To appear: *27th IEEE/ACM Inter'l Symposium on Microarchitecture (MICRO-27)*, Nov. 1994.

Static Branch Frequency and Program Profile Analysis

Youfeng Wu
wu@sequent.com
Sequent Computer Systems, Inc., D2-798
15450 S.W. Koll Parkway
Beaverton, OR 97006-6063

James R. Larus
larus@cs.wisc.edu
Computer Sciences Department
University of Wisconsin–Madison
1210 West Dayton St.
Madison, WI 53706

Abstract: Program profiles identify frequently executed portions of a program, which are the places at which optimizations offer programmers and compilers the greatest benefit. Compilers, however, infrequently exploit program profiles, because profiling a program requires a programmer to instrument and run the program. An attractive alternative is for the compiler to statically estimate program profiles.. This paper presents several new techniques for static branch prediction and profiling. The first technique combines multiple predictions of a branch's outcome into a prediction of the probability that the branch is taken. Another technique uses these predictions to estimate the relative execution frequency (i.e., profile) of basic blocks and control-flow edges within a procedure. A third algorithm uses local frequency estimates to predict the global frequency of calls, procedure invocations, and basic block and control-flow edge executions. Experiments on the SPEC92 integer benchmarks and Unix applications show that the frequently executed blocks, edges, and functions identified by our techniques closely match those in a dynamic profile.

Keywords: compiler optimization, program profile, Dempster-Shafer theory, performance evaluation, static program analysis.

1. Introduction

A compiler improves a program by applying correct and profitable optimizations—which do not change a program's semantics and reduce its running time. Optimization correctness has received more attention than profitability, because incorrect optimizations affect a program's result, but unprofitable optimizations merely slows the program. Recently, however, the advent of ambitious optimizing compilers and the myriad opportunities presented by parallelism have increased the range of optimizations available to a compiler and, consequently, also increased the difficulty of choosing an appropriate one. In particular, increased instruction-level parallelism leads to more speculative execution, which requires a compiler to accurately assess a program's likely path. More generally, to select a profitable optimization, a compiler must predict how often portions of a program execute and use these frequencies to model the costs and benefits of alternative optimizations [Chen-92]. This paper addresses the first step by showing that a compiler can accurately estimate the relative execution frequency of portions of a program by statically predicting both branch frequencies and a program profile.

A profile reports the number of times that a basic block or control-flow graph (CFG) edge executes. A profile typically is reported in absolute values—how often a statement executed—but is mainly used as a relative measure to compare the execution frequencies of portions of a program. Most profiles result from dynamically counting events during a program's execution [Ball-92]. Dynamic profiling produces accurate information about a single program execution, but it requires programmer intervention. A programmer must instrument the program with measurement code—either with a compiler option [Graham-83] or a separate tool such as pixie or qpt [Ball-92]—then run the program with appropriately-chosen input data. Finally at this point, the programmer can compile the program with the benefit of profile information. As the program changes, this process repeats. Another drawback of dynamic profiling is increased execution time, which can affect the behavior of real time and reactive systems.

An alternative is *static profiling,* in which a compiler estimates execution frequencies (not absolute counts) with static program analysis. A static profile eliminates the drawbacks of dynamic profiling—if it accurately captures a program's dynamic behavior. Recent work suggests that static analysis can predict dynamic program behavior. Fisher and Freudenberger [Fisher-92] observed that many programs' dynamic branching behavior is independent of their input data. Ball and Larus developed a simple algorithm that statically predicts the outcome of a conditional branch with good accuracy [Ball-93]. Wagner et al. used simple estimates of branch probabilities to compute static profiles [Wagner-94].

This paper improves on Wagner's work in several ways. We present a new algorithm for statically estimating the probability that a branch is taken. This algorithm uses the Dempster-Shafer theory of evidence to combine several predictions of the outcome of a branch into an estimate of the frequency with which the branch is taken. This paper makes its initial predictions based on Ball and Larus's heuristics. However, the same algorithm can also combine several dynamic profiles, without introducing a bias for longer running executions. We also present new and more efficient techniques for calculating intra-procedural (*local*) and inter-procedural (*global*) block execution frequencies, local and global branch probabilities, and function call and invocation frequencies.

To measure the effectiveness of our static profiles, we compared them against dynamic profiles of the SPECint92 C benchmarks and several Unix applications. We used Wall's weighted-matching technique [Wall-91] to evaluate the two profiles. As an example, our profiles identified the blocks in the group of the 20% most frequently executed (dynamic) blocks that accounted for 82% of this group's counts. By contrast, Wall's heuristics (on a different set of programs) identified the blocks that accounted for approximately 50% of this group's executions. Similarly, our technique accounted for 69% of the intra-procedural edge frequencies, 77% of inter-procedural block frequencies, 79% of global edge frequencies, 85% of global function invocation frequencies, and 72% of global function call frequencies (of the top 20% of each group). These experiments show that our technique can identify the most heavily executed portions of programs.

The paper is organized as follows. Section 2 discusses related work. Section 3 outlines Ball and Larus's heuristics and describes how we use Dempster-Shafer theory to combine branch probabilities. Section 4 shows how to obtain intra-procedural block and branch frequencies. Section 5 presents an algorithm to estimate function call and invocation frequencies. Section 6 shows how to obtain inter-procedural block and edge frequencies. Section 7 presents the performance results. Section 8 summarizes the paper and points out future directions. Appendix A contains a small, complete example to illustrates the algorithms.

2. Related Work

Profiles can guide a programmer or compiler to the most heavily executed portions of a program, which are typically the places at which optimizations produce the greatest benefits. For example, branch probabilities can guide instruction generation and scheduling to reduce stalls on pipelined processors [Fisher-81, Hank-93]. Block and function execution frequencies can identify program bottlenecks during optimization [Chang-91] or assist in performance analysis [Sarkar-89]. Branch and function call frequencies help order code for instruction scheduling [Hwu-89, McFarling-89] or enhance memory reference locality [Pettis-90, Wu-92].

Most previous work on profiling studied techniques for decreasing the cost of profiling or building better tools. For example, Graham et al. [Graham-83] describe a hierarchical program profiler (gprof), which accounted for procedure calls. Sarkar [Sarkar-89] showed that a program-dependence graph can reduce the number of points at which execution frequency must be counted during profiling. Ball and Larus [Ball-92] identified a minimal size and cost set of points at which to record dynamically frequency. While this prior work focused on dynamic program profiling, this paper describes techniques for static profile estimation.

Symbolic complexity analysis attempts to produce an expression relating a program's running time to its input [Hicky-88, Huelsbergen-94]. It is far more ambitious than the work in this paper, which merely predicts an ordering of the execution frequency of program components, rather than producing absolute estimates of their frequencies.

Wall studied the problem of predicting program behavior based on static analysis [Wall-91]. His heuristics were much simpler than those in this paper. For example, to predict basic block frequencies, he tried four heuristics: the block's loop nesting depth, a combination of loop nesting depth and the distance to a call graph leaf, the number of call sites for the containing function, and a combination of the loop nesting depth and call sites. These heuristics, which did not consider details of a program's control-flow graph produced far less accurate predictions than the heuristics in this paper.

This paper builds on earlier work on static branch prediction, most notably Ball and Larus's heuristics and measurements [Ball-93]. We extend their techniques to predict branch probabilities, not just branch direction, and shows how probabilities can be used to estimate the frequencies of basic blocks and edges.

Wagner et al. [Wagner-94] independently developed a static profile technique similar in several respects to the one in this paper. They predict the outcome of conditional branches by applying a subset of Ball and Larus's [Ball-93] branch prediction heuristics to programs' abstract syntax trees (which allowed them source-level information) and converted these predictions into branch probabilities with a fixed weighting of 80% for the predicted edge and 20% for the other edge. Our approach, by contrast, works both for AST and lower-level representations that lack semantic and syntactic information. In addition, our Dempster-Shafer technique (Section 3) answers the open question raised in Wagner's paper of how to convert branch predictions into branch probabilities. Our probabilities had an unweighted standard error 0.7-15.9% lower than their fixed weighting. We reran our experiments (Section 7) with their 20/80% probabilities, which produced roughly as accurate estimations for local blocks and edges, but less accurate profile estimations for global block and edge frequencies and procedure calls and invocations.

