Procedure Mapping Using Static Call Graph Estimation

Amir Hooshang 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 place 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.*

I. 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.

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 [9], [11]. Other techniques rely on applying heuristics based on more detailed control flow analysis [1], [4], [13], [14]. Still others have examined using machine learning techniques to statically predict the control flow at compile-time [2]. 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], [5], [10]. 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. We also point the reader to [6] for further discussion and comparison of our color-based reordering techniques with previous work.

II. Static Call Graph Estimation

The goal of static call graph estimation is to assign estimated call edge frequencies based on procedure call sites. Locations of particular interest are 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. We then propagate edge weights through the graph, based on static branch prediction heuristics. We need to indicate direction (caller-to-callee) in order to propagate weights in the proper direction.

To begin, we handle recursions in the graph by assigning a fixed 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.

We continue our algorithm 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.

The values we use for the *initial weight (wi)*, the *loop weight (wl)*, and the *recursion weight (wr)*, are 10, 10, and 200, respectively. These values where chosen by trial and error.

Program	Input	Miss Rate		
		Original	Static	Dynamic
bison	objc_pars.y	1.7	0.9	1.1
espresso	Z5xp1	1.3	1.1	0.9
	bca	0.3	0.2	0.1
	cps	0.4	0.4	0.3
	dc1	2.8	2.9	2.3
	mlp4	1.1	0.9	0.8
	opa	0.6	0.6	0.5
	ti	0.6	0.4	0.4
	tial	0.9	0.6	0.5
eqntott	int_pri_3	0.2	0.1	0.1
li	8-queens	1.5	0.8	0.3
	9-queens	1.4	0.8	0.3
Average		1.0	0.8	0.6

TABLE I
TRACE DRIVEN SIMULATION RESULTS.

The magnitude of the edge weights in the statically-weighted call graph do not have much significance. Their relative weights are much more important.

III. RESULTS

The applications we study are taken from the SPEC92 benchmark suite and a Unix utility program taken from the gnu toolset. To study the effectiveness of procedure reordering we used the *ATOM* execution-driven simulation tool [12]. 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 caches.

In Table I 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. 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 reordering based on estimated call graph edge weights reduced the miss rate to 0.8%, a 20% reduction. Profile-driven repositioning reduced miss rates 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.

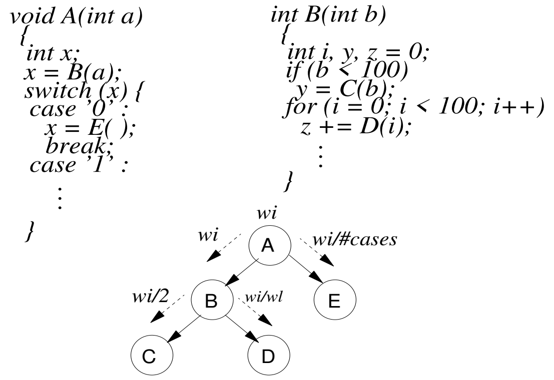

Fig. 1. A sample call graph for procedures A-E. Code snipits for procedures A and B are also shown. Procedure A is the entry node, and is assigned an initial weight wi. This weight is then propagated throughout the rest of the call graph based on the heuristics described above. Each edge assumes the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

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] 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.

[5] A.H. Hashemi, D.R. Kaeli, and B. Calder. Efficient procedure mapping using cache line coloring. In *Proceedings of the SIGPLAN'97 Conference on Programming Language Design and Implementation*, page to appear, Las Vegas, NV, June 1997. ACM.

[6] A.H. Hashemi, D.R. Kaeli, and B. Calder. Procedure mapping using static program estimation and cache line coloring. Northeastern University Computer Architecture Research Laboratory Research Report NUCAR-3/97-1, March 1997.

[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 and J. Hennessy. Reducing the cost of branches. In *13th Annual International Symposium of Computer Architecture*, pages 396–403. Association for Computing Machinery, 1986.

[10] 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.

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

[12] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the SIGPLAN'94 Conference on Programming Language Design and Implementation*, pages 196–205. ACM, 1994.

[13] T.A. Wagner, V. Maverick, S. Graham, and M. Harrison. Accurate static estimators for program optimization. In *Proceedings of the SIGPLAN'94 Conference on Programming Language Design and Implementation*, pages 85–96, Orlando, Florida, June 1994. ACM.

[14] 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.