University of Oregon

Dynamic Performance Analysis Techniques as Software Engineering Assistive Tools

Area Exam Report

by
Ziyad Alsaeed
zalsaeed@cs.uoregon.edu

Department of Computer and Information Science
June 13, 2019

Abstract

Non-functional requirements of typical applications tend to get less attention during software development compared to functional requirements. Software performance, in particular, is one that gets less attention during development, but ahead of shipping apparent performance flaws must be fixed. Dynamic software performance analysis attempts to assist developers locating performance flaws or confirm their understanding of the overall performance behavior.
We evaluate fundamental and recent performance analysis techniques. Moreover, we highlight the strengths and weaknesses of performance analysis tools in terms of efficiency, comprehensiveness, exploration and understandably. Finding inputs that trigger unanticipated performance flaws is an area requiring more work. We review machine learning, genetic algorithms and fuzzing as the three major approaches used to find special performance inputs. Machine learning techniques may be useful for finding data that triggers poor performance.

1 Introduction

The need to understand software performance behavior has long been recognized [28, 5, 1, 47]. In fact, the need for performance satisfactory applications or algorithms has been studied long before software was written at a large scale. The problem existed since the formalization of what is known now as the study of algorithmic complexity. Software performance analysis is still an active research area. The level of abstraction at which the solution works, the definition of the performance issues[1], the diverse nature of software performance requirements and the decision to fix or only highlight performance issues are a few of the factors that make the problem manifold. Moreover, the inherent habit among the software engineering community of "fix it later" [26], makes performance analysis researchers confront a wildly complicated problem. A clear definition of the problem is necessary to narrow down the focus of our work.

As Molyneaux [54] states, "similar to beauty, performance is in the eyes of the beholder". When the performance of an application is satisfactory could greatly differ from one person to another based on their experience and expectations. Also, it could differ based on software applications' nature. For example, a few milliseconds delay on a web-based software might not be as severe as the same amount of delay for real-time software [17].

Jin et al. [38], define performance issues as bugs that with simple solutions the software performance is enhanced while preserving software functionality. However, such definition introduces other issues such as what is a simple solution and how much enhancement is acceptable. We know that the actual measurement is in the cost (monetary value) of having a bad performing application on production [42]. Unfortunately, if such cost is used to define performance issues, it means that it is too late to discover or fix these issues.

For our work, we define performance issues as a negative software performance that contradicts the developer understanding or escape his/her anticipation. As software evolves of several modules, prior developers understanding of how each procedure is performing does not necessarily always hold. In addition, usage of out-source libraries introduces a risk of misusing their interfaces [38]. And as Jin et al. [38] highlighted that the workload mismatch between what the developer anticipated and reality as a major source of performance issues. Such case results in performance issues that are only discovered after software deployment.

There are other less common causes of introducing performance issues (e.g. user behavior change). However, we think that software engineers understanding of the code behavior and anticipation of workload cover such cases.

As it is hard to define performance problems it is also hard to define what performance analysis tools are based on the published literature. The main objective of a performance analysis tool is usually to treat performance issues. Some performance analysis tools are designed to take it upon themselves to fix performance issues [71]. Others focus on very specific but difficult known patterns of issues to highlight them to developers [24, 82, 61, 55, 81]. Tracking the input to understand their influence on performance is also a different objective for some performance analysis tools [20, 45, 31, 82].

[1] The phrases performance issue, performance bug and performance improvement opportunity are used interchangeably in the software performance analysis literature. All of these refer to a spot in software in which if fixed, it will improve the application's overall performance. Through out this document we will use the phrase "*performance issue*" to refer to such case.

In addition to the different objectives, there are different targeted environments. Some performance analysis tools are focused on solutions that are only applicable to parallel programs [21]. Others focus on analyzing software entities and their interactions in the environment of distributed systems [73, 62, 12, 35]. Such diversity on the offered techniques and environments makes it hard to define what should a performance analysis tool do. More importantly, it makes it harder to compare their effectiveness.

One of the challenges in software performance analysis is how to compare techniques' effectiveness in relation to performance. Due to diverse objectives (e.g. targeting performance anti-patterns [81], considering configuration as inputs [87], improving the understandability of the results [23], etc), each solution would emphasize its analysis goal more during evaluation. For example, Curtsinger and Berger [21] argue that the performance improvements opportunities they discover are far more effective than those found by *gprof* [28] for the given benchmark. However, it is known that profiling parallel program was never an objective for *gprof* [28] as it was for Curtsinger and Berger [21]. The overall application speed-up after fixing the highly ranked hot-spots could be seen as a good measurement of the technique's effectiveness. Nevertheless, such an indicator can be biased as the fix is highly dependent on the developer's experience and understanding of the applications. The technique's added value to developers (e.g. ease of use, root cause understandability, etc) could be a good measurement, but hard to capture.

For software engineers, a valuable performance analysis tool would be one that provides fine-grained details efficiently [5], searches for new unanticipated behaviors [48] and clearly identify the root cause of the performance behavior [71]. Such challenging characteristics of a performance analysis tool would help software engineers assert their understanding of their written software or reveal an anticipated performance issue.

