Synthesizing Programs That Expose Performance Bottlenecks

Luca Della Toffola
Department of Computer Science
ETH Zurich
Switzerland

Michael Pradel
Department of Computer Science
TU Darmstadt
Germany

Thomas R. Gross
Department of Computer Science
ETH Zurich
Switzerland

Abstract

Software often suffers from performance bottlenecks, e.g., because some code has a higher computational complexity than expected or because a code change introduces a performance regression. Finding such bottlenecks is challenging for developers and for profiling techniques because both rely on performance tests to execute the software, which are often not available in practice.

This paper presents PerfSyn, an approach for synthesizing test programs that expose performance bottlenecks in a given method under test. The basic idea is to repeatedly mutate a program that uses the method to systematically increase the amount of work done by the method. We formulate the problem of synthesizing a bottleneck-exposing program as a combinatorial search and show that it can be effectively and efficiently addressed using well known graph search algorithms. We evaluate the approach with 147 methods from seven Java code bases. PerfSyn automatically synthesizes test programs that expose 22 bottlenecks. The bottlenecks are due to unexpectedly high computational complexity and due to performance differences between different versions of the same code.

CCS Concepts • **Theory of computation → Discrete optimization**; **Optimization with randomized search heuristics**; • **Computing methodologies → Discrete space search**; • **Software and its engineering → Dynamic analysis**; **Software performance**;

Keywords program synthesis, dynamic analysis, profiling, performance bottlenecks, fuzzing

CGO'18, February 24–28, 2018, Vienna, Austria

ACM ISBN 978-1-4503-5617-6/18/02…$15.00
https://doi.org/10.1145/3168830

ACM Reference Format:
Luca Della Toffola, Michael Pradel, and Thomas R. Gross. 2018. Synthesizing Programs That Expose Performance Bottlenecks. In *Proceedings of 2018 IEEE/ACM International Symposium on Code Generation and Optimization (CGO'18).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3168830

1 Introduction

The performance of software is critical in many domains, e.g., to achieve the desired throughput of a server, to reduce the energy consumption on resource-constrained devices, or to shield users from waiting for an unresponsive application. Previous work has shown that real-world software often suffers from performance bottlenecks [27, 30, 43], and that some of them can be fixed with relatively little effort [22, 33, 49]. Unfortunately, finding such bottlenecks often is non-trivial as it requires executing the inefficient code with input data that brings the bottleneck to the attention of a profiling tool, a developer, or in the worst case, a user. Because developers want to find bottlenecks before deploying the software, techniques for exposing bottlenecks as part of in-house quality control are needed.

As a motivating example, consider the class `Messages` in Listing 1. The class represents an ordered set of unique messages that maintains the messages order using a list-like data structure. The `log` method contains two bottlenecks. The first bottleneck consists of an inefficient containment check for the argument list of messages. The method uses for the check the list representation of the existing messages, this is inefficient because it requires a linear scan of the list, instead of the set representation, which allows for a faster check.

Finding such inefficiencies through profiling involves two aspects: First, it requires inputs that execute the inefficient code in a way that exposes the bottleneck. For the given example above, such inputs must attempt to add a new element multiple times. This input will trigger an increase of the execution time, which exposes the bottleneck. Second, finding a bottleneck requires a dynamic analysis that, given an execution that exposes the bottleneck, detects and reports the problem.

Existing work addresses the second requirement, e.g., by empirically measuring the complexity of code through profiling [11, 21, 60] or by identifying instances of known performance anti-patterns [22, 27, 30, 34, 43, 49, 53, 55]. In contrast,

the first requirement – to trigger an execution that exposes the bottleneck – is currently understudied. Most code has manually crafted or automatically generated correctness tests, but these tests are usually insufficient to detect performance problems. The reason is that detecting such problems requires exercising the inefficient code with particular kinds of inputs and with different input sizes. Instead, correctness testing typically focuses on covering each statement, branch, or path once.

This paper presents PerfSyn, an automatic approach for synthesizing inputs that expose performance bottlenecks. We focus on analyzing code at the unit-level, i.e., to find bottlenecks in individual methods. In this case, "inputs" are test programs that prepare an object under test and then call the method under test. PerfSyn starts with a minimal usage example of the method under test and then applies a sequence of program mutations that add, remove, or modify statements in the program.

Our first key contribution is to formulate the problem of finding a sequence of mutations that yields a bottleneck-exposing program as a combinatorial search problem. To efficiently address the combinatorial search problem we adapt well-known graph search algorithms [15, 25]. To this end, the approach learns from the execution feedback of synthesized programs to steer the search towards mutations that generate bottleneck-exposing programs.

Our second key contribution is to design PerfSyn as a general framework that can expose different kinds of performance bottlenecks. Once the approach hits a program that exposes a specific bottleneck, this program along with information recorded from the program execution is reported to the developers, who then can fix the problem.

We apply PerfSyn in two usage scenarios. The first scenario, PerfSyn allows for comparing the performance of two versions of a program, e.g., for performance regression testing. In this scenario, the approach reports a warning when the performance of the analyzed implementation changes compared to the performance of a reference implementation of the code.

In the second scenario assumes that a developer validates her assumption about the computational complexity of code against the actual performance, i.e., execution time of the code. In this scenario, the developer annotates pieces of code, such as individual methods, with a complexity class, such as $O(n)$. If PerfSyn finds a way to increase the input size so that the execution time of the code increases beyond the assumed complexity class, e.g., superlinear, then it reports a warning. We found this scenario to be a common practice (e.g., in bug repositories) where developers discuss the sub-optimal computational complexity of a method, and in previous work that addresses this kind of problem [11, 37, 60].

To the best of our knowledge, this paper is the first to provide a general framework for synthesizing programs that expose performance-bottlenecks. The two closest existing approaches use symbolic execution to trigger the worst-case execution time of some code [7] and generate tests that expose inefficiencies related to nested loops [14]. In contrast, PerfSyn supports various kinds of performance bottlenecks, e.g., exposed via the usage scenarios described above. Another difference is that both existing approaches rely on a complex whitebox analysis of the code under test, whereas PerfSyn handles the code mostly as a blackbox.

```
1 class Messages {
2   Set<Msg> set = ...
3   List<Msg> list = ...
4
5   Messages(List<Msg> msgs) {
6     /* Add messages to set and list */
7   }
8
9   void log(List<Msg> newMsgs, boolean renameDups) {
10    for (Msg msg : newMsgs) {
11      if (renameDups) {
12        if (set.contains(msg)) {
13          logOne(makeUnique(msg));
14        } else {
15          logOne(msg);
16        }
17      } else {
18        /* Bottleneck 1:
19         * Inefficient containment check */
20        if (!list.contains(msg)) {
21          logOne(msg);
22        }
23      }
24    }
25  }
26
27  void logOne(Msg msg) {
28    /* Add msg to set and list */
29  }
30
31  /* Bottleneck 2: Expensive hash function */
32  Msg makeUnique(Msg msg) {
33    /* append expensive-to-compute hash to msg */
34  }
35 }
```

Listing 1. Class with two bottlenecks.

We implement PerfSyn for Java code and apply it to 147 methods from seven popular code bases. The approach successfully synthesizes programs that expose 22 bottlenecks, which are due to unexpectedly high computational complexity and due to performance differences between different versions of a class.

In summary, this paper contributes the following:

- We present the first general framework for synthesizing bottleneck-exposing programs.
- We are the first to formulate the problem of finding performance problems as a combinatorial search problem.
- We present empirical evidence that the approach is effective and efficient in exposing performance problems in real-world Java classes.

Our implementation and all experimental data are available for download at [1].

2 PerfSyn by Illustration

This section motivates our work with an example and illustrates the key ideas of PerfSyn.

