Accurate Static Estimators for Program Optimization*

Tim A. Wagner[†], Vance Maverick,
Susan L. Graham and Michael A. Harrison

Computer Science Division
University of California, Berkeley

Abstract

Determining the relative execution frequency of program regions is essential for many important optimization techniques, including register allocation, function inlining, and instruction scheduling. Estimates derived from profiling with sample inputs are generally regarded as the most accurate source of this information; static (compile-time) estimates are considered to be distinctly inferior. If static estimates were shown to be competitive, however, their convenience would outweigh minor gains from profiling, and they would provide a sound basis for optimization when profiling is impossible.

We use quantitative metrics to compare estimates from static analysis to those derived from profiles. For C programs, simple techniques for predicting branches and loop counts suffice to estimate intra-procedural frequency patterns with high accuracy. To determine inter-procedural estimates successfully, we combine function-level information with a Markov model of control flow over the call graph to produce arc and basic block frequency estimates for the entire program.

For a suite of 14 programs, including the C programs from the SPEC92 benchmark suite, we demonstrate that static estimates are competitive with those derived from profiles. Using simple heuristics, we can determine the most frequently executed blocks in each function with 81% accuracy. With the Markov model, we identify 80% of the frequently called functions. Combining the two techniques, we identify 76% of the most frequently executed call sites.

1 Introduction

Many compiler optimizations require information about relative execution frequencies to allocate scarce resources or to reduce execution times. Intra-procedural frequency information is useful for per-function register allocation [4], code motion [3], and code layout for instruction cache packing [8]. Tail duplication, which requires trace scheduling, has been shown to improve the performance benefit of classic optimizations [6]. More aggressive optimizations require inter-procedural analyses, such as estimates of function invocation or call site frequencies. Selective function inlining [5], inter-procedural register allocation [10], and inter-procedural memory allocation tuning [2] would all benefit from accurate estimates of this information.

Execution frequencies may be estimated from profiles or by static techniques. Profiles are derived from running the program on inputs that are (usually) different from the one eventually used. Static estimates are derived by examining the structure of the program—loop nests, conditional branch expressions, the static call graph—and using this information to predict elements of the dynamic be-

*This research has been sponsored in part by the Advanced Research Projects Agency (ARPA) under Grant MDA972-92-J-1028, and supported in part by NSF grant CDA-8722788. The content of this paper does not necessarily reflect the position or the policy of the Government.

†For correspondence: `twagner@cs.berkeley.edu` or Tim Wagner, Computer Science Division, 571 Evans Hall, University of California, Berkeley CA 94720

havior.

Since static estimation does not require a separate profiling compilation, selection of sample inputs, or feedback of run-time information to the compiler, it is more convenient to use than profiling. Also, since the collection of profiles or the selection of representative inputs might be difficult, static estimates may sometimes be the best option available. Thus, an assessment of the accuracy of static estimates as input to compiler optimizations, and an idea of which estimation techniques are successful, are important for compiler writers.

Several researchers have explored the estimation of the running time of programs or subprograms, which is a related but somewhat different problem. Ramamoorthy [9] used a Markov model of control flow for this purpose. Sarkar [11] used a combination of profiling and program analysis to estimate execution times and their variances. His approach used profiling because then-current static branch prediction was inadequate for the task.

Fisher and Freudenberger [7] demonstrated that branches in programs behave consistently enough that static branch prediction is feasible. Ball and Larus [1] developed techniques that make accurate branch predictions by identifying "idioms" in executable code. We have employed a similar technique within the compiler, operating at the level of the abstract syntax and the C type system.

Wall [12] addressed essentially the same problem as this paper, constructing estimated profiles and comparing them to real profiles. He found estimated profiles inferior. We extend his techniques by incorporating branch prediction, by modelling control flow more accurately, and by separating the intra- and inter-procedural analyses. We present first a set of simple heuristics, and then explore Markov techniques for both levels of analysis. At the intra-procedural level, the simple AST-based heuristics perform well, but at the inter-procedural level, the greater sophistication of the Markov model yields appreciably more accurate estimates.

2 Experimental Framework

Our experiments were carried out with two tools: a C compiler modified to record analyses of each function and to instrument the code for profiling, and an off-line program that read both profile and analysis information and combined them to create and compare estimates.

The compiler we used for instrumentation and analysis was the GNU C compiler, `gcc`.[1] We augmented its internal representation with an explicit abstract syntax tree (AST) and control-flow graph (CFG) for each function. For each source file, the compiler saved a simplified representation of the ASTs and CFGs, along with additional data needed to do static estimation. This information and the profile output for all sample input files were then read by a single off-line program. Performing estimator construction and evaluation this way simplified the task of changing estimators, since no re-compilation or re-running of our program suite was required.

By working with the AST, we see the structure of the program before some transformations have been applied. It is well known that some compiler analyses, (e.g., constant propagation plus dead code elimination) can eliminate basic blocks or otherwise change the number of branches. Also, computing branch prediction miss rates in the presence of constant test expressions would make the rate appear lower [7]. Our software corrects for this by "predicting", but not counting towards the score, branches whose conditional expressions could be determined via constant folding.

We limited our analysis methods to those whose running time was comparable to conventional sequential compiler optimizations. Our (untuned) implementation required analysis time similar to that of `gcc`'s standard optimization option.

We used all the C programs in the SPEC92 benchmark suite and added several others. Table 1 lists the programs used in this study, along with the number of source lines they contain and a brief description.

