

Procedure Mapping Using Estimated Program Behavior

Amir Hooshang Hashemi John Kalamatianos
David Kaeli Alireza Khalafi Waleed Meleis

Brad Calder

Dept. of Electrical and Computer Engineering
Northeastern University
Boston, MA

Dept. of Computer Science and Engineering
University of California, San Diego
La Jolla, CA

Abstract

*Efficient exploitation of the available memory can have a significant impact on application performance. One technique used by compiler and linkers to improve code mapping on the cache space is called **cache line coloring**. Coloring reorders program units (e.g., procedures, basic blocks) so that temporally local sections of a program are mapped to nonconflicting regions in the cache.*

*A number of coloring schemes have been proposed which use profile data in order to reposition the code in the address space. In this paper we present a link-time procedure mapping algorithm which uses a call graph constructed without the use of profile data. We refer to this scheme as **static call graph estimation**. Our technique uses program-based heuristics to statically estimate (using a form of static branch prediction) the behavior of the call graph. Then once the estimated weighted call graph is formed, we can employ various cache coloring algorithms. Our results show that we were able to reduce instruction cache miss rates by 20% on average when using our estimated static call graph with modern procedure reordering algorithms.*

1 Introduction

Cache can be found on most microprocessors developed today. A cache exploits the temporal and spatial locality exhibited by a program. Instruction caches have been found to be very useful since the instruction stream exhibits a high degree of locality. Even so, cache misses will occur for one of the following reasons:

1. finite cache capacity,
2. program cold start, or
3. address conflicts in the cache.

Low associativity (e.g., direct mapped) caches are especially susceptible to address conflicts. Our work here is focused on remapping a program to reduce this contention, resulting in higher cache hit rates and better overall performance. We perform the remapping without the aid of program execution profiles, allowing our optimizations to be applied to *shrink-wrapped* code.

Profile-based optimizations have been used extensively to tune the performance of programs [4, 12]. Profile-based methods use a set of sample inputs to profile an application. These profiles are then fed back into an optimizer and are used to train the application to the data. The main drawbacks associated with a profile-based approach include:

1. the extra time and effort required to first profile the program, and then recompile using the profile,
2. difficulty finding a representative input suitable for the purposes of profiling, and
3. the inability to modify shared code (e.g., Dynamic Link Libraries).

If a developer can not afford the time and effort to perform profile-based optimizations, the compiler must rely on static techniques to estimate program behavior. To this extent, researchers have examined using program-based heuristics and machine learning techniques to statically estimate a program’s behavior at compile-time.

Program-based estimation methods attempt to predict branches and estimate the control flow of a program based on a program’s structure. Some of these techniques use heuristics based on local knowledge that can be encoded in the branch architecture [13, 18]. Other techniques rely on applying heuristics based on more detailed control flow analysis [1, 19, 21, 22]. Still others have examined using machine learning techniques to statically predict the control flow at compile-time [2]. In [7], Hank et al. showed that these program-based heuristics can be used to accurately guide profile-based optimizations, employing techniques such as superblock formation, achieving performance improvements close to those realized by using profile data. While many of these techniques have been shown to be effective in predicting program flow, none of these previous studies have attempted to apply their heuristics to procedure reordering.

Prior work in procedure reordering used profiles to guide the program layout in order to reduce instruction cache conflicts [6, 8, 9, 10, 11, 14]. Instead, we examine how to perform procedure reordering using heuristics to estimate edge weights in a program call graph. By inspecting the high-level language branch constructs (e.g., loops, switches, conditional branches), we statically predict how often each edge in the call graph is traversed. These estimated weights are then used to guide existing procedure reordering algorithms.

In this paper we will describe our heuristics for statically estimating edge weights in a call graph, and provide cache simulation results showing the improvement in the miss rate after applying procedure reordering optimizations. In Section 2 we will discuss prior work in program-based control flow estimation. In Section 3 we describe the heuristics we use to statically estimate a call graph’s behavior. In Section 4 we discuss how we capture control flow semantics of a program. In Section 5 we provide cache simulation results showing the effects of using our estimated call graph edges to guide procedure reordering optimizations. We then conclude, discussing directions for future work.

