Using Bayesian Learning to Estimate How Hot an Execution Path is

Karim Ali
karim@uwaterloo.ca
CS886 - Bayesian Network Analysis
David R. Cheriton School of Computer Science
University of Waterloo

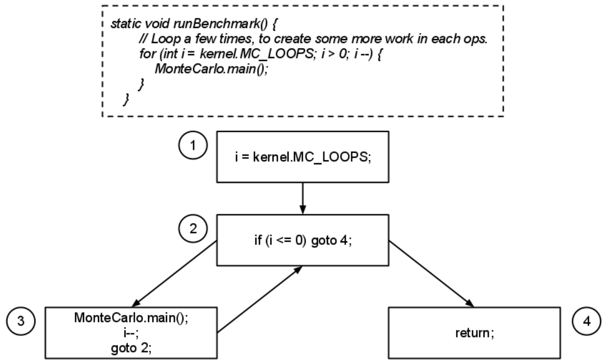
Abstract

Detecting commonly executed program paths (i.e. hot paths) is very essential for many program analyses. It is always advisable to perform code optimizations to those hot paths. Buse and Weimer [7] suggest an approach that uses static path information to build a Bayesian network that can infer the class (hot or cold) of a given path under the assumption that a hot path has minimal effect on the program state. In this project, we propose a modified version of this approach by enumerating static paths for each individual method. We then evaluate our model using the SPECjvm2008 [2] benchmarks. Our experiments show that our model can achieve a precision/recall tradeoff of 0.917/0.909. In the worst case, our model achieves a precision of 0.845 and a recall of 0.735.

1. Introduction

In program analysis, it is advised to perform code optimizations to the paths that have higher likelihood of occurring in a given run of the program. This prioritization process can be tackled by identifying hot paths (i.e. paths of high frequency of execution) and cold paths (i.e. paths of low frequency of execution). The process of identifying hot paths is called *Program/Code Profiling*. The importance of program profiling 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 (i.e. hot paths) [7]. Not only is hot path identification useful for determining cost/benefit ratios for certain compiler optimizations [6], it is also essential for maintenance [13], data-flow analysis [4], and debugging [10]. There are two approaches to perform program profiling: static, and dynamic.

Dynamic program profiling requires the instrumentation of the original source code. In other words, real-life program traces are recorded and gathered by the program to be analyzed by program developers/maintainers later. Due to



```
static void runBenchmark() {
    // Loop a few times, to create some more work in each ops.
    for (int i = kernel.MC_LOOPS; i > 0; i --) {
        MonteCarlo.main();
    }
}
```

Figure 1. An example of a control flow graph for a the *rubBenchmark* method of *SciMark* benchmark from the *SPECjvm2008* benchmarks [2].

the lack of of real-life program traces, the general technique that has been adapted to many domains [12, 17, 9, 14, 15] is to synthetically generate the workloads. Such workloads might be useful in carrying out software stress tests, but have not been shown as indicative of actual usage [7].

In the context of static program profiling, no workloads are generated but rather execution paths are enumerated based on edge information from the control flow graph of various methods in the code. An example of a control flow graph is shown in Figure 1. Consequently, additional information is required to classify the execution frequency of a given enumerated execution path. Otherwise, one can only assume that all execution paths are equally likely to be followed in a given program trace. Buse and Weimer [7] suggests that more information about a static execution path can be obtained by analyzing the relationship between the relative frequency of some features of a path and its effect on program state. The authors show that commonly executed code blocks (i.e. blocks with hot paths) usually do not affect the program state much. Therefore, it is safe to assume that paths with small impact on program state are more likely to be hot paths.

In this project, we plan to use static execution path information (e.g. number of if statements, fields written to) to construct a Bayesian network that is capable of predicting the probability that a given path is hot. We adopt the same approach suggested by Buse and Weimer [7] to classify execution paths after applying two main modifications to it. First, we statically enumerate execution paths based on the control flow graph of each method individually rather than the control flow graph of each individual class. We adjust the feature description model of the execution path accordingly. We then cluster the statically enumerated paths using a simple k-means clusterer into two clusters (hot and cold). The results from the clusterer are used as the ground truth. Afterwards, we train a Bayesian network using one training dataset, and test four different test datasets against it. Second, we evaluate our model using the more recent SPECjvm2008 [2] benchmarks as opposed to the obsolete SPECjvm98 [3] benchmarks.