3 A Metric for Evaluating Estimates

For most optimizations, the benefit of correctly identifying a block, or the cost of misidentification, depends on its frequency of execution. To measure the utility of an estimate, we use the *weight-matching* metric introduced by Wall [12]. This metric is designed to compare estimates and actual measurements for a set of program entities, for ex-

[1] Free Software Foundation distribution for Sun-OS 4.x, version 2.4.5.

Program	Lines	Description
alvinn	272	Back-propagation on neural net
compress	1,503	Unix compression utility
ear	5,239	Simulate sound processing in the ear
eqntott	3,482	Translate boolean functions to truth table
espresso	14,863	Minimize boolean functions
gcc	88,045	GNU C compiler
sc	8,570	Unix spreadsheet
xlisp	7,741	Lisp interpreter
awk	17,962	Unix pattern-matching utility
bison	10,887	LALR(1) parser generator
cholesky	1,898	Cholesky factorize a sparse matrix
gs	78,228	PostScript previewer
mpeg	13,761	Play MPEG video files
water	1,490	Simulate system of water molecules

Table 1: Programs used in this study. The first eight are the C programs from the SPEC92 benchmark suite.

```
/* Find first occurrence of a
   character in a string. */
char *strchr(char *str, int c) {
   while (*str) {
     if (*str == c) return str;
     str++;
   }
   return NULL;
}
```

Figure 1: Our running example: a simple implementation of `strchr`.

ample execution frequencies for the basic blocks in a function. It measures how well the estimate identifies the top n blocks. We always define n as a percentage of the number of blocks, rather than as a fixed count. We select this quantile according to the estimate and according to the measurement; the weight-matching score is the sum of the actual frequencies for the blocks in the estimated quantile, divided by the sum of the actual frequencies for the actual quantile. If the items in the estimated and actual quantiles are identical, this value will be 100%. When the estimate places different items in the quantile, the metric will be lower (or possibly equal—the cut-off point may come between actual items that have the same value).

Figure 1 and Table 2 show the application of this metric to a simple example of intra-procedural frequencies. The function is an implementation of the C string library routine `strchr`. To generate the actual counts in the table, the function was called once with arguments `("abc", 'a')` and once with arguments `("abc", 'b')`. Estimated counts were generated by the techniques described in Section 4. The actual and estimated counts are ranked in increasing order. We examine cut-off points of 20% (1 block) and 60% (3 blocks). At 20%, the estimate correctly identifies the single busiest block, for a score of 100%. At a cutoff of 60%, it identifies the first and second, but not the third; the score is 88%.[2] Often scores are higher for wider cutoffs, but this is by no means universal.

For modeling intra-procedural analysis, we permit the basic blocks of a function to compete only against others within the same function. In measuring the success of function invocation estimates, we compare whole functions. Only in ranking call sites in the entire program do we compare basic blocks from different functions.

We ran each program on several inputs (four or more in almost all cases). Weight-matching scores for an estimate were computed by comparing it separately to each profile, then averaging these scores. Scores for profiling were computed by matching each profile to the aggregate of all the other profiles and averaging the results. To aggregate profiles, we normalized them to have the same total basic block counts, then summed each block's counts.

[2] When the percentage does not exactly divide the number of blocks, we round up, and weight the extra block fractionally.

Block	Actual	Estimate	Actual Ranks	Estimate Ranks	Points at 20%	Points at 60%
while	3	5	1	1	3	3
if	3	4	2	2	—	3
$return_1$	2	0.8	3	5	—	—
incr	1	4	4	3	—	1
$return_2$	0	1	5	4	—	—
				SCORE:	100% (3/3)	88% (7/8)

Table 2: Intra-procedural weight-matching scores for the `strchr` example (Figure 1). The actual counts were generated by searching the string "abc" for the characters 'a' and 'b'. The estimated counts were derived by the **smart** heuristic (Section 4). At the 20% cutoff, the estimate picks the most frequent basic block; at the 60% cutoff it misses the third block.

4 Simple Techniques for Static Prediction

This section explores simple methods for predicting frequencies. At the intra-procedural level an AST-based model is used to form basic block frequency estimates. At the inter-procedural level, simple means are used to combine the call graph and a set of intra-procedural estimates into estimates for function frequencies. The more sophisticated Markov analysis (including the CFG-based intra-procedural model), is discussed in Section 5.

4.1 Branch and Loop Predictions

Both the simple AST-based models and the Markov models depend upon branch and loop predictions in order to estimate local control flow. We adopted a very simple loop model, predicting that all loops iterate five times. The reason for this choice was that the programs in our benchmark set fall roughly into two categories: numerical programs with simple control flow, and others with complex loop behavior. In the numerical category, it is often possible to estimate the iteration counts of loops accurately. Since the control flow of these programs is generally simple, though, the standard loop count was quite sufficient for ordering basic blocks within functions and functions within programs. In the "non-scientific" programs, the distribution of loop iteration counts was highly skewed, with a peak at the low end and a long right tail. For this subset of programs, the standard count of 5 (near the average of the observed values) was a reasonable choice; none of the structural properties we investigated were good predictors of iteration distribution were found.

Ball and Larus [1] describe a branch predictor that identifies programming idioms in executable code. We attempted to design a "smart" branch predictor, which used similar heuristics to turn AST structure, type information, and dataflow information in the compiler into branch predictions.[3]

We used several common programming idioms to predict the likely run-time direction of branches, including:

- Pointers are unlikely to be NULL
- Errors (calling `abort` or `exit`) are unlikely
- When one arm of a conditional construct writes to variables read elsewhere, that arm is more likely
- Multiple logical ANDs make a condition less likely

Figure 2 shows the results of our branch prediction heuristics, and compares them to the results from using other profiles (aggregated as described in Section 3) and to the perfect static predictor.[4] Overall, our branch predictor has a miss rate about twice that for profiling, which is similar to the results in [1]. (The rates are not directly comparable, since our predictions were made at the level of the AST, rather than from the executable code.)

