CrystalBall: Statically Analyzing Runtime Behavior via Deep Sequence Learning

Stephen Zekany*, Daniel Rings*, Nathan Harada*, Michael A. Laurenzano*†, Lingjia Tang*†, Jason Mars*†
{szekany, drings, nharada, mlaurenz, lingjia, profmars}@umich.edu — *University of Michigan - Ann Arbor, MI
†Clinc, Inc. - Ann Arbor, MI

***Abstract*—Understanding dynamic program behavior is critical in many stages of the software development lifecycle, for purposes as diverse as optimization, debugging, testing, and security. This paper focuses on the problem of predicting dynamic program behavior *statically*. We introduce a novel technique to statically identify hot paths that leverages emerging deep learning techniques to take advantage of their ability to learn subtle, complex relationships between *sequences of inputs*. This approach maps well to the problem of identifying the behavior of *sequences of basic blocks* in program execution. Our technique is also designed to operate on the compiler's intermediate representation (IR), as opposed to the approaches taken by prior techniques that have focused primarily on source code, giving our approach *language-independence*. We describe the pitfalls of conventional metrics used for hot path prediction such as accuracy, and motivate the use of Area Under the Receiver Operating Characteristic curve (AUROC). Through a thorough evaluation of our technique on complex applications that include the SPEC CPU2006 benchmarks, we show that our approach achieves an AUROC of 0.85.**

I. Introduction

Runtime behavior analysis is crucial to virtually all aspects of the software lifecycle. It forms the basis of numerous compiler optimizations [36], facilitates understanding software vulnerabilities, improves program testing and debugging, and informs the software development process [24].

Prior work has shown that programs spend most of their execution time along a small percentage of the possible program paths [10]. Analysis of these *hot paths* is one aspect of runtime behavior that is particularly relevant to software construction, debugging, and optimization. Path profiling can be used in branch prediction [50], trace formation [18], and basic block placement optimizations [4]. Testing can take advantage of path execution analysis to estimate code coverage.

Hot path identification and analysis has been and continues to be a focus of research [7] [10] [23] [28] [39] [43]. Dynamic profiling commonly uses instrumentation to calculate path execution information [18] [38]. However, dynamic profiling requires the program to be run in its entirety, introducing several significant hurdles:

1) **Representative environments –** dynamic profiling requires identifying and using inputs and environments representative of production during profiling. Unfortunately, identifying such inputs can be time consuming, and may be difficult to achieve in practice.
2) **Processing overhead and churn –** because a program must be executed completely before a full profile can be generated, the computational expense and latency can prohibit leveraging dynamic profiles for software with a rapid development cycle.
3) **In for a penny, in for a pound –** dynamically characterizing paths for a subset of functions or paths is nearly as computationally expensive as characterizing paths for the whole program. This makes it difficult to quickly obtain path behavior information for a subset of a program's paths or functions.

978-1-5090-3508-3/16/$31.00 © 2016 IEEE

An alternative approach to dynamic profiling is to *statically profile* – to predict runtime properties of the program before the program runs. A path in program execution is defined as a particular sequence of instructions in a program's control flow graph, and these instructions are fixed before runtime. The hypothesis underlying static path profiling is that a useful range of behavioral program characteristics are latent within these instructions [37]. The higher the quality of static analysis performed, the better the prediction of runtime behavior [16]. If the complex, subtle relationships between the static instructions in a program can be understood, the dynamic behavior of the program can be predicted with high accuracy.

Static profilers can overcome the disadvantages of dynamic profilers by exchanging precise, measured execution data for rapidly and easily-generated predictions. However, state-of-the-art static predictors typically rely on hand-crafted features and heuristics [9] [19]. Such features and heuristics are often language-specific, their characteristics being hard to generalize to other languages.

This paper describes a novel approach to static hot path prediction. Our approach is motivated by two insights. The first is that compiler intermediate representation (IR) makes a particularly effective vehicle for performing hot path analysis. IR may contain both high-level semantic information and low-level operation details (e.g., integer and float ops), offering a rich set of data that can be useful for predicting program runtime behavior. IR has been shown to be an effective platform for prior static analysis research [8] [12]. Moreover, the control flow in IR typically bears a very close resemblance to the program's machine code, meaning that predictions regarding the control flow of the IR map directly to predictions about the program's machine code. Finally, IR supports many different languages and platforms. Popular open source compilers such as LLVM or GCC support dozens of source code languages and target platforms [35] [5]. Techniques that can be applied to IR therefore have a high degree of generality, as they can be applied to many different languages out-of-the-box and without language-specific modification.

Our second insight is that emerging data-driven machine-learning is suited to the problem of hot path prediction for two reasons. First, the amount of training data available is enormous; individual SPEC CPU2006 benchmarks have path

counts numbering in the tens and hundreds of billions (this point is elaborated in Section V). Second, these machine learning techniques are capable of learning extremely sophisticated and subtle relationships in information. In particular, recurrent neural networks (RNNs) are designed to perform *sequence learning* – identifying the relationships between *sequences of inputs* [26] – a capability that can be particularly useful in understanding the behavior of *sequences of blocks* along a program's control flow graph. Such models can be designed to incorporate both the *order* and *content* of instructions.

Leveraging these insights, this paper presents *CrystalBall*, a novel technique for identifying hot paths at compile time. *CrystalBall* relies solely on IR, both in training and in inference, and thus achieves language and platform-independence. *CrystalBall* uses an RNN, a sequence learning approach, as its underlying prediction mechanism. This has the advantages of (1) being able to learn and identify complex relationships along sequences of basic blocks in a program's control flow graph and (2) doing so with no human involvement in selecting and hand-crafting input features. *CrystalBall* is implemented as a pass in the compiler, able to perform hot path prediction on individual paths, functions or the entire program with the inclusion of a simple command line switch.

The specific contributions of our work are as follows:

- **Path Prediction Metrics –** we describe and motivate the use of Area Under the Receiver Operating Characteristic curve (AUROC), a significantly more rigorous metric than prior work for comparing the effectiveness of hot path prediction mechanisms (Section III).
- **State-of-the-art Hot Path Prediction –** we present *CrystalBall*, an open source[1], novel approach to hot path prediction that
  1) leverages compiler IR, achieving language and platform independence, and
  2) is based on a sequence learning approach rather than hand-engineered feature selection (Section IV).
- **Hot Path Indicators –** we tested several classification models in the design process of *CrystalBall*. We also provide insight into features of static program code that are indicative of hot and cold paths (Section V).

Using a suite of 21 test programs in three languages, *CrystalBall* achieves an average AUROC of 0.85.

II. Background

Finding frequently executed code in an arbitrary program is a problem of interest. Program profiling, for example, aims to identify frequently-executed code. Work has also been done to find efficient ways to enumerate execution paths, such as Ball-Larus path profiling [10].

A. Dynamic vs. Static Profiling

Dynamic profiling executes a program and uses runtime data to optimize the program at compile-time. Profiling is typically performed dynamically, by executing the program code with a set of input data that is representative of real system usage. However, dynamic profiling has several limitations.

First, it is difficult to perform full dynamic profiling on a single path of execution or a subset of execution paths without executing the entire program, even when techniques such as hybrid slicing are used to reduce the overhead of analysis computation [31].

Second, for many applications, it is also difficult to generate inputs representative of real execution environments. The selection of inputs and execution environment is crucial to realize high quality dynamic profiling. In addition, a particular dynamic profiling instance is guaranteed to be correct only for the given input data [14]. Proper inputs must therefore cover a breadth of workloads, as well as error conditions for the program to be profiled accurately.