Motivating Example. Listing 1 shows the code of `Messages`, a container class for logging messages. The class ensures that each message is logged only once and optionally renames messages to make them unique. To store messages the class provides the method `log`, which takes two arguments: a list of new messages and a flag to indicate whether duplicate messages should be renamed into unique messages. The implementation of the class stores messages in two data structures. First, it uses a set to quickly check whether a message has already been logged. Second, it uses a list to represent the order of messages. The helper method `logOne` adds a new message to both the set and the list. If the user wants duplicate messages to be renamed, then another helper method, `makeUnique`, changes the message by appending a unique string. The example contains two performance bottlenecks:

- The first bottleneck at line 20 is an inefficient containment check that accidentally uses the expensive `List.contains` instead of `Set.contains`. A developer may expect the containment check to require constant time, but instead, the required time is linear w.r.t. the number of already logged messages, making the overall complexity of `log` quadratic in the number of messages.
- The second bottleneck causes suboptimal performance when already contained messages get renamed. The reason is that `makeUnique` uses an expensive hash function to append a unique string to the message (line 33). Instead, a more efficient implementation would be to append a unique number obtained via a global counter.

Challenge: Expose Bottlenecks. Finding bottlenecks, such as those in our motivating example, in complex software is non-trivial. The most successful approach to find bottlenecks in practice is profiling. However, effective profiling depends on inputs that trigger an execution that exposes a bottleneck. In practice, most code comes without extensive performance tests but only a correctness test suite, or maybe even only a set of minimal usage examples [29].

For the example, program p_0 in Figure 1 shows a minimal usage example of the `Messages` class. The program initializes the class with an empty list of messages and then calls `log` with two arguments: an empty list and the constant `true`. Profiling an execution of this program does not expose the performance bottlenecks in the class. For bottleneck 1, the program fails to trigger the inefficient check because it requires setting the flag to `false` and to log at least two messages. For bottleneck 2, the program fails to expose the inefficiency because it requires repeatedly logging the same message.

Synthesizing Bottleneck-Exposing Programs. PerfSyn searches for bottlenecks in the method under test m_{ut} starting from an initial program p_0. The program p_0 in Figure 1 is generated by PerfSyn and it contains the minimal number of statements required to execute m_{ut} without a crash [1]. To search for a bottleneck PerfSyn modifies p_0, e.g., by inserting method calls or by modifying the values passed to the calls. The approach identifies a program as bottleneck-exposing based on a configurable performance oracle.[2] For example, the performance oracle may report that the measured complexity differs from the expected worst-case complexity [11, 21, 60] or that two implementations with supposedly equal performance have different performance properties.

For the two bottlenecks in our example, let `log` be the method under test. Suppose that PerfSyn modifies the initial program p_0 into the programs p_m and p_n in Figure 1, respectively. The programs that led to p_m initialize the class and then add an increasingly large list of messages while setting the `renameDups` flag to `false`. That is, the programs reach the inefficient check at line 20 with an increasingly large input. PerfSyn profiles the execution of `log` with these programs and creates the performance plot shown next to program p_m. The plot shows the execution time depending on the input size and compares the measured values to the expected complexity.

For the second bottleneck, suppose that an alternative version of the `Messages` class uses a more efficient implementation of `makeUnique`. PerfSyn exposes the bottleneck in Listing 1 using the programs that lead to p_n in Figure 1. The programs pass to `log` an increasingly large number of duplicate messages, each of which triggers the `makeUnique` function. The performance plot next to p_n compares the two implementations with each other and shows that the implementation in Listing 1 performs sub-optimally.

Targeted Search. The key challenge in exposing bottlenecks by automatically synthesizing programs is the large space of possible programs. This paper presents a novel approach to address this challenge based on feedback obtained from executing programs. Starting from the initial program p_0, PerfSyn represents the space of possible modifications as a tree, where p_0 is the root, each node is another possible program, and edges represent code modifications that turn one program into another one. The approach explores the tree while gathering feedback about how effective specific mutations are at getting closer to a bottleneck-exposing program. The feedback depends on the performance oracle used to

[1] In the case PerfSyn is not able to automatically generate the program because the setup code is too complicated (e.g., a data-structure required to be in a specific state), PerfSyn can start from an existing test.

[2] The term "oracle" is inspired by test oracles, which decide whether a test exposes an error [4]. In contrast, the performance oracle decides to what extent a program exposes a performance bottleneck.

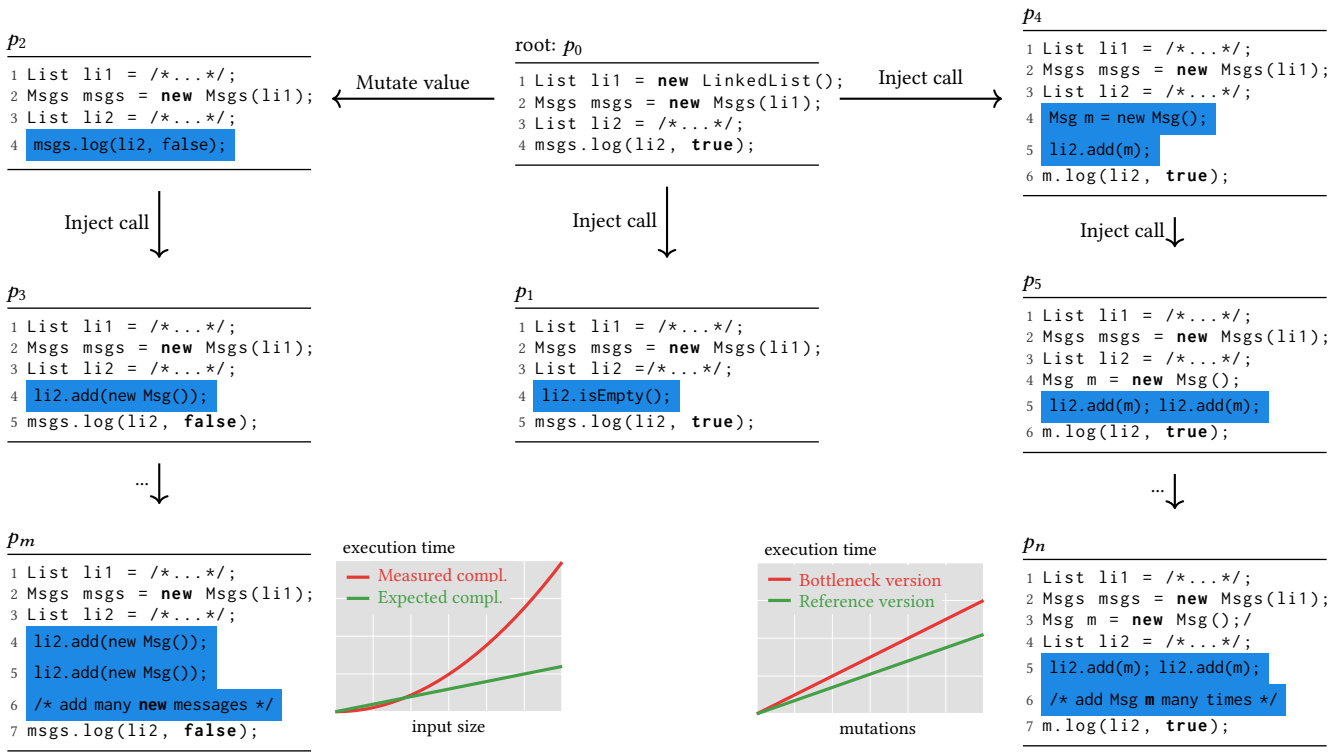

Figure 1. Running example. Each box represents a program. The plots at the bottom summarize execution feedback for the left and right path, respectively.

identify bottleneck-exposing programs. For example, an oracle aimed at exposing an unexpected complexity class steers the approach toward programs where the observed complexity diverges more and more from the expected complexity. Likewise, an oracle aimed at exposing performance differences between two implementations targets the approach toward programs with such differences.