[3] We tried two techniques for handling switches: guessing each arm to be equally likely, and weighting arms by the number of `case` labels on them. The latter performed slightly better, although switches did not represent a large enough fraction of dynamic branches for the choice in handling them to have much effect.

[4] As defined in [1], this uses a single profile to predict *its own* result; it thus represents the upper bound on the performance of static branch prediction, since the remaining misses are intrinsic to any software-based scheme.

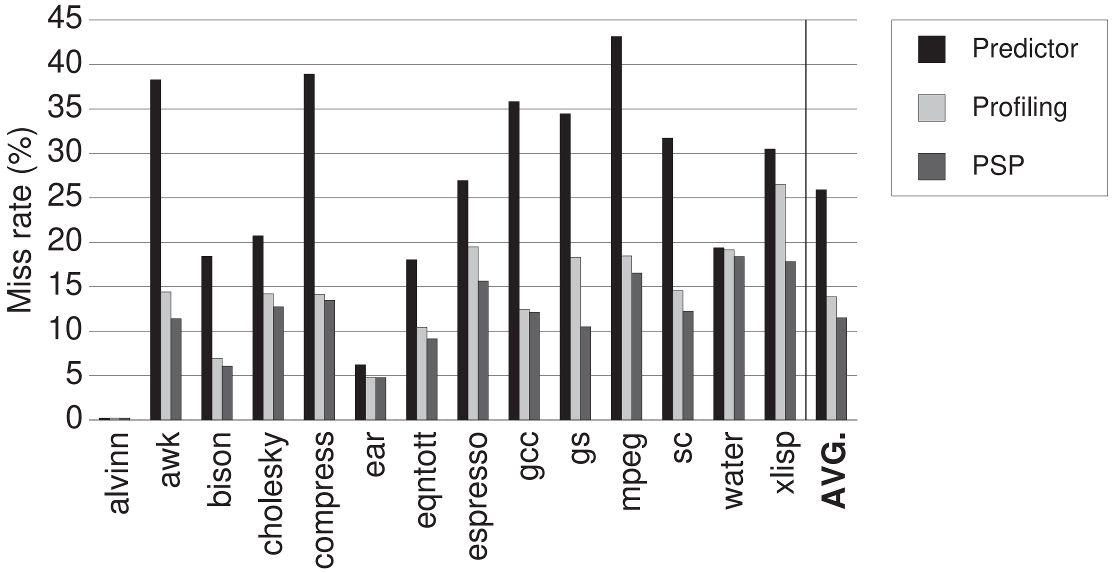


Figure 2: Miss rates (percentages of dynamic branches mispredicted) for the predictor we used, profiling with alternate inputs, and for the perfect static predictor (**PSP**). We omit branches whose condition is constant and all `switch` statements (the latter account for less than 3% of dynamic branches on average). The values for `alvinn` are uniformly low (0.23%), because its only branches are for loops that iterate many times.

4.2 Predicting Basic Block Frequencies within Functions

To predict the ordering of blocks within functions, three methods for producing relative frequency information were considered. **Loop** measures the contribution due to locating loops and assuming they execute five times whenever they are encountered. Each branch direction is considered equally likely (a 50/50 split for an `if` test, for example). **Smart** extends **loop** with the branch prediction heuristics. **Markov** is discussed in Section 5: it uses a CFG-based model to more accurately determine the control flow within a function.

Loop and **smart**, the AST-based methods, require converting individual branch predictions and loop iteration guesses into a relative ordering for the basic blocks within each function. To derive basic block frequencies, which are normalized to a single entry of the function, each branch prediction was converted into a probability.[5] Then we traversed the AST top-down, computing the execution frequency of each basic block based on the structures containing it. The AST-based model ignores `break`, `continue`, `goto`, and `return` statements. Figure 3 shows the abstract syntax tree for our `strchr` example. Since the **if** test is predicted false, the estimated frequency of the statement '`return str;`' is 0.2 times the estimated frequency of its parent. The estimated frequencies from the AST are mapped to the basic blocks in the CFG.

Given the relative ordering from the estimated basic block counts, we used the weight-matching heuristic to compare them to the actual basic block counts derived from profiling. Each function was analyzed independently, and the resulting per-function scores were then averaged, *weighted by the dynamic invocation count of the function in question.* (As with all the analyses, this comparison was done for each available profile; only the mean result is reported.)

The results are shown in Figure 4, for a 5% cutoff margin, and are compared to the results using an aggregate of other profiles for prediction. Essentially *all* the benefit to be derived comes from using loop iteration alone; the addition of smarter branch prediction and the refinement of more precise control flow analysis do not provide a significant improvement. Furthermore, the difference between static estimation and profile-based estimation at the intra-procedural level is small: in almost all cases, intra-procedural optimizations can successfully rely on a simple, compiler-provided analysis instead of

[5] We chose 0.8 for the predicted arm of a binary branch. The exact value chosen did not have a significant effect.

