The Road Not Taken: Estimating Path Execution Frequency Statically

Raymond P.L. Buse
University of Virginia
Charlottesville, VA, USA

Westley Weimer
University of Virginia
Charlottesville, VA, USA

E-mail: {buse, weimer}@cs.virginia.edu

Abstract

A variety of compilers, static analyses, and testing frameworks rely heavily on path frequency information. Uses for such information range from optimizing transformations to bug finding. Path frequencies are typically obtained through profiling, but that approach is severely restricted: it requires running programs in an indicative environment, and on indicative test inputs.

We present a descriptive statistical model of path frequency based on features that can be readily obtained from a program's source code. Our model is over 90% accurate with respect to several benchmarks, and is sufficient for selecting the 5% of paths that account for over half of a program's total runtime. We demonstrate our technique's robustness by measuring its performance as a static branch predictor, finding it to be more accurate than previous approaches on average. Finally, our qualitative analysis of the model provides insight into which source-level features indicate "hot paths."

1 Introduction

"Amdahl's law" is often invoked as advice to prioritize optimization on the most common use case. In the domain of program analysis, this demands *hot path* identification [4, 7, 13, 25]. The importance of hot paths arises from the empirical observation that most or all of the execution time of a typical program is spent along a small percentage of program paths. That is, certain sequences of instructions tend to repeat often.

This high degree of non-uniformity in program execution makes characterizing the runtime behavior of software artifacts an important concern for tasks far beyond code optimization; for propositions as diverse as maintenance [28] and general data-flow analysis [2]. As a result, program profiling remains a vibrant area of research (e.g., [7, 18, 32]) even after many years of treatment from the community. Unfortunately, profiling is severely limited by practical concerns: the frequent lack of appropriate workloads for programs, the questionable degree to which they are indicative of actual usage, and the inability of such tools evaluate program modules or individual paths in isolation.

In a static context, without workloads, there is no clear notion of the relative execution frequency of control-flow paths. The frequency with which any given block of code will be executed is unknown. Without additional information all paths can only be assumed equally likely. This practice is questionable, particularly when an analysis is designed to measure or affect performance. It is reported that large amounts of code exists to handle "exceptional" cases only, with 46%–66% of code related to error handling and 8%–16% of all methods containing error handling constructs (see [34, p. 4] for a survey). Such error handling code is common, but does not significantly impact real performance [23]. Therefore, the assumption that all paths are equally likely or that input space in uniform leads to inaccurate or otherwise suboptimal results in program analysis.

We propose a fully static approach to the problem of hot path identification. Our approach is based on two key insights. First, we observe a relationship between the relative frequency of a path and its effect on program state. Second, we carefully choose the right level of abstraction for considering program paths: interprocedural for precision, but limited to one class for scalability.

Program State. We observe that there are static source-level characteristics that can indicate whether a program path was "intended" to be common, or alternatively, if it was designed to be rare or "exceptional." We claim that infrequent cases in software typically take one of two forms. The first involves error detection followed by a divergence from the current path, such as returning an error code as a response to an invalid argument or raising an exception after an assertion failure. The second involves significant reconfiguration of program state, such as reorganizing a data structure, reloading data from a file or resizing a hash table. Conversely, common cases typically entail relatively small state modifications, and tend to gradually increase available context, subtly increasing system entropy. We hypothesize that paths which exhibit only small impacts on program state, both in terms of global variables and in terms of context and stack frames, are more likely to be hot paths.

Program Paths. When predicting relative path execution frequency, the definition of "path" is critical. In their seminal work on `gprof`, Graham *et al.* note, "to be useful, an execution profiler must attribute execution time in a way that is significant for the logical structure of a program as well as for its textual decomposition." [18] Flat profiles are not as helpful to programmers or as accurate as interprocedural call-graph based profiles. We consider method invocations between methods of the same class to be part of one uninterrupted path; this choice allows us to be very precise in common object-oriented code. For the purposes of our analysis we split paths when control flow crosses a class boundary; this choice allows us to scale to large programs.

These two insights form the heart of our static approach to predicting dynamic execution frequencies. In this paper we enumerate a set of static source-level features on control flow paths. Our features are based on structure, data flow, and effect on program state. We employ these features to train a machine learning algorithm to predict runtime path frequency. Our approach reports, for each path, a numerical estimate of runtime frequency relative either to the containing method, or to the entire program. We evaluate our prototype implementation on the SPECjvm98 benchmarks, using their supplied workloads as the ground truth. We also discuss and evaluate the relative importance of our features.

The main contributions of this paper are:

- A technique for statically estimating the runtime frequency of program paths. Our results can be used to support or improve many types of static analyses.
- An empirical evaluation of our technique. We measure accuracy both in selecting hot paths and in ranking paths by frequency. We demonstrate the flexibility of our model by employing it to characterize the running time and dynamic branching behavior of several benchmarks. The top 5% of paths reported account for over half of program runtime; a static branch predictor based on our technique has a 69% hit rate compared to the 65% of previous work.

The structure of this paper is as follows. We discuss related work, present a motivating example, and enumerate several potential uses of our technique in Section 2. After describing our process for program path enumeration in Section 3, we discuss the features that make up our model in Section 4. We perform a direct model evaluation in Section 5. We then perform a timing based experiment (Section 6) and a branch prediction experiment (Section 7). Finally, we extract and discuss possible implications of our model (Section 8) and then conclude.

2 Context and Motivation

In this section, we present key related work, review a simple example that demonstrates the application and output of our technique, and then highlight a number of tasks and analyses that could benefit from a static prediction of dynamic path frequencies.