Finally, the runtime overhead of performing dynamic profiling can be expensive. Dynamic profiling necessitates re-executing the entire program each time the application is changed and subsequently compiled (irrespective of the size and scope of the change).

A static approach to infer the dynamic behavior of application code has the potential to mitigate these limitations.

[1]https://github.com/szekany/crystalball

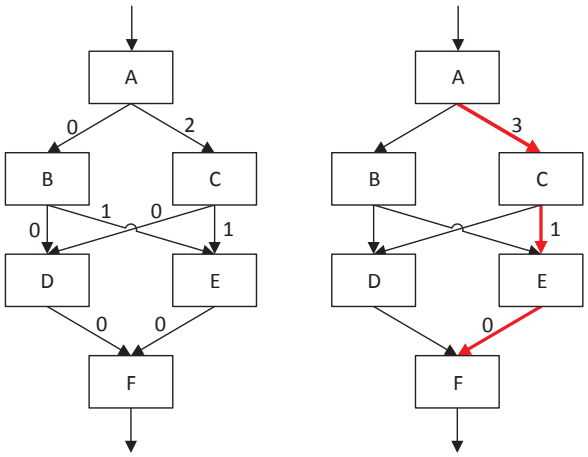

Fig. 1: Example of function path enumeration using Ball-Larus algorithm (left - edge weights between basic blocks, right - example of path reconstruction)

B. Hot Path Definition

Although various definitions of *hot path* have been used throughout prior works, generally a hot path is defined as a sequence of instructions that are frequently executed [46], often further quantified by a measure such as percentage of execution relative to all paths [23] or percentage of runtime [19]. We define a hot path as a path that is executed more than a threshold n.

C. Ball-Larus Path Profiling

While program profiling techniques can measure many kinds of program components, we choose to focus on paths. Unlike basic blocks or edges profiling, paths provide a more complete picture of the execution of a program, because a program is empirically a series of basic blocks executed in a certain order.

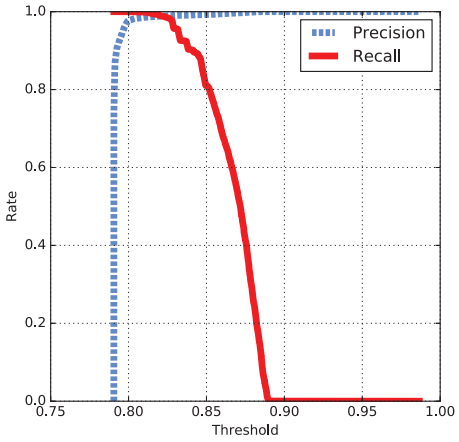

Fig. 2: Example precision and recall rates across a range of threshold values

A path is a sequence of basic blocks executed in a function, typically from the function's entry point to exit point. We utilize the Ball-Larus method for path enumeration and construction [10]. In Ball-Larus, each function is first converted to a directed acyclic graph (DAG), where nodes are represented by basic blocks and the back edges (any edge pointing to a preceding edge in depth-first search) have been removed. All edges are then assigned weights such that the sum of all edge weights for any given path is unique.

This unique sum of weights for a path is also the path identification number. To reconstruct a path, the algorithm sets a variable containing the path value equal to the path number and identifies the entry node. At each node, the algorithm chooses the edge with the largest weight not exceeding the path value. The edge weight is decremented from the path value, and the process is repeated until reaching the exit node.

An example of path reconstruction is shown in Figure 1. This figure shows a simple function represented as a DAG, with basic blocks as nodes. The DAG on the left shows the edge weights, and the DAG on the right shows an example of path reconstruction. The algorithm begins with a path value equal to the path number (in this example, the path number is 3). From basic block A, the algorithm selects the edge with the highest weight not exceeding the current path value. This is the edge to block C with a weight of 2. Having taken the edge, the path value is decremented by the edge weight and is now 1. We select the next edge not exceeding the path value, which is the edge to block E. From block E, we must take the edge of weight 0 to F. The path value is now zero, and we have enumerated the path.

Ball-Larus path profiling is lightweight and efficient, as it requires only the addition of a counter as part of each edge to trace program execution. It is important to note that program execution is not required to enumerate all possible paths.

III. Path Prediction Metric

Hot path prediction is a classification problem. However, measuring the quality of this classification is difficult, because metrics such as accuracy are problematic due to severe class imbalance.

A. Limitations of Current Metrics

Since the vast majority of paths are never executed, it would be trivial to achieve an extremely high accuracy value by classifying each path as cold. For example, if the ratio of hot paths to cold paths was 1:999, we could predict every path to be cold and achieve an accuracy of 99.9%. (In fact, using reference data for profiling, we find that fewer than 1 in 1,000,000 paths are hot in our set of test programs.) Yet, despite achieving high accuracy, this statistic reveals nothing about program behavior or the characteristics of any particular hot path. Therefore, we find that accuracy is an insufficient metric to capture both the salient points and nuances of hot path classification.

Typical machine-learning classifiers work by assigning probabilities to items. A binary classifier must therefore choose a threshold to map probabilities to classifications (such as hot or cold path). The threshold for designating a path as hot or cold will also greatly affect traditional binary classification metrics like precision, recall, and false positive rate. These are defined as follows:

$$\text{precision} = \frac{\text{true positives}}{\text{true positives + false positives}}$$

$$\text{recall} = \frac{\text{true positives}}{\text{true positives + false negatives}}$$

$$\text{false positive rate} = \frac{\text{false positives}}{\text{false positives + true negatives}}$$

For hot path prediction, a threshold that is too low will yield especially poor precision scores because there is so much potential for false positives in a set of paths that are almost all cold.

However, too high a threshold will yield a poor recall because there are not enough true positives to absorb even a handful of false negatives. These effects can be seen in Figure 2, which shows precision and recall for a binary classifier with large class imbalance. The recall drops precipitously as the threshold is increased due to the number of false negatives, despite precision remaining high. At high threshold values (x = .9, for example), many true positives are not included. However, at lower threshold values (x = .75) too many items are incorrectly included as positive.

Additionally, the penalty for a false positive classification versus a false negative differs depending on the exact application of the classification. For example, it may be better to have a handful of false positives than to miss an actual hot path when running a hot path-based compiler optimization. In this case, a low precision score is more acceptable because the marginal penalty for each additional false positive is low.

F_1 score has been used to evaluate hot path classification models in the past [19]. The F_1 score is the harmonic mean of precision and recall [21]: $F_1 = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}}$

However, F_1 scores suffer from the same pitfalls as precision and recall with respect to an arbitrary threshold [42]. It also

considers precision and recall with equal influence. A low threshold will produce a low precision, which will result in an inferior F_1 score. The arbitrarily low threshold causes poor precision, and the F_1 score is lowered significantly, even though precision is not nearly as important as recall in many cases.

As an example, consider a medical screening that tests for a dangerous disease. The highest F_1 is likely to select a tradeoff between precision and recall, while we may care much more about recall (it is better to accidentally send a patient to the doctor than miss the disease entirely). Additionally, the optimal tradeoff between precision and recall may vary on a per-patient or per-hospital basis.

Given these pitfalls, we propose that a superior metric should aim to:

- take into account all possible thresholds, and
- give useful measures of model quality in the presence of severe class imbalance.