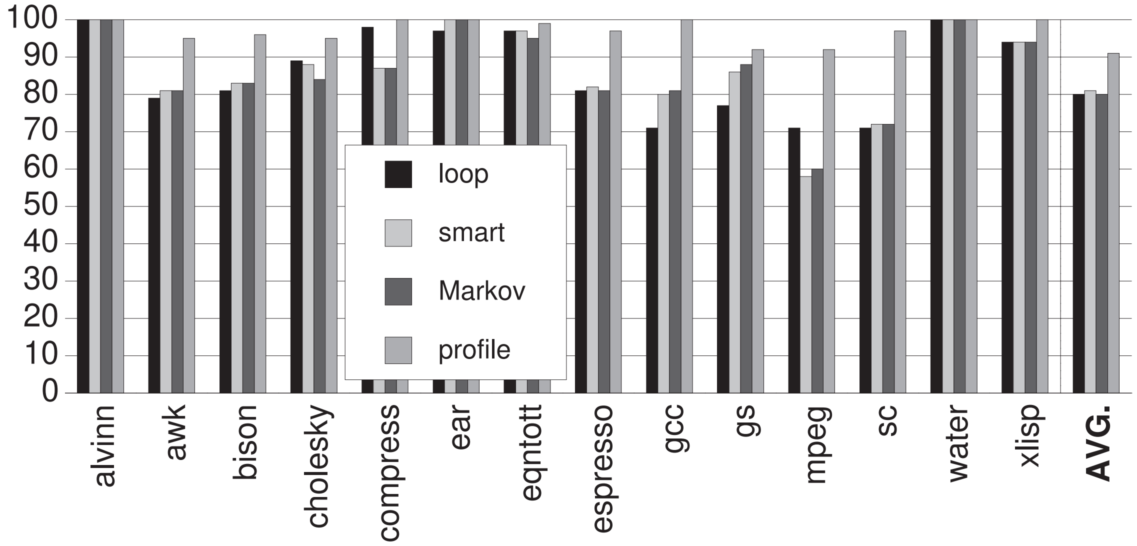


Figure 4: Weight-matching scores for estimates of intra-procedural basic block frequency. For each program, we show the score for the **loop** heuristic, the **smart** heuristic, the **Markov** technique (Section 5), and profiling with alternate inputs. The final column is the average across all programs.

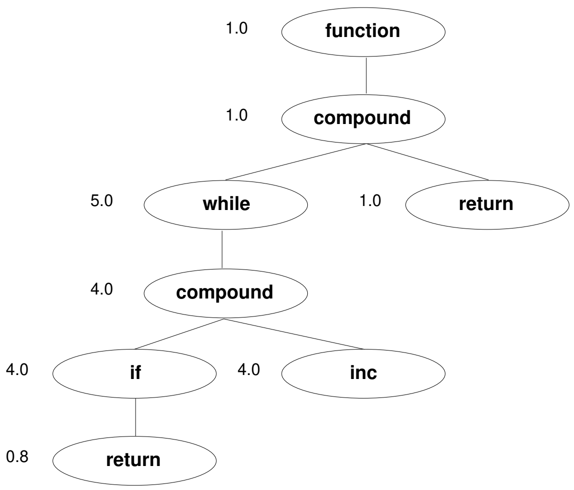


Figure 3: The AST for our running example, `strchr`. The `if` statement is predicted false in this example; the `while` loop is assumed to execute five times, so items in its body execute four times. A single top-down tree walk computes an estimated count (shown to the left of each node) for each basic block. For this function, the mapping from the AST to basic blocks is simple.

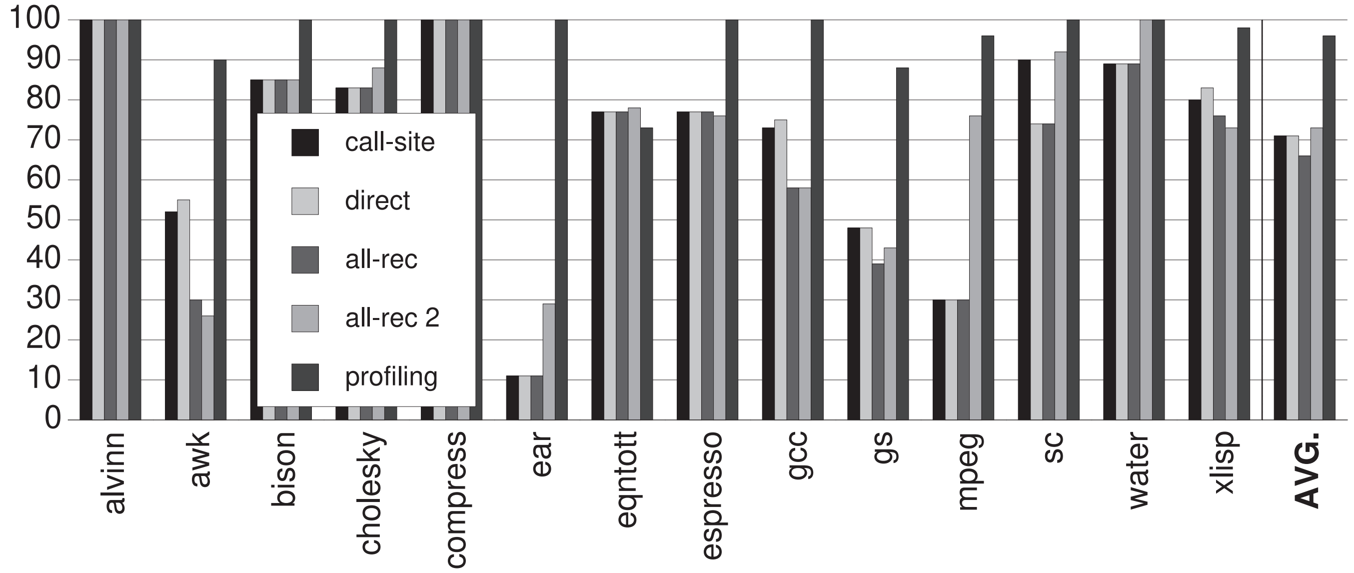
resorting to profile inputs.