The rest of the report is structured as follows. We discuss related work in Section 2. In Section 3, we provide an in depth discussion for the implementation details of our model. We then provide our the experimental results of our model in Section 4 followed by discussion and some proposed future work in Section **??**. Finally, we conclude the report in Section 5.

2. Related Work

Static branch prediction is the closest work to the idea presented in this project. The purpose of static branch prediction is to statically classify branch edges in a given execution path as *taken*, or *not-taken* according to some criteria. Information about branch edges can be gathered from workloads (profile-based branch prediction) or inferred statically from the control flow graph information (program-based branch prediction).

Ball and Larus [5] present a program-based branch predictor that performs well for a large and diverse set of programs written in C and Fortran. The core of their model is based on simple heuristics for predicting non-loop branches, which dominate the dynamic branch count of many programs. A more recent study shows that those heuristics achieve an average of 70% prediction rate with 33% miss rate [8]. The same study further extends those heuristics by using a richer static feature set and employing neural network learning techniques. The authors were able to decrease the miss rate to an average of 21% but this came at the cost of decreasing the prediction rate to an average of 50%.

Focusing only on branch edge information is not useful for hot path identification. Analyzing the whole execution path provides a richer set of information than individual branch edge information [7]. Thus, it is more useful to design a path-based formal model for prediction. This is the option we opted for in this project.

Our work is inspired by the work of Buse and Weimer [7] where they present a technique for statically estimating the runtime frequency of program paths. The results of such technique can be used to support or improve many types of static program analyses. The authors report an accuracy of 90% for their model. However, they do not report on their miss rates. Moreover, they do not provide any details as to how they built the Bayesian network used for prediction or the precision and the recall values recorded for each experiment. In addition, the authors use workloads automatically generated from the SPECjvm98 benchmarks [3] to get the ground truth for path execution frequency to train and evaluate their model. This defies one of their major goals, attempting to model indicative execution frequency by avoiding the usage of automatically generated workloads. In this work, we only depend on information gathered from statically enumerated execution paths and do not make use of automatically generated workloads.

3. Model Construction

In any given imperative programming language (e.g Java), each method can be represented as a control flow graph (cfg). This cfg defines all possible execution paths among which only one will be followed in a given execution scenario. The goal of this work is to describe a feature model for a given static execution path (hereafter, path). Based on that model, the path can be classified as either a hot path or a cold path. Our approach consists of four major steps: path description, path enumeration, dataset generation, and finally the construction of the Bayesian network used for testing. The current implementation of our approach supports Java, but the model still applies to other imperative languages.

3.1. Path Description Model

As mentioned earlier, we adopt a similar path description model to the one proposed by Buse and Weimer [7]. However, we apply some modifications to the path description model to comply with our definition of a path. We define a path as an execution path for a given method that can be statically enumerated from the control flow graph of that method. Consequently, we do not take into consideration features like *this* and *return statements* from their model [7] because these features do not reveal more information in the context of a method. Any given method has a *this* parameter and a *return statement*. Table 1 provides a full list of the features that are used to describe a path. A path is eventually represented by a vector of the counts of those features. Since we assume that hot paths induce minimal change in program state, the lower the values of the feature vector, the more probable it is that the path is hot.

Feature	Description
==	equal condition
new	any new expression
=	assignment statements
.	pointer dereferences
fields	class fields usage
fields written	class fields written to
invocations	all method invocations
goto	goto statements
if	if statements
locals	local variables usage
statements	total number of statements
throws	throw statements

Table 1. The set of features defining a path description model.

