2016 IEEE/ACM 38th IEEE International Conference on Software Engineering

Generating Performance Distributions via Probabilistic Symbolic Execution

Bihuan Chen[*], Yang Liu[*] and Wei Le[†]
[*]School of Computer Engineering, Nanyang Technological University, Singapore
[†]Department of Computer Science, Iowa State University, USA

ABSTRACT

Analyzing performance and understanding the potential best-case, worst-case and distribution of program execution times are very important software engineering tasks. There have been model-based and program analysis-based approaches for performance analysis. Model-based approaches rely on analytical or design models derived from mathematical theories or software architecture abstraction, which are typically coarse-grained and could be imprecise. Program analysis-based approaches collect program profiles to identify performance bottlenecks, which often fail to capture the overall program performance. In this paper, we propose a performance analysis framework `PerfPlotter`. It takes the program source code and usage profile as inputs and generates a *performance distribution* that captures the input probability distribution over execution times for the program. It heuristically explores *high-probability* and *low-probability* paths through probabilistic symbolic execution. Once a path is explored, it generates and runs a set of test inputs to model the performance of the path. Finally, it constructs the performance distribution for the program. We have implemented `PerfPlotter` based on the Symbolic PathFinder infrastructure, and experimentally demonstrated that `PerfPlotter` could accurately capture the best-case, worst-case and distribution of program execution times. We also show that performance distributions can be applied to various important tasks such as performance understanding, bug validation, and algorithm selection.

CCS Concepts

•software and its engineering → software performance;

Keywords

Performance Analysis, Symbolic Execution

1. INTRODUCTION

Understanding software performance is important. Developers want to know whether further optimizations or performance related software assurance tasks need to be done; users are interested to know whether software may run too slowly to use. Up till today, much of the software engineering research has been focusing on the functional correctness, but not much work has been done for performance analysis.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICSE '16, May 14-22, 2016, Austin, TX, USA
© 2016 ACM. ISBN 978-1-4503-3900-1/16/05…$15.00
DOI: http://dx.doi.org/10.1145/2884781.2884794

There are two types of existing techniques for performance analysis: model-based and program analysis-based approaches. Model-based approaches [6, 32] often rely on analytical models (e.g., Petri nets [31]) derived from mathematical theories or design models (e.g., feature models [55]) derived from software architecture abstraction. The problems are that constructing such models requires extensive domain knowledge; the models are coarse-grained, and miss implementation details to derive best-case and worst-case execution times; and it is also hard to keep the models and source code synchronized. As a result, the performance prediction can be imprecise.

On the other hand, program analysis-based approaches derive performance models from source code and program executions. In particular, there have been dynamic analyses that perform statistical inference and derive a performance model based on the measured execution time for each basic block under a set of test inputs [25, 58, 17]. The problem is that the accuracy of the performance model often relies on the given set of test inputs. Differently, symbolic execution and guided testing are used to automatically generate the test inputs that can lead to the worst-case execution times [12, 61, 26, 53]. Their focus is to identify performance bottlenecks but not to characterize the overall performance of a program for all inputs.

To capture insightful and comprehensive characteristics of the performance of a program, in this paper, we propose the concept of *performance distribution* as well as a performance analysis framework `PerfPlotter` to generate the performance distribution for a program under a usage profile. The performance distribution plots the input probability over program execution times. It shows the likely range of execution times, including the best-case and worst-case execution times; and it also visualizes how likely certain execution time will be exhibited. Such information can be used to understand the corner cases of a program, reveal performance problems and prioritize their severity, if any. Comparing the performance distributions for the two versions of a program before and after code changes, we can track unexpected performance regressions or speedups; and comparing the performance distributions for functionally equivalent programs, we can learn the performance advantage of a program over others.

`PerfPlotter` leverages probabilistic symbolic execution [35, 23] to perform a selective exploration of program paths. We classify two types of paths: *high-probability* and *low-probability* paths. Here, the probability refers to how likely a path will be executed under a usage profile. In particular, our path exploration process is divided into two phases. In the first phase, we explore high-probability paths while ensuring path diversity to uncover performance characteristics for frequently executed scenarios in a diverse way. In the second phase, we explore low-probability paths to expose the performance of interest hidden in the less frequently executed paths. In fact, we did find in our experiments that such paths are very important for exposing best-case and worst-case execution times of a program. In the presence of a loop, only a subset of representative loop paths are explored to capture the overall performance of the loop while ensuring scalability. For each explored path, we generate multiple test inputs and model the performance of the path as their average execution time.

We implemented `PerfPlotter` using Symbolic PathFinder (SPF) [47] and its probabilistic symbolic execution engine [23]. In our experiments, we compared the performance distributions generated via `PerfPlotter` with those generated by three other approaches: exhaustive symbolic execution, random symbolic execution, and random testing. Our results demonstrated that, by heuristically exploring a small set of paths, `PerfPlotter` can generate performance distributions that can capture the general trend of the actual ones and expose the best-case and worst-case execution times with high coverage and low overhead. We also applied the performance distributions to assisting performance understanding, bug validation, and algorithm selection.

The main contributions of this paper are three: (1) introducing the concept of performance distribution, (2) designing a framework to automatically generate performance distributions, and (3) evaluating the proposed framework.

2. PRELIMINARIES

In this section, we provide the background on symbolic execution and probabilistic analysis in symbolic execution.

2.1 Symbolic Execution

Symbolic execution is a program analysis technique that systematically explores program execution paths [16, 30]. It uses *symbolic* values in place of *concrete* data as inputs and produces symbolic expressions over symbolic inputs as outputs. The state of a symbolically executed program is defined by an instruction pointer, symbolic expressions of program variables, and a path condition. The path condition is a conjunction of constraints over symbolic inputs, which characterizes the inputs that would execute the path from the initial state to the current state. During symbolic execution, path conditions are checked for satisfiability; when a path condition is not satisfiable, the path is infeasible, and the symbolic execution stops exploring this path and backtracks.

The paths generated during symbolic execution are characterized by a *symbolic execution tree*. The tree nodes represent states, and the arcs represent transitions between states. Here we focus on sequential programs, and hence distinguish two kinds of nodes: *branch* nodes (e.g, loops and conditional statements) and *non-branch* nodes (e.g., method invocations and assignments). The former has one or two outgoing transition(s), representing the true and/or false branches, and the latter has at most one outgoing transition. Branch nodes are the key to select paths of interest during path exploration.

Several tools have been proposed for symbolic execution of programs written in different programming languages, e.g., KLEE [14] for LLVM and SPF [47] for Java. Here we focus on Java programs and use SPF for implementation, but our technique is general and independent of these tools.

2.2 Probabilistic Analysis

Symbolic execution uses satisfiability checking to decide if a path is feasible or not. A feasible path only implies that there is an element (i.e., a solution generated by a constraint solver) in the input domain that satisfies the path condition, but gives no information on the exact number of such solutions. Probabilistic symbolic execution is proposed to allow such a quantification for calculating the probability of satisfying a path condition under uniformly distributed inputs [35, 23].

Since not all input values are equally likely, a *usage profile* is used to probabilistically characterize the program's interactions with the environment [21]. In particular, each input variable is discretized by partitioning the input values into a set of intervals and assigning each of them a probability [21]. By this means, arbitrarily complex probability distributions of the input variables can be handled.

Given a specific usage profile, a symbolic execution tree is constructed as follows. Whenever a condition is encountered, a branch node is introduced, and we compute a pair $\langle pc,\ prob\rangle$. pc is the path condition, which is the conjunction of the conditions of the taken branches along the path from the initial node to the current one. $prob$ is the path probability, which is the product of the conditional probabilities of the taken branches along the path from the initial node to the current one. Along with the branch node, two transitions are introduced: we compute $\langle c,\ pr_c\rangle$ and $\langle \neg c,\ pr_{\neg c}\rangle$ for the true and false branch respectively. pr_c and $pr_{\neg c}$ are the branch probabilities (the probability of the true or false branch taken) under the usage profile. They can be computed using $P(c \mid pc)$ and $P(\neg c \mid pc)$ as proposed in [21] based on the law of total probability and conditional probability [44]. We assume the usage profile is encoded in programs as proposed in [21, 22].

Several tools have been proposed to quantify the solution space of constraints over specific domains, which can be used to calculate branch probabilities, e.g., LattE for linear integer constraints [18], qCORAL for arbitrary float constraints [9, 10], SMC for string constraints [38], and Korat for heap data structures [11]. In this paper, we focus on the domains of finite integer and float numbers, and we use LattE and qCORAL in our implementation; however, our framework is potentially extensible to other domains and tools.