4.3 Predicting Function Invocation Counts

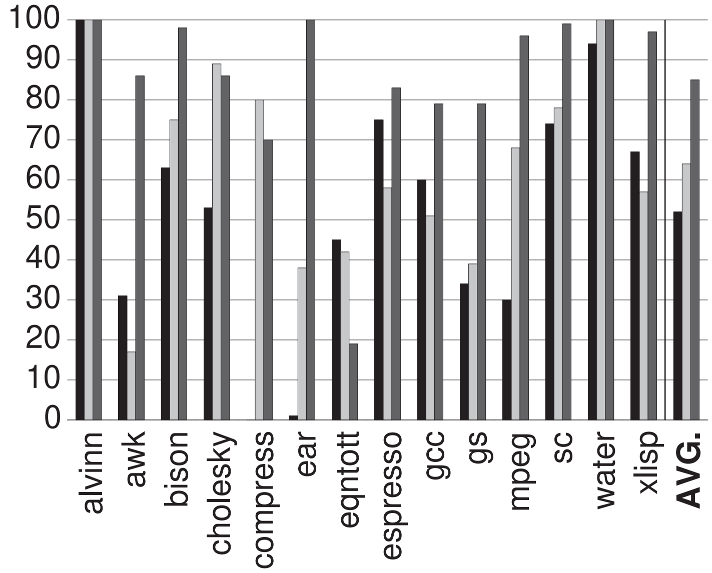
Inlining, aggressive register allocation, and other optimizations require inter-procedural information in addition to intra-procedural knowledge. Since it is used to determine other inter-procedural information, we first examine the prediction of function invocation counts.

Having used some technique to assign per-function basic block frequencies, we can then combine those estimates with information about the program's call graph to arrive at estimates for the dynamic entry count for each function. Several simple models were explored for this conversion. **call site** assigns an estimated invocation count to each function which is the sum of the basic block counts of its call sites. **direct** uses the same approach, but multiplies the final invocation count of any directly-recursive function by five. **all rec** multiplies the invocation count of functions involved in any recursion by five. Finally, **all rec2** uses the function invocation counts of **all rec** to scale up the execution counts of basic blocks, then reapplies the algorithm to compute new function counts.

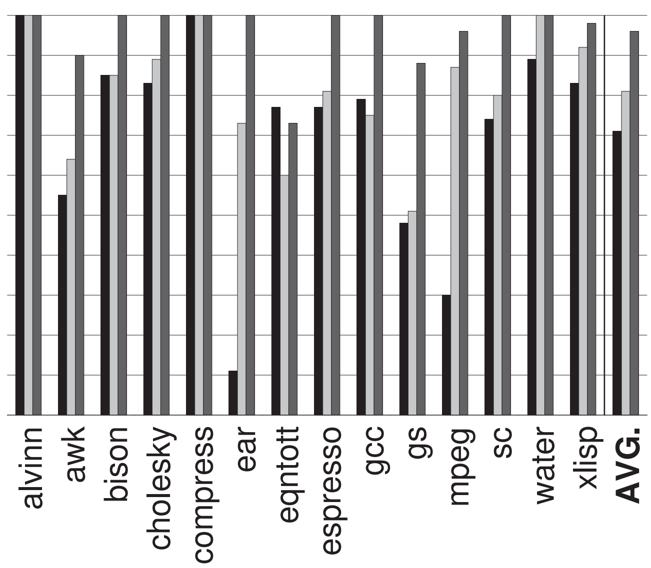
For all four of these predictors, the indirect call



(a). Simple heuristic scores. From left to right: **call-site**, **direct**, **all-rec**, **all-rec 2**, and profiling. All are weight-matching with a 25% cutoff.



(b). **direct**, **Markov**, and profiling, at the 10% cutoff.



(c). **direct**, **Markov**, and profiling, at the 25% cutoff.

Figure 5: Weight-matching scores for function invocations by estimates and profiling. All estimates are built on the **smart** intra-procedural estimator. At the 25% cutoff, the **Markov** function-invocation estimator scores 81% on average.

site counts are summed and divided among the functions whose address is taken, weighted by the (static) number of address-of operations done on that function name.

Figure 5a gives the results of function invocation prediction for the simple predictors, using **smart** to generate the initial basic block counts, and for profiling. **all rec2** does slightly better at the 25% cut-off; **direct** performs nearly as well, and is slightly more stable across a range of cut-offs (for this reason, we will use it hereafter as the best choice among this set of predictors).

As we will see in the next section, function invocation prediction can be distinctly improved by using a more accurate model of control flow.

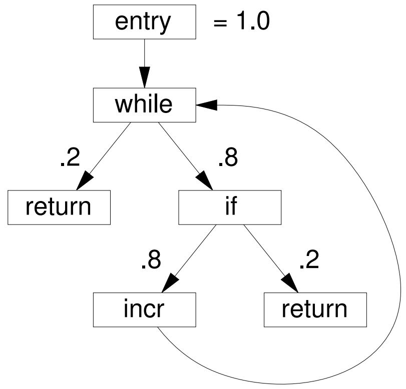


Figure 6: Control-flow graph for `strchr`, annotated with entry value and branch probabilities. Unmarked arcs have probability 1.

