Static Branch Frequency and Program Profile Analysis

Youfeng Wu
James R. Larus

Technical Report #1248

September 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 slow 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 profile 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 por-
tions 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 invoca-
tion 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 illustrate 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 bene-
fits. For example, branch probabilities can guide instruction
generation and scheduling to reduce stalls on pipelined
processors [Fisher-81, Hank-93]. Block and function execu-
tion frequencies can identify program bottlenecks during op-
timization [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 [Petits-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 hierar-
chical program profiler (gprof), which accounted for proce-
dure calls. Sarkar [Sarkar-89] showed that a program-depen-
dence graph can reduce the number of points at which execu-
tion 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
[Hickey-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 heuris-
tics in this paper.

This paper builds on earlier work on static branch predic-
tion, most notably Ball and Larus’s heuristics and mea-
surements [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
algorithm (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% prob-
babilities, which produced roughly as accurate estima-
tions for local blocks and edges, but less accurate profile es-
timations for global block and edge frequencies and proce-
dure calls and invocations.

Wagner also presents three techniques for estimating pro-
files 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^2) 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 exam-
ple, in the following code:

\begin{verbatim}
b1: if ( condition )
b2:     statement 1;
b3: else statement 2;
\end{verbatim}
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 heuristic, 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.

<table>
<thead>
<tr>
<th>Heuristics</th>
<th>Probability of taking branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop branch (LBH)</td>
<td>88 %</td>
</tr>
<tr>
<td>Pointer (PH)</td>
<td>60 %</td>
</tr>
<tr>
<td>Call (CH)</td>
<td>78 %</td>
</tr>
<tr>
<td>Opcode (OH)</td>
<td>84 %</td>
</tr>
<tr>
<td>Loop exit (LEH)</td>
<td>80 %</td>
</tr>
<tr>
<td>Return (RH)</td>
<td>72 %</td>
</tr>
<tr>
<td>Store (SH)</td>
<td>55 %</td>
</tr>
<tr>
<td>Loop header (LHH)</td>
<td>75 %</td>
</tr>
<tr>
<td>Guard (GH)</td>
<td>62 %</td>
</tr>
</tbody>
</table>

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:

\[
\text{if } (i < 0) \text{ then } x = y; \\
\text{else } (x = i; \text{ return; })
\]

\(^1\) 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 branch 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 = {A1, A2, ..., Ak}. 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 → {b1, b2, ..., bk} has k exhaustive and mutually exclusive outcomes A = {b1, b2, ..., bk}. If a heuristic predicts the probability of taking bi is μ and the probability of not taking bi is 1 - μ, we get the following basic probability assignment: m1({bi}) = μ and m1(A - {bi}) = 1 - μ. If another heuristic predicts the probability of taking bj is v, we get another basic probability assignment: m1({bj}) = v and m1(A - {bj}) = 1 - v.

Let m1 and m2 be two basic probability assignments. The Dempster-Shafer algorithm computes a new combined assignment, denoted m1 ⊕ m2, that combines the evidence from both assignments. For a subset B of A:

\[
m1 \oplus m2 (B) = \sum_{X \subseteq B} \frac{m1(X)m2(Y)}{m1(U)m2(W)}
\]

where X and Y are 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 b1 = bi (i.e. the two heuristics predict the same outcome) only the subsets {bi} and A - {bi} have non-zero basic probabilities because all other subsets, S, have m1(S) and m2(S) equal to zero. To find the combined basic probability, notice that m1({bi})m2({bi}) produces μv, and for all other subsets X and Y, if their intersection is {bi}, then m1(X)m2(Y) is zero. Furthermore, m1(A - {bi}) m2(A - {bi}) = (1 - μ)(1 - v), so:

\[
m1 \oplus m2 ({bi}) = \frac{\mu v}{\mu + (1 - \mu)(1 - v)}
\]

\[
m1 \oplus m2 (A - {bi}) = \frac{(1 - \mu)(1 - v)}{\mu + (1 - \mu)(1 - v)}
\]

In this case, m1 ⊕ m2({bi}) ≥ m1({bi}) if and only if m2({bi}) ≥ 0.5 and m1 ⊕ m2(A - {bi}) ≥ m2(A - {bi}) if and only if m1(A - {bi}) ≥ 0.5. This shows that an estimation that bi occurs less than half of the time lowers the probability of another prediction of the same outcome.