For the running example, suppose that PerfSyn at first modifies p_0 by calling `isEmpty` on the list given to `log`, as shown in program p_1 in Figure 1. Because this mutation does not influence the performance of the method under test in the intended way, the approach decreases the priority of this part of the tree. Instead, PerfSyn learns over time that applying the mutations that lead to p_m and p_n are beneficial and steers the search along these mutations.

3 Approach

PerfSyn is a feedback-directed code synthesis approach with the goal to synthesize a program that exposes a bottleneck. The feedback is in the form of runtime performance measurements. We say that a method suffers from a bottleneck when the value of a performance property of the executed method with an input of a particular size exceeds the value expected by the developer. We design PerfSyn as a generic framework that supports different kinds of bottlenecks and strategies to expose them.

The main data structure created and manipulated by PerfSyn is a program that uses a method under test

Definition 3.1 (Program). A program $p \in \mathcal{P}$ for a method under test m_{ut} is a sequence of statements of the form $v_o = v_{arg_0}.m(v_{arg_1}, .., v_{arg_k})$, or $v_o = OP(v_{arg_0}, .., v_{arg_k})$ where:

- $\mathcal{P}$ is the set of all programs,
- v_o and $v_{arg_{0..k}}$ are typed variables,
- m is a method name,
- OP is a type-specific operator, and
- $m = m_{ut}$ for the last statement in the sequence.

The input to PerfSyn is a class that contains the method under test and a possibly empty initial program p_0. The output of PerfSyn is another program, which has been derived from the initial program and which exposes a bottleneck in the method under test. To derive such a program PerfSyn, performs sequences of mutations that, starting from p_0, feed the result of each mutation into the subsequent mutation. The space of all possible sequences of mutations for a specific initial program forms a tree:

Definition 3.2 (Mutation tree). A mutation tree for an initial program p_0 is an acyclic, connected, and directed graph

Algorithm 1 Synthesize bottleneck-exposing program

Input: Initial program p_0, best programs set $\mathcal{F}_{best} = \emptyset$, prediction table $\mathcal{L} = \emptyset$
Output: Program p_{best} that exposes a bottleneck

```
1:  while no timeout do
2:      for nb_mut ← 1 to maxMuts do
3:          p ← copy of p_0
4:          F ← empty sequence
5:          for step ← 1 to nb_mut do
6:              μ ← pickMutation(p, L)            ▷ Mutate program
7:              p ← mutate(p, μ)
8:              (f_1, .., f_j) ← execute(p)       ▷ Execute program
9:              add (f_1, .., f_j) to F
10:             score ← oracle(F)                 ▷ Rank best solutions
11:             update F_best with (p, score)
12:             L ← learn(p, F)                   ▷ Learn from execution of program
13: p_best = maxByScore(F_best)
```

$(\mathcal{P}, \mathcal{M})$, where each node $p \in \mathcal{P}$ is a program and each edge is a mutation $\mu \in \mathcal{M}$ that modifies the source node's program into the destination node's program.

The mutation tree represents all possible sequences of mutations that can be applied to a given initial program. Since the number of possible mutations per program is finite, the number of outgoing edges per node is also finite. In contrast, the height of the mutation tree is unbounded, because each mutation results in a new program that can always be further mutated.

To expose a program with a bottleneck PerfSyn repeatedly performs five steps (Algorithm 1):

1. *Mutate a program.* Transform a program into another program by modifying or adding statements. To this end, function *pickMutation* takes an existing program p and decides which mutation to apply next (line 6).
2. *Execute and gather feedback.* Execute a program while applying a dynamic analysis that runtime properties of the program (line 8) that serve as feedback.
3. *Learn.* Check if the applied mutation changed the performance of the method under test based on the oracle feedback and infer which mutations are most effective (line 12).
4. *Explore.* Steer the synthesis toward mutations of the initial program that may yield a bottleneck-exposing program. To this end, the algorithm iteratively explores the mutation tree (line 3) and ranks the most effective mutations (line 10).

The search continuously updates the set of best programs $\mathcal{F}_{best}$ until exceeding a configurable time budget and then returns the program that has the highest score, i.e., the program that is most likely to expose a bottleneck (line 13).

3.1 Mutating Programs

To change the runtime behavior of the method under test, PerfSyn transforms an existing program by applying a *mutation*.

Definition 3.3 (Mutation)**.** A mutation $\mu \in \mathcal{M}$ transforms a given program p into another program p'.

Each mutation consists of a *mutation operation*, i.e., how to transform a program, and of multiple *mutation operands*, i.e., the program elements subject of the program transformation.

Mutation Operators. The approach supports three kinds of operators:

- *Inject call.* This operator mutates a program by calling a method on any of the existing objects. To this end, the operator inserts a new statement $v_o = v_{arg_0}.m(v_{arg_1}, .., v_{arg_k})$ into the sequence of statements. The rationale for including this operator is to modify the state that may influence the performance of the method under test. In Figure 1, this operator synthesizes p_1 from p_0 by inserting the call to `li2.isEmpty()`.
- *Modify constructor.* This operator mutates a program by replacing an existing constructor call with a subtype constructor call. PerfSyn supports this operator to trigger bottlenecks that require an instance of a specific class. In Figure 1, this operator could synthesize a variant of p_0 where the first statement calls `new ArrayList()` instead of `new LinkedList()`.
- *Mutate value.* This operator applies a type-specific operation to one of the existing variables. Specifically, the operator inserts a statement $v_o = OP(v_{arg_0}, .., v_{arg_k})$ that applies a type-specific operator to a set of variables. PerfSyn supports operators for primitive types, such as incrementing an integer variable or toggling a boolean variable. In Figure 1, the operator synthesizes p_2 from p_0 by mutating the boolean value passed to `log` from `true` to `false`.

Applying Mutation Operators Applying an operator to a specific program requires several decisions. At first, PerfSyn selects a position in the sequence of existing statements where to apply the operator. Then, the approach concretizes the operator by selecting the method to call, the constructor to call, or the type-specific operator to apply. Finally, PerfSyn binds all variables of the operator either to variables in the existing program or to newly created variables. If during this final step, no variable with an appropriate type exists or when PerfSyn decides to use a new variable, then the approach recursively inserts new statements to initialize a variable of the required type.

For the example in Figure 1, consider the mutation that synthesizes p_3 from p_2 by applying the "inject call" operator. When applying this operator, PerfSyn decides to apply the mutation before the call to `log` and to use the `List.add` method. Furthermore, when trying to bind the argument to `add` to a variable, PerfSyn decides to create a fresh value and recursively inserts a statement, for example, a call to the constructor `new Msg()`.

The set of possible mutations for a particular program depends on which mutation operators are applicable and on

the number of ways that these operators can be applied. The number of mutations for a program p is always finite because the number of statements in p, the number of variables in p, the number of methods callable in p, and the number of type-specific operators are finite.

3.2 Gathering Execution Feedback

To determine whether a program p exposes a bottleneck, the approach executes and dynamically analyzes p.

Definition 3.4 (Execution feedback)**.** The feedback obtained by executing a program p is a tuple $(f_1, \ldots, f_j)$, where each f_i $(1 \leq i \leq j)$ represents a dynamically measured property of p's execution.