5 Markov Modeling of Control Flow

Control flow within a function may be modeled as a Markov process. The state of the model is the block currently being executed; the transitions between states are the arcs between blocks. The relative execution frequencies of the blocks may be computed from the probabilities associated with the arcs. We first illustrate this concept on a single function, then apply it to inter-procedural analysis.

This model was introduced by Ramamoorthy as a method of estimating program execution time [9]. We apply it instead to estimating the relative execution frequency of program regions. We also handle certain difficulties in the model that he did not discuss.

5.1 Estimating Block Frequencies

If all the branch probabilities in a function are known, including those for the controlling expressions of loops, the relative execution frequencies of all the blocks in the function may be computed exactly. Consider the example in Figures 6 and 7. The execution frequency of every block in the function's control-flow graph may be expressed as a linear function of the other frequencies, using the branch probabilities associated with the arcs as multipliers. The frequency of the entry block is set arbitrarily at 1. The graph is thus translated into a system of n equations in n unknowns, where n is the number of nodes in the graph. These equations are solved using ordinary methods for linear systems.

The structural heuristics for estimating basic block frequencies take no account of `break`, `continue`, `goto`, or `return` statements. Frequencies inside a loop are multiplied uniformly by a constant. The Markov model, however, reflects these explicit transfers of control. In Figure 6, the probabilities assigned to the `while` statement are 0.8 true and 0.2 false, corresponding to a test count of 5. The solution to the equations yields a test count of only 2.78, because the `return` within the loop reduces the flow back to the top.

For static estimation, we apply this technique with the same estimated probabilities used for the **smart** intra-procedural heuristic. As we see in Figure 4, this model shows no improvement over the simple heuristics. In the next section, though, we will see that the Markov model performs well at the inter-procedural level.

It is an open question whether static branch prediction can be accurate enough to make good use of the intra-procedural Markov model (for example, by using a static predictor that generates probabilities directly, rather than a true/false guess).

5.2 Estimating Function Invocation Frequencies

Computing execution frequencies on the call graph is amenable to the same Markov representation as control-flow graphs within functions. From intra-procedural analysis, whatever the technique, we have an estimate of execution frequency for every

$$
\begin{aligned}
entry &= 1\\
while &= entry + incr\\
if &= 0.8\, while\\
return_1 &= 0.2\, if\\
incr &= 0.8\, if\\
return_2 &= 0.2\, while
\end{aligned}
$$

(a). Linear equations for the relative execution frequencies of nodes in the control flow graph of `strchr` (Figure 6).

$$
\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & 1 & 0 \\ 0 & .8 & -1 & 0 & 0 & 0 \\ 0 & 0 & .2 & -1 & 0 & 0 \\ 0 & 0 & .8 & 0 & -1 & 0 \\ 0 & .2 & 0 & 0 & 0 & -1 \end{pmatrix}
\begin{pmatrix} entry \\ while \\ if \\ return_1 \\ incr \\ return_2 \end{pmatrix}
=
\begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}
$$

$$
\begin{pmatrix} entry \\ while \\ if \\ return_1 \\ incr \\ return_2 \end{pmatrix}
=
\begin{pmatrix} 1 \\ 2.78 \\ 2.22 \\ 0.44 \\ 1.78 \\ 0.56 \end{pmatrix}
$$

(b). Matrix solution.

Figure 7: Deriving block execution frequencies from branch probabilities.

call site, relative to the frequency with which the containing function is called. In inter-procedural analysis, functions are nodes, and these relative frequencies become the multipliers associated with the arcs. Arcs between the same pair of functions are merged. The function `main` is assigned an execution count of 1. Solving this system of equations yields an estimate of the execution frequency of each function in the program.

In Figure 5, we compare the best simple predictor of function invocations, **direct**, to the Markov model; **Markov** improves noticeably on the simple estimate, scoring 10% higher at both the 10% and 25% cutoffs.

Before applying this technique to other inter-procedural questions, we must discuss two problems that arise in constructing the Markov model.

5.2.1 Problem: Function pointers

Function pointers make the call graph inexact. We cannot determine statically which functions will be reached by a call through a pointer, and so are unable to assign correct weights to the potential arcs from that call site. We approximate the weights by creating a special node, the "pointer node", that acts as the target of all calls through pointers. We draw an arc from this node to every function that has its address taken, weighting it according to the number of (static) address-of operations done on that function name.

This approximation is a risk only with codes that use function pointers very heavily. The Markov model cannot assign a high estimated frequency to an indirectly called function, because it is forced to make all of them nearly equiprobable. One program in this category is `xlisp`: all the 173 built-in Lisp functions are called by pointer. In practice, though, regardless of the input program, the Lisp interpreter spends most of its time in the read/eval/print loop and in garbage collection. The Markov model correctly identifies these functions as among the busiest.

The only one of the programs in which a complex system of function pointers is used heavily enough for this analysis to fail is `gs`, in which some 650 functions (about half the functions in the program) are referenced indirectly. Here both the Markov and the simple heuristics do badly.

Indirect control flow can also arise in the intra-procedural case: `switch` statements are one example, as are pointers to labels (a non-ANSI extension supported by `gcc`).