2 Background

We will begin by reviewing existing approaches to program-based static branch prediction and control flow estimation. Statically estimating the control flow graph of a program starts with static

branch prediction. Correctly predicting the probability of “taking a branch” will allow us to have a better chance at accurately estimating which paths in the control flow will be the most important.

One of the simplest program-based methods for static branch prediction is called “backward-taken/forward-not-taken” (BTFNT). This technique relies on the heuristic that backward branches are usually loop branches, and as such, are likely to be taken. One of the main advantages of this technique is that it relies solely on the sign bit of the branch displacement, which is already encoded in the instruction. While simple, BTFNT is also quite successful, since many programs spend a lot of time executing inside of loops and the backwards branch in a loop are correctly predicted as taken when using the BTFNT heuristic.

In [1], Ball and Larus showed that applying a number of simple program-based heuristics can significantly improve the static branch prediction miss rate over BTFNT on tests based on the conditional branch operation. Their heuristics use information about branch opcodes, operands, branch successor blocks, and looping constructs, as they try to encode knowledge about common programming idioms. Two questions arise when employing this type of a approach: 1) which heuristics should be used in general, and 2) how to prioritize heuristics when more than one applies to a given branch. Ball and Larus describe seven heuristics that they considered successful, but also noted that they first tried many heuristics that were unsuccessful [1].

The prioritization problem has existed in the artificial intelligence community for many years and is commonly known as the “evidence combination” problem. Ball and Larus considered this problem in their paper and decided that the heuristics should be applied in a fixed order; thus the first heuristic that applied to a particular branch was used to determine which direction it would take. They determined the “best” fixed order by conducting an experiment in which all possible orders were considered.

In a related paper, Wu and Larus extended the heuristic-based methods of Ball and Larus [22] to statically estimate the edge weights of the program’s control flow graph. In that paper, their goal was to determine *branch probabilities* instead of simple branch prediction in order to provide program-based profile estimation. Wu and Larus abandoned the simplistic evidence combination function of using a best fixed order in favor of an evidence combination function borrowed from Dempster-Shafer theory [5, 17]. By making some fairly strong assumptions concerning the independence of different attributes, the Dempster-Shafer evidence combination function can produce an estimate of the branch probability from any number of sources of evidence. The sources of evidence used by Wu and Larus were the heuristic’s branch prediction success from the paper of Ball and Larus [1]. Their algorithm propagated these branch probabilities throughout each procedure’s basic block graph. After the intra-procedural estimated edge weights were calculated, the algorithm then propagated the call frequencies along the call graph edges to compute the inter-procedural estimated call edge weights. In more recent work by Sokolova and Kaeli [19], these heuristics were further improved upon by expanding the scope of the analysis to multiple basic blocks, and by restricting the analysis to source code level semantics.

Wagner et al. [21] also used heuristics similar to those of Ball and Larus to perform program-based profile estimation. They also applied the heuristics in a fixed order. They used the heuristic probabilities as did Wu and Larus, but instead used Markov Modeling to propagate the probabilities through the control flow graph [15]. This creates basic block graphs and call graphs with estimated edge weights.

Both the Wu and Larus and the Wagner et al. study examined statically estimating the program’s behavior. What they did not provide are results on how effective these estimations would

be when applied to weighting call graphs used for code reordering. To our knowledge, no study has examined the performance of using these statically estimated call graphs to guide profile-based optimizations like procedure reordering. In this paper we use a very small subset of these previously proposed heuristics to create estimated call graph edge weights. We then use these estimated edge weights to guide procedure reordering optimizations. The main contribution of this paper is our examination into how effective static call graph estimation can be used for procedure placement, which typically uses profile data.