3. PERFORMANCE ANALYSIS

In this section, we provide an overview of `PerfPlotter`; then we elaborate on the techniques for selective path exploration and path-based performance modeling; and finally we present the algorithms for generating performance distributions.

3.1 Overview

A program consists of a set of paths. The performance of a program can be exactly captured by characterizing the performance of all program paths. Specifically, we define the *performance of a path* as a 3-tuple $pf = \langle pc,\ time,\ prob\rangle$, where pc is the path condition that constraints the test inputs that lead to this path, $time$ is the average execution time that the program takes for all the inputs that exercise this path, and $prob$ is the probability of executing this path under a usage profile. Then we define the *performance distribution* of a program as a function relating input probability with execution time, where the input probability is the cumulative probability of a set of paths whose execution times fall into an interval. The histogram at the rightmost of Figure 1 shows a performance distribution `PerfPlotter` generates. The X and Y axes denote the execution time and input probability.

The number of actual feasible execution paths can be very large, and it is impossible to run tests for all the paths. Thus, we select representative paths by classifying *high-probability*

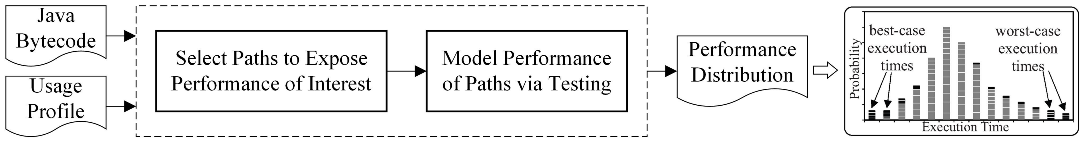
Java Bytecode
Usage Profile
Select Paths to Expose Performance of Interest
Model Performance of Paths via Testing
Performance Distribution
best-case execution times
worst-case execution times
Probability
Execution Time

Figure 1: The Overview of Our Performance Analysis Framework.

and *low-probability* paths during the path exploration. The goal is to generate performance distributions that can expose the performance of interest and capture the general trend of the actual performance distributions over all feasible paths.

Figure 1 presents an overview of `PerfPlotter`. It takes as inputs the *Java bytecode* and the *usage profile* of a program. It generates a *performance distribution* of the program, which highlights the performance of interest. The internal design of `PerfPlotter` aims to address two key challenges: how to select representative paths that can expose the performance of interest, and how to generate the performance distribution based on the probability and execution time of the paths.

To address the first challenge, we employ symbolic execution to selectively explore two types of paths in two phases. In the first phase, we explore high-probability paths where the majority of inputs will execute. Understanding performance for these paths can help developers learn the performance characteristics of a program for the common inputs. Among these high-probability paths, we also seek their path diversity, aiming to explore high-probability paths with a diverse set of behaviors. In the second phase, we explore low-probability paths. Our assumption is that these paths often lead to special execution conditions and thus can be useful to expose the performance of interest, such as the best-case and worst-case execution times. Considering there potentially exists an exponential number of paths, we define a stopping criterion for each phase to terminate (see Sections 3.2.1 and 3.2.2). For handling loops, we apply loop unrolling and a set of heuristics to selecting representative paths in loops (see Section 3.2.3).

To address the second challenge, we apply path-based performance modeling. We run the generated test inputs for the selected paths and use the probabilities of the paths to construct the performance distribution (see Section 3.3).

3.2 Selective Path Exploration

We apply probabilistic analysis and compute the probabilities of outgoing transitions of each branch. This is the key information used for path selection. To determine which paths to select, the symbolic execution makes two decisions. First, during the exploration of a path, we need to decide which outgoing transition of each branch node along the path should be taken. Second, after completing the exploration of a path, we need to decide which unexplored outgoing transition of the branch nodes in the symbolic execution tree should be taken to start the exploration of a new path. In the following subsections, we explain how the two decisions are made in the two phases of our path exploration. We will first show how our approach handles the paths without loops, and then we will discuss how to handle paths with loops.

3.2.1 *Exploring High-Probability Paths*

The goal of the first phase is to explore high-probability paths and ensure the path diversity so that the performance characteristics for the frequently executed scenarios can be exposed in a diverse way. For the first decision of selecting the outgoing transition at each branch node when exploring a path, we always select the outgoing transition that has the higher branch probability. Meanwhile, the other outgoing transition that has the lower branch probability is stored in a list UT^C for further considerations when we make the second decision. This list maintains all the unexplored outgoing transitions.

For the second decision, we use heuristics to combine the goals of exploring high-probability paths and ensuring the path diversity. Specifically, we construct an objective function F_1 shown in Equation 1. F_1 computes the tradeoff of path probability and path diversity using the weight $\alpha \in [0,\ 1]$. Users can configure α dependent on how important the two factors are for a specific application. The transition with the highest value of F_1 will be chosen from UT^C to explore a new path.

In Equation 1, the first term measures the probability of the path for a transition T in UT^C, where *pr* is the branch probability for T, and *prob* is the probability of the path that leads to the branch node of T. The higher value the first term results, the higher path probability T leads to. The second term quantifies the degree of path diversity T implies. Specifically, *br* represents the number of branches along the path from the initial node to transition T shared by the previously explored paths. A smaller value of *br* indicates a shorter common prefix with explored paths, suggesting a higher path diversity. To allow a unified measurement of the metrics independent of their units and ranges, we scale path diversity into a value between 0 and 1 by comparing it with the minimum (br_{min}) and maximum (br_{max}) number of branches computed for all the transitions in UT^C.

$$F_1 = \alpha \cdot pr \cdot prob + (1-\alpha) \cdot \frac{br_{max} - br}{br_{max} - br_{min}} \qquad (1)$$

The path exploration in this phase terminates when the cumulative probability of all the explored paths reaches a coverage probability *cp*. Users can configure *cp* for specific applications. The intuition for this stopping criterion is the 80-20 rule: programs typically spend 80% of the execution time within 20% of the code. Once *cp* is reached, we say the most frequently executed scenarios are covered by the selected paths.

3.2.2 *Exploring Low-Probability Paths*

The goal of the second phase is to explore low-probability paths so that the performance of interest potentially hidden in the less frequently executed scenarios can be exposed. Following this goal, the first decision is made as follows: during the exploration of a path, at each branch node, we always select the outgoing transition that has the lower branch probability. At the same time, the other outgoing transition that has the higher branch probability is stored in the list UT^C for further considerations.

For the second decision of selecting a transition to start a new path, different heuristics can be used to explore paths that potentially expose different performance of interest. We focus on the best-case and worst-case execution times of a program in this paper. We construct two objective functions F_2 (for the best-case execution time) and F_3 (for the worst-case execution time), shown in Equation 2, to rank each unexplored transition T in UT^C. We select the transition with the lowest rank as the starting point of the exploration of a new path with the goal of finding the transitions that most likely lead to the best-case and worst-case execution times.

Shown in Equation 2, F_2 and F_3 introduce the weights β and $\gamma \in [0,1]$ respectively to represent the tradeoff between path probability and path performance. Similar to F_1, path probability is defined using $\beta \cdot pr \cdot prob$ and $\gamma \cdot pr \cdot prob$. Path performance is measured by the number of instructions (in) along the path from the initial node to transition T. A larger-/smaller value of in indicates a worse/better performance. Here, the number of instructions is used as a metric for performance with the assumption that the performance of a path is approximately proportional to the number of instructions executed in the path. Similar to path diversity, path performance is also scaled into a value between 0 and 1 by comparing it with the minimum (in_{min}) and maximum (in_{max}) number of instructions for the transitions in UT^C.

$$\begin{aligned} F_2 &= \beta \cdot pr \cdot prob + (1-\beta) \cdot \frac{in - in_{min}}{in_{max} - in_{min}} \\ F_3 &= \gamma \cdot pr \cdot prob + (1-\gamma) \cdot \frac{in_{max} - in}{in_{max} - in_{min}} \end{aligned} \tag{2}$$

The path exploration in this phase terminates when none of the l consecutive selected paths can expose the performance of interest, or no path can be further explored. To do so, we compare the real execution time of the path under consideration with the best-case and worst-case execution time among previously explored paths. The intuition here is that we seem to already find all the performance of interest as no performance of interest is exposed for a period of time.

3.2.3 Handling Loop Structures

Here our goal is to select a set of representative paths to expose the performance of interest related to loops while capturing the overall performance of loops.