5.2.2 Problem: Recursion

Recursion can make the call graph numerically ill-formed. Because call frequencies are only estimates, we may assign call-graph arc weights that are, strictly speaking, impossible. Specifically, we may give a weight greater than 1 to an arc from a function to itself. If correct, this would mean that

```
/* Count the number of nodes in a
   binary tree */
int count_nodes(struct tree_node *node) {
    if (node == NULL) return 0;
    else return (count_nodes(node->left) +
                 count_nodes(node->right) +
                 1);
}
```

Figure 8: Incorrect branch prediction can lead to invalid estimates of recursion frequency.

for every time the function is called, it calls itself again more than once—that is, it never returns. In solving the linear system, this invalid weight will lead to a negative value for the function's execution frequency.

Figure 8 shows how this problem can arise from an incorrect branch prediction. The conditional statement in `count_nodes` branches on a test of a pointer against NULL. This is one of the patterns our branch predictor recognizes: it predicts that the `else` branch will be taken with greater probability. (In reality, the pointer will be NULL more often than not, because a binary tree, in this implementation, will always have more empty child slots than filled ones.) Because the branch we predict contains two recursive calls, the first formulation of the call graph will give a multiplier of 1.6 to the arc from `count_nodes` to itself.

In the case of direct recursion, this problem can be corrected easily: recursive arcs with a probability greater than 1 are changed to a standard value of 0.8. With more complex recursion, it is difficult to detect by inspection. Accordingly, we first attempt a solution for the entire program; then, if any estimates are negative, we separate the graph into strongly-connected components (SCCs). We consider each of these in isolation, constructing a subgraph with an artificial main node, which calls the various nodes in the SCC in proportion as they are called from other nodes in the real graph. We apply a somewhat stricter criterion to these subproblems than to the graph as a whole: their solution may not include negative values, *or* values larger than some ceiling.[6] If the solution of this subproblem fails, we scale down all the arc probabilities in the SCC by a constant, repeating until the solution succeeds. Once each of the SCCs in the graph has passed this test, the entire graph may be solved without error.

In principle, a loop created by a `goto` could cause a similar problem at the intra-procedural level, if control were transferred out of the loop only by `longjmp` or `exit`. There were no such cases in any of the programs examined.

[6] After some experimentation, we chose a ceiling of 5. The arc from the artificial main node of the subproblem to each of the nodes in the SCC received a flow of m/n, where m is the number of calls to the target from outside the SCC, and n the total number of calls into the SCC from outside.

5.3 Estimating Call Site Frequencies

In function inlining, the crucial information derived from a profile is the frequency of execution of specific call sites [5]. Combining our intra- and inter-procedural heuristics, we can estimate the global ranking of call sites by frequency. It is difficult or impossible to inline calls through pointers, so we omit them from these scores. Figure 9 shows the accuracy of call site estimates. We identify the busiest 1/4 of the call sites in our programs with 76% accuracy.

6 Example: Using Estimates to Guide Optimization

One simple practical use of function invocation estimates is selective optimization: optimizing only those functions that are expected to be called frequently. We measured the performance of selective optimization with `compress`,[7] using function rankings from our heuristics and from two forms of profiling. We timed `compress` on a sample load; first without optimization, then with 1, 2, 3, 4, 5, and 6 functions optimized (chosen according to each of the three rankings), and then with all 16. We used `gcc`'s `-O2` switch to optimize the candidate functions.

The ordering was chosen in three different ways: by the static Markov estimate of function invocations, by the results of the first profile used on `compress` in the earlier sections, and by the aggregated (normalized and summed) results of the remaining three profiles. For each ordering, we added one function at a time to the optimized set, and ran this version of the binary on an input set different from the one used for profiling. Figure 10 shows the results. The performance increases monotonically as

[7] This was the only program that was small enough to make this experiment feasible.

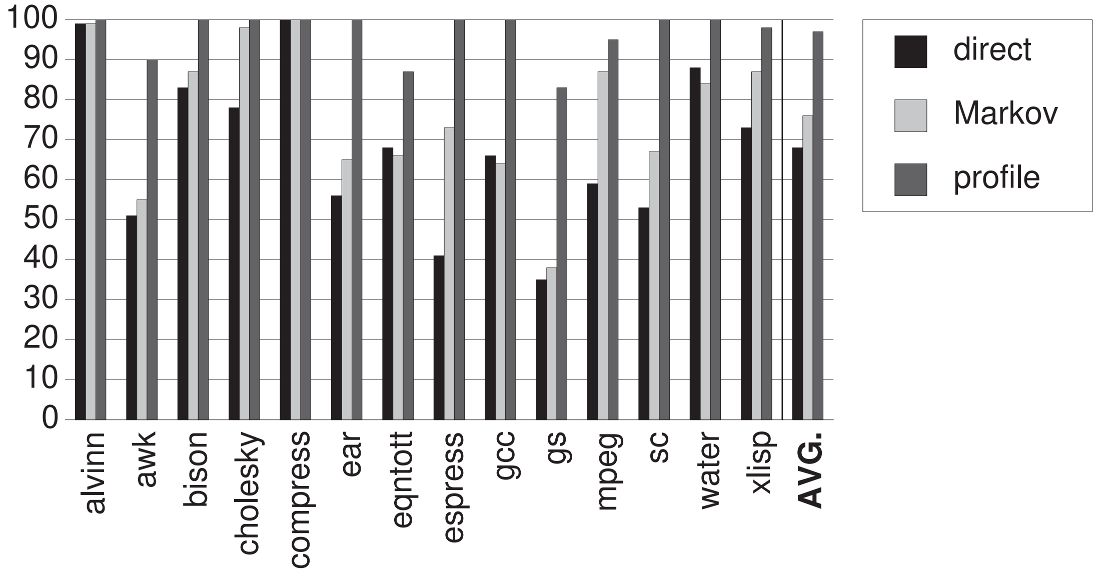


Figure 9: The combination of intra- and inter-procedural heuristics provides estimates of call site frequency. These scores were computed at the 25% cutoff.

