining and cloning (CD) and region formation with aggressive inlining and cloning (CH) is consistently lower for CD. This is most likely due to inlining only a subset of a set of clones. Cloning produced additional code growth in some cases due to the cloning of larger functions which were more likely to exhibit distinct behavior at different call sites, and in other cases due to the inlining of some of the cloned functions. In summary, the current experiments indicate that the CD strategy (i.e. path-spectra-based cloning, then region-formation with demand-driven inlining) provides the best runtime performance.

5 Related Work

Goal-directed cloning [7, 11] first solves a forward interprocedural data flow problem with slight modification to compute a set of cloning vectors for a particular data flow problem 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. [7] presented an experiment on the `matrix300` code from the SPEC89 benchmark suite, in which they showed that significant improvement in code quality could be obtained 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.

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 [8]. This goal-directed technique helps to reduce the number of expensive dynamic dispatches, providing significant improvements in performance and reduced code growth over *customization* [6], 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 spectra have been used successfully for program optimization [1, 9, 10] as well as software maintenance and testing [15]. Path profiling has been used successfully in compiler optimization [1, 10, 18]. Other well-known types of profiling are edge, basic block, and value profiling. Edge profiling measures the execution frequency of each individual flow graph edge; basic block profiling measures how many times each basic block is executed. Edge profiles are good predictors of frequently executed (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]. When variables cannot be conservatively identified as constants at compile time, value profiling [5] can be used to determine whether they exhibit a high degree of invariant behavior at run-time.

In this paper, we use path profiling to discover differences and 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.

Bodik, Gupta, and Soffa [4] developed an interprocedural correlation analysis which they show can direct inlining by post-analysis to enable an unmodified intraprocedural branch elimination optimization to be applied. Procedures that generate a higher correlation can be given a higher priority in the inlining phase. This shares the same motivation as inlining driven by the demand during region formation [16]. While there are similarities in the issues addressed, we believe that their work is orthogonal to region-based compilation, and the techniques could in fact be integrated to increase the quality of regions and reap the benefits of region-based compilation.

6 Summary and Future Work

Region-based compilation has already been shown to help increase ILP performance by enabling interprocedural code motion without the expense of large compilation units or interprocedural data flow analysis. We have designed and implemented a path-spectra-based cloning algorithm and incorporated it into a region-based compiler with the goal of improving the quality of regions to be used for optimization. This has involved both increasing the role of profiling in the compiler as well as utilizing both edge and path profiling. In this paper we describe our algorithm, our overall compilation strategy, and both static and runtime experimental results within two different region-based compilation frameworks. In the context of demand-driven region-based compilation, the path-spectra-based cloning improves performance in all cases, with no degradation due to cloning in any case. In the context of a separate aggressive inlining step performed after cloning, there is less benefit to runtime performance. The combination of cloning and demand-driven region formation consistently gives better runtime performance than the other strategies. We are currently designing experiments to investigate the tradeoffs for a sizable set of heuristics for region formation in the context of both cloning and inlining.

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] A. Ayers, R. Schooler, and R. Gottlieb. Aggressive inlining. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 134–145, 1997.

[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] R. Bodik, R. Gupta, and M. L. Soffa. Interprocedural conditional branch elimination. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 146–158, June 1997.

[5] B. Calder, P. Feller, and A. Eustace. Value profiling. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 259–269, Dec. 1997.

[6] 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.

[7] K. D. Cooper, M. W. Hall, and K. Kennedy. A methodology for procedure cloning. *Computer Languages*, 19(2):105–117, Feb. 1993.

[8] 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.

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

[10] R. Gupta, D. A. Berson, and J. Z. Fang. Path profile-guided partial redundancy 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] R. E. Hank, W. W. Hwu, and B. R. Rau. Region-based compilation: Introduction, motivation, and initial experience. *International Journal of Parallel Programming*, 25(2):113–146, Apr. 1997.

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

[14] A. Nene, S. Talla, B. Goldberg, and R. M. Rabbah. Trimaran - an infrastructure for compiler research in instruction-level parallelism - user manual, 1998. `http://www.trimaran.org`. New York University.

[15] 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 M. Jazayeri and H. Schauer, editors, *Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97*, pages 432–449. Springer–Verlag, 1997.

[16] T. Way, B. Breech, and L. Pollock. Region formation analysis with demand-driven inlining for region-based optimization. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 24–33, Philadelphia, Pennsylvania, Oct. 2000.

[17] T. Way and L. Pollock. Using path spectra to direct function cloning. In *Workshop on Profile and Feedback-Directed Compilation*, pages 40–47, Oct. 1998.

[18] C. Young and M. D. Smith. Static correlated branch prediction. *ACM Transactions on Programming Languages and Systems*, 21(5):1028–1075, 1999.