Wagner also presents three techniques for estimating profiles from branch probabilities, the best of which is similar to the technique (Sections 4 and 5) of this paper, which was originally described by Ramamoorthy [Ramamoorthy-65]. They used Ramamoorthy's formulation of the problem as a set of linear equations and solved it with sparse matrix techniques, which have a worst case $O(n^3)$ running time and do not handle non-terminating programs. We solve directly for profile frequencies, with an elimination algorithm that has worst case $O(n^2)$ running time and easily handles non-termination.

3. Branch Prediction and Probability

A *branch prediction* algorithm predicts if a branch will be taken (a yes–no, binary decision), assuming control reaches the branch's block. A *branch probability* is an estimate of the likelihood (a value between 0 and 1) that a branch will be taken. A *block or branch frequency* is a measure of how often a block is executed or a branch is taken. For example, in the following code:

```
b1:    if ( condition )
b2:        statement 1;
b3:    else statement 2;
```

a branch prediction might claim that branch $b_1 \rightarrow b_2$ is taken and branch $b_1 \rightarrow b_3$ is not taken. A probability estimate might predict that branch $b_1 \rightarrow b_2$ is taken with probability 0.81 and branch $b_1 \rightarrow b_3$ has probability 0.19. Furthermore, a branch frequency estimate might find that block b_1 executes 80 times and consequently branches to block b_2 65 times and block b_3 15 times.

3.1 Branch Probabilities

Our most basic result is a new technique for predicting branch probabilities. Our work starts with Ball and Larus's branch prediction heuristics [Ball-93]. By viewing each heuristic prediction as a binary experiment, we approximate the probability of a branch being taken by the frequency of a correct prediction. In other words, if a heuristic's correctly predicts that a branch is taken *M* of *N* times (and *N* is large enough), then the *observed* probability of taking the branch is *M*/*N*. Ball and Larus measured, on a large number of programs, the frequency of correct predictions (hit rate) for each heuristic [Ball-93]. We use their numbers. If a heuristics, with a hit rate of *R*, predicts a branch is taken, we claim that the probability of taking the branch is *R*. Although our measurements come from a different architecture and compiler than their heuristics, the published hit rates produce good results.

This section review Ball and Larus's branch prediction technique [Ball-93] and describes our algorithm for predicting branch probability. Their techniques uses a small collection of tests of local program characteristics. The first heuristic applies to branches controlling loop execution:

- **Loop branch heuristic (LBH).** Predict as taken an edge back to a loop's head. Predict as not taken an edge exiting a loop .[1]

The following four heuristics analyze the branch comparison and CFG successor for non-loop branches:

- **Pointer heuristic (PH).** Predict that a comparison of a pointer against null or of two pointers will fail.
- **Opcode heuristic (OH)**. Predict that a comparison of an integer for less than zero, less than or equal to zero, or equal to a constant, will fail.
- **Guard heuristic (GH)**. Predict that a comparison in which a register is an operand, the register is used before being defined in a successor block, and the successor block does not post-dominate will reach the successor block.
- **Loop exit heuristic (LEH)**. Predict that a comparison in a loop in which no successor is a loop head will not exit the loop.

The following four heuristics analyze CFG successors:

- **Loop header heuristic (LHH)**. Predict a successor that is a loop header or a loop pre-header and does not post-dominate will be taken.
- **Call heuristic (CH)**. Predict a successor that contains a call and does not post-dominate will not be taken.
- **Store heuristic (SH)**. Predict a successor that contains a store instruction and does not post-dominate will not be taken.
- **Return heuristic (RH).** Predict a successor that contains a return will not be taken.

Heuristics	Probability of taking branch
Loop branch (LBH)	88 %
Pointer (PH)	60 %
Call (CH)	78 %
Opcode (OH)	84 %
Loop exit (LEH)	80 %
Return (RH)	72 %
Store (SH)	55 %
Loop header (LHH)	75 %
Guard (GH)	62 %

Table 1. *Branch probability predicted by Ball-Larus heuristics.*

When a heuristic predicts exactly one successor of a branch as taken, the heuristic *applies* to the branch. If a heuristic applies to a branch and the predicted taken branch is actually taken, the heuristic *hits*. The percentage of predictions that hit is the heuristic's *hit rate*. We can treat a hit as the successful outcome of a binary experiment. By repeating the experiment *N* times, we obtain *M* ($M \leq N$) true outcomes and N - M false outcomes. If *N* is reasonably large, the hit ratio *M*/*N* approximates the probability of a successful outcome. A fundamental insight of this work is that a heuristic's hit rate is a good estimate of the *probability* that the predicted branch will be taken at run time. Table 1 lists Ball and Larus's hit rates. For example, when heuristic PH applies to a branch, we claim that PH predicts that the branch will fail 60% of the time.

3.2 Combining Predictions

Typically, several of Ball and Larus's heuristics apply to a branch. They predicted a branch's outcome with the first heuristic—from a pre-computed, static ordering—that applied to a branch and disregard the other heuristics. We use a new algorithm, based on the Dempster-Shafer theory of evidence [Shafer-76], to combine probability predictions of all applicable heuristics into a stronger probability estimate.

Combining probabilities from several heuristics can produce a result that is more accurate than the individual estimates. Ball and Larus combine heuristics by selecting the first applicable one from the sequence: PH, CH, OH, RH, SH, LHH, GH. This approach worked well for branch prediction, which only produces a binary outcome. For branch probabilities, however, ordering throws away useful information. For example, in the following code:

```
if ( i < 0 ) then  x = y;
    else  { x = 1;  return; }
```

[1] We separated Ball and Larus's loop branch heuristic into the Loop Branch Heuristic (LBH) and Loop Exit Heuristic (LEH). This allowed LEH to be combined with non-loop heuristics.

both OH and RH apply. OH suggests that the else-branch is taken, but RH claims that the then-branch is taken. Ordering resolves the conflict by ignoring RH. This reasonably predicts the else-branch, but it results in a 84% probability for this branch. Intuitively, the negative evidence from RH should reduce the probability.

Dempster-Shafer theory [Shafer-76] provides a mathematical technique for combining evidence of this type into a prediction of the probability of an outcome. It starts from a *basic probability* in the range [0,1] . This value is the degree to which evidence supports a hypothesis. For branch probability estimation, the hypothesis is: “branch b is taken” or “a branches other than b is taken (b is not taken).” The evidence is that a heuristic predicts the branch.

If more than one heuristic supports or denies a hypothesis, Dempster-Shafer theory provides an elegant way to combine basic probabilities. Assume an event has a set of k exhaustive and mutually exclusive possible outcomes $A = \{A_1, A_2, ..., A_k\}$. Each subset of A has a corresponding hypothesis that the events in the subset occur. A piece of evidence assigns a value in [0, 1] to every hypothesis (subset of A), so the values for the evidence sum to 1. This value indicates the likelihood that the event occurs. The empty set is assigned 0. This assignment is called a *basic probability assignment* (denoted by function m). For example, a branch $b \rightarrow \{b_1, b_2, ..., b_k\}$ has k exhaustive and mutually exclusive outcomes $A = \{b_1, b_2, ..., b_k\}$. If a heuristic predicts the probability of taking b_i is μ and the probability of not taking b_i is $1 - \mu$, we get the following basic probability assignment: $m_1(\{b_i\}) = \mu$ and $m_1(A - \{b_i\}) = 1 - \mu$.. If another heuristic predicts the probability of taking b_j is υ, we get another basic probability assignment: $m_2(\{b_j\}) = \upsilon$ and $m_2(A - \{b_j\}) = 1 - \upsilon$.