Consider the case when b1 ≠ b2 (the heuristics predict different outcomes). If k = 2, we have the same case as bi = bj by using bj = A - {bj}. If 2 < k:

\[
m1 \oplus m2 ({bj}) = \frac{\mu (l - v)}{l - \mu}
\]

\[
m1 \oplus m2(A - {bj}) = \frac{\mu l - \mu}{l - \mu}
\]

In this case, m1 ⊕ m2({bj}) ≥ m1({bj}) if and only if m1(A - {bj}) = 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 → {b1, b2} initially (in the absence of a prediction2) has an equal probability of branching to b1 and b2 (m1({b1}) = m1({b2}) = 0.5). If a heuristic predicts that b → b1 occurs 70% of the time (m2({b1}) = 0.7 and m2({b2}) = 0.3), the combined probabilities are:

\[
m1 \oplus m2({b1}) = \frac{0.5 \times 0.7 + 0.5 \times 0.3}{0.5 \times 0.7 + 0.5 \times 0.3} = 0.7
\]

\[
m1 \oplus m2({b2}) = \frac{0.5 \times 0.7 + 0.5 \times 0.3}{0.5 \times 0.7 + 0.5 \times 0.3} = 0.3
\]

Now suppose another heuristic estimates that b → b1 is taken 60% of the time (m2({b1}) = 0.6 and m2({b2}) = 0.4). The estimate then becomes:

\[
m1 \oplus m2 \oplus m3({b1}) = \frac{0.7 \times 0.6 + 0.7 \times 0.3 + 0.3 \times 0.4}{0.7 \times 0.6 + 0.7 \times 0.3 + 0.3 \times 0.4} = 0.778
\]

\[
m1 \oplus m2 \oplus m3({b2}) = \frac{0.7 \times 0.6 + 0.7 \times 0.3 + 0.3 \times 0.4}{0.7 \times 0.6 + 0.7 \times 0.3 + 0.3 \times 0.4} = 0.222
\]

The second heuristic increased the probability that b1 is taken from 0.7 to 0.778. This process can be repeated, in any order, to incorporate other heuristics as the operator ⊕ 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 predict 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 Theoretically, the absence of a prediction should assign the "unknown" probabilities to a hypotheses:

\[
m1({b1}, {b2}) = 1, m1({b1}) = 0, m1({b2}) = 0.
\]

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

\[
bfreq(b_i) = \begin{cases} 
1 & (\text{if } b_i \text{ is the entry block}) \\
\sum_{b_p \in \text{pred}(b_i)} freq(b_p \rightarrow b_i) & (\text{otherwise}) 
\end{cases}
\]

\[
freq(b_i \rightarrow b_j) = bfreq(b_i) \cdot prob(b_i \rightarrow b_j)
\]