B. AUROC

Classifiers for data sets from other domains with similar class imbalance typically use metrics based on the Receiver Operating Characteristic curve (ROC) [17] [25]. Medical diagnosis classifiers must pick out the few patients with a rare disease from a vast number of healthy patients. Website vulnerability classifiers must predict which sites have a high risk of being compromised in the near future from the sea of low-risk websites. In each of these domains, the penalty for a false positive is very different from the penalty for a false negative. A false positive medical diagnosis could subject a patient to expensive and life-disrupting treatment unnecessarily, while a false negative could result in an untreated patient dying. A vulnerable website could be compromised if the webmaster is not notified of the vulnerability, but too many false warnings will result in the webmaster ignoring them. The choice of threshold of the vulnerabilities reported to the webmaster is therefore important. In order to choose a good threshold, it is necessary to have insight into the rate of true positives and false positives that will result.

An ROC curve is generated by plotting the true positive rate (recall) versus the false positive rate across the entire range of possible thresholds. Figure 3 shows an example of a perfect recall (at y = 1), random recall (where x = y), and an example of real data from our hot path classifier. The y-value plotted at (x = 0.5) is the true positive rate at the threshold required to give a false positive value of 0.5. The ROC curve gives insight into the effects of picking any threshold. A suitable threshold can then be chosen based on the desired trade-off between true positives and false positives.

Evaluation of the classifier as a whole in a single metric, independent of the threshold, can be performed using the Area Under the ROC Curve (AUROC, or simply AUC) [17].

A perfect ROC curve, shown in green in Figure 3, would always have a true positive rate of 1 and a false positive rate of 0, except when the threshold is 0 and everything is classified as positive. Therefore, the only two points on the plot would be at (0, 1), when the threshold is anything besides 0, and (1, 1), when the threshold is 0. The area under such an ROC curve would be 1.

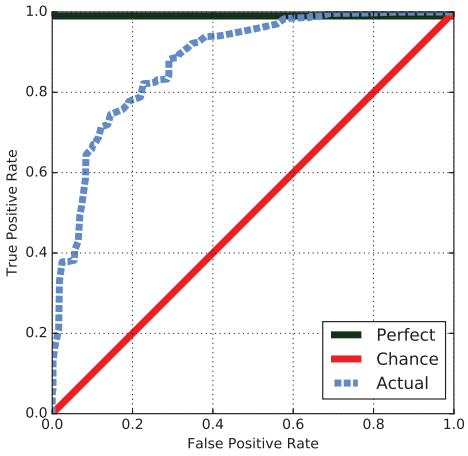

Fig. 3: Example ROC curves showing a perfect classifier (has 100% true positive rate for all values of false positive rate), a classifier that is no better than random chance (false positive rate = true positive rate), and an example of a classifier that falls in between. (Note that it is possible for a classifier to perform worse than random chance, but then the classification can simply be inverted.)

A classifier based on random chance would be expected to have a true positive rate that equals the false positive rate for any threshold. This would mean that for any decrease in threshold, there would be an increase in just as many false positives as the increase in true positives. Such an ROC curve is shown in red in Figure 3 and yields an AUROC of 0.5. An AUROC $a < 0.5$ means that the positive and negative classes are mislabeled and should be switched, after which will result in an AUROC of 1 - $a > .5$.

Another way to interpret the AUROC is as the likelihood that given a random positive instance and a random negative instance, the classifier will score the positive instance higher than the negative instance. In this way, AUROC is an excellent metric for evaluating classifiers independent of thresholds.

IV. CrystalBall

The *CrystalBall* system has three major components, which will be elaborated in the following section. First, the train and test program feature vectors are collected from the compiler's IR. Next, the RNN model trains on the train program feature data. Finally, the model infers on test program data and is evaluated. The system components are diagrammed in Figure 4.

A. Training and Data

CrystalBall's hot path prediction can be applied to a program with the inclusion of a single compiler pass.

1) Obtaining Ground-Truth: Training examples for classification models require that their actual classification be known. *CrystalBall* obtains the ground truth of hot path via program execution with path profiling instrumentation. The inserted instrumentation tracks each path's execution count. These execution counts are used as the ground truth for whether each particular path is hot or cold.

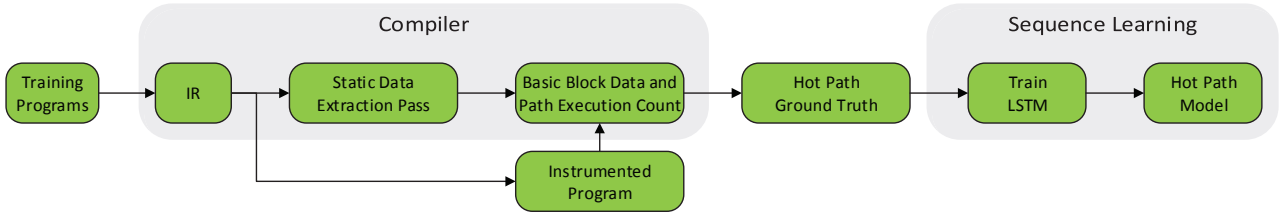

Fig. 4: CrystalBall overview

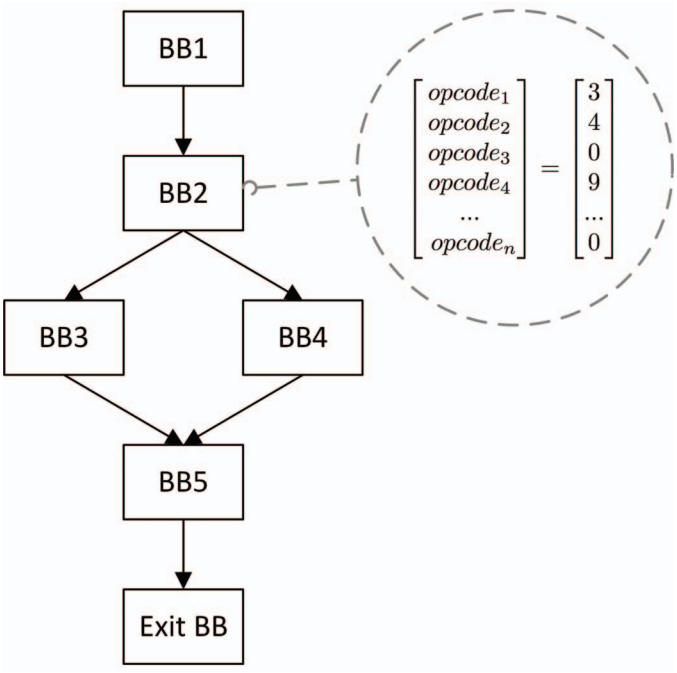

Fig. 5: Basic block feature vector extraction

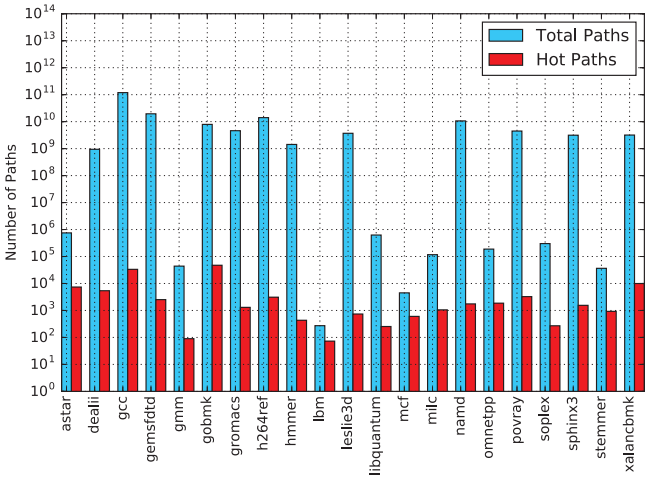

Fig. 6: Hots paths and total paths for the SPEC CPU2006 benchmarks and Sirius suite kernels

2) Static Data Extraction: *CrystalBall* describes features of the IR code at the basic block level. These code characteristics may be relevant to path behavior and should be quantifiable, although their relationship with path behavior may extend beyond what is obvious to human perception.