Let m_1 and m_2 be two basic probability assignments. The Dempster-Shafer algorithm computes a new combined assignment, denoted $m_1 \oplus m_2$, that combines the evidence from both assignments. For a subset B of A:

$$m_1 \oplus m_2 (B) = \frac{\sum m_1(X)m_2(Y)}{\sum m_1(U)m_2(W)}$$

where X and Y run over all subsets of A whose intersection is B and U and W are subsets of A with at least one element in common. To continue the example from above, when $b_i = b_j$ (i.e. the two heuristics predict the same outcome) only the subsets $\{b_i\}$ and $A - \{b_i\}$ have non-zero basic probabilities because all other subsets, S, have $m_1(S)$ and $m_2(S)$ equal to zero. To find the combined basic probability, notice that $m_1(\{b_i\})m_2(\{b_i\})$ produces $\mu\upsilon$, and for all other subsets X and Y, if their intersection is $\{b_1\}$, then $m_1(X)m_2(Y)$ is zero. Furthermore, $m_1(A - \{b_i\})\ m_2(A - \{b_i\}) = (1 - \mu)(1 - \upsilon)$, so:

$$m_1 \oplus m_2 (\{b_i\}) = \frac{\mu\upsilon}{\mu\upsilon+(1-\mu)(1-\upsilon)}$$

$$m_1 \oplus m_2(A - \{b_i\}) = \frac{(1-\mu)(1-\upsilon)}{\mu\upsilon+(1-\mu)(1-\upsilon)}$$

In this case, $m_1 \oplus m_2(\{b_i\}) \geq m_1(\{b_i\})$ if and only if $m_2(\{b_i\}) \geq 0.5$ and $m_1 \oplus m_2(\{b_i\}) \geq m_2(\{b_i\})$ if and only if $m_1(\{b_i\}) \geq 0.5$. This shows that an estimation that b_i occurs less than half of the time lowers the probability of another prediction of the same outcome.

Consider the case when $b_i \neq b_j$ (the heuristics predict different outcomes). If $k = 2$, we have the same case as $b_i = b_j'$ by using $b_j' = A - \{b_j\}$. If $2 < k$:

$$m_1 \oplus m_2(\{b_i\}) = \frac{\mu(1-\upsilon)}{1-\mu\upsilon}$$

$$m_1 \oplus m_2(\{b_j\}) = \frac{\upsilon(1-\mu)}{1-\mu\upsilon}$$

$$m_1 \oplus m_2(A - \{b_i, b_j\}) = \frac{(1-\mu)(1-\upsilon)}{1-\mu\upsilon}$$

In this case, $m_1 \oplus m_2(\{b_i\}) \geq m_1(\{b_i\})$ if and only if $m_1(\{b_i\}) = 1$ or 0. This shows that a contradictory prediction lowers the probability unless one prediction is certain (1 or 0).

As a concrete example, suppose $b \rightarrow \{b_1, b_2\}$ initially (in the absence of a prediction[2]) has an equal probability of branching to b_1 and b_2 ($m_1(\{b_1\}) = m_1(\{b_2\}) = 0.5$). If a heuristic predicts that $b \rightarrow b_1$ occurs 70% of the time ($m_2(\{b_1\}) = 0.7$ and $m_2(\{b_2\}) = 0.3$), the combined probabilities are:

$$m_1 \oplus m_2(\{b_1\}) = \frac{0.5\times0.7}{0.5\times0.7+0.5\times0.3} = 0.7$$

$$m_1 \oplus m_2(\{b_2\}) = \frac{0.5\times0.3}{0.5\times0.7+0.5\times0.3} = 0.3$$

Now suppose another heuristic estimates that $b \rightarrow b_1$ is taken 60% of the time ($m_3(\{b_1\}) = 0.6$ and $m_3(\{b_2\}) = 0.4$). The estimate then becomes:

$$m_1 \oplus m_2 \oplus m_3(\{b_1\}) = \frac{0.7\times0.6}{0.7\times0.6+0.3\times0.4} = 0.778$$

$$m_1 \oplus m_2 \oplus m_3(\{b_2\}) = \frac{0.3\times0.4}{0.7\times0.6+0.3\times0.4} = 0.222$$

The second heuristic increased the probability that b_1 is taken from 0.7 to 0.778. This process can be repeated, in any order, to incorporate other heuristics as the operator $\oplus$ is associative.

Algorithm 1 computes the probability for two-way branches by combining predictions from all applicable heuristics. For multiway (>2) branches, it assigns equal probability to each outcome since no heuristics predicts these branches. If heuristics are developed for multiway branches, the algorithm can use the general Dempster-Shafer algorithm (omitted here) to combine the basic branch probabilities.

A similar algorithm can also combine the probabilities from dynamic profiles. A common way to combine these profiles is to add counts for each block, which weights a profile in proportion to its execution length. By first converting counts into predictions of branch probabilities, Dempster-Shafer theory can combine profiles without this bias.

---

[2] Theoreticaly, the absence of a prediction should assign the "unknown" probabilities to a hypotheses:

$m_1(\{b_1, b_2\}) = 1,\ m_1(\{b_1\}) = 0,\ m_1(\{b_2\}) = 0.$

But this treatment makes propogation of branch frequencies more difficult. Instead, we reasonably assume equal probabilities when a prediction is unavailable.

```
Input: Control-flow graph G for function. Each node is a basic
block and an edge bi→bj represents a branch from block bi to bj.
For each heuristic H, the predicted taken probability is
taken_prob(H), and the not taken probability is not_taken_prob(H).
---
Output: Assignment of a branch probability prob(bi→bj) to each
edge bi→bj in G.
---
Process:
foreach block b with n successors
        and m back edge successors (m ≤ n) do
  if n == 0 then          // No successors
    continue;
  else if b calls exit() then
    foreach successor s of b do
      prob(b→s) = 0.0;  // Never reach successors
  else if m > 0 and m < n then
    // Both back edges and exit edges
    foreach back edge successor s of b do
      prob(b→s) = taken_prob(LBH) / m;
    foreach exit edge successor s of b do
      prob(b→s) = not_taken_prob(LBH) / (n - m);
  else if m > 0 or n ≠ 2 then
    // Only back edges, or not a 2-way branch
    foreach successor s of b do
      prob(b→s) = 1.0 / n;
  else          // None of the above
    let s1 and s2 be the successors of b
    prob(b→s1) = prob(b→s2) = 0.5
    foreach heuristic H that applies do
      Assume H predicts (b→s1) taken,
        and (b→s2) not taken
      d = prob(b→s1) × taken_prob(H)
        + prob(b→s2) × not_taken_prob(H);
      prob(b→s1) = prob(b→s1) × taken_prob(H) / d;
      prob(b→s2) = prob(b→s2) × not_taken_prob(H) / d;
```

Algorithm 1. *Calculate branch probabilities.*

4. Local Block and Edge Frequency

After computing branch probabilities, we calculate *intra-procedural (or local) basic block and CFG edge frequencies* by propagating branch probabilities over a single procedure's control-flow graph. The frequency of a branch $b_i \rightarrow b_j$, is the execution frequency of block b_i times the branch probability of $b_i \rightarrow b_j$. The execution frequency of block b_i is the sum of the frequencies of incoming edges. Let $bfreq(b_i)$ be the execution frequency of block b_i and $freq(b_i \rightarrow b_j)$ be the edge frequency of $b_i \rightarrow b_j$. The following flow equations state this relation precisely [Ramamoorthy-65]:

$$bfreq(b_i) = 1 \qquad (\text{if } b_i \text{ is the entry block})$$

$$bfreq(b_i) = \sum_{b_p \in pred(b_i)} freq(b_p \rightarrow b_i) \qquad (\text{otherwise})$$

$$freq(b_i \rightarrow b_j) = bfreq(b_i)\, prob(b_i \rightarrow b_j)$$

For a flow graph without cycles, these equations can be solved top-down in a single pass. When a graph contains cycles, these equations are mutually recursive and must be solved by finding a least fixed point. An algorithm described in [Forman-81] solves the equations, but it is slow and, more seriously, does not handle non-terminating flow graphs. For example, the following loop:

```
for (;;) { ...;
do_a_few_times_then_exit(); }
```

results in a flow graph that appears to loop forever, which Forman's algorithm cannot solve. Below, we present an elimination algorithm that is fast and handles non-terminating, reducible flow graphs.

