Appeared in the Workshop on Interaction between Compilers and Computer Architectures, San Antonio, Texas, Feb. 1997

Procedure Mapping Using Static Call Graph Estimation

Amir H. Hashemi David R. Kaeli

Dept. of Electrical and Computer Engineering
Northeastern University
Boston, MA
{ahashemi,kaeli}@ece.neu.edu

Brad Calder

Dept. of Computer Science and Engineering
University of California, San Diego
La Jolla, CA
calder@cs.ucsd.edu

Abstract

As the gap between memory and processor performance continues to grow, it becomes increasingly important to exploit cache memory effectively. One technique used by compiler and linkers to improve the performance of the cache is code reordering. Code reordering optimizations rearrange a program so that sections of the program with temporal locality will be placed next to each other in the final program layout.

A number of software approaches to code reordering have been proposed. Their goal is to reduce the number of cache line conflicts. Most of these schemes 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 will refer to this scheme as **static call graph estimation**. *In this approach we use program-based heuristics to statically estimate the behavior of the call graph. Then once the estimated weighted call graph is formed, we can employ various procedure remapping 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

Profile-based feedback optimizations have been used extensively to tune the performance of programs [3, 8]. 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 drawback of profile-based optimizations is the extra time and effort required to first profile the program, and then recompile using the profile. It can also be hard to find a representative input suitable for the proposes of profiling. If a developer can not afford the time and effort to perform profile-based optimizations, the compiler must rely on static techniques to estimate the program's 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 [10, 14]. Other techniques rely on applying heuristics based on more detailed control flow analysis [1, 16, 17]. Still others have examined using machine learning techniques to statically predict the control flow at compile-time [2]. In [5], 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 [7, 6, 11]. In this paper we examine how to perform procedure reordering using heuristics to estimate the behavior of a program's 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 describe our heuristics for statically estimating the call graph, and provide cache simulation results showing the improvement in the miss rate after applying procedure reordering optimizations. In §2 we will discuss prior work in program-based control flow estimation. In §3 we describe the heuristics we use to statically estimate a call graph's behavior. In §4 we describe the experimental methodology and the procedure reordering algorithm used in our results. We then provide cache simulation results showing the effects of using our estimated call graph edges to guide procedure reordering optimizations in §5. In §6 we conclude, and discuss directions for future work.

2 Background

In this section we discuss 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 is correctly predicted as taken when using the BTFNT heuristic.

In recent work, Ball and Larus [1] 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, 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 "We 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 what 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 [17] 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 [4, 13]. 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.

Wagner et al. [16] 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 [12]. 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 es-

Variable	Value
wi	10
wl	10
wr	200

Table 1: Weight propagation factors

timated 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 is when performing optimizations that are typically guided by profiles.

3 Static Call Graph Estimation

In this section we describe the heuristics we used for our 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 call edge frequencies based on procedure call sites. 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 to a single procedure produce a single edge in the graph. 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.

One issue that needs to be addressed when constructing our call 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 assigning 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 (wr)*, breaking the recursion.

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

1. **Postdom-entry**: if the basic block containing the procedure call Y post-dominates all the entry points into procedure X, then we know procedure Y will be called if X is executed. Assign the node weight of procedure X to a call 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 (wl)* 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 (wi)*, the *loop weight (wl)*, and the *recursion weight (wr)* for the results in this paper. These values where chosen by trial and error. Future work we will study what effects varying these values have on the procedure mapping algorithms.

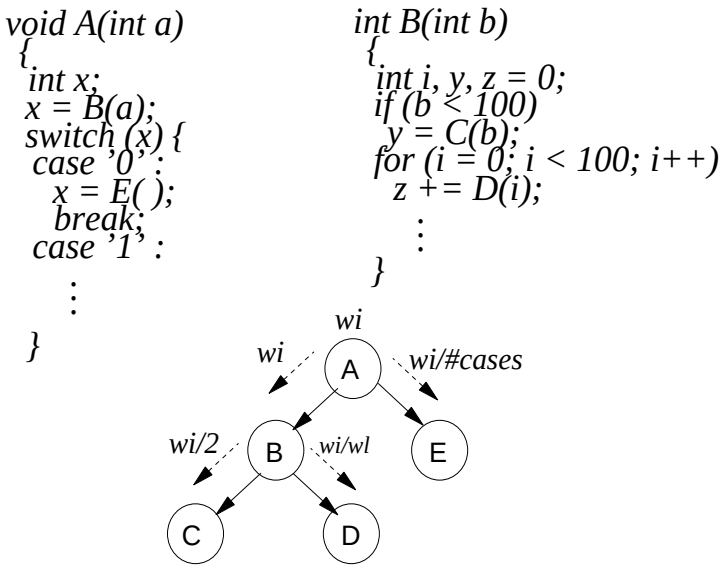

Figure 1: A sample call graph for procedures A-E. Code snipits for procedures A and B are also shown. Procedure A is the entry node, and is assigned an initial weight wi. This weight is then propagated throughout the rest of the call graph based on the heuristics described above. Each edge assumes the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