The work most similar to ours is *static branch prediction*, a problem first explored by Ball and Larus [5]. They showed that simple heuristics are useful for predicting the frequency at which branches are taken. The heuristics typically involve surface-level features such as, "if a branch compares a pointer against null or compares two pointers, predict the branch on false condition as taken." Extensions to this work have achieved modest performance improvements by employing a larger feature set and neural network learning [11].

We have chosen to focus on paths instead of branches for two reasons. First, we claim that path-based frequency can be more useful in certain static analysis tasks; see Ball *et al.* [8] for an in-depth discussion. Second, paths contain much more information than individual branches, and thus are much more amenable to the process of formal modeling and prediction.

Automatic *workload generation* is one strategy for coping with a lack of program traces. This general technique has been adapted to many domains [22, 24, 27]. While workload generation is useful for stress testing software and for achieving high path coverage, it has not been shown suitable for the creation of indicative workloads. In contrast, we attempt to model indicative execution frequencies.

More recent work in *concolic testing* [30, 31] explores all execution paths of a program with systematically selected inputs to find subtle bugs. The technique interleaves static symbolic execution to generate test inputs and concrete execution on those inputs. Symbolic values that are too complex for the static analysis to handle are replaced by concrete values from the execution. Our approach makes static predictions about dynamic path execution frequency, but does not focus on bug-finding.

2.1 Motivating Example

We hypothesize that information about the expected runtime behavior of imperative programs, including their relative path execution frequency, is often embedded into source code by developers in an implicit, but nonetheless predictable, way. We now present an intuition for how this information is manifested.

Figure 1 shows a typical example of the problem of hot path identification in a real-world algorithm implementations. In this case there are three paths of interest. The path corresponding to insertion of the new entry on lines

```
 1  /**
 2  * Maps the specified key to the specified
 3  * value in this hashtable ...
 4  */
 5  public synchronized V put(K key, V value) {
 6
 7    // Make sure the value is not null
 8    if (value == null) {
 9      throw new NullPointerException();
10    }
11
12    ...
13
14    modCount++;
15    if (count >= threshold) {
16      // Rehash the table if the
17      // threshold is exceeded
18      rehash();
19
20      tab = table;
21      index = (hash & 0x7FFFFFFF) % tab.len;
22    }
23
24    // Creates the new entry.
25    Entry<K,V> e = tab[index];
26    tab[index] =
27      new Entry<K,V>(hash, key, value, e);
28    count++;
29    return null;
30  }
```

Figure 1. The **put** method from the Java SDK version 1.6's **java.util.Hashtable** class. Some code has been omitted for illustrative simplicity.

26–27 is presumably the "common case." However, there is also the case in which the **value** parameter is **null** and the function terminates abruptly on line 9, as well as the case where the **rehash** method is invoked on line 18, an operation likely to be computationally expensive.

This example helps to motivate our decision to follow paths within class boundaries. We leverage the object-oriented system paradigm whereby code and state information are encapsulated together. Following paths within classes enables our technique to discover that **rehash** modifies a large amount of program state. Not tracing all external method invocations allows it to scale.

Our technique identifies path features, such as "throws an exception" or "reads many class fields", that we find are indicative of runtime path frequency. The algorithm we present successfully discovers that the path that merely creates a new entry is more common than the **rehash** path or the exceptional path. We now describe some applications that might make use of such information.

2.2 Example Client Applications

Profile-guided optimization refers to the practice of optimizing a compiled binary subsequent to observing its runtime behavior on some workload (e.g., [3, 9, 19]). Our technique has the potential to make classes of such optimization more accessible; first, by eliminating the need for workloads, and second by removing the time required to run them and record profile information. A static model of relative path frequency could help make profile-guided compiler optimizations more mainstream.

Current work in computational *complexity estimation* has been limited to run-time analyses [17]. At a high level, the task of estimating the complexity of the **put** procedure in Figure 1 requires identifying the paths that represent the common case and thus dominate the long-run runtime behavior. This is critical, because mistakenly assuming that the path along which **rehash** is invoked is common, will result in a significant over-estimate of the cost of **put**. Similarly, focusing on the path corresponding to the error condition on line 9 will underestimate the cost.

Static *specification mining* [1, 33], which seeks to infer formal descriptions of correct program behavior, can be viewed as the problem of distinguishing correct paths from erroneous ones in programs that contain both. Behavior that occurs on many paths is assumed to be correct, and deviations may indicate bugs [15]. In practice, static specification miners use static path counts as a proxy for actual execution behavior; a static model of path frequency would improve the precision of such techniques.

Many static analyses produce false alarms, and *statistical ranking* is often used to sort error reports by placing likely bugs at the top [21]. These orderings often use static proportion counts and z-rankings based on the assumption that all static paths are equally likely. A static prediction of path frequency would allow bug reports on infrequent paths, for example, to be filtered to the top of the list.

In general any program analysis or transformation that combines static and dynamic information could benefit from a model of path frequency. Areas as disparate as *program specialization* [29], where frequency predictions could reduce the annotation and analysis burden for indicating what to specialize, and *memory bank conflict detection* in GPU programs [10], where frequency predictions could reduce the heavy simulation and deployment costs of static analysis, are potential clients. In the next sections we describe the details of our analysis.

3 Path Enumeration

In an imperative language, a method is comprised of a set of control flow paths, only one of which will be executed on any actual invocation. Our goal is to statically quantify the relative runtime frequency with which each path through a method will be taken. Our analysis works by analyzing features along each path.

Our algorithm operates on a per-class-declaration basis. This enables it to consider interactions between the methods of a single class that are likely to assist in indicating path frequencies, but avoids the large explosion of paths that would result if it were fully context sensitive.

Central to our technique is *static path enumeration*, whereby we enumerate all acyclic intra-class paths in a target program. To experimentally validate our technique, we also require a process for *dynamic path enumeration* that counts the number of times each path in a program is executed during an actual program run.