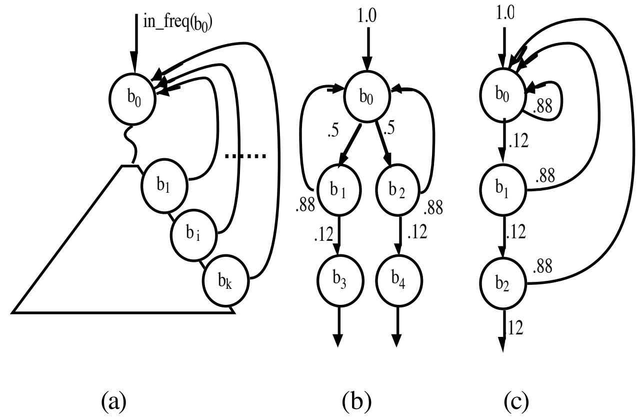


Figure 1. *Loops with single loop entries.*

Consider first a structured flow graph (e.g. Figure 1a) in which a single loop head dominates a loop body (this could be a single loop or nested loops that share the same head). In the flow graph of Figure 1a, block b_0 is the loop head, $in\_freq(b_0)$ is the total frequency of the edges entering b_0, and blocks $b_1, b_2, ..., b_k$ contain back edges leading to b_0. Since b_0 is the only entry to the loop, we can propagate $bfreq(b_0)$, without recursion, to $b_1, b_2, ..., b_k$, and obtain $bfreq(b_i) = bfreq(b_0) r_i$, for $i = 1, ..., k$, where r_i is the probability that control passes from b_0 to b_i. From this, we find:

$$\begin{aligned} bfreq(b_0) &= in\_freq(b_0) + \sum_{i=1}^{k} freq(b_i \rightarrow b_0) \\ &= in\_freq(b_0) + \sum_{i=1}^{k} (bfreq(b_i) prob(b_i \rightarrow b_0)) \\ &= in\_freq(b_0) + \sum_{i=1}^{k} (bfreq(b_0) r_i prob(b_i \rightarrow b_0)) \\ &= in\_freq(b_0) + bfreq(b_0) \sum_{i=1}^{k} (r_i prob(b_i \rightarrow b_0)) \end{aligned}$$

Let

$$p_i = r_i\, prob(b_i \rightarrow b_0)$$

$$cp(b_0) = \sum_{i=1}^{k} (r_i prob(b_i \rightarrow b_0)) = \sum_{i=1}^{k} p_i,$$

and if $0 \le cp(b_0) < 1$, we have:

$$\begin{aligned} bfreq(b_0) &= in\_freq(b_0) + bfreq(b_0)\, cp(b_0) \\ &= \frac{in\_freq(b_0)}{1 - cp(b_0)} \end{aligned}$$

In this derivation, p_i is the probability that control goes from b_0 to b_0 through block b_i, and $cp(b_0)$ is the probability along all paths that control goes from b_0 to b_0. We call $cp(b_0)$ the *cyclic probability* of block b_0. To find the cyclic probability, first assume b_0 executes once and propagate branch probabilities from b_0 to all back edges leading to b_0, and sum the probabilities of the back edges.

Applying this formula to the examples in Figure 1, we have:

$$bfreq(b_0) = \frac{1}{1\text{-}0.5\times0.88\text{-}0.5\times0.88} = 8.33$$

for the flow graph in Figure 1b, and:

$$bfreq(b_0) = \frac{1}{1\text{-}0.88\text{-}0.88\times0.12\text{-}0.88\times0.12\times0.12} = 578.70$$

for the flow graph in Figure 1c.[3]

For a loop that terminates, $cp(b_0) < 1$. If the loop appears not to terminate, we could have $cp(b_0) \geq 1$. When this happens, we can easily set $cp(b_0)$ to a value (less than 1) that represents the maximum cyclic probability.

Now consider a flow graph with two loop heads, one of which is nested in the other (Figure 2). For this flow graph, we first find the cyclic probability of the inner loop and then treat the outer loop the same manner as a single-level loop, except that we use the formula

$$bfreq(b_{inner}) = \frac{in\_freq(b_{inner})}{1\text{-}cp(b_{inner})}$$

to find the frequency of the inner loop head, where b_{inner} is the head block of the inner loop structure and $cp(b_{inner})$ is b_{inner}'s cyclic probability.

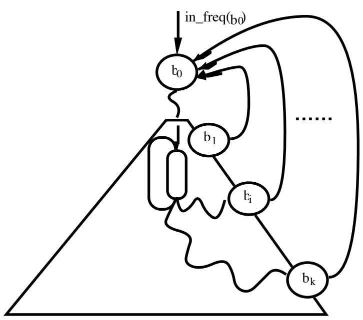


Figure 2*. A two-head nested loop structure.*

If a flow graph is reducible, every loop head dominates the blocks in the loop. The method described above works for these flow graphs. We visit the inner-most loop first and use the cyclic probabilities of inner loops to compute frequencies for the outer loops.

Algorithm 2 calculates the edge frequency of control-flow graph edges and the execution frequency of blocks. Although it assumes that the flow graph is reducible, the algorithm terminates for non-reducible flow graphs, although the resulting estimates may be less accurate.

Input: Control-flow graph G for function, in which each node is a basic block and each edge $b_i \rightarrow b_j$ represents a branch from block b_i to block b_j. Each edge $b_i \rightarrow b_j$ has branch probability $prob(b_i \rightarrow b_j)$.

Output: Assignments of frequency $freq(b_i \rightarrow b_j)$ to edge $b_i \rightarrow b_j$ and $bfreq(b)$ to block b.

```
Subroutine: propagate_freq(b, head)
  if b has been visited then
    return;
  // 1. find bfreq(b)
  if b == head then
    bfreq(b) = 1;
  else
    foreach predecessor bp of b do
      if bp is not visited and (bp->b) is not a back edge then
        return;
    bfreq(b) = 0;
    cyclic_probability = 0;
    foreach predecessor bp of b do
      if (bp->b) is back edge to loop head b then
        cyclic_probability += back_edge_prob(bp->b);
      else
        bfreq(b) += freq(bp->b);
    if ( cyclic_probability > 1 - epsilon ) then
      cyclic_probability = 1 - epsilon;
    bfreq(b) = bfreq(b) / (1-cyclic_probability);
  // 2. calculate the frequencies of b's out edges
  mark b as visited
  foreach successor bi of b do
    freq(b->bi) = prob(b->bi) × bfreq(b);
    // update back_edge_prob(b->bi) so it
    // can be used by outer loops to calculate
    // cyclic_probability of inner loops
    if bi == head then
      back_edge_prob(b->bi) = prob(b->bi) × bfreq(b);
  // 3. propagate to successor blocks
  foreach successor bi of b do
    if (b->bi) is not back edge then
      propagate_freq(bi, head);
```

```
Process:
  foreach edge do
    back_edge_prob(edge ) = prob(edge );
  foreach loop from inner-most to out-most do
    let head be the head block of the loop
    mark all blocks reachable from head as not visited
    and mark all other blocks as visited.
    propagate_freq(head, head);
```

Algorithm 2*. Compute block and edge frequencies.*

5. Function and Function Call Frequency

The local block frequencies enable us to calculate the *local frequency of calls* on other functions. We then propagated these call frequencies along call-graph edges to compute *inter-procedural (or global) function invocation frequencies*. Finally, we obtain *global basic block and edge frequencies* by multiplying each a local frequencies by its procedure's global invocation frequency.

The *local call frequency* is the number of times that *f* calls *g*, assuming one invocation of *f*. This information is readily available from the block frequencies computed previously. If function *f* calls *g* in blocks b_1, ... b_k, the local call frequency of *f* calling *g* is the sum of the execution frequency of these blocks, The *global call frequency* of function *f* calling *g* is the number of times that *f* calls *g* during all invocations of *f*, which is just the product of the local call frequency times the global invocation frequency of *f*.

Computing global call frequencies from local call frequencies is similar to propagating branch probabilities in a flow graph. Assume $cfreq(f)$ is the number of times that

[3] In general, for k properly nested loops that share the same loop head, in which the back edge probabilities are p and the exit edge probabilities are 1 - p, we can verify that

$$\frac{1}{1\text{-}p^*\sum_{i=0}^{k-1}(1\text{-}p)^i} = \left(\frac{1}{1\text{-}p}\right)^k$$

function f is called, $lfreq(f,g)$ is the local frequency of f calling g, and $gfreq(f,g)$ is the global frequency of f calling g. The flow equations relating local and global call frequencies are:

$$cfreq(f) = 1 \qquad (f \text{ is main function})$$

$$cfreq(f) = \sum_{p \in pred(f)} freq(p,f) \qquad (\text{otherwise})$$

$$gfreq(f,g) = lfreq(f,g)\, cfreq(f)$$

A call graph is not reducible when a recursive cycle in the graph can be entered at several points. To handle these cycles, we modify the edge frequency propagation algorithm by treating each node that is the target of a back edge as a loop head and, when calculating the cyclic probability for a loop head that is not the entry function, not using its descendants' cyclic probabilities. The modified algorithm (Algorithm 3) propagates call frequencies.