The tuple of measurements may contain any value of interest to understand the performance of the method under test, such as execution time, memory consumption, or amount of network traffic. Based on the execution feedback obtained for each program in a path through the mutation tree, an oracle decides how useful a program is for exposing a bottleneck:

Definition 3.5 (Performance oracle)**.** Let $(p_0 \xrightarrow{\mu_1} p_1 \xrightarrow{\mu_2} \ldots p_k)$ be a path through the mutation tree. Given a sequence of execution feedbacks F, where each element represents the feedback for a program p_i $(0 \leq i \leq k)$ on the path, the performance oracle is a function $F \rightarrow \mathbb{R}$. A higher value returned by the oracle indicates that the program p_k is closer to exposing a bottleneck.

3.2.1 Learning from Executions

To pick the next mutation, the algorithm selects one of the outgoing edges of the current program. The main challenge is to decide about the potential effectiveness of a mutation $\mu_{predict}$ before having performed it. To address this challenge, after each iteration the algorithm learns from the previously performed mutations to predict the effectiveness of similar mutations (line 12). At first, the algorithm computes the set $\mathcal{M}_{learn}$ of already performed mutations μ_{learn} that fulfill three conditions: (i) μ_{learn} and $\mu_{predict}$ share the same mutation operator; (ii) the operator is applied with the same method, constructor, or type-specific operator; (iii) if the mutation refers to existing variables, then both μ_{learn} and $\mu_{predict}$ share the same choice as to whether to use existing or new variables. Based on this set of already performed mutations to learn from, the algorithm builds a prediction table $\mathcal{L}$ that indicates for each mutation how often it has increased the value returned by the oracle in previous iterations.

3.3 Exploring the Search Space

The key step of PerfSyn is to explore the mutation tree to find a bottleneck-exposing program. Because of the large number of possible mutations, exhaustively exploring the tree is impractical, even when setting a bound on the exploration depth. In principle, any search algorithm, metaheuristic, or other machine learning technique that solves combinatorial optimization problems can be used to explore the mutation tree. We implement and experimentally compare two algorithms in this paper: (i) the A* algorithm [25], because it is one of the most popular and widely used graph search algorithms; and (ii) Ant Colony Optimization (ACO) [15], which we find to be well-suited for our problem.

In the following paragraphs we explain how we concretize the generic parts of the exploration strategy using these two approaches.

3.3.1 A* Search

A* [25] is a path finding algorithm that heuristically and iteratively builds a solution to reach a goal state. The algorithm selects at each iteration step the most promising solution, i.e., a solution that maximizes the cost function $f(n) = g(n) + h(n)$. Function $g(n)$ represents the cost for a path from the root to node n, and $h(n)$ heuristically estimates the cost of traversing node n to reach the goal.

We adapt the A* algorithm to the problem of finding in the mutation tree a path that exposes a bottleneck. A node n is a program, and the function $f(n)$ is the performance oracle's score for this program. Our version of A* does not use the heuristic function h, but PerfSyn uses only the feedback obtained from the execution $g(n)$. In other words, the algorithm prioritizes mutations that increase the *score* value returned by the oracle for a program. To bound the memory used by the search, we use the SMA* variant of the A* algorithm [42].

3.3.2 Ant Colony Optimization (ACO)

ACO [15] is an iterative algorithm to find an approximate solution to a combinatorial optimization problem. The intuition of the algorithm, which gave it its name, is that a set of ants traverse the graph while leaving pheromones on edges between components. Pheromones are numeric weights that are obtained by evaluating partial solutions and that evaporate over time. Pheromones encode how close already explored paths are to its goal. Based on the pheromones from previous iterations, ants prioritize paths through the solution space in such a way that the ants steer toward a solution with high pheromone values. This process iteratively improves the explored solutions until meeting some stopping criterion.

We adapted ACO to iteratively explore paths through the mutation tree using a set of ants that independently select the next mutation. During an iteration, each ant traverses the mutation tree and picks the next mutation based on the probability distribution described by *prob* for all outgoing edges:

$$prob(\mu) = \frac{\mathcal{T}(\mu) \cdot \mathcal{L}(\mu)}{\sum_{\mu' \in outgoing(p)} \mathcal{T}(\mu') \cdot \mathcal{L}(\mu')}$$

The map $\mathcal{T}$ yields a default pheromone value for all not yet performed mutations and it yields the current pheromone

for the already explored mutations. In other words, the algorithm initially assigns equal probabilities to all the mutations, and then focuses more and more on promising mutations. The map $\mathcal{L}$ is the prediction table that assigns to each mutation its expected effectiveness in changing the performance. The algorithm updates pheromones values τ after all ants have completed a sequence of nb_{mut} mutations:

$$\tau_{new} = \begin{cases} (1-\varphi) * \tau_{old} + \tau_{old} * score & \text{if on best path} \\ (1-\varphi) * \tau_{old} & \text{otherwise} \end{cases}$$

The algorithm reduces each pheromone by a constant factor φ, i.e., the pheromones evaporate. Furthermore, the pheromones of all mutations involved in the best path found so far are increased proportionally to the score returned by the oracle. Moreover, the algorithm bounds pheromone values in such a way that no mutation ever becomes impossible, and that the search continues to explore new mutations even after finding a promising path [46].

3.4 Feedback and Performance Oracle

Which of the synthesized programs is the best is determined by the performance oracle and depends on the kind of performance problem to search. Instantiating our framework for a specific kind of bottleneck requires to define a feedback function that evaluates to a higher score when the program is closer to reach the goal of exposing a bottleneck. The following presents two feedback functions targeted at specific kinds of bottlenecks.

3.4.1 Exposing Changes in Relative Performance

The first application of our generic framework detects performance bottlenecks fixed or introduced into another functionally equivalent implementation. Prior work to detect performance regressions [10, 17, 57] requires inputs that exercise the analyzed code. In contrast, we focus on how to automatically create inputs that expose a performance difference. To use PerfSyn to find bottlenecks due to changes in relative performance, the developer provides a second implementation of the analyzed code, e.g., an earlier version of the same class. We call this alternative implementation the *reference implementation*.

Execution Feedback The feedback for a path through the mutation tree is a sequence $F = [(c_0, c_0^{ref}), \ldots, (c_k, c_k^{ref})]$ where each pair (c_j, c_j^{ref}) contains the execution cost for the implementation under analysis and the reference implementation, respectively. During both executions, the approach gathers feedback about the execution cost. Because naively measuring the execution cost as wallclock time is inaccurate for short methods we instead measure the number of evaluated conditional checks [40, 58].

Performance Oracle The oracle computes the score of a program p_j as the difference in execution cost of the implementation under analysis and the reference implementation using the formula:

$$score(F) = |c_j^{ref} - c_j|$$

That is, the oracle steers the search toward paths that maximize the difference in execution cost between the two versions of the method. This formulation of the score enables PerfSyn to find paths in the mutation tree where the reference implementation is faster (i.e., the new implementation introduces a bottleneck) and where the reference implementation is slower (i.e., the new implementation fixes a bottleneck).

3.4.2 Exposing Unexpected Asymptotic Complexity

The second application of the PerfSyn framework focuses on *unexpected asymptotic complexity*, where an implementation requires more resources to handle increasingly larger inputs than expected. Prior work has studied how to detect such problems under the assumption that the inputs that expose the problem are available [11, 21, 60]. In contrast, we focus on how to automatically synthesize inputs that expose an unexpected asymptotic complexity. To use PerfSyn to find such a problem, the developer specifies the expected complexity class, e.g., $O(1)$, $O(n)$, $O(n^2)$, of the method under test.

Execution Feedback The feedback for a path through the mutation tree is a sequence $F = [(i_0, c_0), \ldots, (i_k, c_k)]$, where each pair (i_j, c_j) represents a measurement of the input size and the execution cost of a program p_j in the path. In general, the problem of determining this size is impossible because the notion of "input size" is program-specific and typically not explicitly specified. In this work, we measure the size of the input based on the number of distinct memory locations first accessed by a method [11, 12] and we measure the execution cost as the number of evaluated conditional checks.