3.1 Static Path Enumeration

Static intra-class path enumeration begins at each member function of each concrete program class. A *static intra-class path* is a flat, acyclic whole-program path that starts at a public method of a class T and contains only statements within T and only method invocations to methods outside of T. We first enumerate the paths in each method separately; we then combine them to obtain intra-class paths.

For a given method, we construct the control flow graph and consider possible paths in turn. A method with loops may have an unbounded number of potential paths; we do not follow back edges to ensure that the path count is finite. For the purposes of our analysis, a path through a loop represents all paths that take the loop one or more times. We find that this level of description is adequate for many client analyses, but higher bounds on the number of loop iterations are also possible. Our path definition is similar to the one used in efficient path profiling [7], but richer in that we consider certain interprocedural paths. We process, record and store information about each path as we enumerate it.

Once all intra-method paths are enumerated they can be merged, or "flattened", into intra-class paths. We use a fixpoint worklist algorithm. We iterate though each path, and at intra-class invocation sites we "splice-in" the paths corresponding to the invoked method, cloning the caller path for each invoked callee path. The process terminates when all such invocation sites in all paths have been resolved. Since we do not allow a statement to appear more than once on any single path and the number of invocation sites is finite, this process always terminates.

Figure 2 shows the pseudocode for our static path enumeration algorithm. While our static intra-class path definition is intuitive, we show the enumeration process because our path frequency estimates are given only for static paths we enumerate and our rankings of path frequencies are given only relative to other static paths. Our intra-class path definition thus influences the results of our analysis.

Input: Concrete Class T.
Output: Mapping P : method $\rightarrow$ static intra-class path set

```
for all methods $meth$ in $T$ do
  $P(meth) \leftarrow$ acyclic paths in $meth$
end for
let $Worklist \leftarrow$ methods in $T$
while $Worklist$ is not empty do
  let $meth \leftarrow$ remove next from $Worklist$
  for all paths $path$ in $P(meth)$ do
    let $changed \leftarrow$ false
    for all method invocations $callee()$ in $path$ do
      if $callee \in T$ then
        for all paths $path_c$ in $P(callee)$ do
          let $path' \leftarrow$ copy of $path$
          $path' \leftarrow path'[callee() \mapsto path_c]$
          $P(meth) \leftarrow P(meth) \cup \{path'\} - \{path\}$
          $changed \leftarrow$ true
        end for
      end if
    end for
    if $changed$ then
      $Worklist \leftarrow Worklist \cup \{meth\}$
    end if
  end for
end while
```

Figure 2. High-level pseudocode for enumerating static intra-class paths for a concrete class T.

3.2 Dynamic Path Enumeration

To train and evaluate our model of path execution frequency we require "ground truth" for how often a given static path is actually executed on a typical program run. Our dynamic path enumeration approach involves two steps. First, we record each statement visited on a program run; before executing a statement we record its unique identifier, including the enclosing class and method. One run of the program will yield one single list of statements.

Second, we partition that list into static paths by processing it in order while simulating the run-time call stack. We split out intra-class paths when the flow of control leaves the currently enclosing class or when the same statement is revisited in a loop. Once the list of statements has been partitioned into static intra-class paths, we count the number of times each path occurs. This *dynamic count* is our ground truth and the prediction target for our static analysis.

4 Static Path Description Model

We hypothesize that there is sufficient information embedded in the code of a path to estimate the relative frequency with which it will be taken at runtime. We characterize paths using a set of *features* that can be used to render this estimate. With respect to this feature set, each path can be viewed as a vector of numeric entries: the *feature values* for that path. Our path frequency estimation model takes as input a feature vector representing a path and outputs the predicted frequency of that path.

In choosing a set of features we have two primary concerns: predictive power and efficiency. For power, we base our feature set on our earlier intuition of the role that program state transformations play in software engineering practice. To scale to large programs, we choose features that can be efficiently computed in linear time.

The features we consider for this study are enumerated in Figure 3; they are largely designed to capture the state-changing behavior that we hypothesize is related to path frequency. We formulate these features for Java, but note that they could easily be extended to other object-oriented languages. Many of our features consist of a static count of some source-level feature, such as the number of `if` statements. Others measure the percentage of some possible occurrences that appear along the path. For example, "field coverage" is the percentage of the total class fields that appear along the path, while "fields written coverage" is the percentage of the non-`final` fields of that class that are updated along the path. "Invoked method statements" refers to statements only from the method where the path originates. Our count features are discrete, non-negative and unbounded; our coverage features are continuous in the closed interval $[0, 1]$.

Given vectors of numeric features, we can use any number of machine learning algorithms to model and estimate path frequency. In our experiments, accurate estimates of path frequency arise from a complex interaction of features. We view the choice of features, the hypothesis behind them, and the final performance of a model based on them as more important than the particular machine learning technique used. In fact, we view it as an advantage that multiple learning techniques that make use of our features perform similarly. In our experiments, for example, Bayesian and Perceptron techniques performed almost equally well; this suggests that the features, and not some quirk of the learning algorithm, are responsible for accurate predictions. In the experiments in this paper we use logistic regression because its accuracy was among the best we tried, and because the probability estimates it produced showed a high degree of numerical separation (i.e., the probability estimates showed few collisions). This property proved important in creating a ranked output for a large number of paths.

Count	Coverage	Feature
✓		`==`
✓		`new`
✓		`this`
✓		all variables
✓		assignments
✓		dereferences
✓	✓	fields
✓	✓	fields written
✓	✓	invoked method statements
✓		`goto` stmts
✓		`if` stmts
✓		local invocations
✓	✓	local variables
✓		non-local invocations
✓	✓	parameters
✓		`return` stmts
✓		statements
✓		`throw` stmts

Figure 3. The set of features we consider. A "Count" is a raw static tally of feature occurrences. A "Coverage" is a percentage measuring the fraction of the possible activity that occurs along the path.