6. Global Block and Edge Frequencies

To obtain global block and edge frequencies, multiply each local block or edge frequency by the execution frequency of the function that contains the block or edge.

7. Experimental Evaluation

To measure the effectiveness of our static profile techniques, we compared static and dynamic profiles of the SPECint92 C benchmarks and a several simple Unix applications. We ran the experiments on a Sequent S2000/750 system with i486 processors and the Sequent DYNIX/ptx C compiler Version 2.1, which supports dynamic function call profiling and static branch and function call frequency prediction. We added dynamic branch profiling to the compiler and added the heuristics to its static analysis.

To measure the effectiveness of our techniques, we used Wall's weighted and unweighted matching method [Wall-91]. Consider block frequencies as an example. Assume that a program contains N blocks and we have a list of blocks from static analysis and another from dynamic profiling, both sorted by frequency. To compute the weighted match score of the top m static blocks, count the number of these blocks that occur in the top m dynamic blocks. If k ($k \leq m$) blocks occur, the ratio of sum of the static frequencies of these k blocks to the sum of the dynamic frequencies of the top m blocks is the *weighted match score*. The ratio k/m is the *unweighted match score*. A perfect match will has a score of one. A random frequency estimate will have an average score m/N. The closer to one, the better the heuristic estimate.

Below, we first present our results for the SPECint92 C benchmarks. In this experiment, we calculate matching scores for the top 10%, 20%, 30%, 40%, and 50% of entries for the following six analyses: block execution frequency, edge frequency, function invocation frequency, function call frequency, global block execution frequency, and global edge frequency. The dynamic profiles came from the standard data sets supplied with the benchmarks.

Table 2 summarizes the unweighted and weighted matching scores for local block frequencies predictions. It

Input: A call graph, each node of which is a procedure and each edge $f_i \rightarrow f_j$ represents a call from function f_i to f_j. Edge $f_i \rightarrow f_j$ has local call frequency $lfreq(f_i \rightarrow f_j)$.

Output: Assignments of global function call frequency $gfreq(f_i \rightarrow f_j)$ to edge $f_i \rightarrow f_j$ and invocation frequency $cfreq(f)$ to f.

```
Subroutine: propagate_call_freq(f, head, final)
  if f has been visited then
    return;
  // 1. find cfreq(f)
  foreach predecessor fp of f do
    if fp is not visited and (fp→f) is not a back edge then
      return;
  if f == head then
    cfreq(f) = 1;
  else
    cfreq(f) = 0;
  cyclic_probability = 0;
  foreach predecessor fp of f do
    if final and (fp→f) is a back edge then
      cyclic_probability += back_edge_prob(fp→f);
    else if (fp→f) is not a back edge
      cfreq(f) += gfreq(fp→f);
  if ( cyclic_probability > 1 - epsilon ) then
    cyclic_probability = 1 - epsilon;
  cfreq(f) = cfreq(f) / (1-cyclic_probability);
  // 2. calculate global call frequencies for f's out edges
  mark f as visited;
  foreach successor fi of f do
    gfreq(f→fi) = lfreq(f→fi) × cfreq(f);
    // update back_edge_prob(f→fi) so it can be
    // used by the outer-most loop to calculate
    // cyclic_probability of inner loops
    if fi == head and not final then
      back_edge_prob(f→fi) = lfreq(f→fi) × cfreq(f);
  // 3. propagate to successor nodes
  foreach successor fi of f do
    if (f→fi) is not a back edge then
      propagate_call_freq(fi, head, final)
```

```
Process:
  foreach edge do
      back_edge_prob(edge ) = lfreq(edge );
  foreach function f in reverse depth-first order do
    if f is a loop head then
      mark all nodes reachable from f as not visited
        and all other as visited.
      propagate_call_freq(f, f, false)
  mark all nodes reachable from entry func as not
      visited and others as visited.
  propagate_call_freq(entry func, entry func, true)
```

***Algorithm 3**. Calculate function call and invocation frequencies.*

shows that our technique identified 79% of the top 20% blocks that accounted for 82% of their frequencies. In most tables below, unweighted scores are lower than weighted scores, which indicates that our algorithms were successful at identifying the heavily-executed regions.

Bench-marks	10%		20%		30%		40%		50%	
	U	W	U	W	U	W	U	W	U	W
espresso	.71	.71	.76	.78	.79	.83	.81	.83	.83	.88
li	.85	.86	.84	.84	.82	.82	.83	.82	.84	.86
eqntott	.84	.85	.86	.87	.88	.88	.88	.88	.87	.91
compress	.79	.89	.77	.87	.82	.88	.85	.88	.83	.91
sc	.80	.76	.78	.78	.82	.81	.85	.81	.87	.85
gcc	.73	.76	.75	.78	.76	.80	.78	.80	.81	.87
Geo-mean	.79	.80	.79	.82	.81	.84	.83	.84	.84	.88

***Table 2**. Scores of SPEC92 local block frequency. U is fraction and W is weighted fraction.*

Table 3 shows scores for the local edge frequencies. These scores are lower than the block frequencies, which indicates that our algorithms were better at identifying heavily executed blocks than edges. For the 20% most frequently executed edges, our technique identified about 64% of the most frequently executed edges, which accounted for 69% of their frequencies.

Benchmarks	10%		20%		30%		40%		50%	
	U	W	U	W	U	W	U	W	U	W
espresso	.57	.61	.64	.68	.67	.75	.71	.80	.76	.83
li	.61	.67	.64	.68	.72	.73	.74	.76	.78	.81
eqntott	.73	.69	.77	.80	.78	.83	.81	.88	.84	.89
compress	.65	.74	.72	.80	.71	.81	.74	.84	.77	.87
sc	.50	.53	.57	.60	.62	.65	.67	.69	.72	.74
gcc	.47	.54	.52	.60	.58	.65	.65	.72	.71	.77
Geo-mean	.58	.62	.64	.69	.68	.73	.72	.78	.76	.82

Table 3. *Scores of SPEC92 local edge frequency.*

Tables 4 and 5 summarize the scores for function invocation and call frequencies. For the 20% most frequently executed functions, our estimates identified 52% of these invocations, which accounted for 85% of their frequencies. For the 20% most frequent function calls, our estimates identified only 39% of the calls, but these calls accounted for over 72% of their frequency.

Benchmarks	10%		20%		30%		40%		50%	
	U	W	U	W	U	W	U	W	U	W
espresso	.57	.66	.68	.83	.70	.97	.77	.98	.77	.99
li	.39	.76	.52	.92	.67	.96	.83	.97	.89	.98
eqntott	.43	.98	.46	.99	.53	1.0	.48	1.0	.65	1.0
compress	1.0	1.0	.50	1.0	.80	1.0	.86	1.0	.88	1.0
sc	.50	.57	.50	.91	.44	.91	.52	.97	.52	.99
gcc	.35	.32	.47	.53	.52	.67	.63	.76	.71	.81
Geo-mean	.51	.67	.52	.85	.69	.91	.67	.94	.73	.96

Table 4. *Scores of SPEC92 function invocation frequency.*

Benchmarks	10%		20%		30%		40%		50%	
	U	W	U	W	U	W	U	W	U	W
espresso	.53	.63	.54	.76	.65	.87	.66	.89	.74	.95
li	.31	.81	.34	.84	.49	.90	.65	.91	.86	.94
eqntott	.30	.97	.37	.98	.45	.99	.47	1.0	.47	1.0
compress	1.0	1.0	.75	1.0	.67	1.0	.71	1.0	.78	1.0
sc	.32	.41	.26	.70	.33	.78	.33	.79	.51	.99
gcc	.18	.16	.26	.32	.33	.46	.42	.57	.50	.66
Geo-mean	.38	.57	.39	.72	.47	.81	.52	.85	.63	.91

Table 5. *Scores of SPEC92 global function call frequency.*

Tables 6 and 7 summarize the scores for global block and edge frequencies. For the 20% most frequently executed blocks, our estimates identified 56% of the blocks that account for 77% of their frequencies. By contrast, Wall's best static block frequency estimate achieved a weighted score of about 50% for the top 25% blocks [Wall-91]. Our results are a significant improvement. For the 20% most frequently executed edges, our estimates identified 49% of the edges that account for 78% of their frequencies.