The *CrystalBall* compiler pass performs static analysis on individual basic blocks in the program. A path is treated as a sequence of basic blocks. For each basic block, we obtain a feature vector of a count of opcode types. The feature vector is therefore a count of each type of instruction contained in the basic block. This process is shown in Figure 5.

3) Path Sampling: The number of paths in a program does not scale with the size of the program, but rather with the complexity of the control flow graph (CFG) of each function. A function with a long case statement, for example, will have a higher number of basic blocks than a function that simply performs calculations without any conditional statements. The more conditional statements, the more paths exist. For example, two control flow graphs with 20 basic blocks and a thousand paths each, when connected by a single edge, would contain one million paths.

This kind of large, highly connected function presents a particular challenge for training a model. Several SPEC CPU2006 programs have functions with over one billion paths, and *gcc* has over 100 billion paths. Using every single path of these programs to train the neural network model requires too much time and memory on today's hardware. We mitigate this by imposing a maximum number of paths from each function used for training and testing. Due to their scarcity, hot paths in a function are always included. Figure 6 shows the number of hot paths compared to the number of total paths in SPEC CPU2006 programs and Sirius kernels. In all but one case, the number of cold paths exceeds the number of hot paths by at least an order of magnitude, and typically several orders of magnitude.

We attempted to scale the number of paths extracted from each function based on the logarithm of the number of paths. This method produced poor results, even when the total number of paths exceeded the standard value. We therefore chose to sample a standard number of paths equally from every function. Any function with fewer paths will be sampled in its entirety. We sample 2000 cold paths from each function due to the limitation of system memory.

4) Splitting Training and Testing Data: Splitting training and testing data is important for building a valid classification model. The training data should capture the feature-label relationships while avoiding manmade correlations between the training and testing data.

A common approach to splitting training and testing data sets is to pool all information from all benchmarks together, and proceed to split the collected information into 30% as testing data vs 70% as training data [48]. Neural network models also require a small validation set. The validation data is evaluated after each training epoch to signal the model to stop training before overfitting the training data.

However, considering the strong similarities between individual path data collected from the same program, we use leave-one-program-out testing, illustrated in Figure 7. This prevents paths in the same program from being present in both the training and testing sets. Such a scheme mirrors the application of hot path classifiers. A compiler is not statically given the execution counts of half the paths in a program to use as a model in predicting the execution counts of the remaining paths. It is given a model based on a number of different programs to make inferences on a new, entirely unseen program. The training and testing scheme for a path classifier should reflect this. A scheme that mixes paths of the same program between training and testing data gives the model an unrealistic advantage. If we had used such a scheme, our results would likely have appeared better, but would have no bearing on real-world applications.

During training, we take data collected from all but one of the benchmark programs as the training data, while using the data collected from the remaining program as the testing data. An additional program is removed from the training data to be used for validation. This process is repeated using every benchmark program as the testing data once.

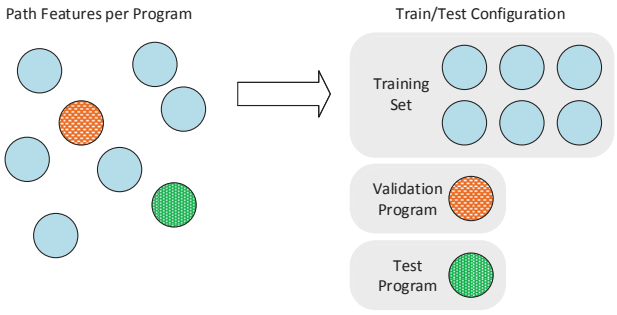

Fig. 7: Training set construction in *CrystalBall*. Different colors encode different programs (test and validation programs are not included in training).

B. Sequence Learning Model

In recent years, the growing amount of available data combined with widespread adoption of general purpose graphics processing units (GPGPUs) has resulted in a resurgence of neural networks, particularly for deep learning, where many non-linear layers and complex architectures are used to automatically learn features from data [8]. Sequence learning has benefited heavily from this paradigm, facilitating dependency modeling too complex or tedious for humans to generate manually [29] [44] [30] [40].

1) Recurrent Neural Network: The task of predicting hot paths lends itself naturally to this model. Given that a path is an ordered sequence of basic blocks, a model that can use this sequential nature should be particularly effective. This is exactly the advantage that recurrent neural networks (RNNs) provide. RNNs do not only output a classification for the input data, they provide a classification at each step in the input sequence *and* use the results of the previous steps to influence the result of the current step [41]. This ability to remember context allows the RNN to better classify much more nuanced inputs. For example, if a basic block with many allocation instructions occurs towards the beginning of a path, it might suggest the creation of variables in preparation for frequently executed computation. Whereas a basic block with similar allocation instructions towards the end of a path might suggest variables created for debugging and error handling procedures, which are less likely. RNNs can account for this and for much more complex relationships related to the sequence of inputs beyond the intuition of humans.

Because conventional RNNs suffer from numerical instability over large numbers of timesteps, we implemented ours with long short-term memory (LSTM) [26] [33] units to mitigate vanishing and exploding gradients.

2) LSTM Architecture: We use a two-layer LSTM model with 256 cells per layer as illustrated in Figure 8. During training, a path, represented as a sequence of instruction count vectors, feeds into to the LSTM network for t timesteps, where t is the number of basic blocks in the path. At each timestep:

- **(a)** A basic block's instruction count vector feeds into all 256 cells of layer 1.
- **(b)** The output of the cells in layer 1 feeds into the input of the cells in layer 2.
- **(c)** Additionally, each cell uses its previous timestep's output as input into the current timestep.
- **(d)** The output of layer 2 feeds into a time distributed softmax, which compresses the output into a range from 0 to 1, representing a class label. In our case, a cold path is represented as 0 and a hot path is represented as 1.
- **(e)** This output is compared to whether the path is actually hot or not, which is our ground truth obtained through dynamic profiling.
- **(f)** The loss is calculated between the timestep's output and the ground truth, and we apply back propagation through time [49] to modify the cell weights to bring the model output in line with the ground truth.

After the model is trained on all paths in the training set, it is ready to be applied to the test program's paths. The inference results on the test paths are measured against the ground truth to determine how well the model would have performed inference on an uknown program.