5 Model Evaluation

We performed two experiments to evaluate the predictive power of our model with respect to the `SPECjvm98` benchmarks. The first experiment evaluates our model as a binary classifier, labeling paths as either "high frequency" or "low frequency." The second experiment evaluates the ability of our model to correctly sort lists of paths by frequency.

5.1 Experimental Setup

We have implemented a prototype version of the algorithm described in Section 4 for the Java language. We use the SOOT [16] toolkit for parsing and instrumenting. We use the Weka [20] toolkit for machine learning. We characterize our set of benchmarks in Figure 4. Note that the `javac` program is significantly larger and, in our view, more realistic than the other benchmarks; we see it as a more indicative test of our model in terms of program structure. However, all of the benchmarks have been specifically designed to be indicative of of real programs in terms of runtime behavior.

Figure 5 details the behavior of our static path enumeration algorithm (see Section 3.1) on our benchmarks. The `javac` and `jack` benchmarks had the largest number of static paths. Despite its relatively large size in lines of code, `jess` actually had very few static intra-class paths because

Name	Description	LOC	Class	Meth
check	check VM features	1627	14	107
compress	file compression	778	12	44
db	data management	779	3	34
jack	parser generator	7329[a]	52	304
javac	java compiler	56645	174	1183
jess	expert system shell	8885	5	44
mtrt	ray tracer	3295	25	174
Total		79338	275	1620

[a]no source code was given for jack; we report lines from a decompiled version. Our analysis runs on Java bytecode.

Figure 4. The set of `SPECjvm98` benchmark programs used. "Class" counts the number of classes, "Meth" counts the number of methods.

Name	Paths	Paths/method	Runtime
check	1269	11.9	4.2s
compress	491	11.2	2.91s
db	807	23.7	2.8s
jack	8692	28.6	16.9s
javac	13136	11.1	21.4s
jess	147	3.3	3.12s
mtrt	1573	9.04	6.17s
Total or Avg	26131	12.6	59s

Figure 5. Static intra-class path enumeration.

of its relatively sparse call graph. The `jack` and `db` benchmarks had the highest number of static paths per method because of their more complicated object structures. In total we were able to enumerate over 26,000 static paths in under a minute on a 2GHz dual-core architecture; the process was largely disk bound to 150 MBytes/sec.

5.2 Classifier Training

We formulate the problem of hot-path identification as a classification task: given a path, does it belong to the set of "high frequency" or "low frequency" paths? We can phrase the problem alternatively as: with what probability is this a "high frequency" path?

Machine learning algorithms designed for this purpose take the form of *classifiers* which operate on *instances*, or object descriptions [26]. In our case, each instance is a feature vector of numerical values describing a specific static path, and the classifier used is based on logistic regression.

Such classifiers are supervised learning algorithms that involve two phases: training and evaluation. The training phase generates a classifier from a set of instances along with a labeled correct answer derived from the path counts obtained by dynamic path enumeration (Section 3.2). This answer is a binary judgment partitioning the paths into low frequency and high frequency. We consider two ways of distinguishing between these categories: an "intra-method" approach and an "inter-method" approach. The preferred scheme necessarily depends on the client application.

In an intra-method experiment we first normalize path counts for each method so that the sum of all path counts for that method becomes 1.0. We label paths as high frequency if they have a normalized intra-method frequency greater than 0.5 (i.e., they have a frequency greater than all other paths in the method combined). In an inter-method experiment, we consider a path as high frequency if it was taken more than 10 times. Our approach is largely insensitive to the threshold chosen; we achieve very similar results for thresholds between two and twenty.

After training, we apply the classifier to an instance it has not seen before, obtaining an estimate of the probability that it belongs in the high frequency class. This is our estimate for the runtime frequency of the path.

To help mitigate the danger of over-fitting (i.e., of constructing a model that fits only because it is very complex in comparison to the amount of data), we always evaluate on one benchmark at a time, using the other benchmarks as training data. We thus avoid training and testing on the same dataset.

5.3 F-score Analysis

First, we quantify the accuracy of our approach as an information retrieval task. In doing so we evaluate only discrete (nominal) classifier output. Two relevant success metrics in an experiment of this type are *recall* and *precision*. Here, recall is the percentage of high frequency paths that were correctly classified as high frequency by the model. Precision is the fraction of the paths classified as high frequency by the model that were actually high frequency according to our dynamic path count. When considered independently, each of these metrics can be made perfect trivially (e.g., a degenerate model that always returns high frequency has perfect recall). We thus weight them together using the F-score statistic, the harmonic mean of precision and recall [12]. This reflects the accuracy of the classifier with respect to the high frequency paths.

For each benchmark X, we first train on the dynamic paths from all of the other benchmarks. We then consider all of the high frequency paths in X, as determined by dynamic path count. We also consider an equal number of low frequency paths from X, chosen at random. We then use our model to classify each of the selected paths. Using an equal number of high- and low-frequency paths during evaluation allows us to compare accuracy in a uniform way across benchmarks. The F-scores for each benchmark, as

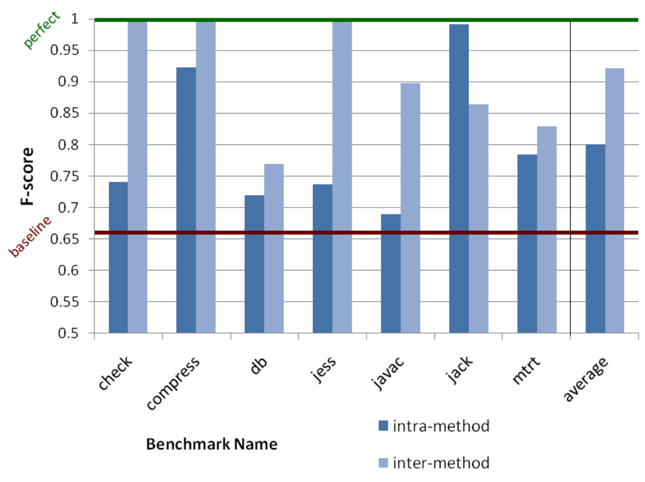

Figure 6. F-score of our performance at classifying high frequency paths. 0.67 is baseline; 1.0 is perfect.