Benchmarks	10%		20%		30%		40%		50%	
	U	W	U	W	U	W	U	W	U	W
espresso	.53	.53	.66	.74	.70	.92	.79	.98	.79	1.0
li	.44	.75	.48	.84	.52	.94	.66	.96	.73	.97
eqntott	.66	.30	.69	.98	.56	1.0	.60	1.0	.60	1.0
compress	.54	.65	.67	.78	.66	.90	.73	.97	.71	.98
sc	.54	.81	.54	.90	.56	.96	.56	.99	.62	.99
gcc	.26	.27	.38	.50	.46	.64	.55	.78	.65	.86
Geo-mean	.48	.51	.56	.77	.57	.88	.64	.94	.68	.97

Table 6. *Scores of SPEC92 global block frequency.*

Benchmarks	10%		20%		30%		40%		50%	
	U	W	U	W	U	W	U	W	U	W
espresso	.50	.55	.61	.75	.69	.93	.73	.98	.80	.99
li	.35	.76	.41	.85	.61	.94	.72	.95	.78	.97
eqntott	.63	.30	.60	.99	.57	1.0	.54	1.0	.68	1.0
compress	.47	.55	.63	.82	.67	.91	.64	.94	.71	.98
sc	.40	.77	.43	.91	.45	.94	.56	.98	.68	1.0
gcc	.26	.28	.34	.49	.44	.67	.58	.78	.69	.86
Geo-mean	.42	.50	.49	.78	.56	.89	.62	.94	.72	.97

Table 7. *Scores of SPEC92 global edge frequency.*

Next we present our results for several Unix commands. To collect the dynamic profiles, each command was run to exercise all of its command line options. Table 8 summarizes the weighted scores for the top 20% of block execution frequencies, edge frequencies, function invocation frequencies, function call frequencies, global block execution frequencies, and global edge frequencies. The average scores are around 80% for the top 20%.

Benchmarks	LBK	LBR	FI	FC	GBK	GBR
cal	.49	.55	1.0	1.0	.80	.80
cmp	.86	.93	0.0	1.0	.86	.93
ed	.74	.65	.96	.31	.82	.81
grep	.66	.52	1.0	1.0	.86	.92
factor	.89	.87	0.0	1.0	.94	.94
pack	.90	.77	1.0	1.0	.42	.43
split	.99	.99	1.0	0.0	.99	.99
sum	.69	.71	1.0	0.0	.69	.71
tsort	.77	.76	1.0	1.0	.87	.81
uniq	.81	.73	1.0	1.0	.39	.45
wc	.92	.55	1.0	1.0	.95	.91
Average	.79	.73	.82	.76	.78	.79

Table 8. *Scores for Unix commands (top 20%): LBK-local block, LBR-local branch, FI-function invocation, FC-function call, GBK-global block, GBR-global branch.*

8. Conclusion and Future Work

We have presented an algorithm that statically estimates program profiles in three steps. First, we estimate branch probabilities with a new combination of Ball and Larus's branch prediction heuristics and the Dempster-Shafer theory. Next, we propagate branch probabilities along each procedure's control-flow graph to obtain local block and edge frequencies. Finally, we use these local estimates to compute function call and invocation frequencies. With the function invocation frequencies, we can then obtain global block and edge frequencies. Table 9 summarizes the weighted scores for the top 20% of profiles from experiments with the SPECint92 Benchmarks and several Unix commands.

Profiles	SPEC 92 Benchmarks	Unix Commands
Local block	82%	79%
Local edge	69%	73%
Funct. invocation	85%	82%
Function call	72%	76%
Global block	77%	78%
Global edge	78%	79%

***Table 9.** Average scores of static profiles. (20%)*

Several improvements might increase the accuracy of our techniques. First, Ball and Larus's heuristics apply to two-way branches, so we assume equal probabilities for multi-way branches. This assumption may affect programs that heavily use switch statements (e.g., gcc). We tried rewriting some multiway branches into a series of conditional branches, but it did not improve the accuracy because the new blocks were too small for the heuristics. New heuristics for multiway branches are needed.

Second, current heuristics use only local information about a block and its immediate successors. This makes the heuristics simple and fast, but looking deeper into the CFG might improve the accuracy.

Third, our experiments used the hit rates reported in Ball and Larus' paper. Their experiments ran on a different processor (MIPS R2000) and used C, C++, and Fortran programs. Since our programs were all written in C, we might improve accuracy by using hit rates from C programs compiled by the same compiler.

Finally, indirect function calls are difficult to analyze statically. Currently our compiler exploits some knowledge about the run-time library—for example that `qsort` calls its last argument. Calls through pointers in user code are not analyzed. This reduces the accuracy of the function call and execution frequencies for programs, such as gcc, that indirectly invoke many functions. Techniques that find the functions bound to a function parameter (e.g., [Hall-92]) are too imprecise to determine local function call frequencies for a set of possible functions. Furthermore, in gcc (and object-oriented programs), many functions are called indirectly through tables. New analysis techniques are needed.

Acknowledgments

Thomas Ball, James Bash, Dorsey Drane, Mark Hill, Xiaoning Ling, Vivek Sarkar, Guri Sohi, Gary Tracy, T. Vijaykumar, and David Wall provided many helpful comments on the paper. The second author was supported in part by NSF NYI Award CCR-9357779, with matching funds from Digital Equipment Corp., and NSF Grants CCR-9101035 and MIPS-9225097.

References

[Ball-92] Ball, Thomas, and James R. Larus, "Optimally Profiling and Tracing Programs," *Conference Record of the Nineteenth ACM Symposium on Principles of Programming Languages* (Jan., 1992) pp. 59-70.

[Ball-93] Ball, Thomas, and James R. Larus, "Branch Prediction for Free," *Proceedings of ACM SIGPLAN'93 Conference on Programming Language Design and Implementation* (June, 1993) pp. 300-313.

[Chang-91] Chang, P.P., S.A. Mahlhe, and W. W. Hwu,, "Using profile information to assist classic code optimizations." *Software Practice and Experience*, 21, 12 (Dec., 1991) pp. 1301-1321.

[Chen-92] Chen, W., R. Bringmann, S. Mahlke, S. Anik, T. Kiyohara, N. Warter, D. Lavery, W.-M. Hwu, R. Hank, J. Gyllenhaal, "Using Profile Information to Assist Advanced Compiler Optimization and Scheduling," *Languages and Compilers for Parallel Computing*, Springer-Verlag, 1993, pp. 31–48.

[Fisher-81] Fisher, Joseph A, "Trace Scheduling: A Technique for Global Microcode Compaction." *IEEE Trans. Computers*, C-30, 7 (July, 1981) pp. 478-490.

[Fisher-92] Fisher, Joseph A. and S. M. Freudenberger, "Predicting Conditional Branch Directions From Previous Runs of a Program," *Fifth International Conference on Architectural Support for Programming Language and Operating Systems* (October 1992) pp. 85-95.

[Forman-81] Forman, Tra R., "On the Time Overhead of Counters and Traversal Markers," *Proceedings of the 5th International Conference on Software Engineering (*March 1981) pp. 164-169.

[Graham-83] Graham, S.L, P.B. Kessler, and M.K. McKusick, "An Execution Profiler for Modular Programs," *Software—Practice and Experience*, 13 (1983) pp. 671-685.

[Hall-92] Hall, M.W. and Ken Kennedy, "Efficient Call Graph Analysis," *ACM Letters on Programming Languages and Systems*, Vol. 1, No. 3 (Sept. 1992) pp 227-242.

[Hank-93] Hank, R.E., S.A. Mahlke, R.A. Bringmann, Gyllehaal, and W.W. Hwu, "Superblock Formation Using Static Program Analysis," *Micro-26* (Dec. 1993) pp. 247-255.

[Hicky-88] Hickey, T. and Cohen, J., "Automating Program Analysis," *JACM*, 35, 1, 1988, pp. 185-220.

[Huelsbergen-94] Huelsbergen, L., Larus, J., and Aiken, A., "Using the Run-Time Sizes of Data Structures to Guide Parallel-Thread Creation," *Proceedings of the ACM Conference on Lisp and Functional Programming* (June, 1994) pp. 79-90.