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:

```python
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.

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, \( infreq(b_0) \) 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 \( bfreq(b_i) = bfreq(b_0) \cdot r_i \) for \( i = 1, ..., k \), where \( r_i \) is the probability that control passes from \( b_0 \) to \( b_i \). From this, we find:

\[
bfreq(b_0) = \text{infreq}(b_0) + \sum_{i=1}^{k} \text{freq}(b_i \rightarrow b_0)
\]

\[
= \text{infreq}(b_0) + \sum_{i=1}^{k} (bfreq(b_i) \cdot \text{prob}(b_i \rightarrow b_0))
\]

\[
= \text{infreq}(b_0) + \sum_{i=1}^{k} (bfreq(b_0) \cdot r_i \cdot \text{prob}(b_i \rightarrow b_0))
\]

\[
= \text{infreq}(b_0) + bfreq(b_0) \sum_{i=1}^{k} (r_i \cdot \text{prob}(b_i \rightarrow b_0))
\]

Let

\[
p_i = r_i \cdot \text{prob}(b_i \rightarrow b_0)
\]

\[
cp(b_0) = \sum_{i=1}^{k} (r_i \cdot \text{prob}(b_i \rightarrow b_0)) = \sum_{i=1}^{k} p_i,
\]

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

\[
bfreq(b_0) = \text{infreq}(b_0) + bfreq(b_0) \cdot cp(b_0)
\]

\[
= \text{infreq}(b_0) + \sum_{i=1}^{k} \text{bfreq}(b_0) \cdot p_i
\]

\[
= \text{infreq}(b_0) + \sum_{i=1}^{k} \text{bfreq}(b_0) \cdot \frac{p_i}{1 - cp(b_0)}
\]

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:

\[
\text{bfreq}(b_0) = \frac{1}{1-0.5\times0.88-0.5\times0.88} = 8.33
\]
for the flow graph in Figure 1b, and:

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

\[
\text{bfreq}(b_{\text{inner}}) = \frac{\text{freq}(b_{\text{inner}})}{1-cp(b_{\text{inner}})}
\]
to find the frequency of the inner loop head, where \( b_{\text{inner}} \) is the head block of the inner loop structure and \( cp(b_{\text{inner}}) \) is \( b_{\text{inner}} \)'s cyclic probability.

\[\text{Figure 2. A two-head nested loop structure.}\]

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.

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

\[
\text{cfreq}(f) = 1 \quad (f \text{ is main function})
\]

\[
\text{cfreq}(f) = \sum_{p \in \text{pred}(f)} \text{cfreq}(p) \quad (\text{otherwise})
\]

\[
\text{gfreq}(f,g) = \text{lfreq}(f,g) \times \text{cfreq}(f)
\]

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 SPECint92C benchmarks and a several simple Unix applications. We ran the experiments on a Sequent S2000/750 system with 1486 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 [Wall91]. 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 \( \text{kim} \) is the unweighted match score. A perfect match will have 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 SPECint92C 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 shows that our technique identifies 79% of the top 20% blocks that account 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.

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>10%</th>