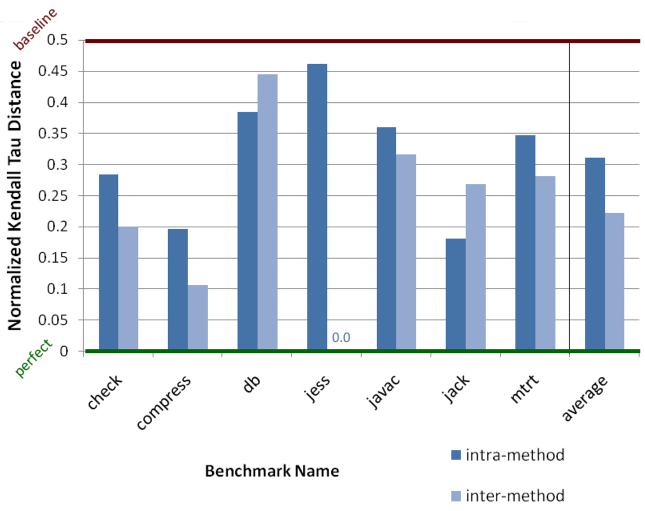

Figure 7. Kendall's tau performance of our model when used to rank order paths based on predicted dynamic frequency. 0.0 is perfect; 0.25 is a strong correlation; 0.5 is random chance.

well as the overall average, are shown in Figure 6.

In our case, an F-score of 0.67 serves as a baseline, corresponding to a uniform classification (e.g., predicting all paths to be high frequency). Perfect classification yields an F-score of 1.0, which our model achieves in 3 cases. On average we obtain an F-score of 0.8 for intra-method hot paths and 0.9 for inter-method ones.

Our model is thus accurate at classifying hot paths. One example of an application that would benefit from such a classifier is static specification mining. A specification miner might treat frequent and infrequent paths differently, in the same way that previous miners have treated error paths and normal paths differently for greater accuracy [33]. However, for other analysis tasks it is important to be able to discriminately rank or sort paths based on predicted frequency. Our next experiment extends our model beyond binary classification to do so.

5.4 Weighted Rank Analysis

In this experiment we evaluate our model's ability to rank, sort and compare relative execution frequencies among paths. Rather than a binary judgment of "high frequency" or "low frequency", we use the classifier probability estimates for high frequency to induce an ordering on paths that estimates the real ordering given by the dynamic path counts.

We use the same method for data set construction, training and testing as before, but evaluation requires a new metric. Intuitively, if the dynamic path counts indicate that four paths occur in frequency order $A > B > C > D$, then the prediction $A > B > D > C$ is more accurate than the prediction $D > C > B > A$. Kendall's tau [14] is distance metric between ranked orderings that formalizes this intuition; we use it as our evaluation criterion.

Kendall's tau is equivalent to the number of bubble sort operations or swaps needed to put one list in the same order as a second normalized to the size of the lists. Kendall's tau is a distance metric for ordered lists, and smaller numbers indicate a stronger correlation. We use a normalized Kendall's tau to facilitate comparison between lists of different sizes. Lists in reverse order (i.e., exactly wrong) would score 1.0, a random permutation would score 0.5 on average, and perfect agreement (i.e., the same order) yields 0.0.

Figure 7 shows the Kendall's tau values for each benchmark individually as well as the overall average. For the particular case of inter-method hot paths on the `jess` benchmark we obtain a perfect score of 0.0 (i.e., exactly predicting the real relative order). On average, our model yields a tau value of 0.30 for intra-method and 0.23 for inter-method paths. We view numbers in this range as indicative of a strong correlation. Note that small perturbations to the ordering, especially among the low frequency paths, are not likely to represent a significant concern for most clients. Static branch prediction [5], program specialization [29], and error report ranking [21] are examples of client analyses that could make use of ranked paths. In the next section we show how selecting "top paths" from this ranking is sufficient to quickly characterize a large percentage of run time program behavior.

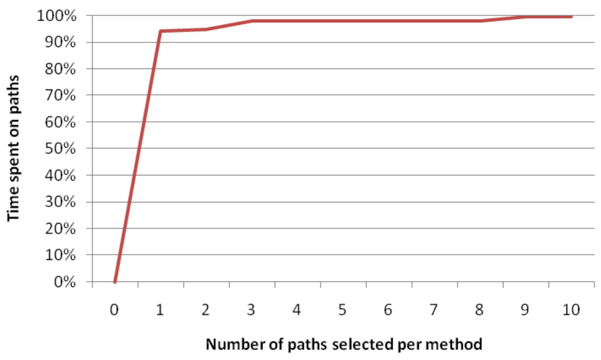

Figure 8. Percentage of total run time covered by examining the top paths in each method.