Performance Oracle To compute a score that indicates how close a path is to expose a bottleneck the oracle performs polynomial curve fitting using the measured values in F. The curve represents how the execution cost varies depending on the input size. To determine the best-fitting curve, the oracle tries to fit the data points to curves of different degrees and it selects the curve that yields the smallest normalized root mean square error. Based on the fitted curve, the oracle computes the score as the area under the curve using the trapezoidal numerical integration over the data points in F:

$$score(F) = \frac{1}{2} \cdot \sum_{(i_j, c_j) \in F} (i_j - i_{j-1}) \cdot (c_j + c_{j-1})$$

Intuitively, the larger the area under the curve the more the actual performance deviates from the expected performance.

Table 1. Classes and methods used in the evaluation.

Project	Version	Classes (# of methods under test)
Changes in relative performance:		
Ant	1.9.1 vs. 1.9.4	IdentityStack (6), VectorSet (20)
Commons-Collections	3.2.1 vs. 4.1	TreeList (9), ListOrderedMap (14), ListOrderedSet (11)
Commons-Lang	3.4 vs. 3.5	ArrayUtils (51), CharSetUtils (5), StringUtils (1)
Unexpected asymptotic complexity:		
Lucene	5.5.4	Operations (17)
Commons-Collections	3.2.1 4.1	BoundedFifoBuffer (8) CompositeMap (1)
Commons-Math	3.6	EnumeratedIntegerDistribution (1)
Guava	8ea0f20 6cec4d2	Sets (1) NavigableMap (1)
SunFlow	0.07.2	SunFlowAPI (1)

4 Implementation

We implement PerfSyn as a tool to analyze Java classes. PerfSyn is implemented in Scala and Java. To gather feedback about program executions, we instrument the analyzed code using the Java bytecode manipulation library ASM [5]. To collect information about the program structure, the class hierarchy, and type information of the initial program, we use the Soot framework [51]. The implementation of curve fitting and other mathematical computations builds upon the Apache-Commons Math library. The oracle that exposes changes in relative performance is implemented by executing a synthesized program with two version of the class under test, each running in a separate Java virtual machine. To avoid issues of incompatible APIs across versions, PerfSyn ignores mutations that involve methods that do not exist in both versions or methods that have different signatures.

5 Evaluation

5.1 Experimental Setup

Benchmarks We apply PerfSyn to 147 methods from 15 classes in seven popular Java projects [47] (Table 1). The methods are selected in two ways. First, we analyze 17 methods with bottlenecks known from previous work [34] and from publicly available bug reports, all of which have been confirmed by the developers. These methods allow us to evaluate whether PerfSyn detects developer-confirmed problems. Second, we analyze 130 additional methods to evaluate how many additional bottlenecks PerfSyn reports and whether it detects any previously unknown problems. For the oracle that targets changes in relative performance, we compare different versions of the same class. For example, for methods with known and now fixed bottlenecks, we use the version before and after the fix. For the oracle that targets unexpected asymptotic complexity, we analyze methods that explicitly annotate the expected complexity in their Javadoc documentation.

The initial program for most benchmarks is simply an empty program, i.e., PerfSyn creates programs without requiring any manually written initial program. For four benchmarks, we manually wrote initial programs. They initialize classes in a way that is currently not supported by our implementation, e.g., providing an anonymous class. Writing these initial program is straightforward.

Inspecting Bottlenecks For each method, PerfSyn yields a synthesized program and its performance score. In practice, we expect developers to inspect only methods with a high score. For the evaluation, we inspect the best synthesized program of a method if the following holds:

- *Changes in Relative Performance.* The difference in performance is larger than $1.2x$.
- *Unexpected Asymptotic Complexity.* The error of the fitted curve is below 5%, $|F| \geq 4$, and the fitted complexity exceeds the expected complexity.

Parameters and Hardware We give a 5-minute time budget per method under test, and we limit the maximum exploration depth of the mutation tree $maxMuts = 32$. For A*, we set the maximum number of nodes in the open set to 8,192; when the memory is full, all but the best 16 nodes are removed. For ACO, we use 16 ants. All experiments are done on a 42-core machine with two 2.2 GHz Intel Xeon processors E5-2699, 512GB memory, running 64-bit Ubuntu 16.04.1 LTS with GNU/Linux 4.4, Java 1.6.0_27 using OpenJDK 1.8.0.111, with 256GB of memory assigned to the VM.

5.2 Effectiveness in Finding Bottlenecks

For 22 out of the 147 methods, PerfSyn synthesizes a program that exposes a bottleneck and that we inspect based on the criteria given in Section 5.1. Table 2 summarizes these bottlenecks. The table shows for each method the best program synthesized with A* and ACO, along with the number of mutations applied to reach this program and the performance effect exposed by the program. For the changes in relative performance, we show speedups and slowdowns with a plus and minus, respectively. For example, for bottleneck ID 7, the A* search synthesizes a program by applying three mutations, and this program shows the method to be 80x faster than in the previous version. For unexpected asymptotic complexities, the table reports the complexity class that PerfSyn finds. For all bottlenecks in the table, the methods were expected to be $O(1)$, but turn out to have linear complexity.

Eight of the 22 methods reveal a previously unknown performance property. Table 2 highlights these bottleneck in

Table 2. Bottlenecks detected by PerfSyn (Muts=number of mutations, Perf=Outcome of performance oracle).

		Best program			
		A*		ACO	
ID	Method	Muts	Perf	Muts	Perf
Changes in relative performance:					
1	ArrayUtils.removeAll	1	-2.4x	3	-2.4x
2	ArrayUtils.removeElements	31	+1.7	29	+1.5x
3	ArrayUtils.indexOf	1	+1.2x	1	+1.2x
4	ArrayUtils.isNotEmpty	1	-1.7x	1	-1.7x
5	ArrayUtils.isSameLength	1	+1.4x	1	+1.4x
6	CharSetUtils.squeeze	14	+1.7x	13	+1.3x
7	StringUtils.getLevenDist	3	+80.0x	1	+50.6x
8	IdentityStack.containsAll	7	-41x	14	-41x
9	IdentityStack.removeAll	12	-6360x	8	-57x
10	IdentityStack.retainAll	12	-6360x	14	-3.2x
11	ListOrderedMap.remove	7	-1.4x	6	+2.4x
12	ListOrderedSet.addAll	12	+1.4x	14	+1.4x
13	ListOrderedSet.addAllAtIndex	1	-1.4x	3	-1.4x
14	ListOrderedSet.toArray	5	+2.6x	2	+2.6x
15	ListOrderedSet.remove	-	-	23	+1.8x
16	VectorSet.addAll	9	-1.2x	12	-1.2x
17	VectorSet.clone	5	+1.2x	2	+1.2x
18	TreeList.addAll	1	+6.2x	10	+7.5x
19	TreeList.addAllAtIndex	-	-	12	+1.5x
Unexpected asymptotic complexity:					
20	EnumIntDist.probability	32	$O(n)$	5	$O(n)$
21	NavigableMap.isEmpty	7	$O(n)$	6	$O(n)$
22	Sets.powerSet	10	$O(n)$	4	$O(n)$

grey. Bottleneck ID 1 is a previously unknown slowdown in a method without a previously known problem. For the other highlighted bottlenecks, PerfSyn triggers an unexpected slowdown introduced by a change supposed to optimize the code with a previously known problem. For example, for bottleneck ID 10, PerfSyn shows that applying a change supposed to be an optimization is not beneficial for a usage scenario. These cases show that our approach helps developers to find unexpected effects of modifying code.

The remaining reports confirm that optimizations applied to methods with a previously known problems by developers are indeed beneficial. This kind of report is useful for a developer who wants to verify her performance assumptions after a code change.