<th>20%</th>
<th>30%</th>
<th>40%</th>
<th>50%</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>espresso</td>
<td>.71</td>
<td>.71</td>
<td>.76</td>
<td>.78</td>
<td>.79</td>
</tr>
<tr>
<td>li</td>
<td>.83</td>
<td>.86</td>
<td>.84</td>
<td>.82</td>
<td>.83</td>
</tr>
<tr>
<td>eqnott</td>
<td>.84</td>
<td>.85</td>
<td>.86</td>
<td>.87</td>
<td>.88</td>
</tr>
<tr>
<td>compress</td>
<td>.89</td>
<td>.89</td>
<td>.77</td>
<td>.87</td>
<td>.88</td>
</tr>
<tr>
<td>se</td>
<td>.80</td>
<td>.76</td>
<td>.78</td>
<td>.82</td>
<td>.81</td>
</tr>
<tr>
<td>gcc</td>
<td>.73</td>
<td>.76</td>
<td>.75</td>
<td>.78</td>
<td>.76</td>
</tr>
<tr>
<td>Geo-mean</td>
<td>.79</td>
<td>.80</td>
<td>.79</td>
<td>.81</td>
<td>.84</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>10%</th>
<th>20%</th>
<th>30%</th>
<th>40%</th>
<th>50%</th>
</tr>
</thead>
<tbody>
<tr>
<td>espresso</td>
<td>57</td>
<td>61</td>
<td>64</td>
<td>68</td>
<td>66</td>
</tr>
<tr>
<td>li</td>
<td>61</td>
<td>67</td>
<td>64</td>
<td>68</td>
<td>75</td>
</tr>
<tr>
<td>eqntott</td>
<td>73</td>
<td>69</td>
<td>77</td>
<td>80</td>
<td>78</td>
</tr>
<tr>
<td>compress</td>
<td>65</td>
<td>74</td>
<td>72</td>
<td>80</td>
<td>71</td>
</tr>
<tr>
<td>sc</td>
<td>50</td>
<td>53</td>
<td>57</td>
<td>60</td>
<td>62</td>
</tr>
<tr>
<td>gcc</td>
<td>47</td>
<td>54</td>
<td>32</td>
<td>60</td>
<td>58</td>
</tr>
<tr>
<td>Geo-mean</td>
<td>58</td>
<td>67</td>
<td>64</td>
<td>69</td>
<td>68</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>10%</th>
<th>20%</th>
<th>30%</th>
<th>40%</th>
<th>50%</th>
</tr>
</thead>
<tbody>
<tr>
<td>espresso</td>
<td>57</td>
<td>66</td>
<td>68</td>
<td>83</td>
<td>70</td>
</tr>
<tr>
<td>li</td>
<td>39</td>
<td>76</td>
<td>52</td>
<td>92</td>
<td>67</td>
</tr>
<tr>
<td>eqntott</td>
<td>43</td>
<td>98</td>
<td>46</td>
<td>99</td>
<td>53</td>
</tr>
<tr>
<td>compress</td>
<td>1.0</td>
<td>1.0</td>
<td>30</td>
<td>50</td>
<td>1.0</td>
</tr>
<tr>
<td>sc</td>
<td>50</td>
<td>57</td>
<td>50</td>
<td>91</td>
<td>44</td>
</tr>
<tr>
<td>gcc</td>
<td>35</td>
<td>32</td>
<td>47</td>
<td>53</td>
<td>52</td>
</tr>
<tr>
<td>Geo-mean</td>
<td>51</td>
<td>67</td>
<td>52</td>
<td>85</td>
<td>69</td>
</tr>
</tbody>
</table>

Table 4. Scores of SPEC92 function invocation 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.

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>10%</th>
<th>20%</th>
<th>30%</th>
<th>40%</th>
<th>50%</th>
</tr>
</thead>
<tbody>
<tr>
<td>espresso</td>
<td>53</td>
<td>55</td>
<td>66</td>
<td>74</td>
<td>70</td>
</tr>
<tr>
<td>li</td>
<td>44</td>
<td>73</td>
<td>48</td>
<td>84</td>
<td>52</td>
</tr>
<tr>
<td>eqntott</td>
<td>66</td>
<td>60</td>
<td>69</td>
<td>56</td>
<td>56</td>
</tr>
<tr>
<td>compress</td>
<td>54</td>
<td>65</td>
<td>67</td>
<td>78</td>
<td>66</td>
</tr>
<tr>