3.2. Path Enumeration

We use the Soot [16] APIs to analyze the input source code, and generate the control flow graphs for each method from the input classes. For a given method, we generate its control flow graph, then apply a modified version of the depth-first search algorithm to the graph to enumerate potential paths. Algorithms 1 and 2 show a pseudo-code for the algorithm used to enumerate the paths for a given control flow graph, and paths starting at a given block in the graph respectively. We assume that a path through a loop construct represents all paths that take the loop once or more. This prevents the generation of an unbounded number of potential paths. Each path is represented as a stack of basic blocks, where each basic block is sequence of source code statements.

Algorithm 1 Enumerate potential paths for a given CFG

```
Input: cfg for a method
Output: potential execution paths

process(cfg) {
  Initialize $potentialPaths$ = {}
  for $head$ in $cfg.getHeads()$ do
    Initialize $path$ = {}
    Initialize $headPaths$ = {}
    $search(head, path, headPaths)$
    $potentialPaths.addAll(headPaths)$
  end for
}
```

Once all potential execution paths are generated, we analyze each one to compute its feature vector (i.e. the vector of count of each feature for this path). This task is performed by our feature extraction tool that takes a given execution path and uses the Soot [16] APIs to extract those features. The feature vector of the path gets updated accordingly. Finally, we store the generated feature vectors in a format that can be recognized by the data mining software we are using, WEKA [11]. We chose to store the feature vectors in ARFF format, an ASCII text file that describes a list of instances sharing a set of attributes [1].

Algorithm 2 Search for paths starting at a given CFG block

```
Input: current block, current path, and the result so far
Output: potential paths starting at a given CFG block

search(block, path, result) {
  if stuck in a cycle then
    $return$
  end if
  $path.push(block)$ {path is a stack of blocks}
  if block is a tail then
    $result.add(path)$
    $path.pop()$ {to get other paths}
    $return$
  end if
  for $successor$ in $block.getSuccessors()$ do
    $search(successor, path, result)$
  end for
  $path.pop()$ {no path found}
}
```

3.3. Dataset Generation

Generating an indicative dataset of execution paths is not an easy task. One has to develop (or gather) real life applications and analyze them to compile such a dataset. SPECjvm2008 [2], fortunately, already provides a benchmark suite for measuring the performance of a Java Runtime Environment (JRE), containing several real life applications and benchmarks (e.g. derby, javac, crypto, xml) focusing on core Java functionality. This dataset was compiled to be used instead of the obsolete SPECjvm98 [3] used in [7].

As with any learning algorithm, we had to divide our dataset into a training set, and a test set. We limit our training set to all paths generated from control flow graphs that have at most 10 nodes (i.e. basic blocks). On the other hand, we generate four test sets each consists of all paths generated from control flow graphs that have, 10-20, 20-30, 30-40, and 40-50 nodes respectively. We could not go beyond that number of nodes per control flow graph due to the limited RAM we have on our machine (1 GB). This was also another reason why we had to split the dataset into 5 subsets, one used as a training dataset, and the others as test datasets.

Since WEKA [11] has to know the ground truth about both the training set and the test sets, we first generate all the datasets with their class attribute missing (Table 2). We

CFG Node Count	No. of Instances	Runtime (msecs)	Codename
1 - 10	2087	619	Training
11 - 20	2667	1130	Test-20
21 - 30	6730	2342	Test-30
31 - 40	71772	47461	Test-40
41 - 50	143078	92895	Test-50

Table 2. The characteristics of the datasets with the class attribute missing.

then feed each of our 5 datasets to WEKA as a training set and apply simple k-means clustering algorithm to them, ignoring the missing class attribute. The output of each run results in two clusters per dataset, each cluster has an average count per feature (centroid of the cluster). The cluster with the smaller centroid values is considered to be the hot paths cluster, the other one is the cold paths cluster (based on our assumption in Section 1). Table 3 summarizes the feature counts of the cluster centroid for what we identified as the hot paths cluster for each dataset.