To expose performance of interest, we aim to first select a path with the maximum number of loop iterations when encountering a loop during the path exploration. Thus, the first decision is made as follows: when a loop is encountered during the path exploration, it is unrolled by selecting the branch transition that keeps the path traversing the loop and its inner loops. In particular, for a branch node resulting from the loop condition, we will select the outgoing transition that can lead to the next loop iteration (i.e., selecting the true branch). For a branch node resulting from a conditional statement inside the loop, if either of the two out-going transitions breaks out from the loop (e.g., due to *break* statements), we will select the transition that keeps the exploration staying in the loop. For the branch nodes that are not relevant to the loop termination conditions, we will select the transition by the approach introduced in Section 3.2.1 and 3.2.2 depending on whether the exploration is in the first or second phase.

Meanwhile, the transitions that represent the unexplored false branch of loop conditions and the transitions that break out from the loop are stored in a list UT^L for further considerations. UT^L maintains the unexplored transitions of a loop for a set of designated loop iterations. Since a path may sequentially have r loops and each loop needs a list to store its unexplored transitions, we maintain $\{UT_i^L \mid 1 \leq i \leq r\}$.

To capture the overall performance of a loop, we need to select a set of paths that contain representative loop iterations. Thus, for the second decision, we select a set of transitions from UT_i^L $(1 \leq i \leq r)$ as the starting points of the exploration using two sampling-based heuristics: (1) we select the first transition (i.e., having the minimum loop iteration) and also every s-th transition in UT_i^L; and (2) we select the transitions with the lowest and highest path probability. As a result, these heuristics explore loop paths with the minimum-/maximum loop iterations and the lowest/highest path probability, and also explore other loop paths at intervals with respect to their loop iterations. s is the sampling interval, which can be tuned based on the time budget for performance analysis. For example, s could be set to a large value if the time budget is tight; in this case, less loop paths will be explored.

3.3 Path-Based Performance Modeling

A path may be infinite, and thus an exploration depth limit (i.e., the number of transitions executed) is usually set for a bounded symbolic execution [47]. When the limit is reached during our selective path exploration, the exploration of one path terminates. As a result, we obtain three types of paths: completed paths (i.e., paths that are completely explored), terminated paths (i.e., paths that are partially explored and terminated due to reaching the limit), and uncompleted paths (i.e., paths that are partially explored but not selected by `PerfPlotter`). Among them, the completed and terminated paths are executed by testing to measure their real execution times.

Different from a completed path, a terminated path is actually a common prefix of a set of different paths. Therefore, executing such a path using different test inputs, we may get quite different execution times. In other words, the execution time of a terminated path is more volatile than that of a completed path. For this reason, we test each completed and terminated path by running the program respectively on d_1 and d_2 ($d_1 < d_2$) number of test inputs and calculating the average of measured execution times. Here d_1 and d_2 are the testing sizes, which can be specified by users. These test inputs are randomly generated from the path condition by the constraint solvers in SPF [47].

We calculate the execution time of a path by amortizing the execution times from all the runs of the generated test inputs. Based on the data and the probabilities of paths, we construct a performance distribution as a histogram to present developers with a graphical representation for a more readily understanding of the performance of a program (see the example in Figure 1). In detail, we divide the entire range of the measured execution times into a set of intervals, and then draw a rectangle for each path with the height proportional to the path probability and put the rectangle into the interval that the execution time of the path falls into. We also highlight the top-k paths that expose specific performance of interest (e.g., worst-case execution times).

3.4 The Algorithm

Algorithm 1 shows the detailed procedure of our performance analysis. It takes program source code and the usage profile as inputs and reports a performance distribution `perfDist`. It works through a number of iterations (Line 6–27). At each iteration, it has three steps: (1) explore a path starting from a transition `stTran`, and if the path contains loops, also explore the representative loop paths starting from the representatives from `transL` (i.e., $\{UT_i^L \mid 1 \leq i \leq r\}$) (Line 7–13); (2) decide if the exploration needs to be switched into the second phase (Line 14–15), or if the exploration needs to be terminated (Line 16–17), based on the two stopping criteria of each phase respectively; and (3) if not terminated, select transition(s) from `transC` (i.e., UT^C) based on the objective

Algorithm 1: AnalyzePerformance

```
input  : program source code, usage profile
output: perfDist
1  phase ← 1;    // phase of the exploration
2  stTran ← null; // the starting transition
3  transC ← {};  // a list of unexplored transitions
4  transL ← {};  // lists of unexplored loop transitions
5  ts ← {};      // a list of selected transitions
6  repeat
7      repeat
8          pf ← ExplorePath(phase, stTran, transC, transL);
9          update perfDist with performance of the path pf;
10         ts ← ts ∪ {representatives from transL};
11         transL ← {};
12         stTran ← ts.removeFirst();
13     until stTran == null;
14     if phase == 1 and stopping criterion 1 then
15         phase ← 2;
16     else if phase == 2 and stopping criterion 2 then
17         return perfDist;
18     if phase == 1 then
19         ts ← ts ∪ {tran in transC that maximizes $F_1$};
20         transC ← transC \ {tran};
21     else if phase == 2 then
22         ts ← ts ∪ {tran in transC that minimizes $F_2$};
23         transC ← transC \ {tran};
24         ts ← ts ∪ {tran in transC that minimizes $F_3$};
25         transC ← transC \ {tran};
26     stTran ← ts.removeFirst();
27 until stTran == null;
28 return perfDist;
```

Algorithm 2: ExplorePath

```
input  : phase, stTran, transC, transL
output: pf
1  start symbolic execution from stTran until a condition $c$;
2  repeat
3      introduce a branch node, compute its $pc$ and $prob$;
4      introduce transitions $t_c$ and $t_{\neg c}$, compute $pr_c$ and $pr_{\neg c}$;
5      if c is from a conditional statement then
6          if c may break out from a loop then
7              select the transition that stays in loop if feasible,
               and add the other transition to transL.last;
8          else if phase == 1 then
9              transC ← transC ∪ {transition with low $pr$};
10             select the transition with high $pr$;
11         else if phase == 2 then
12             transC ← transC ∪ {transition with high $pr$};
13             select the transition with low $pr$;
14     else if c is from a loop structure then
15         if c is feasible then
16             if loop entry then
17                 transL ← transL ∪ {};
18             transL.last ← transL.last ∪ {$t_{\neg c}$};
19             select $t_c$;
20         else
21             select $t_{\neg c}$;
22     continue symbolic execution until a condition $c$;
23 until the path is completed or the limit is reached;
24 get the execution time time of the explored path by testing;
25 return pf ← ⟨$pc$, $prob$, $time$⟩;
```

function(s) (i.e., F_1, or F_2 and F_3) to start the exploration of new path(s) (Line 18–26). The algorithm terminates if the stopping criterion is satisfied or no path could be further explored, and then returns the performance distribution.

Specifically, in the second step at Line 8, Algorithm 2 is invoked for exploring a path starting from the transition `stTran`. The goal of this step is to generate performance model for one path, and the output `pf` reports the path condition, path probability and execution time. Initially, Algorithm 2 starts from the initial state of the program (i.e., `stTran` is `null`). Whenever a condition is encountered, it introduces one branch node and computes its path condition *pc* and path probability *prob*. It also introduces two transitions and computes their probabilities pr_c and $pr_{\neg c}$ (Line 3–4). If the condition is from a conditional statement that might break out from a loop, it aways selects the transition that keeps the exploration staying in the loop if feasible, and stores the other one in `transL` (i.e., $\{UT_i^L \mid 1 \leq i \leq r\}$ for further exploration (Line 5–7); otherwise, it selects a transition based on the specific heuristic of each phase, and stores the other one in `transC` (i.e., UT^C) for further exploration (Line 8–13). If the condition is from a loop structure, it selects the true transition if feasible, while stores the false one in `transL` for further exploration (Line 14–21). Whenever a new loop structure is encountered, an empty list is inserted into `transL` for storing its unexplored transitions (Line 16–17). The iteration stops if the path is completed or the exploration depth limit is reached (Line 23). Then, the algorithm obtains the execution time of the explored path by testing generated inputs (Line 24), and returns the performance of the path (Line 25).

4. IMPLEMENTATION AND EVALUATION

We implemented `PerfPlotter` using Java based on the SPF infrastructure [47] and its probabilistic symbolic execution engine [23]. In `PerfPlotter`, we use Choco [46] to solve linear integer constraints and CORAL [8] to solve floating point constraints. We use LattE [18] to compute the probability of conditional branches related to linear integer constraints and integrate qCORAL [10, 9] to compute the probability of conditional branches related to floating point constraints. `PerfPlotter` is available at our website [3] with all the software artifacts and experimental data used in our evaluation.