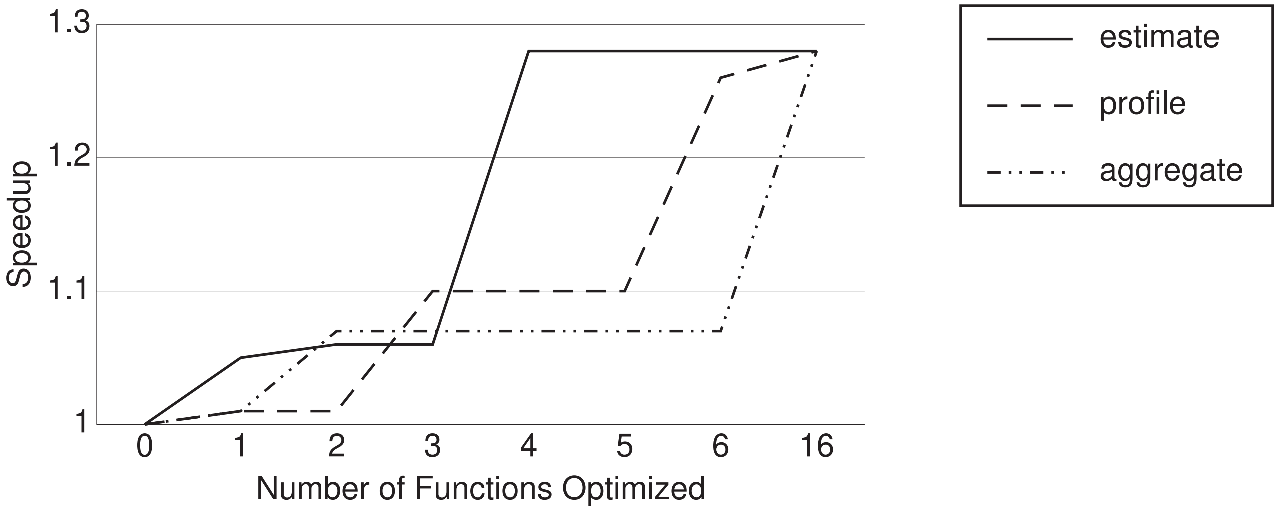


Figure 10: Speedup achieved by selectively optimizing functions in `compress`. The run time of the program is dominated by 4 of its 16 functions. The Markov estimate, which scores 100% at this 25% cutoff, identifies these functions correctly.

functions are added, but the specific ordering determines how the performance improves with each additional function. None of the estimates select the best ordering for this input (in fact, the aggregate is particularly unlucky). However, at the 25% cutoff point, the static estimate has identified the top four functions correctly. Optimizing the remaining 12 functions does not measurably improve the performance.

7 Conclusion

The goal of this work was to provide optimizing compilers with accurate estimates of the relative execution counts of various program entities: basic blocks, functions, and call sites. We have presented a series of estimation techniques and measured their accuracy by comparing them to profiling.

For intra-procedural frequencies, a simple loop-based analysis is sufficient, and rivals the ability of profiles to predict the frequently executed block(s) within each function.

To predict function invocation counts, relative call site use, or other inter-procedural quantities, simple prediction techniques do not perform as well; their performance may be improved by better modeling of control flow using a Markov model. The time to perform this analysis is comparable to that of other optimizations, and may eliminate the expense and complexity of profiling. All the techniques discussed in this paper may be easily implemented in a production compiler.

Acknowledgments

David Wall and Stefan Freudenberger assisted us in collecting profiling inputs. Jim Larus provided further information about the techniques used in [1]. Andrew Gelman and Chris Genovese provided statistical advice. The paper benefited from the comments of many colleagues, including Steven Lucco, Ethan Munson, Robert Wahbe, and the referees. Finally, we are grateful to the Free Software Foundation for providing the GNU C compiler, without which this work could not have been accomplished.

References

[1] Thomas Ball and James R. Larus. Branch prediction for free. *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 300–313, 1993.

[2] David A. Barrett and Benjamin G. Zorn. Using lifetime predictors to improve memory allocation performance. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 187–196, Albuquerque, New Mexico, 1993.

[3] Preston Briggs and Keith D. Cooper. Effective partial redundancy elimination. *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, 1994.

[4] David Callahan and Brian Koblenz. Register allocation via hierarchical graph coloring. *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 192–202, 1991.

[5] Pohua P. Chang, Scott A. Mahlke, William Y. Chen, and Wen-mei W. Hwu. Profile-guided automatic inline expansion for C programs. *Software—Practice and Experience*, 22(5):349–369, May 1992.

[6] Pohua P. Chang, Scott A. Mahlke, and Wen-mei W. Hwu. Using profile information to assist classic code optimizations. *Software—Practice and Experience*, 21(12):1301–1321, December 1991.

[7] Joseph A. Fisher and Stefan M. Freudenberger. Predicting conditional branch directions from previous runs of a program. *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 85–95, 1992.

[8] Scott McFarling. Program optimization for instruction caches. *Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 183–191, 1989.

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

[10] Vasta Santhanam and Daryl Odnert. Register allocation across procedure and module boundaries. *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 28–39, 1991.

[11] Vivek Sarkar. Determining average program execution times and their variance. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 298–312, Portland, Oregon, 1989.

[12] David W. Wall. Predicting program behavior using real or estimated profiles. *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 59–70, 1991.