<td>sc</td>
<td>54</td>
<td>81</td>
<td>54</td>
<td>90</td>
<td>56</td>
</tr>
<tr>
<td>gcc</td>
<td>26</td>
<td>27</td>
<td>38</td>
<td>50</td>
<td>46</td>
</tr>
<tr>
<td>Geo-mean</td>
<td>48</td>
<td>51</td>
<td>56</td>
<td>77</td>
<td>57</td>
</tr>
</tbody>
</table>

Table 5. Scores of SPEC92 global function call frequency.

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>LBK</th>
<th>LBR</th>
<th>FT</th>
<th>FC</th>
<th>GBK</th>
<th>GBR</th>
</tr>
</thead>
<tbody>
<tr>
<td>cal</td>
<td>49</td>
<td>55</td>
<td>1.0</td>
<td>1.0</td>
<td>80</td>
<td>80</td>
</tr>
<tr>
<td>emp</td>
<td>86</td>
<td>93</td>
<td>0.0</td>
<td>1.0</td>
<td>86</td>
<td>93</td>
</tr>
<tr>
<td>ed</td>
<td>74</td>
<td>65</td>
<td>96</td>
<td>31</td>
<td>82</td>
<td>81</td>
</tr>
<tr>
<td>grep</td>
<td>66</td>
<td>52</td>
<td>1.0</td>
<td>1.0</td>
<td>86</td>
<td>92</td>
</tr>
<tr>
<td>factor</td>
<td>89</td>
<td>87</td>
<td>0.0</td>
<td>1.0</td>
<td>94</td>
<td>94</td>
</tr>
<tr>
<td>top</td>
<td>90</td>
<td>77</td>
<td>1.0</td>
<td>1.0</td>
<td>42</td>
<td>43</td>
</tr>
<tr>
<td>split</td>
<td>99</td>
<td>99</td>
<td>1.0</td>
<td>0.0</td>
<td>99</td>
<td>99</td>
</tr>
<tr>
<td>sum</td>
<td>69</td>
<td>71</td>
<td>1.0</td>
<td>0.0</td>
<td>69</td>
<td>71</td>
</tr>
<tr>
<td>sort</td>
<td>77</td>
<td>76</td>
<td>1.0</td>
<td>1.0</td>
<td>87</td>
<td>81</td>
</tr>
<tr>
<td>uniq</td>
<td>81</td>
<td>73</td>
<td>1.0</td>
<td>0.0</td>
<td>39</td>
<td>45</td>
</tr>
<tr>
<td>wc</td>
<td>92</td>
<td>55</td>
<td>1.0</td>
<td>1.0</td>
<td>95</td>
<td>91</td>
</tr>
<tr>
<td>Average</td>
<td>79</td>
<td>73</td>
<td>82</td>
<td>76</td>
<td>78</td>
<td>79</td>
</tr>
</tbody>
</table>

Table 6. Scores of SPEC92 global block frequency.

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%.

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>LBK</th>
<th>LBR</th>
<th>FT</th>
<th>FC</th>
<th>GBK</th>
<th>GBR</th>
</tr>
</thead>
<tbody>
<tr>
<td>cal</td>
<td>49</td>
<td>55</td>
<td>1.0</td>
<td>1.0</td>
<td>80</td>
<td>80</td>
</tr>
<tr>
<td>emp</td>
<td>86</td>
<td>93</td>
<td>0.0</td>
<td>1.0</td>
<td>86</td>
<td>93</td>
</tr>
<tr>
<td>ed</td>
<td>74</td>
<td>65</td>
<td>96</td>
<td>31</td>
<td>82</td>
<td>81</td>
</tr>
<tr>
<td>grep</td>
<td>66</td>
<td>52</td>
<td>1.0</td>
<td>1.0</td>
<td>86</td>
<td>92</td>
</tr>
<tr>
<td>factor</td>
<td>89</td>
<td>87</td>
<td>0.0</td>
<td>1.0</td>
<td>94</td>
<td>94</td>
</tr>
<tr>
<td>top</td>
<td>90</td>
<td>77</td>
<td>1.0</td>
<td>1.0</td>
<td>42</td>
<td>43</td>
</tr>
<tr>
<td>split</td>
<td>99</td>
<td>99</td>
<td>1.0</td>
<td>0.0</td>
<td>99</td>
<td>99</td>
</tr>
<tr>
<td>sum</td>
<td>69</td>
<td>71</td>
<td>1.0</td>
<td>0.0</td>
<td>69</td>
<td>71</td>
</tr>
<tr>
<td>sort</td>
<td>77</td>
<td>76</td>
<td>1.0</td>
<td>1.0</td>
<td>87</td>
<td>81</td>
</tr>
<tr>
<td>uniq</td>