The performance differences reported in Table 2 are based on the number of evaluated conditions (Sections 3.4.1 and 3.4.2). To validate this proxy metric, we execute the synthesized programs and measure their wall-clock execution time. To confirm the performance differences for bottlenecks ID 1 to ID 19, we repeatedly measure the execution time (32 times) and check whether the differences are statistically significant (95% confidence) [19]. To confirm bottlenecks ID 20 to ID 22, we manually inspect the code and verify that the computational complexity reflects what PerfSyn reports. The validation confirms that in general the proxy metric reliably reflects the performance behavior of the synthesized tests when executed on the machine used for the experiments, and that the manually checked code complexity reflects what PerfSyn reports. However for short methods (e.g., like for bottleneck ID 17 and bottleneck ID 14) the proxy metric fails to accurately approximate the performance behavior of the method because the execution time differences for the methods are not significant.

```
1  public class SynProg {
2     public run() {
3        ListOrderedMap v0 = new ListOrderedMap();
4        v0.put("str0", "...");
5        v0.put("str1", "...");
6        ...
7        v0.put("strN", "...");
8        v0.remove("strX");
9     }
10 }
```

Listing 2. Program that triggers bottleneck ID 11

```
1  public class SynProg {
2     public run() {
3        int[] v0 = new int[/*...*/];
4        int v3 = /*...*/;
5        Array.add(v0,/*...*/);
6        Array.add(v0,/*...*/);
7        /*...*/
8        Array.add(v0,/*...*/);
9        EnumIntDist v1 = new EnumIntDist(v0);
10       v1.probability(v3);
11    }
12 }
```

Listing 3. Program that triggers bottleneck ID 20.

5.3 Examples of Synthesized Programs

Listing 2 and Listing 3 show two programs synthesized by PerfSyn. The highlighted statements are inserted by PerfSyn on top of the initial program. The program in Listing 2 exposes a bottleneck in `ListOrderedMap.remove`. The implementation of the faster version of `remove` performs a containment check before removing the element to save time when the collection is non-empty and when it does not contain the element. A previous version the method did not contain this optimization. For bottleneck ID 11 PerfSyn synthesize two different programs, one for each search as indicated in Table 2. PerfSyn discovers a path in the mutation tree that adds multiple elements to the map and that passes an argument to the method. For one synthesized program the argument to `remove` is contained in the collection, and for the other program the argument is not part of the collection.

In the second example, shown in Listing 3, after a few iterations, PerfSyn prioritizes a path through the mutation tree that inserts multiple similar statements. As a result, the

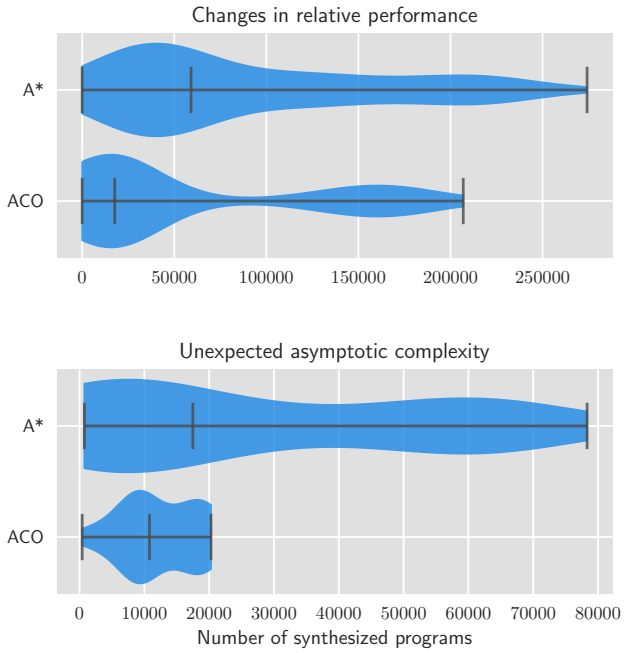

Figure 2. Violin plots of the average number of programs synthesized while analyzing a method.

execution time of the method under test increases, exposing the linear complexity of the method.

Bottleneck ID 1 (not shown for space reason) is caused by two unnecessary calls in `ArrayUtils.remove` that copy and sort the second parameter of the method. These calls are only necessary when the second argument is a non-empty array. PerfSyn detects this case and synthesizes a program that triggers the problem, showing that the bottleneck is triggered independently of the content of the first argument. For bottleneck ID 9, PerfSyn synthesizes a program that removes from an empty collection. The older version of `IdentityStack.removeAll` does not execute any code because the collection is empty. However, the newer version of the method first creates a useless instance of a `HashSet` and then needlessly adds to this newly created set multiple elements supposed to be removed from the empty collection.

Overall, these examples show that our approach detects real-world performance bottlenecks in widely used Java classes. By synthesizing a program that demonstrates each problem, developers can easily understand and then fix the bottlenecks, or more easily understand the performance impacts of code changes.

5.4 Efficiency

Figure 2 shows the distribution of the number of explored nodes in the mutation tree. The results show that both A* and ACO explore a large number of programs in the given time budget. The relative difference between the two search strategies are mostly due to an implementation detail: ACO probabilistically chooses a single mutation at each step, which sometimes causes ACO to expand a longer sequence of programs that cause a crash. These programs take more time to execute because of our handling of exceptions in the profiling VMs (e.g., timeouts to manage infinite loops and recursions). Overall, we conclude that both search strategies efficiently explore the mutation tree, which enables them to detect bottlenecks in at most 5 minutes per method.

6 Limitations and Future Work

Being a form of test input generation, PerfSyn suffers from similar limitations as existing test generators. For example, the current implementation is unlikely to be effective for methods that require complex string inputs [50], e.g., parsers. Targeted support for a specific parser could be implemented by adding appropriate mutation operators. Similarly PerfSyn fails to synthesize programs for classes that require specific method sequences to initialize an object state, however to alleviate this issue PerfSyn may benefit from approaches that address a similar issue [48].

Our approach can be instantiated with various performance oracles, of which we only present two in this paper. Future work may explore other kinds of performance oracles, e.g., to find algorithmic complexity attacks [38, 45] by trying to maximize the resource usage triggered an input of fixed size. Other limitations of our current approach is to consider only a limited set of mutation operators and to evaluate only two search algorithms to explore the mutation tree. Based on our general framework, it is relatively easy to implement and evaluate further mutation operators and search algorithms.

7 Related Work

7.1 Test Generation

Wise [7] steers symbolic execution toward inputs that trigger worst-case complexity. In contrast, PerfSyn uses a blackbox approach. SpeedGun [39] detects performance regressions via test generation, which is a special case of detecting unexpected relative performance. Instead of generating tests at random, PerfSyn uses feedback to steer toward bottleneck-exposing tests. EventBreak [40] also exploits performance feedback to generate tests, but for UI-level instead of unit-level tests. Dhok et al. [14] propose to generate tests that expose loop inefficiencies. Their work focuses on a particular kind of performance problem and relies on a manual inspection of generated tests. Travioli [37] dynamically identifies functions that traverse data structures, which can help developers in manually constructing performance tests. SlowFuzz [38] uses a resource-usage-guided evolutionary search to find inputs that expose an algorithmic complexity vulnerability. Their work differs from PerfSyn by considering inputs to be byte sequences instead of sequences of programs, and by maximizing the slowdown within a fixed input size instead of being guided by a performance oracle. Applying our approach to find algorithmic complexity vulnerabilities is

an interesting direction for future work. Compared to all of the above, we are the first to (i) provide a generic framework that creates tests for different kinds of bottlenecks and to (ii) formulate the problem as a graph exploration problem.