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 profile-based procedure reordering algorithms function on the edge weights. If the estimated edge weights mirror those of the profiled edge weights, our static call graph estimation should result in procedure orderings similar or close to the those generated using dynamic profiles.

4 Experimental Methods

We modified `gcc` version 2.7.2 to build the estimated call graphs needed as input to our procedure mapping algorithm. This provided us with the intermediate representation necessary to guide our heuristics described in §3.

The applications we study are taken from the SPEC92 benchmark suite and a Unix utility program taken from the gnu toolset. In the final presentation of this paper, we will have results from the SPEC95 benchmark suite and as well as a number of other programs with much higher cache miss ratios. For `espresso` and `li` we provide results across a range of inputs. In the final version of this paper, we will include a range of inputs for all of the programs studied.

To examine the performance of our statically estimated call graphs we apply these estimated edge weights to an existing procedure reordering algorithm. The procedure reordering algorithm we chose is our *color mapping* algorithm described in [6]. The algorithm processes the edges from the heaviest weighted to the lightest weighted. When processing an edge, the procedure's associated with the edge are mapped to the address space, and each procedure is assigned the cache lines (colors) used by that procedure in the address space. These colors are then used to avoid cache conflicts with other procedures as they are mapped into the address space (see [6] for a complete description of the algorithm). Other algorithms have been proposed for procedure reordering such as the depth-first algorithm of Hwu and Chang [7], and the greedy edge weight algorithm of Pettis and Hansen [11]. In [6], we showed that our color mapping algorithm consistently outperformed the Pettis and Hansen greedy algorithm. Therefore, in this paper we only provide results for our cache line color mapping algorithm.

To study the effectiveness of procedure reordering we used the *ATOM* trace-driven simulation tool [15]. ATOM is an execution-driven simulation tool for the DEC Alpha processor. Using ATOM, we model an 8KB direct-mapped instruction cache with a 32 byte

line, similar in design to the DEC Alpha 21064 and 21164 first-level instruction cache.

5 Results

In this section, we provide simulation results for the *Original* program layout, the procedure layout generated using our *Static* estimated call graph edge weights, and the layout using a *Dynamic* profile-generated call graph. These results are shown in Table 2. The results show that static call graph estimation can be effectively used to determine edge weights. While the average miss rate for programs studied was around 1%, using procedure ordering based on estimated call graph edge weights reduced the miss rate to 0.8%, a 20% reduction. Compared to profile-driven repositioning, the miss rate was reduced from 1% down to 0.6% on average. This shows that our static call graph estimation achieved almost half the reduction in misses that the dynamic profile achieved. This is a very encouraging result, indicating that static code repositioning can significantly improve a program's performance.

We can see from Table 2 that for all of the benchmarks and input combinations, we obtain a reduction in the cache miss rate, except for a couple of inputs for `espresso`. When using the `dc1` input to `espresso`, the instruction cache miss rate actually increases for our algorithm. Note that this input generates the shortest trace, and thus is more susceptible to small differences. The actual number of misses between the 2 runs differ by less than 900 instruction cache misses. The result for `bison` is also of special interest. Here we are actually outperforming the profile-driven color-based repositioning. This highlights the fact that profile data does not capture the temporal locality exhibited by a single procedure, and that finding an optimal mapping to minimize conflicts is NP-complete [9].

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

1. Incorporate static branch prediction techniques to improve upon the *one-half* heuristic currently used for conditional branches.
2. Identify commonly called, but infrequently visited, procedures (e.g., exit(), error()).
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. Our procedure mapping algorithm uses a threshold value on the call graph edge weights in order to split a graph into 2 sets [6]. 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 edges weights entering and exiting a node. The algorithm concentrates on accurately mapping out 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 dynamically-formed call graphs should also appear in the popular set for the statically-formed graphs.