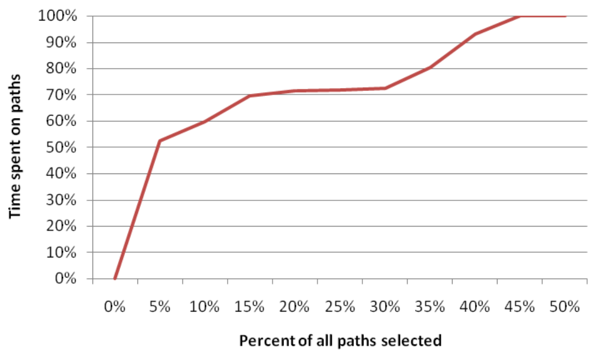

Figure 9. Percentage of the total run time covered by examining the top paths of the entire program.

6 Path Frequency and Running time

We have shown that our static model can accurately predict runtime path frequency. In this section, we use our model for hot paths to characterize program behavior in terms of actual running time. In essence, we evaluate our model as a "temporal coverage metric."

To acquire timing data we re-instrument each benchmark to record entry and exit time stamps for each method to nanosecond resolution. This adds an average of 19.1% overhead to benchmark run time, as compared to over 1000% for our full dynamic path enumeration. We combine timing data with path enumeration and calculate the total time spent executing each path.

We again use our model to sort static program paths by predicted frequency. We then select the X most highly ranked paths and calculate the percentage of the total runtime spent along the selected paths. We experiment with selecting X paths from each method in Figure 8, and X paths from the entire program in Figure 9.

Our results indicate a strong correlation between our static model for path frequency and run time program behavior. In the intra-method case, selecting the single "hottest path" from each method is sufficient to capture nearly 95% of program run time. Recall that, on average, these benchmarks have over 12 paths per method. Furthermore, in the inter-method case selecting 5% of program paths is sufficient for over 50% temporal coverage on average. This analysis indicates that our model is capable of characterizing real program run time behavior, with accuracy that is likely to be sufficient for improving or enabling many types of program analyses. For example, a feedback-directed compiler optimization that is too expensive to be applied to all paths could be applied to the single hottest path predicted by our model and have a high probability of improving common-case performance.

7 Path Frequency for Branch Prediction

In this section we demonstrate the robustness of our approach by coercing our static path frequency prediction model into a static branch predictor. A static branch predictor is one that formulates all predictions prior to running the program. Intuitively, to make a branch prediction, we ask our model to rank the paths corresponding to the branch targets and we choose the one with the greatest frequency estimate.

To achieve this comparison, we start by constructing the control flow graph for each method. At the method entry node, we enumerate the list of paths for the method and sort that list according to our intra-method frequency estimate. We then traverse the control flow graph (CFG) top-down, at every node maintaining a set of static paths that could pass through that node.

Consider, for example, the code fragment below:

```
1  a = b;
2  c = d;
3  if (a < c) {     /* choice point */
4    e = f;
5  } else {
6    g = h;
7  }
```

If execution is currently at the node corresponding to line 2, the future execution might proceed along the path 1-2-3-4 or along the path 1-2-3-6. At each branching node we perform two actions. First, we partition the path set entering the node into two sets corresponding to the paths that conform to each side of the branch. Second, we record the prediction for that branch to be the side with the highest frequency path available. In the code fragment above, we will query our model about the relative ordering of 1-2-3-4 and 1-2-3-6; if 1-2-3-6 were ranked higher we would record the prediction "not taken" for the branch at line 3. Finally, because

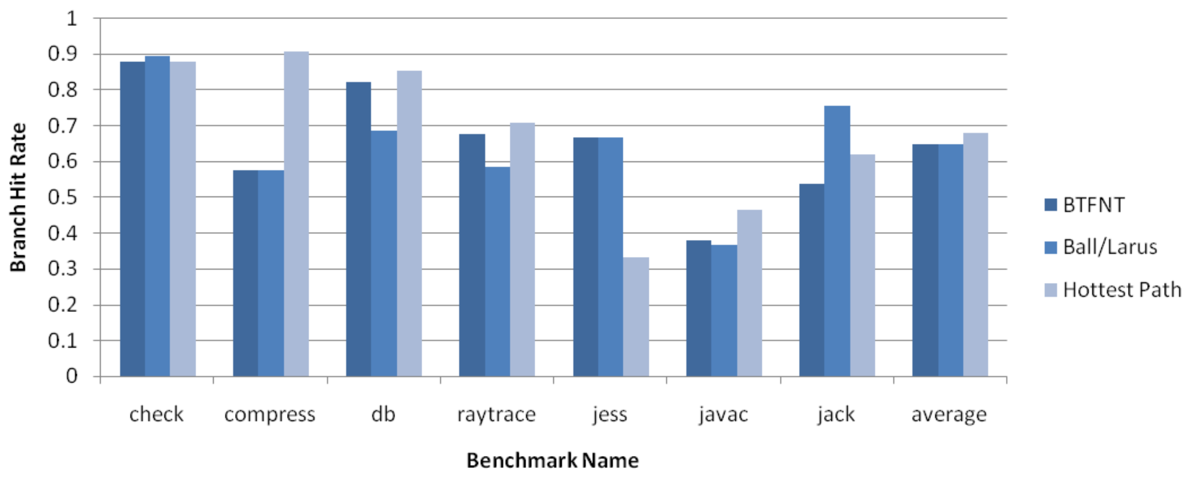

Figure 10. Static branch prediction hit rate. BTFNT is a baseline heuristic; Ball/Larus represents the state-of-the-art; Hottest Path corresponds to using our static path frequency prediction to predict branches.

our path model is acyclic, at branches with back edges we follow the standard "backward taken / forward not taken" (BTFNT) heuristic, always predicting the back edge will be followed. This scheme is based on the observation that back edges often indicate loops, which are typically taken more than once.