4.1 Evaluation Setup

To evaluate whether `PerfPlotter` is effective and can compute meaningful performance distributions that highlight the performance of interest, we compared the performance distributions generated by `PerfPlotter` with those generated by the exhaustive symbolic execution (`ESE`) approach. In the `ESE` approach, we use symbolic execution to exhaustively explore all program paths and generate performance distributions by testing all these program paths. Thus, the ESE approach can be regarded as the baseline. To evaluate whether `PerfPlotter` performs better than a random testing (`RT`) approach and thus can be considered worthwhile, we compared `PerfPlotter` with a `RT` approach, where we generate performance distributions using a set of randomly generated test inputs. To evaluate if the heuristics applied in `PerfPlotter` are effective in selecting the paths of interest, we compared `PerfPlotter` with the random symbolic execution (`RSE`) approach, where the two decisions for the path exploration (see Section 3.2) are made randomly. Note that, to make a fair comparison across these approaches, the `RT` and `RSE` approaches are limited to run the same amount of time as `PerfPlotter`.

We selected the following software artifacts as our experimental subjects. Note that our implementation is dependent on SPF, and SPF is not yet able to handle large artifacts because of the challenges in symbolic execution (similar experience has been shown in other work [23, 21]). To the best of our knowledge, we have used all the largest and publicly available artifacts SPF can handle.

- **Quick Sort**: This program performs the quick sort on an array of size n. It contains $n!$ feasible paths, and there is only one path that leads to the worst-case time complexity. The performance of this program depends on the choice of pivot, and we set the middle element as the pivot. We analyzed a version for $n = 7$ with 5,040 paths (the number is reported by the exhaustive symbolic execution approach).
- **Single Loops**: This program contains two single loops executed in sequence. The iterations of each loop depend on an input variable. We limited the two input variables to [1, 100]. This program contains a total of 10,000 paths.
- **Nested Loop**: This program contains a nested loop. The iterations of the inner and outer loop depend on two input variables. We limited the two input variables to $[1, 30]$, which leads to a total of 900 paths.
- **TreeMap**: This program is the red-black tree implementation in the Java library. We analyzed the performance of a sequence of 5 calls to `put` and `remove` methods. The program has 0.6K lines of code with 3,840 paths.
- **Apollo**: Apollo Lunar Autopilot is a Java program automatically translated from a Simulink model [43]. The model is available from MathWorks [2]. It decides the transition between *fire* and *coast* states as well as the reaction jet to fire. It contains 2.6K lines of code with 2,110 paths.
- **CDx**: The CDx system [1] is a simulator for aircraft collision detection. The *Detector* module detects collisions according to the current and next positions of the aircrafts, which has 3.2K lines of code. We analyzed this module for 10 aircrafts, which leads to a total of 2,476 paths.
- **TSAFE**: Tactical Separation Assisted Flight Environment is designed by NASA and FAA to predict flight conflicts and alert air traffic controllers. We analyzed the *Conformance Monitor* module, which checks if a flight is conforming to the planned route and assigns or synthesizes a trajectory. This module has 4.0K lines of code with 1,289 paths.

In the evaluation, we assumed a uniform usage profile for all input variables. To configure `PerfPlotter`, we set the weights α, β and γ to 0.5 and testing sizes d_1 and d_2 to 10 and 20 (see Section 3 for details of the parameters). We studied the settings of other three key parameters in our experiments: *cp*, the coverage probability for stopping exploring high-probability paths; l, the number of consecutive paths executed but not exposing any best-case or worst-case execution times; and s, the sampling interval for selecting loop paths. We report our empirical results regarding these parameter settings in the following sections. We ran our experiments on a desktop with 3.50 GHz Intel Xeon E5-1650 v2 CPU and 16 GB RAM.

Our evaluation answers the following research questions:

- **RQ1**: Can `PerfPlotter` generate meaningful performance distributions that well capture the general trend of the actual performance distributions?
- **RQ2**: Can `PerfPlotter` expose the performance of interest with high coverage and low overhead?
- **RQ3**: Are the heuristics applied in `PerfPlotter` effective in selecting the paths of interest?
- **RQ4**: What are the potential application scenarios of the generated performance distributions of `PerfPlotter`?

4.2 Performance Distributions (RQ1 and RQ3)

Figure 2 shows the generated performance distributions for the subject programs by the `ESE`, `RT`, `RSE` and `PerfPlotter` approaches. The X axis denotes the execution time, and the Y axis denotes the input probability (i.e., the cumulative probability of a set of paths whose execution time falls into an interval). For clarity, we compared the results of the approaches using the line charts. The histogram versions of performance distributions are available at our website [3].

The `RT` approach is expected to approximate the actual performance distributions generated by the `ESE` approach; however, Figure 2 shows that it greatly over-/under-approximated the probability for certain execution time intervals, e.g., [13.3, 14.8) for Quick Sort, [525, 555) for Nested Loop, [12500, 15000) for CDx, and [5250, 5500) for TSAFE. Besides, it falsely identified the execution time that most frequently occurs for Single Loops, Nested Loop, TreeMap, Apollo, and CDx. Such deviations are caused by the stochastic nature of the test input generation, where certain regions of the input domain might be densely sampled while others might be sparsely sampled. Moreover, the `RT` approach failed to capture the entire range of execution times for Apollo (i.e., the execution times larger than 57,500 nanoseconds) and CDx (i.e., the execution times larger than 25,000 nanoseconds), because the corresponding test inputs are hard to hit during random testing.

The `RSE` and `PerfPlotter` approaches are expected to generate similarly shaped performance distributions to the `ESE` approach because they sample a much smaller space of the input domain than the `ESE` approach. As shown in Figure 2, their cumulative probabilities for all inputs are less than the `ESE` approach. However, the `RSE` approach failed to capture the entire range of execution times (i.e., the worst execution times) for Single Loops, Nested Loop and TreeMap. This is because of the fact that random symbolic execution often tends to uncover shorter paths more easily than longer paths due to the stochastic nature especially when the program has deep paths. In addition, the `RSE` approach produced much less complete performance distributions than `PerfPlotter` for TreeMap, Apollo, CDx and TSAFE since it often fails to explore high-probability paths due to its stochastic nature.

Differently, Figure 2 demonstrates that `PerfPlotter` generated performance distributions that capture the entire range of execution times and the general trend of the actual one for all the programs. The curves generated by `PerfPlotter` have similar shapes as the ones generated by the `ESE` approach. Further, it better exposed the best-case and worst-case execution times than the `RSE` approach (see details in Section 4.3). It indicated that, by heuristically exploring high-probability paths while seeking path diversity, we are able to capture performance characteristics for a variety of most frequently executed paths; and by heuristically exploring low-probability paths while considering path performance, we are able to expose corner cases hidden in less frequently executed paths.

Apart from the comparisons with the three approaches, we evaluated the configuration parameters of `PerfPlotter` and observed how different settings of *cp*, l and s could generate different performance distributions. In Figure 2 (a), (e), (g), (i) and (k), we compared the two performance distributions generated when l was fixed and *cp* was changed. Our finding is that, when the value of *cp* increases, the performance distribution becomes more complete and captures the general trend more effectively because more paths are explored in the first phase. Similarly, we fixed *cp* and changed l in Figure 2 (b),