Test generation for purposes other than performance has been widely studied. Concolic execution [20], feedback-directed random test generation [36], evolutionary algorithm-based test generation [18], and combinations of static and dynamic analysis [48] share the idea of using feedback from past executions to steer the generation of future tests. These and other [8] approaches steer toward high code coverage. Instead, PerfSyn exposes bottlenecks, which typically does not require covering all statements or branches, but to repeatedly trigger specific statements and branches.

7.2 Dynamic Analysis

PerfSyn relates to dynamically analyzing the asymptotic complexity and scalability of software. Goldsmith et al.[21] pioneered the idea of empirically estimating the computational complexity of a program. Their work requires a developer to specify the input size of the analyzed code – a problem addressed more recently [11, 60]. Our work builds on one of these approaches [11] to estimate the input size for determining unexpected asymptotic complexities. A limitation shared by all of the above is to require inputs that may expose complexity issues; PerfSyn addresses this limitation. Instead of analyzing scalability w.r.t. input size, Calotoiu et al.[9] propose a dynamic analysis to model the scalability w.r.t. the number of processors that execute a program. Our work differs by targeting performance problems that are independent of the underlying hardware.

Beyond asymptotic complexity issues, dynamic analysis is widely used to detect various other kinds of bottlenecks. General purpose profilers highlight functions where most CPU time is spent [23] or performance-critical paths [41], extract a model that summarizes performance properties of a program [6], or highlight code locations that, if optimized, will speed up the execution [13]. Other profilers target particular classes of performance problems, such as JIT-unfriendly code [22], memoization opportunities [49], unnecessarily repeated behavior [34], inefficient use of object-oriented language features [31, 32, 52–56], and UI delays [28]. PerfDiff [61] compares the performance of a single implementation in different environments; in contrast, one of our oracles compares the performance of different implementations. The effectiveness of all these approaches depends in test inputs that trigger interesting behavior. PerfSyn contributes an automated way to generate such test inputs.

To help developers understand and localize observed bottlenecks, several dynamic analyses have been proposed. They help understand idle times of servers [2], associate bottlenecks with particular configuration options [3], and reveal impact relations between code locations [59]. Some approaches combine runtime data from multiple users to pinpoint the root cause of bottlenecks [24, 44]. SyncProf [58] analyzes waits-for relationships between critical sections to localize synchronization-related bottlenecks. A recent survey [26] summarizes work on performance anomalies.

7.3 Static Analysis

Several static analyses have been proposed that detect performance bottlenecks, e.g., redundant traversals of data structures [35] and other unnecessary loop iterations [33]. Dufour et al. [16] propose a technique that combines static and dynamic analysis to find excessive uses of temporal objects.

7.4 Studies of Performance Issues

Studies of performance issues in real-world code show that developers spend non-negligible amounts of time dealing with bottlenecks [27, 30, 43]. One of them [30] reports that many bottlenecks only manifest with specific input data and with input data of specific sizes, a problem addressed here. A recent study shows the that the lack of writing and maintaining performance tests is commonplace across many popular open-source projects [29].

8 Conclusions

This paper presents the first general framework for synthesizing inputs that expose performance bottlenecks in a given method under test. The approach synthesizes a program that calls the method under test and iteratively this program to systematically increase the amount of work done by the method. A key insight enabling our work is that the problem of finding a sequence of mutations that leads to a bottleneck-exposing program can be effectively and efficiently addressed by the ant colony optimization algorithm. Based on this insight, the approach uses feedback from executions of synthesized programs to steer the exploration toward the most promising sequences of mutations. The presented approach is applicable to different kinds of performance bottlenecks, including unexpected asymptotic complexity and unexpected performance relative to another implementation. We evaluate the idea by applying it to widely used Java classes, in which PerfSyn reveals 22 bottlenecks in only five minutes of profiling time per bottleneck. Our work addresses the understudied problem of creating bottleneck-exposing inputs and expands the applicability of existing profiling techniques, helping software developers in their efforts to reduce performance bottlenecks.

Acknowledgments

This work was supported by SNF grant 206021 133835, by the German Federal Ministry of Education and Research and by the Hessian Ministry of Science and the Arts within CRISP, and by the German Research Foundation within the Emmy Noether project ConcSys.

References

[1] 2017. PerfSyn - Repository. https://github.com/lucadt/perfsyn.

[2] Erik R. Altman, Matthew Arnold, Stephen Fink, and Nick Mitchell. 2010. Performance analysis of idle programs. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 739–753.

[3] Mona Attariyan, Michael Chow, and Jason Flinn. 2012. X-ray: Automating Root-Cause Diagnosis of Performance Anomalies in Production. In *Symposium on Operating Systems Design and Implementation (OSDI)*. 307–320.

[4] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Trans. Software Eng.* 41, 5 (2015), 507–525.

[5] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. 2002. ASM: a code manipulation tool to implement adaptable systems. In *Adaptable and extensible component systems*.

[6] Marc Brünink and David S. Rosenblum. 2016. Mining Performance Specifications. In *European Software Engineering Conference and International Symposium Foundations of Software Engineering (ESEC/FSE)*.

[7] Jacob Burnim, Sudeep Juvekar, and Koushik Sen. 2009. WISE: Automated test generation for worst-case complexity. In *International Conference on Software Engineering (ICSE)*. 463–473.

[8] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Symposium on Operating Systems Design and Implementation (OSDI)*. 209–224.

[9] Alexandru Calotoiu, Torsten Hoefler, Marius Poke, and Felix Wolf. 2013. Using automated performance modeling to find scalability bugs in complex codes. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 45:1–45:12.

[10] Tim Chen, Leonid I. Ananiev, and Alexander V. Tikhonov. 2007. Keeping Kernel Performance from Regressions. In *Linux Symposium*.

[11] Emilio Coppa, Camil Demetrescu, and Irene Finocchi. 2012. Input-sensitive profiling. In *Conference on Programming Language Design and Implementation (PLDI)*. 89–98.

[12] Emilio Coppa, Camil Demetrescu, Irene Finocchi, and Romolo Marotta. 2014. Estimating the Empirical Cost Function of Routines with Dynamic Workloads. In *International Symposium on Code Generation and Optimization (CGO)*. 230.

[13] Charlie Curtsinger and Emery D. Berger. 2015. Coz: finding code that counts with causal profiling. In *Symposium on Operating Systems Principles (SOSP)*. 184–197.

[14] Monika Dhok and Murali Krishna Ramanathan. 2016. Directed Test Generation to Detect Loop Inefficiencies. In *European Software Engineering Conference and International Symposium on Foundations of Software Engineering (ESEC/FSE)*.

[15] M. Dorigo, M. Birattari, and T. Stutzle. 2006. Ant colony optimization. *IEEE Computational Intelligence Magazine* 1, 4 (2006), 28–39.

[16] Bruno Dufour, Barbara G. Ryder, and Gary Sevitsky. 2007. Blended analysis for performance understanding of framework-based applications. In *International Symposium on Software Testing and Analysis (ISSTA)*. 118–128.

[17] King Chun Foo, Zhen Ming Jiang, Bram Adams, Ahmed E. Hassan, Ying Zou, and Parminder Flora. 2010. Mining Performance Regression Testing Repositories for Automated Performance Analysis.. In *International Conference on Quality Software (QSIC)*. 32–41.

[18] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: automatic test suite generation for object-oriented software. In *European Software Engineering Conference and International Symposium Foundations of Software Engineering (ESEC/FSE)*. 416–419.

[19] Andy Georges, Dries Buytaert, and Lieven Eeckhout. 2007. Statistically rigorous Java performance evaluation. In *Conference on Object-Oriented Programming, Systems, Languages, and Application (OOPSLA)*. 57–76.