3 Static Call Graph Estimation

Next we describe the heuristics used during static call graph estimation, and describe how we propagate these estimated edge weights throughout the call graph. The goal of static call graph estimation is to assign estimated edge weights based on procedure call site location. Locations of interest are control transfer constructs present in high-level language codes. We are especially interested in loops (e.g., do while, for loops), switch statements (e.g., switch/case blocks), and conditional branches (e.g., if/then/else).

Our algorithm begins by constructing a call graph, where the nodes of the graph represent the procedures in the program, and the edges connecting the nodes represent call paths. Multiple call sites found in one procedure to a second procedure produce a single edge between the two procedures. It is important to note that this graph is directed, from caller to callee. We need to indicate direction in order to propagate weights in the proper direction.

After a call graph has been constructed, we label the edges with weights. The relative ordering of edge weights dictates the order in which edges are considered for layout during cache line coloring algorithm. Our goal is then to provide a weighting scheme which produces a similar ordering as would results from using dynamic profiles. The magnitude of the weight is only relevant in the context of determining a final ordering of edges.

One issue that needs to be addressed when building and labeling our graph is how to handle the occurrence of cycles (i.e., recursions) in the graph. This can be handled either during the construction of the call graph, or during the propagation of the edge weights. We decided to handle recursive calls by detecting them during construction, and assign a weight at graph creation time. As each edge is added, we see if there are alternative paths between the caller and callee. If another path exists directed from the callee to the caller in the already graphed procedures, we will identify the current edge as a recursion. In this case, we will assign the edge a fixed *recursion weight* (w_r), breaking the recursion. We are left with a Directed Acyclic Graph (DAG).

After the DAG has been formed, we start from the starting procedure (i.e., main), and assign weights to the edges in our graph. We traverse the DAG in a depth-first order, assigning edge weights in the following manner. We begin by assigning an *initial weight* (w_i) to the starting procedure and propagate this weight to procedures called using four heuristics: *postdom-entry*, *loop*, *cond-branch*, and *switch*.

We describe each of these four heuristics next. In the description, we will refer to node X as the parent procedure, and node Y as the child procedure. We use the following rules to determine the weight of edge $X \rightarrow Y$. All heuristics are used where they apply (i.e., multiple heuristics may be applied for a single procedure call site).

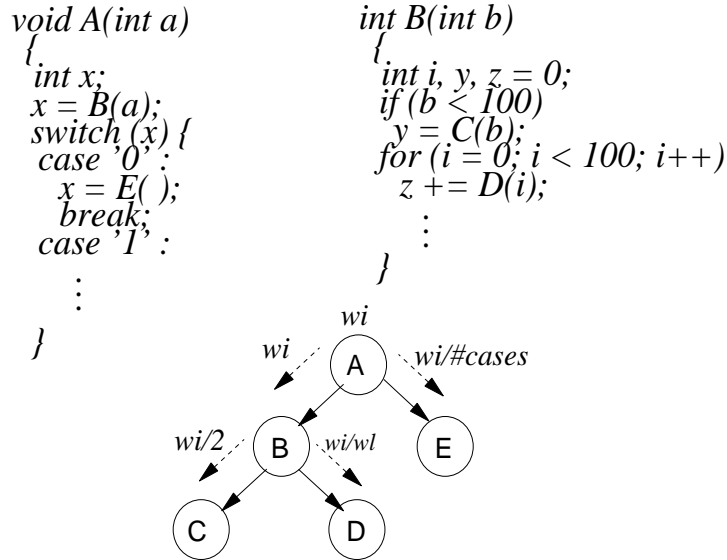


Figure 1: A sample call graph for procedures *A-E*. Code snippets for procedures *A* and *B* are also shown. Procedure *A* is the entry node, and is assigned an initial weight w_i . This weight is then propagated throughout the rest of the call graph based on the heuristics described above. Each edge assumes the propagated weight. Each node assumes the weight of all edges entering the node.