C. Path Prediction

Throughout an inference, the instruction count vectors for each basic block feed into the network just like the training

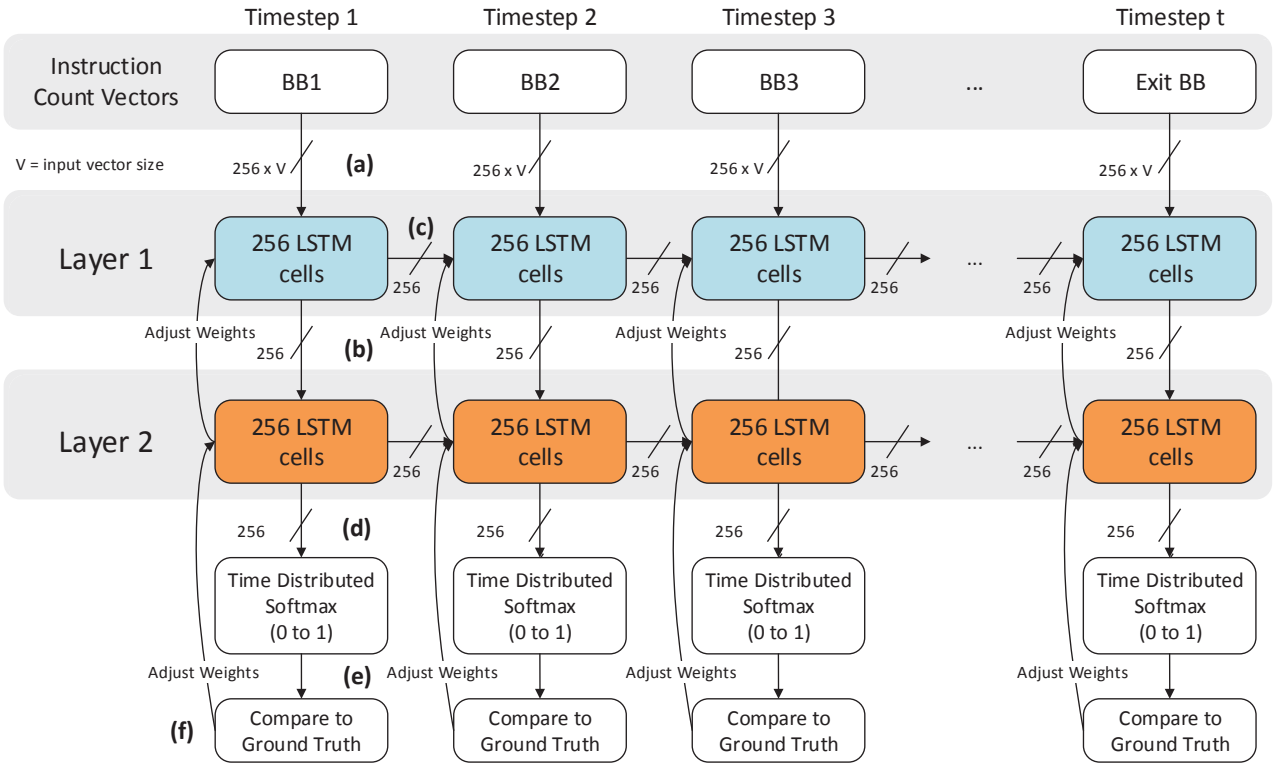

Fig. 8: Recurrent Neural Network architecture and training steps

process, with one basic block per timestep. However, loss calculation and reweighting do not occur. Only the softmax of the last timestep is used to determine the class label of the path being classified. The label indicates whether a path is likely to be executed at least n times. For our testing, we define n to be 1.

Any sequence of paths can feed into the model. The model outputs a probability representing whether each path is likely to be hot or not.

A user of *CrystalBall* can simply run the data extraction compiler pass on any program and get the predictive path profile for the program. These profiles can be used for further optimization and analysis of the program.

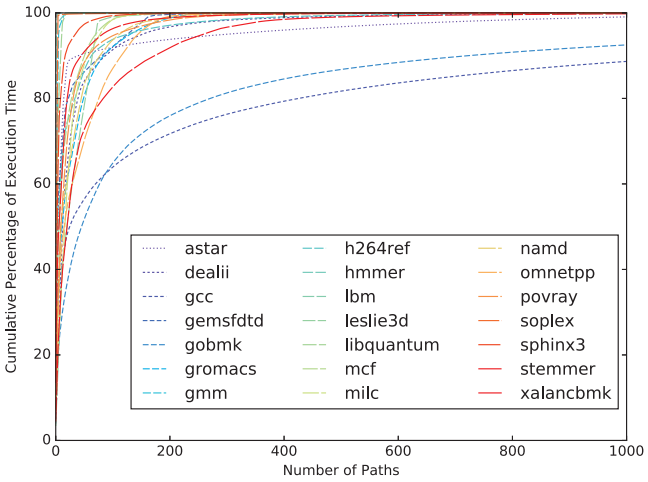

Fig. 9: Paths responsible for cumulative runtime

V. Evaluation

A. Methodology

1) Experimental Setup: Experiments were performed using a server with a 6-core Intel Xeon E5-2620 v2 2.10GHz processor, 256 gigabytes of DRAM, and 8 NVidia K40M GPUs. The server was running Ubuntu 14.04, LLVM 3.3, and Python 2.7.6. The neural network models were implemented in Python using Theano [13] and Keras [22].

2) Benchmarks: We chose benchmarks representative of real-world applications and workloads. Our primary benchmarks are from SPEC CPU2006, both integer and floating-point programs. These programs represent a wide array of real-world applications in C, C++, and Fortran. Additionally, we utilized two kernels from Sirius, an open-source end-to-end personal assistant pipeline: *gmm* and *stemmer* [32]. These

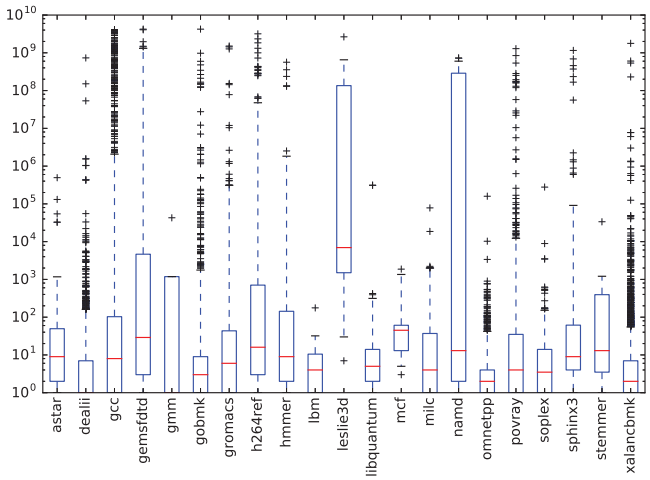

Fig. 10: Path counts per function

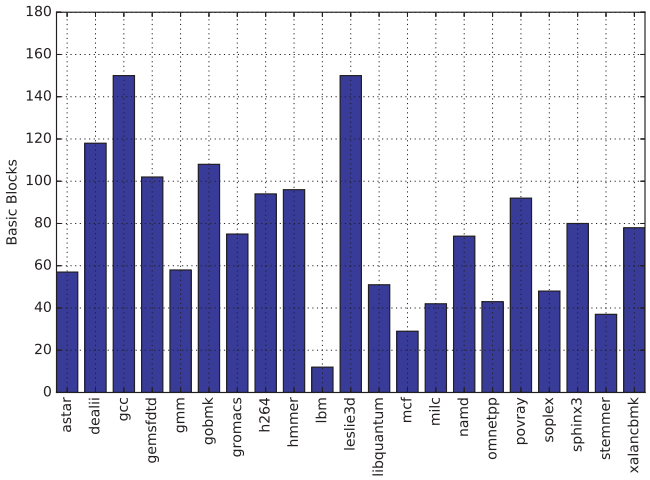

Fig. 11: Max path length by program

Program	Language	Lines of Code	Suite
astar	C++	5,842	SPEC CPU2006
dealii	C++	81,810	SPEC CPU2006
gcc	C	484,953	SPEC CPU2006
gemsfdtd	Fortran	11,580	SPEC CPU2006
gmm	C++	236	Sirius
gobmk	C	190,118	SPEC CPU2006
gromacs	C	72,220	SPEC CPU2006
h264ref	C	51,578	SPEC CPU2006
hmmer	C	35,992	SPEC CPU2006
lbm	C	1,155	SPEC CPU2006
leslie3d	Fortran	3,807	SPEC CPU2006
libquantum	C	3,454	SPEC CPU2006
mcf	C	2,685	SPEC CPU2006
milc	C	15,042	SPEC CPU2006
namd	C++	2,127	SPEC CPU2006
omnetpp	C++	14,200	SPEC CPU2006
povray	C++	140,892	SPEC CPU2006
soplex	C++	41,463	SPEC CPU2006
sphinx3	C	18,280	SPEC CPU2006
stemmer	C++	865	Sirius
xalancbmk	C++	296,028	SPEC CPU2006

TABLE I: Benchmark details

benchmark programs each have a relatively small codebase, but are representative of intelligent personal assistant server applications. Details of the benchmarks are provided in Table I. Each program was compiled to IR using Clang with the -O2 optimization flag.