We compare our branch prediction strategy to a baseline BTFNT scheme and to an approach proposed by Ball and Larus [5] based on a set of nine heuristics such as "if a branch checks an integer for less than zero... predict the branch on false condition." We compare the predictions from these three strategies against the ground truth of running each benchmark. We measure success in terms of the branch "hit rate" (i.e., the fraction of time the static prediction was correct). Figure 10 reports the hit rate for each benchmark individually as well as the overall average.

Static branch prediction is a difficult task. Even with an optimal strategy a 90% hit rate is typically maximal [5], and no one strategy is perfect for all cases. For example, Ball and Larus perform well on `jack` because their heuristics can be frequently employed. Our technique performs quite well on `compress` because of its relatively large number of branches with two forward targets. On average, however, a static branch predictor based on our static model of path frequencies outperforms common heuristics and previously-published algorithms by about 3%. This result serves to indicate that our frequency model can be robust to multiple use cases.

8 Model Implications

We have shown that our model is useful for estimating the runtime frequency of program paths based solely on a set of static features. In this section we present a brief qualitative analysis of our model and discuss some of its possible implications. We conduct a singleton feature power analysis to evaluate individual feature importance as well as a principle component analysis (PCA) to access the degree of feature overlap.

8.1 Feature Power Analysis

In Figure 11 we rank the relative predictive power of each feature and indicate the direction of correlation with runtime frequency. For example, the single most predictive feature in our set is the number of statements in a path. On average, more statements imply the path is *less* likely to be taken.

This data is gathered by re-running our F-score analysis on each benchmark as before, but training on only a single feature and measuring the predictive power of the resulting model. We then normalize the data between 0 and 1. For correlation direction, we compare the average value of a feature in the high and low frequency classes. We prefer a singleton analysis to a "leave-one-out" analysis (where feature power is measured by the amount of performance degradation resulting from omitting the feature from the feature set) due to the high degree of feature overlap indicated by a PCA, where we find that 92% of the variance can be explained by 5 principle components.

The singleton feature power analysis, while not a perfect indicator of the importance of each feature, does offer some

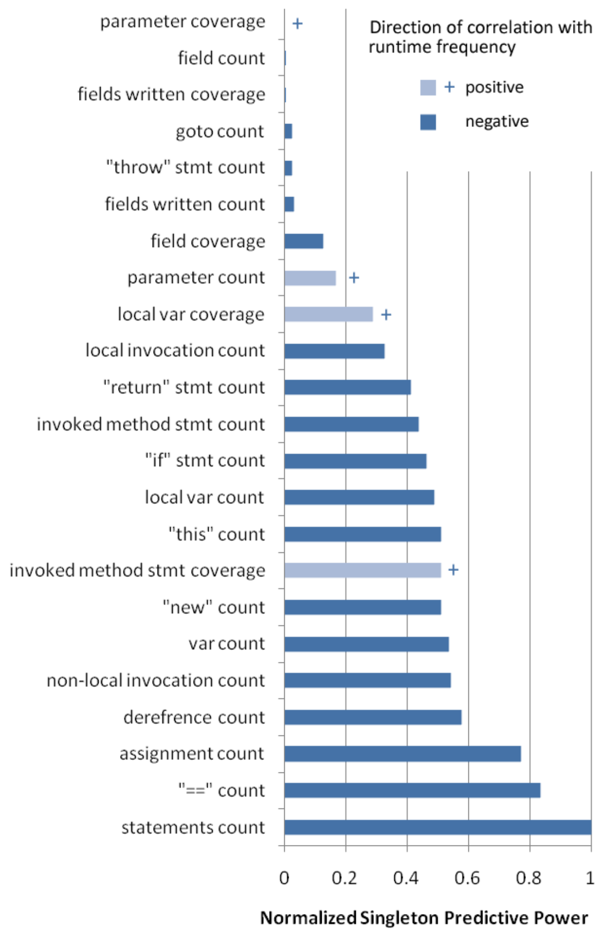

Figure 11. Relative predictive power of features as established with a singleton analysis.

qualitative insight into our model. For example, the relatively high predictive power of statement counts, particularly assignment statements, would seem to support our hypothesis that paths with large changes to program state are uncommon at runtime. We observe much weaker predictive power when training only on class field metrics, suggesting that updates to local variables are also important for uncommon paths. On the other hand, local variable coverage correlates with frequent paths: since variable updates correlate with uncommon paths, our model bears out our intuition that common paths read many local variables but do not update them. Features characterizing path structure, such as "if" and "return" count, show modest predictive power compared to features about state updates.

8.2 Threats To Validity

One threat to the validity of these experiments is overfitting — the potential to learn a model that is very complicated with respect to the data and thus fails to generalize. We mitigate that threat by never testing and training on the same data; to test one program we train on all others. Our results may fail to generalize beyond the benchmarks that we presented; we chose the SPEC benchmarks as indicative examples. Finally, our results may not be useful if our notion of a static intra-class path does not capture enough essential program behavior. Since our output in formulated on these static paths, a client analysis must make queries in terms of them. Our success at building a branch predictor on top of our path frequency analysis helps to mitigate that danger; while some program behavior is not captured by our notion of static paths, in general they form a natural interface.

9 Conclusion

We have presented a descriptive statistical model of path frequency based on features that can be readily obtained from program source code. Our static model eliminates some of the need for profiling by predicting runtime behavior without requiring indicative workloads or testcases. Our model is over 90% accurate with respect to our benchmarks, and is sufficient to select the 5% of paths that account for over half of the total runtime of a program. We also demonstrate our technique's robustness by measuring its performance as a static branch predictor, finding it to be more accurate on average than previous approaches. Finally, the qualitative analysis of our model supports our original hypothesis: that the number of modifications to program state described by a path is indicative of its runtime frequency. We believe that there are many program analyses and transformations that could directly benefit from a static model of dynamic execution frequencies, and this work represents a first step in that direction.