1. **Postdom-entry**: if the basic block in procedure *X* containing the call site to procedure *Y* post-dominates all the entry points into procedure *X*, then we know procedure *Y* will be called whenever procedure *X* is executed. Assign the node weight of procedure *X* (i.e., the sum of all incoming edges) to an edge $X \rightarrow Y$, if the call to procedure *Y* will always be executed whenever procedure *X* is executed.
2. **Loop**: give a larger weight to procedure call edges that are contained within loops. Assign the node weight of procedure *X* multiplied by a constant *loop weight* (w_l) to a call edge $X \rightarrow Y$, if the call to procedure *Y* is contained within a loop in procedure *X*.
3. **Cond-branch**: predict that for a conditional branch point that both paths have equal probability of being taken. Assign the node weight of procedure *X* divided in half to a call edge $X \rightarrow Y$, if the call to procedure *Y* is contained within a conditional path in procedure *X*.
4. **Switch**: predict that each switch case has an equal probability of being executed. Assign the node weight of procedure *X*, divided by the number of cases appearing in the switch block, to a call edge $X \rightarrow Y$, if the call to procedure *Y* is contained within a subcase of a switch block in procedure *X*.

Figure 1 shows an example of how five procedures *A-E* are assigned weights, based on their individual locations in C source code. Dashed lines indicate weights that are propagated.

Table 1 shows the values we use for the *initial weight* (w_i), the *loop weight* (w_l), and the *recursion weight* (w_r) for the results in this paper. We arrived at these values by using a *corpus* of training programs [2], and then applied these values to our programs. Note that the value of w_i is selected

Variable	Value
w_i	10
w_l	10
w_r	200

Table 1: Weight propagation factors

so we avoid edge weight values less than one (in general). A value of 200 for recursive edges is selected based on the average weight that is present at a recursive call site. A value of 10 in our loop heuristic is selected based on the average number of iterations of a typical loop. We do not attempt to use the constant loop count values available to us for each loop. While this would be more accurate, we have found that many loops use variable loop count expressions. Our *Cond-Branch* heuristic could be improved by further utilizing more sophisticated static branch prediction techniques [2, 19]. Also, our *Switch* heuristic is probably much too pessimistic since the chances that all cases within a switch are equally probable is low. In future work we will study what effects varying these values have on the procedure mapping algorithms.

The magnitude of the edge weights in the statically-weighted call graph do not have much significance. Their relative weights are much more important, since the procedure reordering algorithms consider edges in decreasing weight order and does not care about the magnitude of an edge (coloring uses a greedy algorithm, using edge weight to guide the layout). If the estimated edge weight order mirrors the order generated using profiled-guided edge weights, similar cache layout (and performance) should result.

4 Experimental Methods

To work with an intermediate representation of a program, we used `gcc`. We modified version 2.7.2 to provide the information necessary for producing our statically generated edge weights. The programs included in this paper are taken from the SPECInt95 suite, SPECInt92 suite, and GNU applications.

We used trace-driven simulation to quantify the instruction cache performance of our algorithm. The trace driven simulation results were obtained using the ATOM toolset [20]. In our simulations we model a direct-mapped 8 kilobyte instruction cache with a 32 byte line size, similar to the organization used for the DEC Alpha 21064 and DEC Alpha 21164 first-level instruction caches. Therefore, in our color mapping, the number of colors is equal to 256, which is equal to the number of direct-mapped cache lines.

Table 2 describes attributes of the programs we studied. The first column contains the program name, and the second column shows the input used to profile each program. The third column shows the number of instructions traced for the input used. The fourth column shows the size of each program in kilobytes, and the fifth column shows the number of static procedures in the program.