The path profiler in LLVM does not support certain types of indirect branches [1] [2]. This affected functions in six of our benchmarks: *dealii*, *namd*, *omnetpp*, *povray*, *soplex*, and *xalancbmk*. Functions containing these instructions could not be instrumented and were therefore excluded from our training and testing protocols.

B. Hot Path Prediction

We dynamically profile each program with the SPEC-provided reference data. Figure 9 shows the execution time distribution for each of these benchmarks. Most programs have very few hot paths responsible for a large percentage of runtime. In fact, fewer than a thousand paths are responsible for over 80% of runtime for every test program except *gmm*.

Figure 10 shows the distribution of paths per function for each benchmark via whiskers to the 95th percentile, as well as the additional outliers. Several programs have only a few functions with a small number of paths. Others, such as *gcc*, have billions of paths.

The maximum path length varied by program, but was usually less than 150, as shown in Figure 11.

CrystalBall achieves an average AUROC of 0.85. Although we have pointed out the weaknesses of the metric, *CrystalBall* achieves an F_1 score of 0.82 using benchmarks in three different languages, learning of features, and a difficult and realistic training and testing scheme.

C. Comparison to Prior Work

We created a logistic regression (LR) model for hot path classification during the development of *CrystalBall* in order to compare our work to that of Buse and Weimer's static path classifier [19].

Buse and Weimer's LR model extracts features from Java source code, while ours uses features from LLVM IR. Our LR model is based as closely as we can approximate on the features used in their model. However, we do not include features that are specific to Java source code and cannot be extracted from IR, such as field coverage. Our aim is to provide the best representation of what their approach would look like if it were ported to a language-agnostic environment.

We also added IR-specific features to our model. For example, our model makes a distinction between integer and float operations. Additional feature engineering beyond this may result in slightly better results, but we found the marginal returns quickly diminished with each new IR feature.

We call our hand-crafted LR model the *B&W* model and use it as a benchmark to compare *CrystalBall*. We used the same set of programs to train and test the *B&W* model as with *CrystalBall*. The only other difference besides the features is the *B&W* training and testing contains one feature vector per path, while *CrystalBall* contains a feature vector for each basic block.

We find the *B&W* model achieves an average AUROC of 0.83, compared to *CrystalBall*'s average AUROC of 0.85. However, any LR model necessarily requires selection and

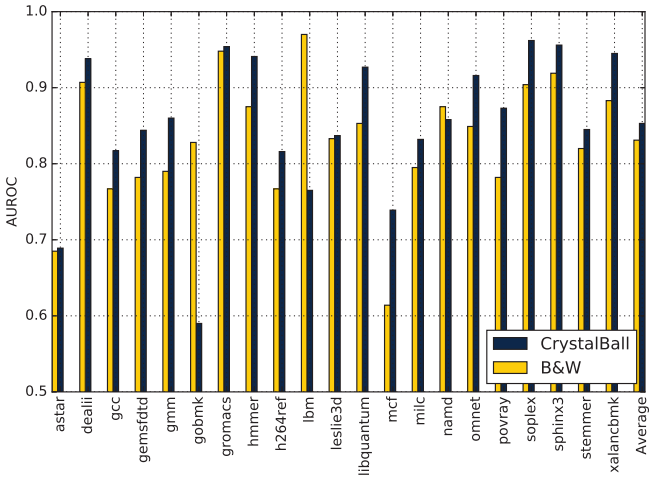

Fig. 12: AUROC by program

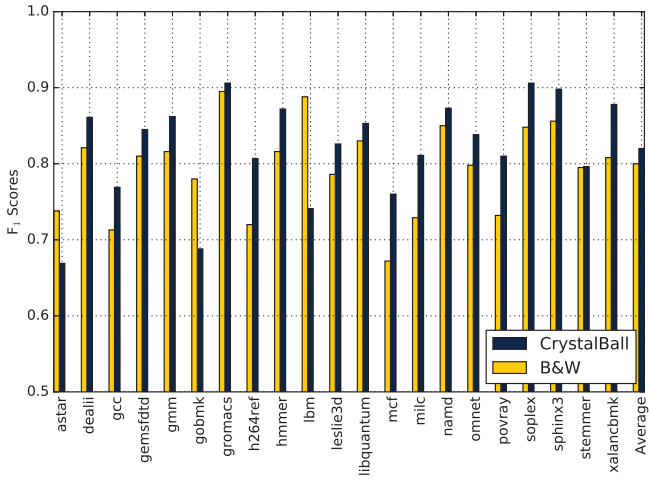

Fig. 13: F_1 scores by program

hand-engineering of features. This "top-down" approach is in direct contrast to CrystalBall's "bottom-up" deep learning of features from opcode frequency in sequential basic blocks.

D. Program-Level Analysis

Figure 12 presents a breakdown of the prediction performance of each model by program. There is considerable variance across programs in both models. This is a tendency of leave-one-out testing. A greater number of smaller, more distinct test sets is prone to higher variance. The variance could have been reduced by doing cross-fold validation instead, where a handful of programs are held out for testing in a group at once. This can also reduce testing time. However, leave-one-out testing is more reflective of real-world compiler scenarios. In exchange for longer testing time, we get data specific to how the model performs on each program, which provides more opportunity for detailed analysis.

The *B&W* model performs better than *CrystalBall* on a few benchmarks, particularly *lbm* and *gobmk*. *lbm* is one of the smallest programs by lines of code and number of paths, although *CrystalBall* is quite a bit better on *stemmer*, another very small program.

The LR models have a set of features to train on and have only one weight associated with each feature. LSTMs, as a much more powerful model, are capable of recognizing complex patterns that LR models cannot. In many of the larger programs, where very few paths are hot, the LSTM is capable of recognizing and interpreting the patterns that could represent a hot path.

CrystalBall performs well across the spectrum of benchmarks, only dropping below 0.7 AUROC in two benchmarks. We also calculated the F_1 score for each program, shown in Figure 13. Though we believe this metric is not as appropriate for hot path classifiers as AUROC, *CrystalBall* still compares favorably to the *B&W* model.

The LSTM is demonstrably capable of capturing the nuance of a broad spectrum of different program lengths, complexity, and class imbalance. This is in addition to the consideration that *CrystalBall* must learn relevant features from much more raw data than the *B&W* model. The feature learning, in combination with the advantage of a sequential input model, permits *CrystalBall* to encapsulate the subtleties of hot paths and removes the burden of feature creation.