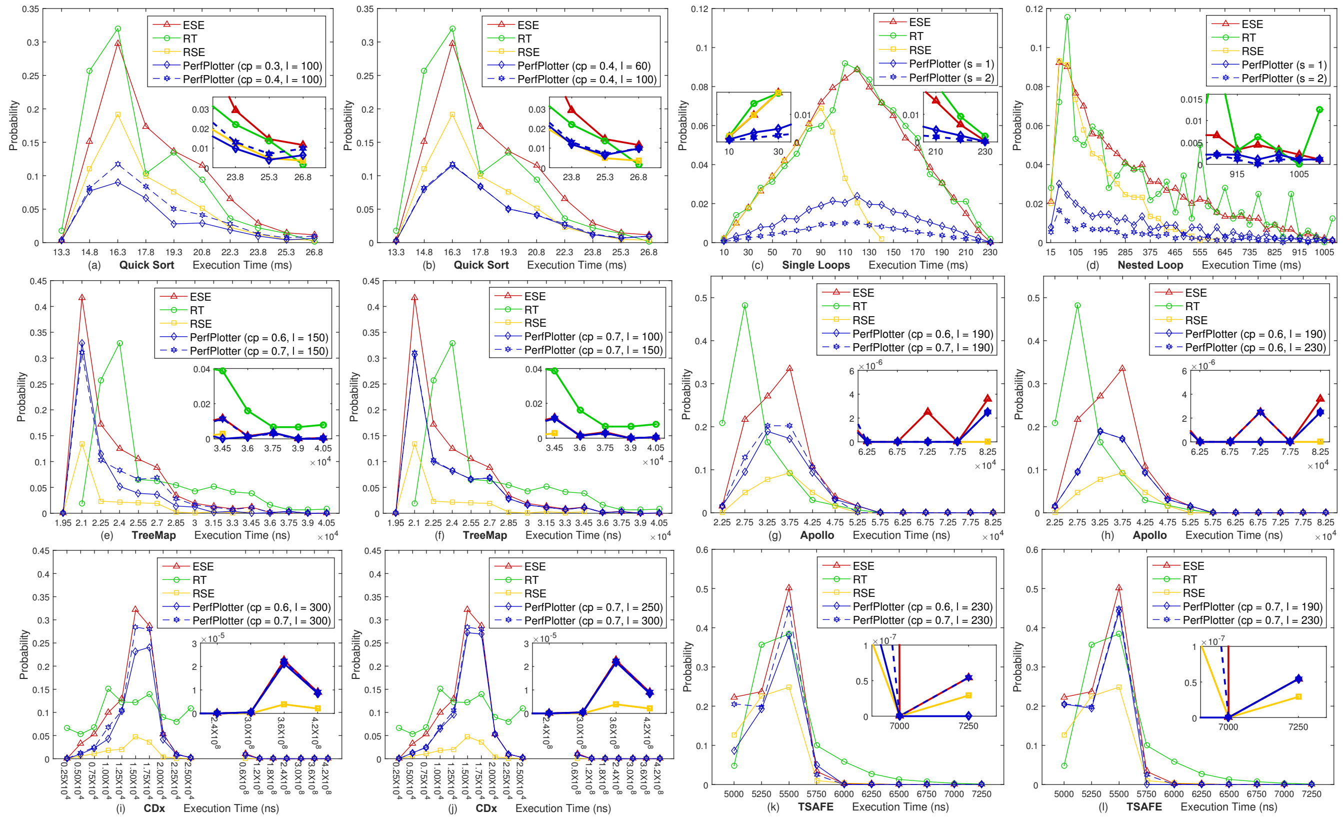
Figure 2: Generated Performance Distributions for the Software Artifacts.

(f), (h), (j) and (l). We found that, with the increase of l, the generated performance distributions are similar, but PerfPlotter can better expose the best-case and worst-case execution times. The reason is that a stricter stopping criterion for exploring the low-probability paths is used as l increases, which is helpful for finding corner cases. Finally, we reported our experiments on different sampling intervals for the loops in Figure 2 (c) and (d). When s increases, a smaller subset of representative loop paths are selected; as a result, the generated performance distributions become less complete and capture the general trend of the actual ones less effectively.

In summary, the experimental results in Figure 2 demonstrated that the RT and RSE approaches are not dependable to compute performance distributions, and their results may mislead users and developers about the performance of programs. It also showed that the heuristics used in PerfPlotter are effective in selecting paths of interest, and thus PerfPlotter can generate meaningful performance distributions that capture the general trend and entire range of execution times with a small set of test inputs.

4.3 Coverage and Overhead (RQ2 and RQ3)

Table 1 reports the comparison of PerfPlotter and the RT and RSE approaches on two metrics: the coverage for performance of interest and the time overhead used to generate the performance distributions. Under *Subject* and *Approach*, we provide involved software artifacts and approaches (with the setting of parameters for running PerfPlotter). Under *Path*, we report the percentage of the paths explored by the PerfPlotter and RSE approaches. This dimension is not analyzed for the RT approach because it is not path-based. Under *Best-Case ET* and *Worst-Case ET*, we show that among the top 10, 20 and 30 best-case or worst-case paths found by the ESE approach, the percentage of such paths that are explored by the PerfPlotter, RT and RSE approaches. In particular, for PerfPlotter, we provide further details: the number before and after "+" is respectively the percentage obtained during the first and second phase. Under *Analysis Time*, we give the total time used for each approach under *Total*. For the PerfPlotter and RSE approaches, we distinguish the time for exploring paths under *Exp* and the time for running tests under *Test*. The performance is a relative measure and reports the percentage of the time of the ESE approach.

Table 1 indicates that the RT approach achieved the lowest coverage for the best-case and worst-case paths. In particular, for the larger artifacts (i.e., TreeMap, Apollo, CDx, and TSAFE), it almost failed to expose any best-case and worst-case paths. This is because, with the extremely large space of the input domain, random testing often fails to hit the corner cases hidden in the extremely low-probability paths.

The RSE approach achieved a higher coverage for the best-case and worst-case paths than the RT approach. In particular, for Single Loops, Nested Loop and TreeMap, it achieved a very high coverage for best-case paths but an extremely low coverage for worst-case paths, as illustrated in Figure 2 (c-f), since the RSE approach tends to uncover shorter paths more easily than longer paths due to its stochastic nature. However, when a program does not have relatively shorter paths as in the case of Apollo, CDx and TSAFE, the RSE approach has a similar coverage for the best-case and worst-case paths.

PerfPlotter obtained a higher coverage for the best-case and worst-case paths than the RSE approach on average, as illustrated in Figure 2. On average, the RSE and PerfPlotter approaches respectively found 41.9% and 52.7% of the top 30 best-case and worst-case paths by exploring 43.4% and 32.6% of the total paths within the same amount of time. This re-

Table 1: Comparison of `PerfPlotter` with the `RT` and `RSE` approaches on Coverage and Overhead

Subject	Approach		Path (%)	Best-Case ET (%)			Worst-Case ET (%)			Analysis Time (%)		
				Top 10	Top 20	Top 30	Top 10	Top 20	Top 30	Exp	Test	Total
Quick Sort	$cp = 0.3$	$l = 60$	29.9	0 + 30	15 + 25	13 + 27	20 + 40	20 + 40	20 + 30	85.1	28.8	48.2
		$l = 100$	32.1	0 + 80	15 + 60	13 + 70	20 + 50	20 + 45	20 + 33	85.1	31.0	49.6
	$cp = 0.4$	$l = 60$	41.3	0 + 90	20 + 65	17 + 57	50 + 40	45 + 40	37 + 50	96.3	40.3	59.6
		$l = 100$	42.1	0 + 100	20 + 75	17 + 73	50 + 40	45 + 40	37 + 50	96.3	41.2	60.2
	RT		–	60	65	70	70	50	53	–	–	60.2
	RSE		57.3	70	75	73	30	20	17	70.7	54.6	60.2
Single Loops	$s = 1$		26.5	50 + 0	45 + 0	43 + 0	60 + 0	45 + 0	40 + 0	55.9	26.6	27.9
	$s = 2$		11.9	40 + 0	25 + 0	30 + 0	20 + 0	15 + 0	20 + 0	38.0	11.8	13.0
	RT		–	30	30	23	40	35	43	–	–	27.9
	RSE		43.4	100	100	100	0	0	0	24.3	28.1	27.9
Nested Loop	$s = 1$		28.4	40 + 0	40 + 0	37 + 0	50 + 0	45 + 0	43 + 0	66.4	29.8	33.2
	$s = 2$		13.6	20 + 0	25 + 0	23 + 0	30 + 0	30 + 0	23 + 0	48.0	14.3	17.4
	RT		–	30	35	37	40	40	33	–	–	33.2
	RSE		61.3	100	100	100	0	0	0	29.9	33.5	33.2
TreeMap	$cp = 0.6$	$l = 100$	15.2	0 + 30	10 + 30	7 + 27	30 + 10	25 + 5	23 + 7	75.9	15.2	15.5
		$l = 150$	20.3	0 + 70	10 + 55	7 + 50	30 + 10	25 + 5	23 + 10	76.3	20.2	20.5
	$cp = 0.7$	$l = 100$	19.1	0 + 30	15 + 25	20 + 20	30 + 0	25 + 0	30 + 0	76.2	19.0	19.3
		$l = 150$	26.3	0 + 80	15 + 50	20 + 40	30 + 10	25 + 5	30 + 7	76.7	26.0	26.2
	RT		–	0	20	17	0	0	10	–	–	26.2
	RSE		26.1	50	40	47	0	10	13	27.7	26.1	26.2
Apollo	$cp = 0.6$	$l = 190$	25.3	0 + 30	5 + 25	3 + 17	0 + 40	0 + 45	3 + 47	34.6	27.0	28.6
		$l = 230$	37.5	0 + 50	5 + 40	3 + 37	0 + 60	0 + 55	3 + 57	45.4	39.6	40.8
	$cp = 0.7$	$l = 190$	33.6	0 + 30	5 + 30	7 + 30	0 + 80	0 + 60	0 + 50	44.4	38.7	39.9
		$l = 230$	35.5	0 + 30	5 + 35	7 + 37	0 + 80	0 + 65	0 + 54	45.3	40.4	41.4
	RT		–	0	5	3	0	0	3	–	–	41.4
	RSE		38.6	20	30	33	50	35	37	43.5	40.9	41.4
CDx	$cp = 0.6$	$l = 250$	21.0	30 + 0	20 + 0	23 + 0	10 + 60	10 + 60	7 + 57	46.7	23.5	30.7
		$l = 300$	23.0	30 + 0	20 + 0	23 + 3	10 + 60	10 + 60	7 + 57	47.6	25.4	32.3
	$cp = 0.7$	$l = 250$	25.6	30 + 0	35 + 0	27 + 0	10 + 60	10 + 55	10 + 53	55.8	29.3	37.5
		$l = 300$	27.7	30 + 0	35 + 0	27 + 3	10 + 70	10 + 70	10 + 63	57.8	31.6	39.7
	RT		–	0	0	0	0	0	0	–	–	39.7
	RSE		34.7	30	30	30	50	45	40	50.9	34.7	39.7
TSAFE	$cp = 0.6$	$l = 190$	20.2	0 + 30	0 + 20	0 + 13	0 + 50	0 + 65	0 + 67	50.0	26.7	44.3
		$l = 230$	23.4	0 + 40	0 + 25	0 + 20	0 + 50	0 + 65	0 + 67	52.6	29.7	49.5
	$cp = 0.7$	$l = 190$	26.9	0 + 30	0 + 20	0 + 20	0 + 50	0 + 70	0 + 70	55.5	32.9	45.2
		$l = 230$	41.0	0 + 60	0 + 35	0 + 30	0 + 50	0 + 70	0 + 70	68.2	50.3	60.1
	RT		–	0	5	7	10	5	10	–	–	60.1
	RSE		42.7	50	50	53	40	45	43	63.0	56.6	60.1