The feature counts of the centroid of each dataset provide us with an average threshold for the feature counts for hot paths in that dataset. Therefore, we have to properly classify the paths as hot or cold in each dataset based on the respective centroid values. Therefore, we had to go back to our previously generated datasets, and label each path accordingly. We consider the resulting class as the ground truth that will be used afterwards to evaluate our model. The pseudo-code for our path classification algorithm is shown in Algorithm 3. To classify a path, we loop over all the feature counts of the given path. If the count of a feature is greater than the average count of the feature for the corresponding cluster centroid. We then increment the probability that this path is not hot by $\frac{1}{numberoffeatures}$. Finally, the path is a hot path if this probability is less than or equal to 0.5, it is cold otherwise.

Algorithm 3 Classify a given execution path

Input: path, and cluster centroid feature vector
Output: class (hot, or cold)

```
label(path, centroid) {
  Initialize $probability$ = 0
  for $feature$ in $path.getFeatures()$ do
    if path.feature.count() $\geq$ centroid.feature.count() then
      $probability$ += 1 / no. of features
    end if
  end for
  if probability $\leq$ 0.5 then
    return hot
  else
    return cold
  end if
}
```

3.4. Bayesian Network Construction

We use WEKA to build our Bayesian network based on the training set only (i.e. paths from control flow graphs with at most 10 nodes). The network is built using K2 (a greedy Bayesian network construction algorithm) with maximum number of parent nodes = 5 and a simple estimator with prior probability = 0.5. The average time taken to build our Bayesian network is 0.22 seconds. Due to the settings of WEKA, the Bayesian network has to be rebuilt each time a new test set is supplied. Figure 2 shows our Bayesian network using Training as the training dataset and Test-20 as the test dataset.

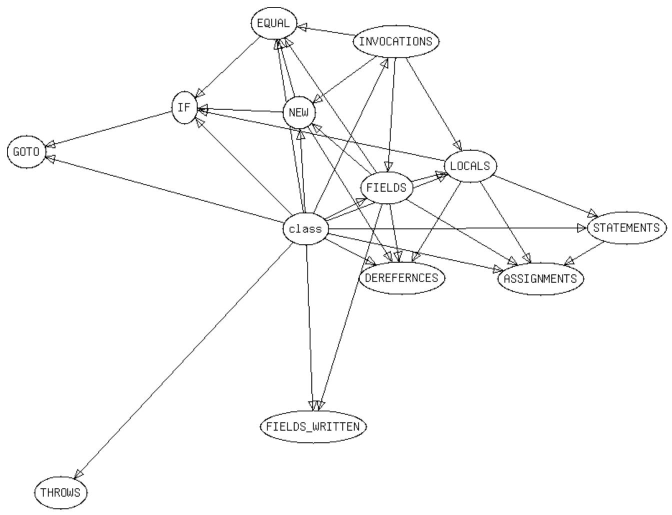


Figure 2. Bayesian network using Training dataset, and Test-20 test dataset.

4. Model Evaluation

We performed 4 experiments to evaluate the predictive power of our model. In each experiment, we used the same training dataset (Training) and a different test dataset. All experiments model our problem as a classification problem, i.e. label each path as *hot* or *cold*.

4.1. Experimental Setup

As mentioned earlier, we used the Soot [16] APIs to implement the algorithms described in Section 3, and WEKA [11] APIs and Explorer GUI for machine learning. We ran all our experiments on an HP laptop with Intel Centrino Duo 2.0 processor, 1GB RAM, running Ubuntu 10.10 (Maverick) with 2.6.3.23 kernel version.

4.2. Results

We evaluated our model based on 4 parameters:

- true positive rate: correctly classified paths (compared to our ground truth),

CFG Node Count	**1 - 10**	**11 - 20**	**21 - 30**	**31 - 40**	**41 - 50**
ASSIGNMENTS	3.714	16.9907	23.1365	69.3282	56.5917
DEREFERNCES	2.9033	4.0865	6.5968	12.4859	13.8164
EQUAL	0.142	1.9292	2.5739	5.554	13.8748
FIELDS	1.1788	2.0805	4.2011	9.7243	10.5063
FIELDS_WRITTEN	0.4753	0.2735	0.6598	0.7413	2.4874
GOTO	0.5498	7.7237	8.4264	13.8151	23.5838
IF	0.4308	6.5747	7.0163	10.756	19.88
INVOCATIONS	2.3236	12.4984	16.3752	74.8947	40.7381
LOCALS	10.4997	45.1786	58.4237	194.84	132.9246
NEW	0.4252	0.8422	1.7663	8.1021	2.6921
STATEMENTS	7.9081	30.7885	38.9947	120.8035	88.2042
THROWS	0.0334	0.025	0.0556	0.0238	0.0088
No. of Iterations	13	20	5	9	9

Table 3. Hot paths clusters centroids.