E. Model Parameter Sensitivity

Many SPEC programs contain functions with large control-flow graphs. Enumerating all possible paths to build a training model would be computationally intractable. We to explore two possibilities to mitigating this problem: setting a maximum number of paths that would be enumerated per function and scaling the number of enumerated paths with the total number of paths of the function. We expected to obtain better results using scaling, but when we implemented a simple function to scale with the logarithm of the total number of paths and trained a new model using the same programs, the scaling model actually performed worse than the max-paths model even though it contained more paths overall.

We found AUROC in most cases increased as we increased the maximum paths threshold, as seen in Figure 14. Our model performance improved the more paths we extracted per training function, even though a lower number of maximum paths per function allows more programs to be included in the training set.

1) Training Set and Hardware Limitations: The training set was limited by the memory limit of our hardware configuration. While there are future opportunities to optimize the efficiency of *CrystalBall*, we were nonetheless limited by the size of the model we could feasibly train. We chose to train on the smallest 15 programs and exclude the larger programs. With improvements to the system, we could likely train the model even more accurately.

2) LSTM Model Parameters: We explored varying the number of retraining generations necessary to improve accuracy. We find that retraining provides a small improvement in AUROC value and F_1 score for certain programs. For our experiments, we elected to perform three training iterations.

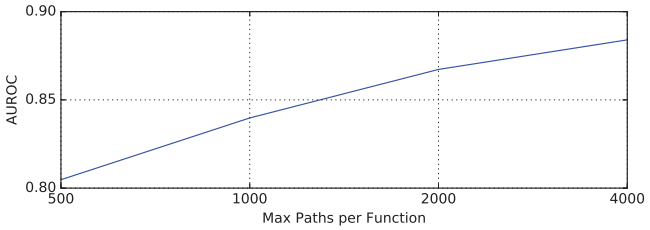

Fig. 14: Performance by max paths for *povray*

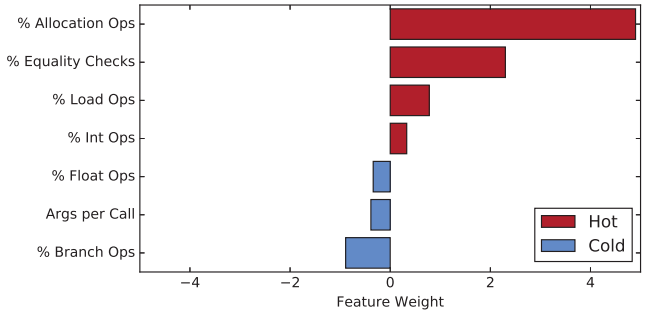

Fig. 15: Most important feature weights

F. Feature Importance

There are some interesting insights that can be gleaned from the logistic regression model described in Section V-C. The feature weights for the LR model correspond directly with how much influence a particular feature has on whether a path will be classified as hot or cold.

The three most influential features, shown in Figure 15, are the percentage of allocation operations, the percentage of equality checks, and the percentage of branch operations in a path. We note that many hot paths have high percentage of allocation operations. The allocation instruction designates memory on the stack frame to be used for the current function [3]. Intuitively, it makes sense that allocation instructions would predict a hot path, since allocation of memory generally precedes heavy computational operations where a storage location is required for intermediate or final values. In a function containing these operations, a path that does not allocate memory is probably handling exceptions or special cases instead of the typical input the function is intended to handle.

A path with more control flow complexity, represented by a higher branch operation count, normally indicates that a path is cold. However, this insight may be less valuable than it first appears. A function with higher control flow complexity will have many more paths than a function with lower complexity. If only a few of those paths are hot, there will be an abundance of cold paths in a high complexity function. This will make the ratio of cold paths to hot paths much greater for paths with many branch operations, thus resulting in the branch operation feature being heavily weighted towards cold paths.

VI. Related Work

Many areas of research are interested in analyzing and improving program performance via optimization at compile-time or runtime. Most of this research falls into two categories: dynamic profiling, which attempts to measure program performance at runtime and use this information to improve performance in some way (e.g. tools such as Pin and DynamoRIO) [18] [38], and static profiling, which attempts to predict program performance at compile-time. In general, static profiling attempts to define a scope of features to analyze and optimize, and then identifies those features within a given program and attempts to classify them via heuristics or more sophisticated models, such as neural networks. Static profiling has been used for determining inlining of functions [6], complexity [27], reliability [47], and precision [15].

A. Static Branch Prediction

A common way to perform static profiling is via static branch prediction [20]. Ball and Larus demonstrated a language-independent, low-overhead classification system for branches. They build a control-flow graph for the executable and profile to discover which branches are taken vs fall-through. Their model is built on the idea that non-loop branches consist of different features than loop branches. Additionally, they argue that predicting non-loop branches is very important for obtaining good results. Their model reveals that using simple heuristics, both loop and non-loop branches can be predicted sufficiently to be useful for code optimization, though not quite as accurately as dynamic profiling [9].

However, static branch prediction does not capture the sophistication of program execution in the way that branch tracing can. In the context of a control-flow graph, branch prediction deals simply with a specific edge, rather than examining an entire path through the graph. Most programs do not take the same path through a function on every execution, and binary classification of branches reduces real-world program execution to a simple binary choice. The path contains additional information that could be used to predict future program state. For example, a path that executes a specific prior basic block may be more likely to execute a different block later on. Finally, paths are likely to be related to each other. For example, one path may be identical to another except for the addition of some error handling code. We aim to identify these kinds of interactions in our work.

B. Hot Path Enumeration

With our work, we aim to find and leverage the additional information obtained by tracing path execution to identify whether a particular path is likely to be taken.

This work is inspired by prior works of static hot path prediction showing feasibility in detecting hot-paths using whole-program techniques [34] and other heuristics [23] [11]. Of particular interest is the work done by Buse and Weimer to apply simple machine learning algorithms to learn statistical models with features extracted from Java source code [19]. Our work differs from their work in three respects: (1) their model works only on programs written in Java while our approach is language-independent, operating on the intermediate representation (IR); (2) their approach uses features from function and variable calls while we use only the opcodes from

the IR; and (3) their model uses traditional machine learning techniques with hand-engineered features while ours leverages deep learning of sequences of individual basic blocks.

C. Loop Iteration Count

Tetzlaff and Glesner apply machine learning to the problem of predicting loop iteration count. They formulate the problem of predicting loop iteration count as a regression problem and identify static features within the loop that relate to runtime behavior [45]. Their machine-learning technique is to use random forest classification trees trained on these feature vectors to predict hot-spots within a program. By extracting loop-specific features from a wide set of programs, the model can learn the relationship between features and loop execution counts. Our work is more general as we focus on paths rather than loops.

VII. Conclusion

This paper examines static prediction of runtime behavior. Utilizing the ability of deep learning models to learn sophisticated relationships between *sequences of inputs*, we introduce a new approach to statically identify hot paths. Our system operates on the compiler's intermediate representation, offering *language-independence*. *CrystalBall* surpasses prior work in identifying hot paths with an AUROC of 0.85.

VIII. Acknowledgments

Many thanks to Zakaria Aldeneh and Yuan Shangguan for their work on our preliminary concept and research of deep learning-based hot path identification. We also thank our reviewers for their valuable time and comments. This research was supported by the National Science Foundation under grants CCF-XPS-1438996, CCF-SHF-1302682, CNS-CSR-1321047, and NSF-CAREER-1553485.

References

[1] "Address of Label and Indirect Branches in LLVM IR". http://blog.llvm.org/2010/01/address-of-label-and-indirect-branches.html.