sult also indicated the effectiveness of the heuristics used in `PerfPlotter` to select paths of interest. Besides, for the top 30, 20 and 10 best and worst cases, `PerfPlotter` had an increasing coverage, which demonstrated that it uncovers the best and worst cases with a good accuracy.

As *cp* and l increased in the case of Quick Sort, TreeMap, Apollo, CDx and TSAFE, the coverage for the best-case and worst-case paths also increased. This is because the first phase diversely explores more high-probability paths as *cp* increases, and thus paves the way for the second phase; and the second phase applies a stricter stopping criterion as l increases, and thus explores more paths of interest. We also found that, for these artifacts, most best-case and worst-case paths were exposed in the second phase. This empirically validates our assumption that corner cases like best-case and worst-case execution times likely occur in low-probability paths, and our path exploration in two phases is reasonable and effective.

As s increased in the case of Single Loops and Nested Loop, `PerfPlotter` exposed less best-case and worst-case paths, as illustrated in Figure 2 (c-d). This is reasonable because `PerfPlotter` selects a smaller set of representative loop paths for exploration as s increases. We found that, for these two loop programs, all the best and worst cases are uncovered in the first phase. The reason is that there are no other branches (except for loop conditions) in these two loop programs, and all the representative loop paths are designed to be explored together to capture the overall performance of loops.

In terms of time overhead, Table 1 shows that, `PerfPlotter` took 36.7% of the total time of the `ESE` approach for exploring 27.0% of the total paths on average since the path exploration in `PerfPlotter` is selective and thus needs more computation time than the `ESE` approach. If the time for running tests is more dominant than that for path exploration, which is the case for most artifacts except for Quick Sort and TSAFE, `PerfPlotter`'s overhead for path exploration will be insignificant. Note that, we analyzed a version of Quick Sort for $n = 8$, the `ESE` approach ran out of memory, while `PerfPlotter` produced a similar distribution to Figure 2 (a) (here we omit the distribution but it is available at our website [3]).

In summary, our experimental data in Table 1 showed that, `PerfPlotter` can find the best-case and worst-case paths that manifest the best-case and worst-case execution times with high coverage and low overhead by heuristically exploring a small set of program paths; and `PerfPlotter` is more accurate than the `RT` and `RSE` approaches and more time-efficient than the `ESE` approach to expose the performance of interest and generate performance distributions.

4.4 Application Scenarios (RQ4)

In this section, we report our experience of applying performance distributions generated by `PerfPlotter` in three typical scenarios: applying the performance distribution of a single version of a program to performance understanding, comparing the two performance distributions generated from the two versions of a program to bug validation, and comparing the performance distributions of functionally equivalent pro-

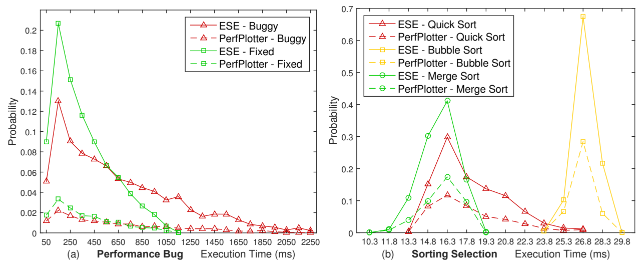

Figure 3: Application Scenarios of `PerfPlotter`.

grams to algorithm selection.

Performance Understanding. From the performance distribution of a program, we can obtain various performance indicators such as best-case, worst-case, average, most-likely and least-likely execution time as well as the corresponding test inputs. These indicators, especially the ones whose values are out of expectation, can help reveal performance or even functional bugs. For example, the artifact CDx contains a divide-by-zero bug in method `isInVoxel` of class `Reducer`, reported by SPF. To fix this bug, we need to add an exception handling routine and either return `true` or `false` for the method. However, the code is complex and it is not straightforward on which value should be returned to fix the bug. We generated a performance distribution for the case we return `true`, and discovered that CDx had abnormal worst-case execution times that were extremely larger than the others (see Figure 2 (i)). By investigating the collected path conditions of those abnormal worst cases and running the generated corresponding test inputs, we found that all those worst cases executed the true branch we added for fixing the bug. That is, our fix was the cause of those abnormal worst cases. By closely looking into the code, we found that the bug is complex and just adding a `return true` statement is not sufficient to fix it. Thus, we have reported the bug to the authors.

Bug Validation. Fixing bugs that manifest for some inputs may slow down the executions for other inputs [41]. Thus it is very important to comprehensively validate if a bug fix will cause any performance regressions. The performance distributions generated via `PerfPlotter` before and after the bug fix can serve this purpose. Figure 3 (a) shows such performance distributions for a program that has a performance bug in a nested loop. The inner loop has a redundant and expensive computation that computes the same result in each iteration of the outer loop. Thus, the redundant computation can be moved to the outer loop so that it is computed only once in each iteration of the outer loop. This bug was a simplified version adapted from the example in [42]. From Figure 3 (a), we can observe that, by comparing buggy and fixed versions, `PerfPlotter` accurately captured the performance speedup as the `ESE` approach did: the worst-case execution time sped up from 2250 to 1150 ms, and the average execution time sped up from 1150 to 600 ms. We also see that the probability of the most-likely execution time increased. Note that `PerfPlotter` cannot validate the correctness of bug fixes, but it can provide some hints on how the bug fix can impact the program performance in general, from which, we then can determine whether the bug fix can be correct, as shown for the divide-by-zero bug in CDx. It is our future work to apply difference algorithms [39] to paths so that we can track path-based performance regressions and locate their root cause.

Algorithm Selection. Developers often choose algorithms (e.g., sorting algorithms and data structures) randomly or by experience, which may lead to performance bugs [34, 52, 29]. The performance distributions can help visually compare the performance of different algorithms, and hence facilitate the selection. Figure 3 (b) shows the performance distributions of quick sort, bubble sort and merge sort. Compared to theoretical time complexity analysis, `PerfPlotter` can provide more fine-grained performance indicators, as we use the implementation details that often deviate from the theoretical time complexity. For example, both quick sort and bubble sort have the worst time complexity of $O(n^2)$, but `PerfPlotter` shows that quick sort's worst case is faster than bubble sort's by around 3 ms in this context (i.e., $n = 7$). This also indicates that the constant in algorithm analysis (using *Big-O* notation) really matters for performance analysis. By comparing their performance distributions, merge sort is the better choice in this context. Note that `PerfPlotter` cannot guarantee that the worst-case paths will actually be explored due to the heuristics nature. Another potential application is to apply `PerfPlotter` to configurable systems [54, 51] for predicting the performance distributions under different configuration options and then helping select the optional configuration.