References

[1] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *Principles of Programming Languages*, pages 4–16, 2002.

[2] G. Ammons and J. R. Larus. Improving data-flow analysis with path profiles. *SIGPLAN Not.*, 33(5):72–84, 1998.

[3] M. Arnold, S. J. Fink, V. Sarkar, and P. F. Sweeney. A comparative study of static and profile-based heuristics for inlining. In *Workshop on Dynamic and Adaptive Compilation and Optimization*, pages 52–64, 2000.

[4] T. Baba, T. Masuho, T. Yokota, and K. Ootsu. Design of a two-level hot path detector for path-based loop optimizations. In *Advances in Computer Science and Technology*, pages 23–28, 2007.

[5] T. Ball and J. R. Larus. Branch prediction for free. In *Programming language design and implementation*, pages 300–313, 1993.

[6] T. Ball and J. R. Larus. Efficient path profiling. In *International Symposium on Microarchitecture*, pages 46–57, 1996.

[7] T. Ball and J. R. Larus. Efficient path profiling. In *International Symposium on Microarchitecture*, pages 46–57, 1996.

[8] T. Ball, P. Mataga, and M. Sagiv. Edge profiling versus path profiling: the showdown. In *Principles of programming languages*, pages 134–148, 1998.

[9] P. Berube and J. Amaral. Aestimo: a feedback-directed optimization evaluation tool. In *Performance Analysis of Systems and Software*, pages 251–260, 2006.

[10] M. Boyer, K. Skadron, and W. Weimer. Automated dynamic analysis of CUDA programs. In *Workshop on Software Tools for MultiCore Systems*, 2008.

[11] B. Calder, D. Grunwald, M. Jones, D. Lindsay, J. Martin, M. Mozer, and B. Zorn. Evidence-based static branch prediction using machine learning. *ACM Trans. Program. Lang. Syst.*, 19(1):188–222, 1997.

[12] T. Y. Chen, F.-C. Kuo, and R. Merkel. On the statistical properties of the f-measure. In *International Conference on Quality Software*, pages 146–153, 2004.

[13] E. Duesterwald and V. Bala. Software profiling for hot path prediction: less is more. *SIGPLAN Not.*, 35(11):202–211, 2000.

[14] E. J. Emond and D. W. Mason. A new rank correlation coefficient with application to the consensus ranking problem. In *Journal of Multi-Criteria Decision Analysis*, pages 17–28, Department of National Defence, Operational Research Division, Ottawa, Ontario, Canada, 2002. John Wiley Sons, Ltd.

[15] D. R. Engler, D. Y. Chen, and A. Chou. Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In *Symposium on Operating Systems Principles*, pages 57–72, 2001.

[16] R. V.-R. et. al. Soot - a java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.

[17] S. F. Goldsmith, A. S. Aiken, and D. S. Wilkerson. Measuring empirical computational complexity. In *Foundations of software engineering*, pages 395–404, 2007.

[18] S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof: a call graph execution profiler. *SIGPLAN Not. (best of PLDI '82)*, 39(4):49–57, 2004.

[19] R. Gupta, E. Mehofer, and Y. Zhang. Profile-guided compiler optimizations. In *The Compiler Design Handbook*, pages 143–174. 2002.

[20] G. Holmes, A. Donkin, and I. Witten. Weka: A machine learning workbench. *Australia and New Zealand Conference on Intelligent Information Systems*, 1994.

[21] T. Kremenek and D. R. Engler. Z-ranking: Using statistical analysis to counter the impact of static analysis approximations. In *Static Analysis Symposium*, pages 295–315, 2003.

[22] D. Krishnamurthy, J. A. Rolia, and S. Majumdar. A synthetic workload generation technique for stress testing session-based systems. *IEEE Trans. Softw. Eng.*, 32(11):868–882, 2006.

[23] J. R. Larus. Whole program paths. In *Programming language design and implementation*, pages 259–269, 1999.

[24] P. Mehra and B. Wah. Synthetic workload generation for load-balancing experiments. *IEEE Parallel Distrib. Technol.*, 3(3):4–19, 1995.

[25] M. C. Merten, A. R. Trick, C. N. George, J. C. Gyllenhaal, and W. Hwu. A hardware-driven profiling scheme for identifying program hot spots to support runtime optimization. In *International Symposium on Computer architecture*, pages 136–147, 1999.

[26] T. Mitchell. *Machine Learning*. McGraw Hill, 1997.

[27] A. Nanda and L. M. Ni. Benchmark workload generation and performance characterization of multiprocessors. In *Conference on Supercomputing*, pages 20–29. IEEE Computer Society Press, 1992.

[28] T. Reps, T. Ball, M. Das, and J. Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. *SIGSOFT Softw. Eng. Notes*, 22(6):432–449, 1997.

[29] U. P. Schultz, J. L. Lawall, and C. Consel. Automatic program specialization for java. *ACM Trans. Program. Lang. Syst.*, 25(4):452–499, 2003.

[30] K. Sen. Concolic testing. In *Automated Software Engineering*, pages 571–572, 2007.

[31] K. Sen and G. Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *Computer Aided Verification*, pages 419–423, 2006.

[32] A. Srivastava and A. Eustace. Atom: a system for building customized program analysis tools. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 196–205, New York, NY, USA, 1994. ACM.

[33] W. Weimer and G. C. Necula. Mining temporal specifications for error detection. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 461–476, 2005.

[34] W. Weimer and G. C. Necula. Exceptional situations and program reliability. *ACM Trans. Program. Lang. Syst.*, 30(2):1–51, 2008.