[20] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: directed automated random testing. In *Conference on Programming Language Design and Implementation (PLDI)*. 213–223.

[21] Simon Goldsmith, Alex Aiken, and Daniel Shawcross Wilkerson. 2007. Measuring empirical computational complexity. In *European Software Engineering Conference and International Symposium on Foundations of Software Engineering (ESEC/FSE)*. 395–404.

[22] Liang Gong, Michael Pradel, and Koushik Sen. 2015. JITProf: Pinpointing JIT-unfriendly JavaScript Code. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 357–368.

[23] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. 1982. Gprof: A call graph execution profiler. In *SIGPLAN Symposium on Compiler Construction*. 120–126.

[24] Shi Han, Yingnong Dang, Song Ge, Dongmei Zhang, and Tao Xie. 2012. Performance debugging in the large via mining millions of stack traces. In *International Conference on Software Engineering (ICSE)*. 145–155.

[25] P. E. Hart, N. J. Nilsson, and B. Raphael. 1968. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics* 4, 2 (1968), 100–107.

[26] Olumuyiwa Ibidunmoye, Francisco Hernández-Rodriguez, and Erik Elmroth. 2015. Performance Anomaly Detection and Bottleneck Identification. *ACM Comput. Surv.* (2015), 4:1–4:35.

[27] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and detecting real-world performance bugs. In *Conference on Programming Language Design and Implementation (PLDI)*. 77–88.

[28] Milan Jovic, Andrea Adamoli, and Matthias Hauswirth. 2011. Catch me if you can: performance bug detection in the wild. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 155–170.

[29] Philipp Leitner and Cor-Paul Bezemer. 2017. An Exploratory Study of the State of Practice of Performance Testing in Java-Based Open Source Projects. In *International Conference on Performance Engineering (ICPE)*. 373–384.

[30] Yepang Liu, Chang Xu, and S.C. Cheung. 2014. Characterizing and Detecting Performance Bugs for Smartphone Applications. In *International Conference on Software Engineering (ICSE)*. 1013–1024.

[31] Darko Marinov and Robert O'Callahan. 2003. Object equality profiling. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 313–325.

[32] Khanh Nguyen and Guoqing Xu. 2013. Cachetor: Detecting Cacheable Data to Remove Bloat. In *European Software Engineering Conference and International Symposium Foundations of Software Engineering (ESEC/FSE)*. 268–278.

[33] Adrian Nistor, Po-Chun Chang, Cosmin Radoi, and Shan Lu. 2015. Caramel: Detecting and fixing performance problems that have non-intrusive fixes. In *International Conference on Software Engineering (ICSE)*. 902–912.

[34] Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. 2013. Toddler: Detecting Performance Problems via Similar Memory-Access Patterns. In *International Conference on Software Engineering (ICSE)*. 562–571.

[35] Oswaldo Olivo, Isil Dillig, and Calvin Lin. 2015. Static detection of asymptotic performance bugs in collection traversals.. In *Conference on Programming Language Design and Implementation (PLDI)*. 369–378.

[36] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-Directed Random Test Generation. In *International Conference on Software Engineering (ICSE)*. 75–84.

[37] Padhye, Rohan and Sen, Koushik. 2017. Travioli: A Dynamic Analysis for Detecting Data-Structure Traversals. (2017), 1–11.

[38] Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. 2017. SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*. 2155–2168.

[39] Michael Pradel, Markus Huggler, and Thomas R. Gross. 2014. Performance Regression Testing of Concurrent Classes. In *International Symposium on Software Testing and Analysis (ISSTA)*. 13–25.

[40] Michael Pradel, Parker Schuh, George Necula, and Koushik Sen. 2014. EventBreak: Analyzing the Responsiveness of User Interfaces through Performance-Guided Test Generation. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 33–47.

[41] Lenin Ravindranath, Jitendra Padhye, Sharad Agarwal, Ratul Mahajan, Ian Obermiller, and Shahin Shayandeh. 2012. AppInsight: mobile app performance monitoring in the wild. In *Conference on Operating Systems Design and Implementation (OSDI)*. 107–120.

[42] Stuart Russell. 1992. Efficient memory-bounded search methods. (1992).

[43] Marija Selakovic and Michael Pradel. 2016. Performance Issues and Optimizations in JavaScript: An Empirical Study. In *International Conference on Software Engineering (ICSE)*. 61–72.

[44] Linhai Song and Shan Lu. 2014. Statistical debugging for real-world performance problems. In *Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*. 561–578.

[45] Cristian-Alexandru Staicu and Michael Pradel. 2017. *Freezing the Web: A Study of ReDoS Vulnerabilities in JavaScript-based Web Servers*. Technical Report TUD-CS-2017-0305. TU Darmstadt.

[46] Thomas Stützle and Holger H. Hoos. 2000. MAX-MIN Ant System. *Future Generation Comp. Syst.* 16, 8 (2000), 889–914.

[47] Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. 2010. Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies. In *Asia Pacific Software Engineering Conference (APSEC)*.

[48] Suresh Thummalapenta, Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Zhendong Su. 2011. Synthesizing method sequences for high-coverage testing.. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 189–206.

[49] Luca Della Toffola, Michael Pradel, and Thomas R. Gross. 2015. Performance Problems You Can Fix: A Dynamic Analysis of Memoization Opportunities. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 607–622.

[50] Luca Della Toffola, Cristian-Alexandru Staicu, and Michael Pradel. 2017. Saying "hi!" Is Not Enough: Mining Inputs for Effective Test Generation. In *International Conference on Automated Software Engineering (ASE)*.

[51] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java bytecode optimization framework. In *Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*. IBM, 125–135.

[52] Guoqing Xu. 2012. Finding reusable data structures. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 1017–1034.

[53] Guoqing Xu, Matthew Arnold, Nick Mitchell, Atanas Rountev, and Gary Sevitsky. 2009. Go with the flow: profiling copies to find runtime bloat. In *Conference on Programming Language Design and Implementation (PLDI)*. 419–430.

[54] Guoqing Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, Edith Schonberg, and Gary Sevitsky. 2010. Finding low-utility data structures. In *Conference on Programming Language Design and Implementation (PLDI)*. 174–186.

[55] Guoqing Xu and Atanas Rountev. 2010. Detecting inefficiently-used containers to avoid bloat. In *Conference on Programming Language Design and Implementation (PLDI)*. 160–173.

[56] Dacong Yan, Guoqing (Harry) Xu, and Atanas Rountev. 2012. Uncovering performance problems in Java applications with reference propagation profiling. In *International Conference on Software Engineering (ICSE)*. 134–144.

[57] Cemal Yilmaz, Arvind S. Krishna, Atif M. Memon, Adam A. Porter, Douglas C. Schmidt, Aniruddha S. Gokhale, and Balachandran Natarajan. 2005. Main effects screening: a distributed continuous quality assurance process for monitoring performance degradation in evolving software systems. In *International Conference on Software Engineering (ICSE)*. 293–302.

[58] Tingting Yu and Michael Pradel. 2016. SyncProf: Detecting, Localizing, and Optimizing Synchronization Bottlenecks. In *International Symposium on Software Testing and Analysis (ISSTA)*. 389–400.

[59] Xiao Yu, Shi Han, Dongmei Zhang, and Tao Xie. 2014. Comprehending performance from real-world execution traces: a device-driver case.. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 193–206.

[60] Dmitrijs Zaparanuks and Matthias Hauswirth. 2012. Algorithmic profiling. In *Conference on Programming Language Design and Implementation (PLDI)*. 67–76.

[61] Xiaotong Zhuang, Suhyun Kim, Mauricio J. Serrano, and Jong-Deok Choi. 2008. Perfdiff: a framework for performance difference analysis in a virtual machine environment. In *Symposium on Code Generation and Optimization (CGO)*. 4–13.