4.5 Discussions

There are two main threats in our evaluation. First, the target artifacts were not very large and the target input domain was bounded tightly. Further studies are required to generalize our findings in real-life programs. Second, symbolic execution could fail to reflect real program execution behaviors due to the bounded exploration depth. Thanks to the impact of stratified sampling as in [10], the `ESE` approach can provide a satisfiable baseline. In fact, `PerfPlotter`, which is based on symbolic execution, can still improve the performance distributions over random testing.

`PerfPlotter` is configurable via a set of parameters. The testing sizes d_1 and d_2 can be tuned based on the variance of execution times as well as the number of solutions executing the path. A larger variance or number may suggest a larger testing size. For the key parameters cp, l and s, our empirical results suggest that, if a program contains paths that report high probability, we need to set cp to a high value so that a sufficient number of high-probability paths are explored. As in the case of Apollo, we set cp to 0.6. Meanwhile, l should be set to a large value, e.g., 190, so that the low-probability paths (which may have extremely small probability in such programs) could be sufficiently explored. If a program does not have paths of high-probability, cp and l should be set to a small value as in the case of Quick Sort. For s, it can be set to a small value, e.g., 1, if the time budget is loose; otherwise, it can be set to a large value but may reduce the coverage.

`PerfPlotter` has a few underlying assumptions, which may limit its application and threat its validity. First, we focus on sequential programs with integer and floating point inputs. We are extending `PerfPlotter` to support non-deterministic programs with the advances in probabilistic symbolic execution for such programs [37] and to support path conditions over strings with SMC [38]. Second, we assume usage profiles can be provided by developers. In fact, such profiles can be inferred by formal specifications [24] or automatic extraction techniques [40]. Lastly, we assume the performance of a path is approximately proportional to the number of instructions executed in the path, which is similarly done in the literature (e.g., [61, 59]), but do not explicitly consider the impact of environments (e.g., instruction pipelines, caches, JVM optimizations, or just-in-time compilation) on performance. As a result, the generated performance distributions might not be accurate. One remedy is to apply a more sophisticated performance model, e.g., incorporating low-level details for the instructions. Another remedy is to model the performance of a path itself as a distribution (which is more informative) but not an average value. This can also improve the applicability of `PerfPlotter` to programs where black-box APIs are invoked and take a huge amount of time for specific inputs.

5. RELATED WORK

Instead of listing all related work, we focus on the most related ones: static and dynamic program analysis-based performance analysis, and probabilistic symbolic execution.

5.1 Static Analysis-Based Approaches

Burnim et al. [12] use symbolic execution to generate worst-case test inputs for large input sizes, and Zhang et al. [61] propose a directed symbolic execution to derive high-load test inputs. In addition, a large body of work analyzes the worst-case execution time for real-time and embedded systems [56, 48]. These approaches focus on the worst-case performance, while our framework derives a performance distribution.

Zhang [59] predicts the performance of a program in terms of some performance indicators as a weighted sum of the performance of all feasible program paths. This work is the closest to ours. However, their description is quite general, and exhaustive symbolic execution is used to explore all the paths; while our framework partially explores the paths to generate a performance distribution and expose corner cases.

Buse and Weimer [13] predict the relative frequency of control flow paths by machine learning techniques. Differently, our framework uses probabilistic symbolic execution to get the absolute path frequency, which could be more accurate to construct the performance distribution. Gulwani et al. [27] compute an upper bound on the worst-case computational complexity for a program, while we generate a performance distribution of the actual execution times.

Chattopadhyay et al. [15] analyze the cache performance of real-time and embedded programs. They partition the input domain by path programs and compute the bound on cache misses of each path program by static invariant generation. Similarly, we also partition the input domain based on paths. Differently, we study execution time instead of cache misses, and explore paths heuristically rather than exhaustively.

5.2 Dynamic Analysis-Based Approaches

Several program profiling techniques (e.g., [4, 33, 19]) have been proposed to analyze execution traces of an instrumented program for predicting execution frequency of program paths and identifying hot paths. In contrast to our framework, these techniques need a larger number of executions to statistically cover the input domain for a high accuracy, and thus heavily rely on the given inputs. Moreover, our framework can automatically generate the test inputs by constraint solving.

Besides these path-sensitive profiling techniques, there has been some recent work on input-sensitive profiling techniques (e.g., [25, 58, 17]). They run an instrumented program with a set of different input sizes, measure the performance of each basic block, and fit a performance function with respect to input sizes. The accuracy of the performance function heavily relies on the chosen inputs, but our framework can automatically generate the test inputs. Moreover, different from their basic block profiling, we focus on paths because paths contain more information than individual basic blocks and path-based frequency can be more useful for developers to understand a program (see Ball et al. [5] for a detailed discussion).

Finally, some research efforts have been made on identifying the critical path of distributed systems [57, 7] and mobile applications [49, 60] by analyzing execution traces. Critical path analysis is often used to find performance bottlenecks. They deal with non-deterministic programs, while we currently focus on deterministic programs. In addition, these approaches identify the critical path, while our framework produces a performance distribution and exposes performance of interest with its probability of occurrence.

5.3 Probabilistic Symbolic Execution

Probabilistic symbolic execution [35, 23] has been recently proposed to support probabilistic analysis at the source code level. It has been applied to program understanding and debugging [23], reliability analysis [21], and quantification of information leaks [45] and software changes [20]. Here we apply probabilistic symbolic execution to performance analysis.

For the scalable probabilistic symbolic execution, Sankaranarayanan et al. [50] propose to compute interval bounds on the probability from an adequate set of paths; and Filieri et al. [22] propose a statistical symbolic execution. Their purpose is to compute the path probability efficiently, while we use some performance-specific heuristics to partially explore the paths to generate a performance distribution.

Further, several advances have been made to quantify the solution space for constraints. Sankaranarayanan et al. [50] propose a volume computation technique to quantify the solution space for linear constraints over bounded integers and floats. Luu et al. [38] propose a model counter to quantify the solution space for constraints over unbounded strings. These approaches are all orthogonal to our framework, and we plan to integrate [38] to support path constraints over strings.

6. CONCLUSIONS

In this paper, we proposed the concept of *performance distributions* as well as a performance analysis framework `PerfPlotter` for automatically generating them. `PerfPlotter` applies probabilistic symbolic execution to perform a selective path exploration and generate meaningful performance distributions for programs. We have implemented `PerfPlotter` and experimentally demonstrated that, by exploring a small set of program paths, `PerfPlotter` can generate the performance distribution that captures the general trend of the actual one and exposes the performance of interest with high coverage and low overhead; and such performance distributions can be useful in performance understanding, bug validation and algorithm selection. In the future, we plan to extend our framework to detect performance bugs [28, 36] and understand other aspects of software qualities (e.g., energy).

7. ACKNOWLEDGMENT

This work is supported in part by the National Research Foundation, Prime Minister's Office, Singapore under its National Cybersecurity R&D Program (Award NRF2014NCR-NCR001-30) and administered by the National Cybersecurity R&D Directorate, and by the US National Science Foundation (NSF) under Award 1542117.

8. REFERENCES

[1] Collision Detector. https://www.cs.purdue.edu/sss/projects/cdx/.

[2] MathWorks. http://www.mathworks.com/help/simulink/examples.html.

[3] PerfPlotter. http://pat.sce.ntu.edu.sg/bhchen/perfplotter.

[4] T. Ball and J. R. Larus. Efficient path profiling. In *MICRO*, pages 46–57, 1996.

[5] T. Ball, P. Mataga, and M. Sagiv. Edge profiling versus path profiling: The showdown. In *POPL*, pages 134–148, 1998.

[6] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni. Model-based performance prediction in software development: A survey. *IEEE Trans. Softw. Eng.*, 30(5):295–310, 2004.

[7] P. Barford and M. Crovella. Critical path analysis of tcp transactions. In *SIGCOMM*, pages 127–138, 2000.