Program	Input	# Instrs Traced in Millions	Exe Size K-Bytes	# Static Procs	Popular Procedures		Unpopular Filler % Exe Size
					% Exe Size	% Procs	
xlisp	9-queens	6938 M	417 K	575	6% (24 K)	15%	0.5% (2 K)
m88ksim	dcrand.lit	27942 M	557 K	460	7% (41 K)	9%	0.9% (5 K)
perl	scrabbl	29612 M	819 K	557	4% (20 K)	4%	0.6% (5 K)
espresso	tial	1145 M	516 K	539	12% (60 K)	17%	0.5% (3 K)
eqntott	int_pri_3	2021 M	400 K	498	11% (46 K)	7%	0.8% (3 K)
bison	objc_pars.y	77 M	352 K	369	9% (32 K)	10%	1.0% (4 K)
flex	fixit.l	24 M	492 K	668	11% (52 K)	8%	0.8% (4 K)
gzip	gcc-2.7.2.tar	9242 M	344 K	140	3% (10 K)	21%	0% (0 K)

Table 2: Measured attributes of traced programs. The input file specified was used to generate the profile used for both the profile-guided and statically generated cache simulation studies. The attributes include the number of instructions traced when simulating the program, the executable size of the program, and the number of static procedures in the program.

5 Results

To evaluate the how well estimation can guide code reordering, we compare cache simulation results with executables generated using: 1) the link order specified by the makefile accompanying the benchmarks (*Original*), 2) an estimated call graph (*Estimated*), and 3) a dynamic profile as described in [9] (*Profile*).

The results in Table 3 indicate that estimated call graph can effectively be used to determine edge weights. While the average miss rate for programs studied was around 2.3%, using procedure ordering based on estimated call graph edge weights reduces the average miss rate to 1.9%, a 20% reduction. Compared to profile-driven repositioning, the miss rate was reduced from 2.3% down to 1.4% on average. This shows that our estimated call graph achieved about half of the improvement obtained using profile-driven layout. This is an encouraging result, indicating that estimated code repositioning can improve a program’s performance.

The reason for the reduction in accuracy using the estimated edge weights is:

1. heavier weights tend to unnaturally propagate to the leaf nodes, and
2. error routines tend to be called from many places in the program.

For most programs, edge weights of the synthetically generated program call graph increase by increasing graph depth. Therefore, popular procedures of a synthetic call graph are usually closer to the leaves of the graph. However, a study of dynamic behavior of programs showed that some programs have very popular edges in the upper regions of the graph [8]. Also, in some programs, error handling routines are called extensively. Therefore, error handling routines are commonly reported as popular procedures in synthetically generated call graphs.

To further improve on these results, we plan to investigate the following issues:

1. Incorporate static branch prediction techniques to improve upon the *cond-branch* heuristic currently used for conditional branches [19].
2. Identify commonly called, but infrequently visited, procedures (e.g., `exit()`, `error()`).

Program	Input	I-Cache Miss Rate		
		Original	Estimated	Real
xlisp	8-queens	1.2%	1.2%	0.3%
	9-queens	1.4%	1.4%	0.3%
m88ksim	dhry.lit	4.3%	3.8%	2.3%
	dcrand.lit	3.0%	2.9%	1.4%
perl	scrabbl	5.2%	5.1%	4.6%
	primes	4.6%	4.0%	3.2%
espresso	tial	0.9%	0.8%	0.5%
	Z5xp1	1.3%	1.1%	0.9%
	bca	0.2%	0.1%	0.1%
	cps	0.4%	0.4%	0.3%
	dc1	2.6%	2.4%	2.1%
	mlp4	1.1%	0.8%	0.8%
	opa	0.6%	0.6%	0.4%
ti	0.5%	0.4%	0.3%	
bison	objc_pars.y	1.5%	1.1%	1.1%
	c_pars.y	1.7%	1.3%	1.3%
flex	fixit.l	2.2%	1.7%	1.7%
	unfixit.l	2.7%	2.1%	2.1%
gzip	gcc-2.7.2.tar	1.1%	0.0%	0.0%
	bison-1.25.tar	1.1%	0.0%	0.0%
Average		2.3%	1.9%	1.4%

Table 3: Instruction cache miss rates for the original, estimated and profile-guided color mapping.

3. Provide a more deterministic approach for loop iteration estimation.
4. Examine alternative techniques for propagating the estimated edge weights throughout the call graph.