<td>81</td>
<td>73</td>
<td>1.0</td>
<td>0.0</td>
<td>39</td>
<td>45</td>
</tr>
<tr>
<td>wc</td>
<td>92</td>
<td>55</td>
<td>1.0</td>
<td>1.0</td>
<td>95</td>
<td>91</td>
</tr>
<tr>
<td>Average</td>
<td>79</td>
<td>73</td>
<td>82</td>
<td>76</td>
<td>78</td>
<td>79</td>
</tr>
</tbody>
</table>

Table 8. Scores for Unix commands (top 20%): LBK-local block, LBR-local branch, FT-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 programs from experiments with the SPECint92 Benchmarks and several Unix commands.
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 multiway 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 `sqrt` 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


[Spre92] SPEC Cint92, Release V1.1


Appendix A

This appendix contains a small, complete example that illustrates the algorithm 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 \( b_0 \rightarrow (b_1, b_2) \), 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 b5 contains a call (CH) and a return (RH). The combined probability of 0.95 for b0→b1 is much higher than the probability estimated by any of the five heuristics.

For the second non-loop branch b1→{b2, b4}, both the Loop Header (LHH) and Return heuristics (RH) apply. Table 10 summarizes the probabilities.

Table 10. Branch probabilities for b1→{b2, b4}.

<table>
<thead>
<tr>
<th>n</th>
<th>b4−b2</th>
<th>b4−b4</th>
</tr>
</thead>
<tbody>
<tr>
<td>LHH</td>
<td>0.75</td>
<td>0.25</td>
</tr>
<tr>
<td>RH</td>
<td>0.72</td>
<td>0.28</td>
</tr>
<tr>
<td>Combined</td>
<td>0.89</td>
<td>0.12</td>
</tr>
</tbody>
</table>

Figure 5. Branch probabilities of atoi.

The branch b3→{b3, b4} contains a back edge, and only the loop branch heuristic (LBB) applies, so:

\[ \text{prob}(b3 \rightarrow b2) = 0.88 \]
\[ \text{prob}(b3 \rightarrow b4) = 0.12 \]

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 (b3→b2) the cyclic probability is the same as prob(b3→b3). For the outer loop, the block frequencies and edge frequencies for the atoi function are calculated as follows:

- freq(b0) = 1
- freq(b0→b1) = prob(b0→b1) × 1 = 0.95
- freq(b0→b2) = prob(b0→b2) × 1 = 0.52
- freq(b1) = freq(b0→b1) = 0.95
- freq(b1→b2) = prob(b1→b2) × freq(b1) = 0.89 × 0.95 = 0.84
- freq(b1→b3) = prob(b1→b3) × freq(b1) = 0.12 × 0.95 = 0.11
- freq(b2) = freq(b1→b2) = 0.84
- freq(b2→b3) = prob(b2→b3) × freq(b2) = 1 × 0.84 = 0.84
- freq(b3) = 1 - prob(b3→b2) = 0.88
- freq(b3→b3) = freq(b3) × 0.84 = 0.69
- freq(b3→b4) = freq(b3→b4) × 0.12 = 0.84
- freq(b4) = freq(b3→b4) + freq(b3→b4) = 0.11 + 0.84 = 0.95
- freq(b5) = freq(b4→b5) = 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(b0→b2) + freq(b1→b4) + freq(b3→b4) = 0.05 + 0.11 + 0.84 = 1.

The atoi function calls printf in block b5 and block b5’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.