- false positive rate: misclassified paths (compared to our ground truth),
- precision: the fraction of correct instances among those that the algorithm classifies as hot paths (i.e. a measure of exactness or fidelity), and
- recall: the fraction of correct instances among all instances that are actually hot (i.e. a measure of completeness).

Figure 3 shows a plot for the precision values for the various test datasets. Test-40 yields the best precision value (0.917), while the lowest precision value achieved by our model (0.829) is obtained using Test-30. On the other hand, Figure 4 shows a plot for the recall values for various test datasets. The best recall value (0.909) is achieved using Test-40, while the lowest recall value achieved by our model (0.735) is obtained using Test-20. Figure 5 compares the precision and recall values for each test dataset. Table 4 reports the F-measure values for each test dataset.

We were very intrigued by the fact that Test-40 achieved the best possible tradeoff between precision and recall (Figure 5), and the highest F-measure (Table 4). This observation needs more thorough investigation to be able to arrive at the reasons why this specific test dataset achieves such high scores. However, our intuition is that for Test-20 and Test-30 the model probably overfits the Training dataset which consists of only 2087 instances as opposed to 2667 and 6730 instances in the case of Test-20 and Test-30 respectively (Table 2). When we use a test dataset of much larger number of instances (Test-40 has 71772 instances), the overfitting problem is no longer there due to the big difference between the number of instances in the training and the test datasets. However, this conclusion requires carrying out more experiments to be validated.

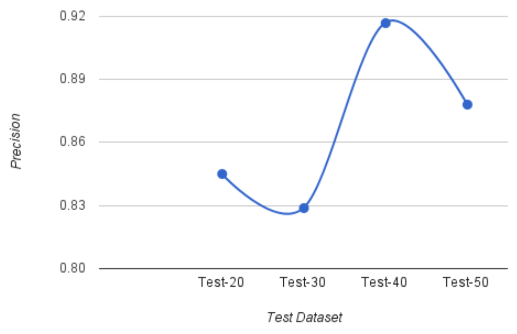


Figure 3. Precision values for the various test datasets.

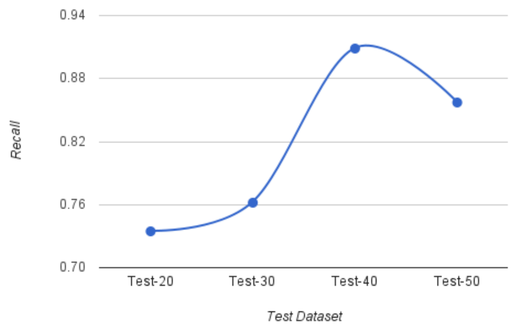


Figure 4. Recall values for the various test datasets.

4.3. Threats to Validity

There are two threats to validity for the experiments we carried out. First, overfitting is inevitable when the sizes of the training data and the test data are very small and close to each other (e.g. the case of Training and Test-20). However,

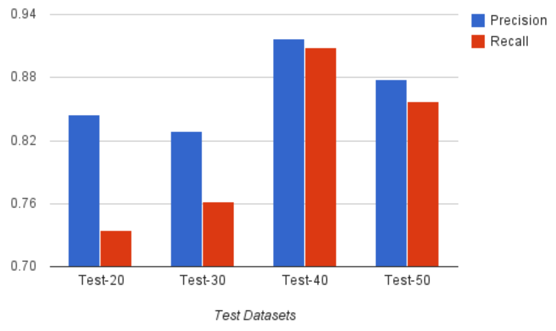


Figure 5. Precision vs Recall values for the various test datasets.