5.1 Synthetic graphs versus profile-driven graphs

To judge how well our heuristics compared to the dynamic profiles, we next examine some statistics comparing the call graphs generated using estimation with the profile-generated call graphs. We refer the reader to [9] for a thorough description of the profile-generated results. Our procedure mapping algorithm uses a threshold value on the call graph edge weights in order to split a graph into 2 sets. The edges and procedures with weights above the threshold value are included into the *popular* set of procedures and edges, and the remaining edges and procedures are labeled as *unpopular*. We compute the weight of a procedure (node) as the sum of all edge weights entering and exiting a node. The algorithm concentrates on accurately remapping the popular procedures, since the unpopular procedures rarely cause a change in control flow. For our approach to be useful, a high percentage of the popular nodes that reside in the profile-generated call graphs should also appear in the popular set for the estimated graphs.

Program	DT	ET	D_Siz	E_Siz	Int	Total	% Static	% W_Crrl
bison	1000	100	38	87	23	331	26.4	47.4
	1000	200	38	77	22	331	28.6	47.4
	1000	500	38	63	19	331	30.2	44.9
espresso	200	8000	94	177	85	539	48.0	91.6
	200	10000	94	174	85	539	48.9	91.4
	200	20000	94	162	83	539	51.2	84.3
li	200	500	88	110	56	575	50.9	86.9
	200	800	88	100	52	575	52.0	81.0
	200	1200	88	89	47	575	52.8	87.1

Table 4: Variance in popular procedures in the estimated call graph.

In Table 4 we compare the popular sets generated for three of the programs we used in this study. In the Table, the second and third columns are the dynamic profile (DT) and estimated (ET) thresholds used when pruning the call graphs, splitting the graphs into the popular and unpopular parts. The fourth and fifth columns are the dynamic profile (D_Siz) and estimated (E_Siz) sizes of the popular procedure sets. The sixth column (Int) is the intersection between these two sets. The seventh column (Total) is the number of procedures in the executable. The next column, (Static) shows the static percentage of popular procedures in the set that were also in the dynamic profile set. The last column, *W_Crrl*, shows the percentage of popular procedures executed that were both in the estimated set and the dynamic profile set. This is calculated as the *(dynamic weight of the intersection set)/(total dynamic weight of the entire dynamically-generated call graph)*. This last column gives an indication of how close the static call graph estimation comes to correctly finding the popular procedures. The results show that we come close to partitioning the program into popular sets for **espresso** and **li**, with 91% and 87% accuracy. For **bison** we achieve less success, only capturing 47% of the procedures in profile-generated popular set. As can be seen in Table 4, one can adjust the correlation by adjusting the threshold value for the estimated call graphs.

6 Summary

The performance of the cache-based memory system is critical in today's processors. Research has shown that compiler optimizations can significantly reduce this latency, and every opportunity should be taken by the compiler to do so.

In this paper we have shown how to statically generate an estimated call graph profile and use this for procedure reordering. We presented a new technique which employs a simple set of heuristics. The performance of this approach generally lacks the accuracy of dynamically trained program layout, but can still provide significant improvements over standard link orders.

In this study we concentrated on applying our color mapping algorithm to procedure reordering. Our algorithm can be combined and benefit from other code reordering techniques such as basic block reordering, taking into consideration looping structures, and procedure splitting. These are topics of future research, along with applying our algorithm to object oriented languages [3, 16]. In

this paper we also concentrated on the performance achieved using call edge profiles to guide the optimizations in order to eliminate first-generation cache conflicts. We are currently investigating how to apply our algorithm to use full path profiling and other collection techniques in order to collect improved temporal locality information [11].

7 Acknowledgements

This research has been supported by the National Science Foundation, Microsoft Research, and by equipment grants from Digital Equipment Corporation. We would like to acknowledge the contributions of Shantanu Tarafdar for developing the graph library used in this work, and to the DTJ editors for their help in preparing this paper for publication.