[8] M. Borges, M. d'Amorim, S. Anand, D. Bushnell, and C. S. Pasareanu. Symbolic execution with interval solving and meta-heuristic search. In *ICST*, pages 111–120, 2012.

[9] M. Borges, A. Filieri, M. D'Amorim, and C. S. Păsăreanu. Iterative distribution-aware sampling for probabilistic symbolic execution. In *FSE*, 2015.

[10] M. Borges, A. Filieri, M. D'Amorim, C. S. Păsăreanu, and W. Visser. Compositional solution space quantification for probabilistic software analysis. In *PLDI*, pages 123–132, 2014.

[11] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on java predicates. In *ISSTA*, pages 123–133, 2002.

[12] J. Burnim, S. Juvekar, and K. Sen. Wise: Automated test generation for worst-case complexity. In *ICSE*, pages 463–473, 2009.

[13] R. P. L. Buse and W. Weimer. The road not taken: Estimating path execution frequency statically. In *ICSE*, pages 144–154, 2009.

[14] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224, 2008.

[15] S. Chattopadhyay, L. K. Chong, and A. Roychoudhury. Program performance spectrum. In *LCTES*, pages 65–76, 2013.

[16] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Trans. Softw. Eng.*, 2(3):215–222, 1976.

[17] E. Coppa, C. Demetrescu, and I. Finocchi. Input-sensitive profiling. In *PLDI*, pages 89–98, 2012.

[18] J. A. De Loera, B. Dutra, M. KöPpe, S. Moreinis, G. Pinto, and J. Wu. Software for exact integration of polynomials over polyhedra. *Comput. Geom. Theory Appl.*, 46(3):232–252, 2013.

[19] E. Duesterwald and V. Bala. Software profiling for hot path prediction: Less is more. In *ASPLOS*, pages 202–211, 2000.

[20] A. Filieri, C. S. Pasareanu, and G. Yang. Quantification of software changes through probabilistic symbolic execution (N). In *ASE*, pages 703–708, 2015.

[21] A. Filieri, C. S. Păsăreanu, and W. Visser. Reliability analysis in symbolic pathfinder. In *ICSE*, pages 622–631, 2013.

[22] A. Filieri, C. S. Păsăreanu, W. Visser, and J. Geldenhuys. Statistical symbolic execution with informed sampling. In *FSE*, pages 437–448, 2014.

[23] J. Geldenhuys, M. B. Dwyer, and W. Visser. Probabilistic symbolic execution. In *ISSTA*, pages 166–176, 2012.

[24] M. Gittens, H. Lutfiyya, and M. Bauer. An extended operational profile model. In *ISSRE*, pages 314–325, 2004.

[25] S. F. Goldsmith, A. S. Aiken, and D. S. Wilkerson. Measuring empirical computational complexity. In *ESEC-FSE*, pages 395–404, 2007.

[26] M. Grechanik, C. Fu, and Q. Xie. Automatically finding performance problems with feedback-directed learning software testing. In *ICSE*, pages 156–166, 2012.

[27] S. Gulwani, K. K. Mehra, and T. Chilimbi. Speed: Precise and efficient static estimation of program computational complexity. In *POPL*, pages 127–139, 2009.

[28] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu. Understanding and detecting real-world performance bugs. In *PLDI*, pages 77–88, 2012.

[29] C. Jung, S. Rus, B. P. Railing, N. Clark, and S. Pande. Brainy: Effective selection of data structures. In *PLDI*, pages 86–97, 2011.

[30] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[31] S. Kounev. Performance modeling and evaluation of distributed component-based systems using queueing petri nets. *IEEE Trans. Softw. Eng.*, 32(7):486–502, 2006.

[32] H. Koziolek. Performance evaluation of component-based software systems: A survey. *Perform. Eval.*, 67(8):634–658, 2010.

[33] J. R. Larus. Whole program paths. In *PLDI*, pages 259–269, 1999.

[34] X. Li, M. J. Garzarán, and D. Padua. A dynamically tuned sorting library. In *CGO*, pages 111–, 2004.

[35] S. Liu and J. Zhang. Program analysis: From qualitative analysis to quantitative analysis. In *ICSE (NIER Track)*, pages 956–959, 2011.

[36] Y. Liu, C. Xu, and S.-C. Cheung. Characterizing and detecting performance bugs for smartphone applications. In *ICSE*, pages 1013–1024, 2014.

[37] K. Luckow, C. S. Păsăreanu, M. B. Dwyer, A. Filieri, and W. Visser. Exact and approximate probabilistic symbolic execution for nondeterministic programs. In *ASE*, pages 575–586, 2014.

[38] L. Luu, S. Shinde, P. Saxena, and B. Demsky. A model counter for constraints over unbounded strings. In *PLDI*, pages 565–576, 2014.

[39] E. W. Myers. An o(nd) difference algorithm and its variations. *Algorithmica*, 1(1-4):251–266, 1986.

[40] M. Nagappan, K. Wu, and M. A. Vouk. Efficiently extracting operational profiles from execution logs using suffix arrays. In *ISSRE*, pages 41–50, 2009.

[41] A. Nistor, P.-C. Chang, C. Radoi, and S. Lu. Caramel:

Detecting and fixing performance problems that have non-intrusive fixes. In *ICSE*, 2015.

[42] A. Nistor, L. Song, D. Marinov, and S. Lu. Toddler: Detecting performance problems via similar memory-access patterns. In *ICSE*, pages 562–571, 2013.

[43] C. Pasareanu, J. Schumann, P. Mehlitz, M. Lowry, G. Karsai, H. Nine, and S. Neema. Model based analysis and test generation for flight software. In *SMC-IT*, pages 83–90, 2009.

[44] W. R. Pestman. *Mathematical Statistics: An Introduction*. Walter de Gruyter, 1998.

[45] Q.-S. Phan, P. Malacaria, C. S. Păsăreanu, and M. D'Amorim. Quantifying information leaks using reliability analysis. In *SPIN*, pages 105–108, 2014.

[46] C. Prud'homme, J.-G. Fages, and X. Lorca. *Choco3 Documentation*. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S., 2014.

[47] C. S. Păsăreanu, W. Visser, D. Bushnell, J. Geldenhuys, P. Mehlitz, and N. Rungta. Symbolic PathFinder: Integrating symbolic execution with model checking for Java bytecode analysis. *Automat. Softw. Eng.*, 20(3):391–425, 2013.

[48] P. Puschner and A. Burns. Guest editorial: A review of worst-case execution-time analysis. *Real-Time Systems*, 18(2):115–128, 2010.

[49] L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Shayandeh. Appinsight: Mobile app performance monitoring in the wild. In *OSDI*, pages 107–120, 2012.

[50] S. Sankaranarayanan, A. Chakarov, and S. Gulwani. Static analysis for probabilistic programs: Inferring whole program properties from finitely many paths. In *PLDI*, pages 447–458, 2013.

[51] A. Sarkar, J. Guo, N. Siegmund, S. Apel, and K. Czarnecki. Cost-efficient sampling for performance prediction of configurable systems. In *ASE*, 2015.

[52] O. Shacham, M. Vechev, and E. Yahav. Chameleon: Adaptive selection of collections. In *PLDI*, pages 408–418, 2009.

[53] D. Shen, Q. Luo, D. Poshyvanyk, and M. Grechanik. Automating performance bottleneck detection using search-based application profiling. In *ISSTA*, pages 270–281, 2015.

[54] N. Siegmund, S. S. Kolesnikov, C. Kästner, S. Apel, D. Batory, M. Rosenmüller, and G. Saake. Predicting performance via automated feature-interaction detection. In *ICSE*, pages 167–177, 2012.

[55] R. Tawhid and D. Petriu. Automatic derivation of a product performance model from a software product line model. In *SPLC*, pages 80–89, 2011.

[56] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem – Overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, 2008.

[57] C.-Q. Yang and B. Miller. Critical path analysis for the execution of parallel and distributed programs. In *ICDCS*, pages 366–373, 1988.

[58] D. Zaparanuks and M. Hauswirth. Algorithmic profiling. In *PLDI*, pages 67–76, 2012.

[59] J. Zhang. Performance estimation using symbolic data. In *Theories of Programming and Formal Methods*, pages 346–353. 2013.

[60] L. Zhang, D. R. Bild, R. P. Dick, Z. M. Mao, and P. Dinda. Panappticon: Event-based tracing to measure mobile application and platform performance. In *CODES+ISSS*, pages 33:1–33:10, 2013.

[61] P. Zhang, S. Elbaum, and M. B. Dwyer. Automatic generation of load tests. In *ASE*, pages 43–52, 2011.