[Hwu-89] Hwu, W.W. and P.P. Chang, "Achieving High Instruction Cache Performance with a Optimizing Compiler," *Proceedings 16th International Symposium on Computer Architecture*, Jerusalem, Israel (May 1989) pp. 242-250.

[McFarling-89] McFarling, S., "Program Optimization for Instruction Caches," *Third International Conference on Architectural Support for Programming Language and Operating Systems* (April 1989) pp. 183-191.

[Pettis-90] Pettis, Karl, and Robert C. Hansen, "Profile Guided Code Positioning." *Proceedings of ACM SIGPLAN'90 Conference on Programming Language Design and Implementation* (June, 1990) pp. 16-27.

[Ramamoorthy-65] Ramamoorthy, C.V., "Discrete Markov Analysis of Computer Programs," In *ACM Proceedings 20th National Conference* (August 1965) pp. 386-391.

[Sarkar-89] Sarkar, V., "Determining Average Program Execution Times and their Variance," *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation* (June 1989) pp. 298-312.

[Shafer-76] Shafer, G., *A Mathematical Theory of Evidence*. Princeton University Press, 1976.

[Spec92] SPEC Cint92, Release V1.1.

[Wagner-94] Wagner, T.A., V. Maverick, S.L. Graham, and M.A. Harrison, "Accurate Static Estimators for Program Optimization," *Proceedings of ACM SIGPLAN'94 Conference on Programming Language Design and Implementation,* (June 1994) pp. 85-96.

[Wall-91] Wall, David W., "Predicting Program Behavior Using Real or Estimated Profiles," *Proceedings of ACM SIGPLAN'91 Conference on Programming Language Design and Implementation,* Toronto, Ontario, Canada (June, 1991) pp. 59-70.

[Wu-92] Wu, Y. "Ordering Functions for Improving Memory Reference Locality in Shared Memory Multiprocessor Systems", *25th Annual International Symposium on Microarchitecture,* Dec. 1992.

Appendix A

This appendix contains a small, complete example that illustrates the algorithms in this paper. Consider the `atoi` function in Figure 3 as an example. Figure 4 shows its flow graph. It contains two non-loop branches and one loop branch. Five heuristics apply to the first non-loop branch $b_0 \rightarrow \{b_1, b_5\}$, because b_0 performs a comparison of a pointer with NULL (PH), b_1 uses s without first defining it

(GH) and stores to *val (SH), and b_5 contains a call (CH) and a return (RH). The combined probability of 0.95 for $b_0 \rightarrow b_1$ is much higher than the probability estimated by any of the five heuristics.

For the second non-loop branch $b_1 \rightarrow \{b_2, b_4\}$, both the Loop Header (LHH) and Return heuristics (RH) apply. Table 10 summarizes the probabilities.

```
/*     permutation program: find
   all the permutations for {1, 2, ...
   max} */
main(argc, argv)
   int argc;
   char *argv[];
{
   int max;
   char *a;
   atoi(argv[1], &max);
   a = (char *) malloc(max);
   permute(a,0,max);
}

permute(a,n,max)
   char *a;
   int n,max;
{
   if (n == max)
      report_one(a,max);
   else
      permute_next_pos
   (a,n,max);
}

permute_next_pos(a,n,max)
   char *a;
   int n,max;
{
   int i;
   for (i=0;i < max;i++) {
      if ( ! in_prefix(i, a, n) ) {
         a[n] = i;
         permute(a,n+1,max);
} } }
```

```
report_one(a,max)
   char *a;
   int max;
{
   int i;
   for (i=0;i < max;i++)
      printf("%c ",a[i]+'0');
   putchar('\n');
}

int in_prefix(i, a, n)
   char *a;
   int i, n;
{
   int found = 0,j;
   for (j=0;j < n;j++)
      if (a[j] == i) {
         return 1;
      }
   return 0;
}

int atoi ( char *s, int *val)
{
   int i;
   if ( s != 0 ) {
      *val = 0;
      for ( ; *s; s++ )
         *val = *val * 10 + *s -
   '0';
      return 1;
   } else {
      printf ("Invalid Input!\n");
      return 0;
} }
```

Figure 3. *The permutation program.*

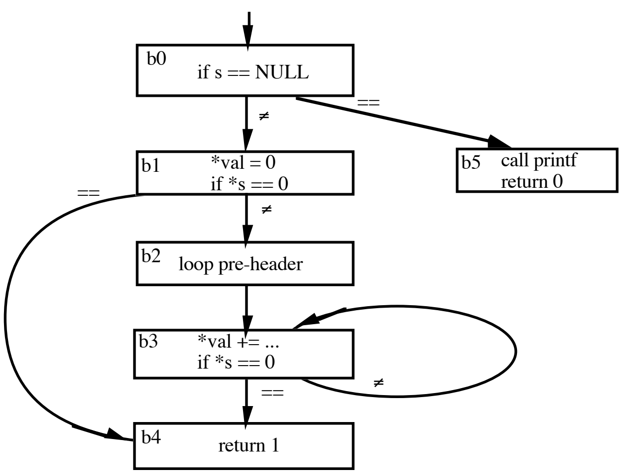


Figure 4. *Flow graph for* `atoi`.

Heuristic	$b_0 \rightarrow b_1$	$b_0 \rightarrow b_5$
CH	.78	.22
RH	.72	.28
SH	.45	.55
PH	.60	.40
GH	.62	.38
Combined	.95	.05

Table 9. *Branch probabilities for $b_0 \rightarrow \{b_1, b_5\}$.*

Heuristic	$b_1 \rightarrow b_2$	$b_1 \rightarrow b_4$
LHH	.75	.25
RH	.72	.28
Combined	.89	.12

Table 10. *Branch probabilities for $b_1 \rightarrow \{b_2, b_4\}$.*

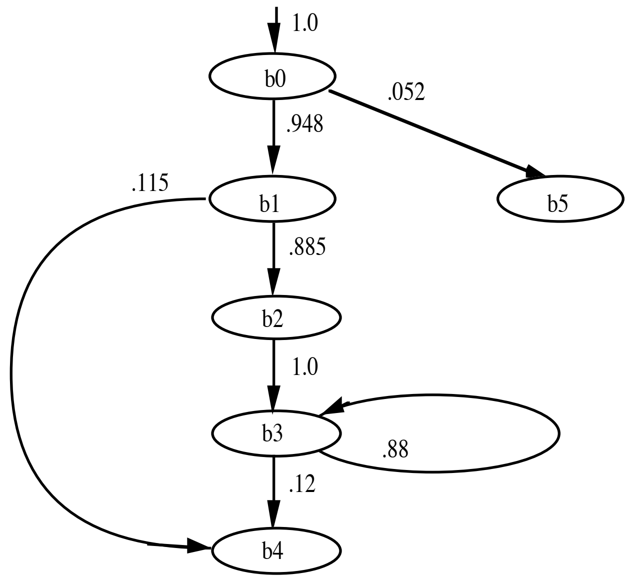


Figure 5. *Branch probabilities of* `atoi`.

The branch $b_3 \rightarrow \{b_3, b_4\}$ contains a back edge, and only the loop branch heuristic (LBH) applies, so:

$prob(b_3 \rightarrow b_3) = LBH(b_3 \rightarrow b_3) = 0.88$

$prob(b_3 \rightarrow b_4) = LBH(b_3 \rightarrow b_4) = 0.12$

Figure 5 shows the flow graph labeled with each edge's branch probabilities. As an example of Algorithm 2, consider the `atoi` function in Figure 5. For the inner loop $(b_3 \rightarrow b_3)$ the cyclic probability is the same as $prob(b_3 \rightarrow b_3)$. For the outer loop, the block frequencies and edge frequencies for the `atoi` function are calculated as follows:

$bfreq(b0) = 1$

$freq(b_0 \rightarrow b_1) = prob(b_0 \rightarrow b_1) \times 1 = 0.95$

$freq(b_0 \rightarrow b_5) = prob(b_0 \rightarrow b_5) \times 1 = 0.052$

$bfreq(b_1) = freq(b_0 \rightarrow b_1) = 0.95$

$freq(b_1 \rightarrow b_2) = prob(b_1 \rightarrow b_2) \times bfreq(b_1) = 0.89 \times 0.95 = 0.84$

$freq(b_1 \rightarrow b_4) = prob(b_1 \rightarrow b_4) \times bfreq(b_1) = 0.12 \times 0.95 = 0.110$