Test Dataset	F-Measure
Test-20	0.734
Test-30	0.729
Test-40	0.876
Test-50	0.798

Table 4. The F-measure for the various test datasets.

we tried as much as possible to overcome this challenge by selecting our datasets to be mutually exclusive of each other. It remains to test our model, using our current training set, against different set of benchmarks to experimentally prove that the overfitting problem has been resolved. Second, we compare the resulting classification of our model to the ground truth that we obtain from our clustering process in the dataset generation step (Section 3.3). This ground truth might not match the real-life ground truth, i.e. the average counts of hot paths from real program traces. Eliminating this threat to the validity of our model requires carrying out more experiments with real-life program traces.

5. Conclusion and Future Work

In this project, we used static execution path information (e.g. number of if statements, fields written to) to construct a Bayesian network that is capable of predicting the probability that a given path is hot. We modified the approach suggested by Buse and Weimer [7] to classify execution paths after applying two main modifications to it. First, we enumerate static paths for each individual method. Second, we evaluate our model using the more recent SPECjvm2008 [2] benchmarks as opposed to the obsolete SPECjvm98 [3] benchmarks. Our experiments have shown that our model achieves the best precision/recall tradeoff when using Training as the training dataset, and Test-40 as the test dataset. In the worst case (i.e. using Test-20 as the test dataset), our model achieves a precision of 0.845 and a recall of 0.735. In the future, it would be useful to investigate other benchmarks to check whether our model suffers from overfitting or not. Finally, we are curious to know the effect of using the ground truth obtained from real-life program traces as opposed to our clustering technique.

References

[1] Attribute-Relation File Format (ARFF). "http://www.cs.waikato.ac.nz/~ml/weka/arff.html".

[2] SPECjvm2008 Benchmarks. "http://www.spec.org/jvm2008/".

[3] SPECjvm98 Benchmarks. "http://www.spec.org/jvm98/".

[4] G. Ammons and J. Larus. Improving data-flow analysis with path profiles. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 72–84. ACM, 1998.

[5] T. Ball and J. Larus. Branch prediction for free. *ACM SIGPLAN Notices*, 28(6):300–313, 1993.

[6] C. Boogerd and L. Moonen. On the Use of Data Flow Analysis in Static Profiling. In *Source Code Analysis and Manipulation, 2008 Eighth IEEE International Working Conference on*, pages 79–88. IEEE, 2008.

[7] R. Buse and W. Weimer. The road not taken: Estimating path execution frequency statically. In *Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 144–154. IEEE Computer Society, 2009.

[8] B. Calder, D. Grunwald, M. Jones, D. Lindsay, J. Martin, M. Mozer, and B. Zorn. Evidence-based static branch prediction using machine learning. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(1):188–222, 1997.

[9] G. Casale, A. Kalbasi, D. Krishnamurthy, and J. Rolia. Automatically generating bursty benchmarks for multitier systems. *ACM SIGMETRICS Performance Evaluation Review*, 37(3):32–37, 2010.

[10] T. Chilimbi, B. Liblit, K. Mehra, A. Nori, and K. Vaswani. HOLMES: Effective statistical debugging via efficient path profiling. In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pages 34–44. IEEE, 2009.

[11] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. Witten. The WEKA data mining software: An update. *ACM SIGKDD Explorations Newsletter*, 11(1):10–18, 2009.

[12] D. Krishnamurthy, J. Rolia, and S. Majumdar. A synthetic workload generation technique for stress testing session-based systems. *IEEE Transactions on Software Engineering*, pages 868–882, 2006.

[13] T. Reps, T. Ball, M. Das, and J. Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. *Software EngineeringESEC/FSE'97*, pages 432–449, 1997.

[14] K. Sen. Concolic testing. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 571–572. ACM, 2007.

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

[16] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot-a Java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999.

[17] A. van Hoorn, M. Rohr, and W. Hasselbring. Generating probabilistic and intensity-varying workload for Web-based software systems. *Performance Evaluation: Metrics, Models and Benchmarks*, pages 124–143, 2008.