In Table 3 we compare the popular sets generated for the programs we examined. In the Table, the second and third columns are the dynamic profile (DT) and static (ST) 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 static (S_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 percentage of popular procedures in the static set that were also in the dynamic profile set. The last column, *W_Crrl*, shows the percentage of popular procedures executed that where both in the static 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 the programs `bison` and `eqntott` we achieve a much less success, only capturing 47%, percentage of procedures in the estimated popular set. As can be seen in Table 3, one can adjust the correlation by adjusting the threshold value for the statically-formed call graphs.

Program	Input	# Instrs Traced in Millions	Miss Rate		
			Original	Static	Dynamic
bison	objc_pars.y	77 M	1.7	0.9	1.1
espresso	Z5xp1	29 M	1.3	1.1	0.9
	bca	486 M	0.3	0.2	0.1
	cps	591 M	0.4	0.4	0.3
	dc1	.9 M	2.8	2.9	2.3
	mlp4	84 M	1.1	0.9	0.8
	opa	136 M	0.6	0.6	0.5
	ti	74 M	0.6	0.4	0.4
	tial	1145M	0.9	0.6	0.5
eqntott	int_pri_3	2021 M	0.2	0.1	0.1
li	8-queens	1314 M	1.5	0.8	0.3
	9-queens	6938 M	1.4	0.8	0.3
Average			1.0	0.8	0.6

Table 2: Trace Driven Simulation Results.

Program	DT	ST	D_Siz	S_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
eqntott	200	50	35	47	22	498	46.8	47.9
	200	200	35	49	22	498	44.9	47.9
	200	800	35	42	19	498	45.2	47.9
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 3: Variance in popular procedures in the estimated call graph.

6 Summary

Instruction caches are commonly used to bridge the performance gap between the speed of the processor and the supporting memory. Code-reordering compiler optimizations can be employed to make better use of a cache by reducing cache conflicts and improving page usage. These optimizations are traditionally guided by profiles which are fed back into an optimizing compiler. Even though profile-based optimizations are effective, they are often expensive due to the extra time needed to perform program profiling. Code developers may choose not to use these optimizations because of this feedback step, or they might not be able to use the optimizations if their compiler does not provide any profiling mechanism. An alternative approach is to attempt to reorder statically (i.e., at compile time) by estimating the behavior of the program. This would allow these optimizations to be applied even in the absence of profile information.

In this paper, we examined the use of static call graph estimation in order to perform procedure mapping optimizations to reduce instruction cache conflicts. We estimated the edge weights in the static call graph using simple heuristics which take into consideration the control flow structure of the program (e.g., loops, conditional branches, switch statements). We then used these estimated call graph edge weights to guide our color-based procedure reordering algorithm [6]. Our results show that the static call graph estimation was able to reduce the cache miss rate of the original program on average by 20%. In compar-

ison to using profiles to perform the same optimizations, the dynamic profiles reduced the miss rate of the original program on average by 40%. This indicates that we were able to achieve almost half of the performance improvement typically seen by profile-based optimizations when using static call graph estimation.

We see many future directions for this work. We are currently modifying our algorithm to include more detailed branch heuristics and to use normalized probabilities for the estimated edge weights as in previous research [1, 16, 17]. We plan to extend the heuristics of Ball and Larus to more accurately predict which call paths will be frequently executed, and to examine other evidence combination techniques to propagate the estimated edge weights in the call graph. In addition, we plan on examining how well these techniques can be applied to other profile-based optimizations such as basic block reordering, inlining, and procedure splitting.

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] 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.

[4] A. P. Dempster. A generalization of bayesian inference. *Journal of the Royal Statistical Society*, 30:205–247, 1968.

[5] 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.

[6] A.H. Hashemi, D.R. Kaeli, and B. Calder. Efficient procedure mapping using cache line coloring. WRL Research Report 96/3, October 1996.

[7] 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.

[8] 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.

[9] S. McFarling. Program optimization for instruction caches. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, pages 183–191, April 1989.

[10] 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.

[11] 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.

[12] C.V. Ramamoorthy. Discrete markov analysis of computer programs. In *20th National Conference*, pages 386–391. ACM, 1965.

[13] G. Shafer. *A Mathematical Theory of Evidence*. Princeton University Press, Princeton, NJ, 1976.

[14] J. E. Smith. A study of branch prediction strategies. In *8th Annual International Symposium of Computer Architecture*, pages 135–148. ACM, 1981.

[15] 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.

[16] 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.

[17] 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.