$bfreq(b_2) = freq(b_1 \rightarrow b_2) = 0.84$

$freq(b_2 \rightarrow b_3) = prob(b_2 \rightarrow b_3) \times bfreq(b_2) = 1 \times 0.84 = 0.84$

$$bfreq(b_3) = \frac{freq(b_2 \rightarrow b_3)}{1-prob(b_3 \rightarrow b_3)} = \frac{0.84}{1-0.88} = 6.99$$

$freq(b_3 \rightarrow b_3) = bfreq(b_3) \times 0.84 = 6.15$

$freq(b_3 \rightarrow b_4) = bfreq(b_3) \times 0.12 = 0.84$

$bfreq(b_4) = freq(b_1 \rightarrow b_4) + freq(b_3 \rightarrow b_4) = 0.11 + 0.84 = 0.95$

$bfreq(b_5) = freq(b_0 \rightarrow b_5) = 0.05$

Figure 6 shows the block and edge execution frequencies. Note that because we start the entry block with a frequency of one, the exit blocks' total frequency is also one. That is, $freq(b_0 \rightarrow b_5) + freq(b_1 \rightarrow b_4) + freq(b_3 \rightarrow b_4) = 0.05 + 0.11 + 0.84 = 1$.

The `atoi` function calls `printf` in block b_5 and block b_5's execution frequency is 0.05. So, in the call graph, `atoi` calls `printf` with a local frequency of 0.05. We can continue this process to find the local call frequencies for the permutation program (Figure 7 (a)).

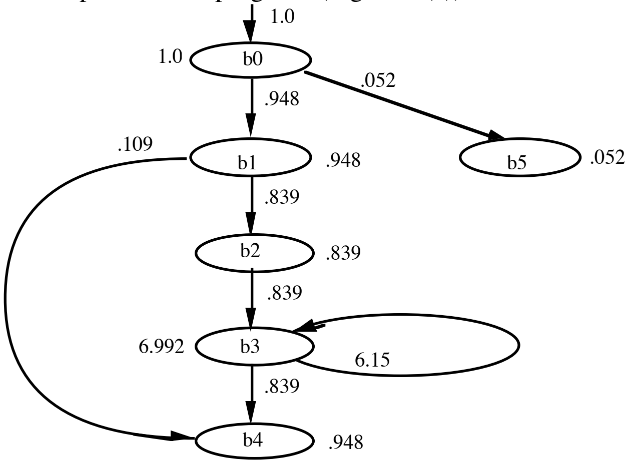
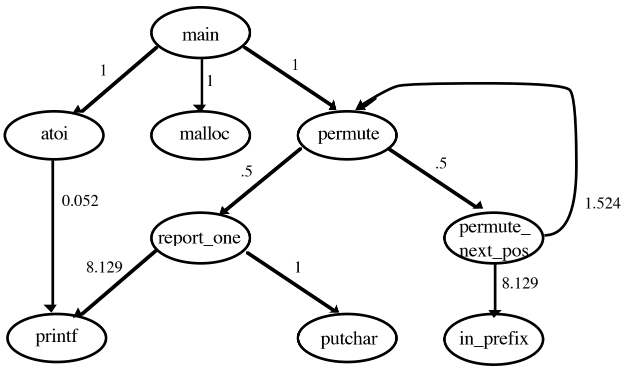
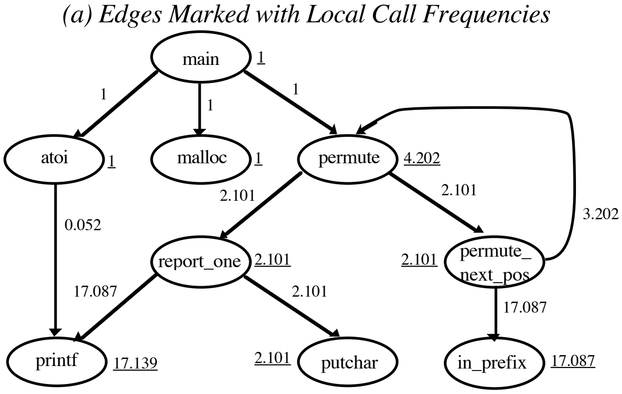


Figure 6. *Block and edge frequencies of* `atoi`



(a) Edges Marked with Local Call Frequencies



(b) Edges Marked with Local Call Frequencies

Figure 7 *Call graph of the permutation program.*

Figure 7(a) shows that the permutation program has a recursive cycle in which permute is the head. The first call to propagate_call_freq updates lfreq(permute_next_pos → permute) from 1.52 to 0.5 ×1.52 = 0.76. In the final call to propagate_call_freq, we obtain function call and invocation frequencies shown in Figure 7(b). Table 11 lists local edge frequencies in col. 3, function invocation frequencies in col. 4, and global edge frequencies in col. 5.

functions	edges	local freq.	func. invoc.. freq.	global freq.
report_one	b0→b3	.02	2.1	.05
report_one	b0→b5	.98	2.1	2.1
report_one	b1→b3	.98	2.1	2.1
report_one	b1→b1	7.2	2.1	15.0
report_one	b5→b1	.98	2.1	2.1
in_prefix	b0→b5	.02	17.1	.41
in_prefix	b0→b7	.98	17.1	16.7
in_prefix	b1→b2	.29	17.1	4.9
in_prefix	b1→b3	5.9	17.1	10.0
in_prefix	b3→b5	.70	17.1	12.0
in_prefix	b3→b1	5.2	17.1	88.0
in_prefix	b7→b1	.98	17.1	16.7
permute_next_pos	b0→b5	.02	2.1	.05
permute_next_pos	b0→b7	.98	2.1	2.1
permute_next_pos	b1→b2	1.5	2.1	3.2
permute_next_pos	b1→b3	6.6	2.1	13.9
permute_next_pos	b2→b3	1.5	2.1	3.2
permute_next_pos	b3→b5	.98	2.1	2.1
permute_next_pos	b3→b1	7.2	2.1	15.0
permute_next_pos	b7→b1	.98	2.1	2.1
atoi	b0→b1	.95	1.0	.95
atoi	b0→b5	.05	1.0	.05
atoi	b1→b4	.11	1.0	.11
atoi	b1→b2	.84	1..0	.84
atoi	b2→b3	.84	1.0	.84
atoi	b3→b3	6.2	1.0	6.2
atoi	b3→b4	.84	1.0	.84
permute	b0→b1	.50	4.2	2.1
permute	b0→b2	.50	4.20	2.1

Table 11. *Global frequencies of program.*

For example, Table 12 lists the global edge frequencies from static and dynamic profiling for the program, with input max = 6. The top 10% of the edges match, so the top 10% matching score is 1. For the top 20% of edges, however, 4 of 5 edges match and the weighted score is 0.89.

Static			Dynamic			top %	sco-res
f#	**brs**	**freqs**	**f#**	**brs**	**freqs**		
1	1→3	100.0	1	1→3	17580		
1	3→1	88.0	1	3→1	15630		
1	0→7	16.7	1	0→7	7416	10	1.0
1	7→1	16.7	1	7→1	7416		
2	1→1	15.0	3	3→1	6185	20	.89
3	3→1	15.0	3	1→3	5466		
3	1→3	13.9	1	1→2	5466		
1	3→5	12.0	2	1→1	3600	30	.92
4	3→3	6.15	3	1→2	1956		
1	1→2	4.87	3	2→3	1956	40	.95
3	1→2	3.20	1	3→5	1950		
3	2→3	3.20	3	0→7	1237	50	.98
5	0→2	2.10	5	0→2	1237		
5	0→1	2.10	3	3→5	1237	60	.97
3	0→7	2.05	3	7→1	1237		
2	0→5	2.05	2	0→5	720		
2	1→3	2.05	2	1→3	720	70	1.0
3	3→5	2.05	5	0→1	720		
2	5→1	2.05	2	5→1	720	80	1.0
3	7→1	2.05	1	0→5	6		
4	0→1	.95	4	0→1	1		
4	1→2	.84	4	1→2	1	90	1.0
4	3→4	.84	4	3→4	1		
4	2→3	.84	4	2→3	1	100	1.0

Table 12. *Edges frequency matching for program. Function key: 1 - in_prefix; 2 - report_one; 3 - permute_next_pos; 4 - atoi; 5 - permute.*