[2] "Implementation of Path Profiling in the LLVM Infrastructure". http://llvm.org/pubs/2010-12-Preuss-PathProfiling.pdf.

[3] "LLVM Language Reference Manual". http://llvm.org/docs/LangRef.html#alloca-instruction.

[4] "LLVM's Analysis and Transform Passes". http://llvm.org/docs/Passes.html#block-placement-profile-guided-basic-block-placement.

[5] "Programming Languages Supported by GCC". https://gcc.gnu.org/onlinedocs/gcc/G_002b_002b-and-GCC.html.

[6] M. Arnold, S. Fink, V. Sarkar, and P. F. Sweeney. "A Comparative Study of Static and Profile-based Heuristics for Inlining". In *Proceedings of the Workshop on Dynamic and Adaptive Compilation and Optimization*, 2000.

[7] 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 (ACST)*, 2007.

[8] D. Babic and A. J. Hu. "Calysto: Scalable and Precise Extended Static Checking". In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2008.

[9] T. Ball and J. R. Larus. "Branch Prediction for Free". In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 1993.

[10] T. Ball and J. R. Larus. "Efficient Path Profiling". In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 1996.

[11] T. Ball, P. Mataga, and M. Sagiv. "Edge Profiling versus Path Profiling: The Showdown". In *Symposium on Principles of Programming Languages (POPL)*, 1998.

[12] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. "CoreDet: a Compiler and Runtime System for Deterministic Multithreaded Execution". In *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

[13] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio. "Theano: A CPU and GPU Math Expression Compiler". In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, 2010.

[14] D. Binkley. "Source Code Analysis: A Road Map". In *Future of Software Engineering (FOSE)*, 2007.

[15] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. "A Static Analyzer for Large Safety-Critical Software". In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 2003.

[16] C. Boogerd and L. Moonen. "On the Use of Data Flow Analysis in Static Profiling". In *Conference on Source Code Analysis and Manipulation*, 2008.

[17] A. P. Bradley. "The Use of the Area Under the ROC Curve in the Evaluation of Machine Learning Algorithms". *Pattern Recognition*, 1997.

[18] D. Bruening, T. Garnett, and S. Amarasinghe. "An Infrastructure for Adaptive Dynamic Optimization". In *International Symposium on Code Generation and Optimization (CGO)*, 2003.

[19] R. P. Buse and W. Weimer. "The Road Not Taken: Estimating Path Execution Frequency Statically". In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2009.

[20] B. Calder, D. Grunwald, M. Jones, D. Lindsay, J. Martin, M. Mozer, and B. Zorn. "Evidence-Based Static Branch Prediction Using Machine Learning". *Transactions on Programming Languages and Systems (TOPLAS)*, 1997.

[21] T. Y. Chen, F.-C. Kuo, and R. Merkel. "On the Statistical Properties of the F-measure". In *International Conference on Quality Software (ICQS)*, 2004.

[22] F. Chollet. "Keras", 2015. https://keras.io/.

[23] E. Duesterwald and V. Bala. "Software Profiling for Hot Path Prediction: Less is More". *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2000.

[24] M. E. Fagan. "Design and Code Inspections to Reduce Errors in Program Development". In *Pioneers and Their Contributions to Software Engineering*. 1976.

[25] P. A. Flach. "The Geometry of ROC Space: Understanding Machine Learning Metrics through ROC Isometrics". In *International Conference on Machine Learning (ICML)*, 2003.

[26] F. A. Gers, J. Schmidhuber, and F. Cummins. "Learning to Forget: Continual Prediction with LSTM". *Neural Computation*, 2000.

[27] S. F. Goldsmith, A. S. Aiken, and D. S. Wilkerson. "Measuring Empirical Computational Complexity". In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2007.

[28] S. L. Graham, P. B. Kessler, and M. K. Mckusick. "Gprof: A Call Graph Execution Profiler". In *Proceedings of the Symposium on Compiler Construction (CC)*, 1982.

[29] A. Graves. *"Supervised Sequence Labelling with Recurrent Neural Networks"*. Springer, 2012.

[30] A. Graves, A.-r. Mohamed, and G. Hinton. "Speech Recognition with Deep Recurrent Neural Networks". In *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2013.

[31] R. Gupta, M. L. Soffa, and J. Howard. "Hybrid Slicing: Integrating Dynamic Information with Static Analysis". *Transactions on Software Engineering and Methodology (TOSEM)*, 1997.

[32] J. Hauswald, M. A. Laurenzano, Y. Zhang, C. Li, A. Rovinski, A. Khurana, R. G. Dreslinski, T. Mudge, V. Petrucci, L. Tang, et al. "Sirius: An open end-to-end voice and vision personal assistant and its implications for future warehouse scale computers". In *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[33] S. Hochreiter and J. Schmidhuber. "Long Short-Term Memory". *Neural Computation*, 1997.

[34] J. R. Larus. "Whole Program Paths". In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 1999.

[35] C. Lattner and V. Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2004.

[36] M. A. Laurenzano, Y. Zhang, L. Tang, and J. Mars. Protean Code: Achieving Near-Free Online Code Transformations for Warehouse Scale Computers. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2014.

[37] Y.-T. S. Li and S. Malik. "Performance Analysis of Embedded Software Using Implicit Path Enumeration". In *Proceedings of the Design Automation Conference (DAC)*, 1995.

[38] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. "Pin: Building Customized

Program Analysis Tools with Dynamic Instrumentation". In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[39] M. C. Merten, A. R. Trick, C. N. George, J. C. Gyllenhaal, and W.-m. W. Hwu. "A Hardware-Driven Profiling Scheme for Identifying Program Hot Spots to Support Runtime Optimization". In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 1999.

[40] T. Mikolov, M. Karafiat, L. Burget, J. Cernocky, and S. Khudanpur. "Recurrent Neural Network Based Language Model". In *Proceedings of the International Speech Communication Association (INTERSPEECH)*, 2010.

[41] K. P. Murphy. *"Machine Learning: A Probabilistic Perspective"*. MIT Press, 2012.

[42] D. M. Powers. "Evaluation: From Precision, Recall and F-measure to ROC, Informedness, Markedness and Correlation". Technical report, Flinders University, Adelaide, Australia, 2011.

[43] A. Srivastava and A. Eustace. "ATOM: A System for Building Customized Program Analysis Tools". In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 1994.

[44] I. Sutskever, O. Vinyals, and Q. V. Le. "Sequence to Sequence Learning with Neural Networks". In *Advances in Neural Information Processing Systems (NIPS)*, 2014.

[45] D. Tetzlaff and S. Glesner. "Static Prediction of Loop Iteration Counts Using Machine Learning to Enable Hot Spot Optimizations". In *Proceedings of the Conference on Software Engineering and Advanced Applications (SEAA)*, 2013.

[46] D. Ung and C. Cifuentes. "Optimising Hot Paths in a Dynamic Binary Translator". *SIGARCH Computer Architecture News*, 2001.

[47] W. Weimer and G. C. Necula. "Exceptional Situations and Program Reliability". *Transactions on Programming Languages and Systems (TOPLAS)*, 2008.

[48] S. M. Weiss and I. Kapouleas. An Empirical Comparison of Pattern Recognition, Neural Nets and Machine Learning Classification Methods. *Readings in Machine Learning*, 1990.

[49] P. J. Werbos. "Backpropagation Through Time: What It Does and How To Do It". *Proceedings of the IEEE*, 1990.

[50] C. Young and M. D. Smith. "Improving the Accuracy of Static Branch Prediction Using Branch Correlation". *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1994.