References

- [1] T. Ball and J.R. Larus. Branch prediction for free. In *Proceedings of the SIGPLAN'93 Conference on Programming Language Design and Implementation*, pages 300–313, June 1993.
- [2] B. Calder, D. Grunwald, M. Jones, D. Lindsay, J. Martin, M. Mozer, and B. Zorn. Evidence-based static branch prediction using machine learning. *ACM Transactions on Programming Languages and Systems*, 19(1), 1997.
- [3] B. Calder, D. Grunwald, and B. Zorn. Quantifying behavioral differences between C and C++ programs. *Journal of Programming Languages*, 2(4), 1994.
- [4] P. P. Chang, S. A. Mahlke, and W. W. Hwu. Using profile information to assist classic compiler code optimizations. *Software Practice and Experience*, 21(12):1301–1321, 1991.
- [5] A. P. Dempster. A generalization of bayesian inference. *Journal of the Royal Statistical Society*, 30:205–247, 1968.
- [6] N. Gloy, T. Blackwell, M.D. Smith, and B. Calder. Procedure placement using temporal ordering information. In *30th International Symposium on Microarchitecture*, pages 303–313, December 1997.
- [7] R. Hank, S. Mahlke, R. Bringmann, J. Gyllenhaal, and W. Hwu. Superblock formation using static program analysis. In *26th International Symposium on Microarchitecture*, pages 247–256. IEEE, 1993.
- [8] A.H. Hashemi, D.R. Kaeli, and B. Calder. Efficient procedure mapping using cache line coloring. WRL Research Report 96/3, October 1996.
- [9] A.H. Hashemi, D.R. Kaeli, and B. Calder. Efficient procedure mapping using cache line coloring. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 171–182. ACM, June 1997.
- [10] W.W. Hwu and P.P. Chang. Achieving high instruction cache performance with an optimizing compiler. In *16th Annual International Symposium on Computer Architecture*, pages 242–251. ACM, 1989.

- [11] J. Kalamatianos and D.R. Kaeli. Temporal-based procedure reordering for improved instruction cache performance. In *Proceedings of the 4th International Conference on High Performance Computer Architecture*, pages 244–253, Las Vegas, NV, February 1998.
- [12] G.P. Lowney, S.M. Freudenberger, T.J. Karzes, W.D. Lichtenstein, R.P. Nix, J.S. O'Donnell, and J.C. Ruttenberg. The multiflow trace scheduling compiler. *Journal of Supercomputing*, 7:51–142, 1993.
- [13] S. McFarling and J. Hennessy. Reducing the cost of branches. In *13th Annual International Symposium of Computer Architecture*, pages 396–403. Association for Computing Machinery, 1986.
- [14] K. Pettis and R.C. Hansen. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 16–27. ACM, ACM, June 1990.
- [15] C.V. Ramamoorthy. Discrete markov analysis of computer programs. In *20th National Conference*, pages 386–391. ACM, 1965.
- [16] A. Sampoga. *Architectural Implications of C and C++ Programming Models*. MS Thesis, Northeastern University, August 1995.
- [17] G. Shafer. *A Mathematical Theory of Evidence*. Princeton University Press, Princeton, NJ, 1976.
- [18] J. E. Smith. A study of branch prediction strategies. In *8th Annual International Symposium of Computer Architecture*, pages 135–148. ACM, 1981.
- [19] S. Sokolova and D.R. Kaeli. Improved static branch prediction by inspecting program behavior. Northeastern University Computer Architecture Research Laboratory Research Report NUCAR-5/97-1, May 1997.
- [20] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 196–205. ACM, 1994.
- [21] T.A. Wagner, V. Maverick, S. Graham, and M. Harrison. Accurate static estimators for program optimization. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 85–96, Orlando, Florida, June 1994. ACM.
- [22] Y. Wu and J.R. Larus. Static branch frequency and program profile analysis. In *27th International Symposium on Microarchitecture*, pages 1–11, San Jose, Ca, November 1994. IEEE.