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.

Table 11. Global frequencies of program.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>report_one</td>
<td>b0→b3</td>
<td>0.2</td>
<td>2.1</td>
<td>0.5</td>
</tr>
<tr>
<td>report_one</td>
<td>b0→b5</td>
<td>0.9</td>
<td>2.1</td>
<td>2.1</td>
</tr>
<tr>
<td>report_one</td>
<td>b1→b3</td>
<td>0.9</td>
<td>2.1</td>
<td>2.1</td>
</tr>
<tr>
<td>report_one</td>
<td>b1→b5</td>
<td>0.9</td>
<td>2.1</td>
<td>2.1</td>
</tr>
<tr>
<td>report_one</td>
<td>b5→b1</td>
<td>0.9</td>
<td>2.1</td>
<td>2.1</td>
</tr>
<tr>
<td>in_prefix</td>
<td>b0→b5</td>
<td>0.2</td>
<td>17.5</td>
<td>41</td>
</tr>
<tr>
<td>in_prefix</td>
<td>b0→b7</td>
<td>0.9</td>
<td>17.1</td>
<td>16.7</td>
</tr>
<tr>
<td>in_prefix</td>
<td>b1→b2</td>
<td>0.9</td>
<td>17.1</td>
<td>4.9</td>
</tr>
<tr>
<td>in_prefix</td>
<td>b1→b3</td>
<td>0.9</td>
<td>17.1</td>
<td>10.0</td>
</tr>
<tr>
<td>in_prefix</td>
<td>b3→b5</td>
<td>0.7</td>
<td>17.1</td>
<td>12.0</td>
</tr>
<tr>
<td>in_prefix</td>
<td>b3→b1</td>
<td>0.5</td>
<td>17.1</td>
<td>88.0</td>
</tr>
<tr>
<td>in_prefix</td>
<td>b7→b1</td>
<td>0.9</td>
<td>17.1</td>
<td>16.7</td>
</tr>
<tr>
<td>permute_next_pos</td>
<td>b0→b5</td>
<td>0.2</td>
<td>2.1</td>
<td>0.5</td>
</tr>
<tr>
<td>permute_next_pos</td>
<td>b0→b7</td>
<td>0.9</td>
<td>2.1</td>
<td>2.1</td>
</tr>
<tr>
<td>permute_next_pos</td>
<td>b1→b2</td>
<td>1.5</td>
<td>2.1</td>
<td>3.2</td>
</tr>
<tr>
<td>permute_next_pos</td>
<td>b1→b3</td>
<td>6.6</td>
<td>2.1</td>
<td>13.9</td>
</tr>
<tr>
<td>permute_next_pos</td>
<td>b2→b3</td>
<td>1.5</td>
<td>2.1</td>
<td>3.2</td>
</tr>
<tr>
<td>permute_next_pos</td>
<td>b3→b5</td>
<td>0.9</td>
<td>2.1</td>
<td>2.1</td>
</tr>
<tr>
<td>permute_next_pos</td>
<td>b3→b1</td>
<td>7.2</td>
<td>2.1</td>
<td>15.0</td>
</tr>
<tr>
<td>permute_next_pos</td>
<td>b7→b1</td>
<td>0.9</td>
<td>2.1</td>
<td>2.1</td>
</tr>
<tr>
<td>atoi</td>
<td>b0→b1</td>
<td>0.5</td>
<td>1.0</td>
<td>0.95</td>
</tr>
<tr>
<td>atoi</td>
<td>b0→b5</td>
<td>0.5</td>
<td>1.0</td>
<td>0.05</td>
</tr>
<tr>
<td>atoi</td>
<td>b1→b4</td>
<td>0.11</td>
<td>1.0</td>
<td>0.11</td>
</tr>
<tr>
<td>atoi</td>
<td>b1→b2</td>
<td>0.84</td>
<td>1.0</td>
<td>0.84</td>
</tr>
<tr>
<td>atoi</td>
<td>b2→b3</td>
<td>0.84</td>
<td>1.0</td>
<td>0.84</td>
</tr>
<tr>
<td>atoi</td>
<td>b3→b3</td>
<td>0.84</td>
<td>1.0</td>
<td>0.84</td>
</tr>
<tr>
<td>atoi</td>
<td>b3→b4</td>
<td>0.84</td>
<td>1.0</td>
<td>0.84</td>
</tr>
<tr>
<td>atoi</td>
<td>b3→b5</td>
<td>0.84</td>
<td>1.0</td>
<td>0.84</td>
</tr>
<tr>
<td>atoi</td>
<td>b3→b1</td>
<td>0.5</td>
<td>1.0</td>
<td>0.5</td>
</tr>
<tr>
<td>atoi</td>
<td>b3→b2</td>
<td>0.5</td>
<td>1.0</td>
<td>0.5</td>
</tr>
</tbody>
</table>

Table 12. Edges frequency matching for program.

Function key: 1 - in_prefix; 2 - report_one; 3 - permute_next_pos; 4 - atoi; 5 - permute.