Collecting very detailed traces of an application run is costly. There is a trade-off between how detailed the information provided by a performance analysis tool and how acceptable the overhead is. Details can be of different types. For example, effectively collecting the number of times a path is taken within an application among all possible paths is one type [5]. Another type would be identifying possible inputs of all executed methods [20]. Such fine-grained information is usually associated with a high overhead cost that could preclude their adoption in the real world.

In addition to the overhead trade-off, existing performance analysis tools rarely attempt to discover unanticipated worst case scenarios. For dynamic analysis techniques, the major cause of such limitation is the dependency on the developers written unit tests to drive the analysis of a given application. Such unit tests are usually written to ensure the preservation of the application's functional requirements. Thus, the result of using such unit tests can rarely lead to interesting performance observations. Different inputs alone could have a significant effect on analysis outcomes. Many people were aware of the effect of the inputs on the performance analysis process [20, 45, 31, 18, 82], but they either didn't attempt to generate new and interesting inputs to drive the test or were limited to special cases of input generation and permutation.

Moreover, existing performance analysis tools focus on locating possible performance issues, but not communicating these to the developer in an understandable use case or architectural abstraction [19]. Result understandability is a major criterion that performance analysis tools should provide. Misunderstanding a tool result would lead to wasted optimization opportunities [60]. There is usually a significant trade-off between how comprehensive a performance tool is and the adaptation of it is results due to difficulties in understanding its findings [71, 60]

The goal of this document is to first classify dynamic performance analysis efforts based on their goals and limitations. Second, define what does performance analysis means to software engineers. Third, identify major limitations and the closest work that attempted to solve it. Finally, briefly examine unexplored and promising solutions to the defined limitations.

2 Performance Analysis for Software Engineers

Performance analysis techniques have been developed with different goals in mind. In this section, we go over the diverse approaches of software performance analysis, identify their goals and strengths and examine how are they addressing the needs of software engineers.

2.1 High Level Look at the State of Software Profiling

The basic thinking about performance usually looks at how much time a method is taking. In the simplest form of applications, such intuition is valid. However, real-world programs are ever more complex. Applications are constructed of multiple modules, objects, and methods each of which interact with each other. Such complications require more sophisticated software performance analysis techniques that drive the analysis to interesting results and present them in a meaningful way to the developers.

2.1.1 Control-Flow Based Profiling

Initial clearly identified efforts to software performance analysis started as early as the 1980s [28, 9, 5]. Graham et al. [28] were looking at the software performance analysis in its simplest form. The simplest insight developers are looking for is to understand the method execution time and calls counts at different software architecture abstractions. Provided the software control-flow graph, Graham et al. [28] collected the needed information during the application runtime.

Programs are usually composed of multiple parts that are usually written by multiple developers. Because programs usually use external libraries to implement frequently used methods (e.g. data structure libraries), such composition makes programs difficult to understand. Graham et al. [28] understood such complications and sought to provide results that show the performance cost of routines within the executable program at different abstractions. Such composition can be easier to understand and increase the probability that developers would find appropriate refactoring opportunities for performance gain.

In their solution, Graham et al. [28, 29] explain that merging the two basic measurements of method usage count and time will highlight more meaningfully bottlenecks. Counts are taken within a context (call site), provide the chance to understand the task a method is serving along with its cost. Call site is the identification of a method based on the location in which it was invoked. For example, as shown in Figure-1, if method `foo()` is invoked twice through the program execution, once from within method `bar()` and once from within method `baz()`, then we have two different call sites of method `foo()`. On the other hand, the timing profile provides an insight to asses if a method time consumption is justified given

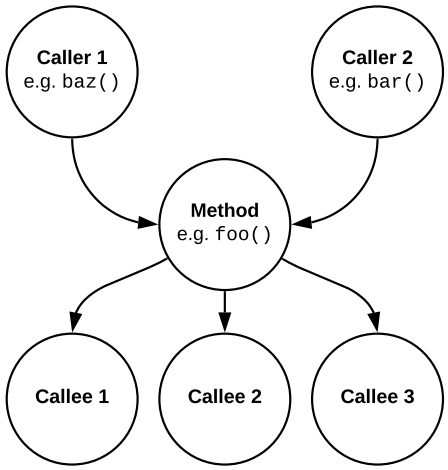

Figure 1: A method within the context of callers and callees.

the task it serves despite the number of times it was called. Once all counts and times of methods are collected and propagated, the results are presented in two forms. A flat representation ranking methods based on which contributed to the program execution time the most. Second, Graham et al. [28] provide the ability to examine methods in a *sub-call graph* that shows all the given method callers and callees (see Figure-1). Such representation allows the users to see how the method is contributing to its callers' time by knowing how much time it was called by a given caller of all the times it was called. Also, it shows which of the callees contributed to its execution time the most by examining how many times a callee was called from that method given all the calls to that particular callee.

Graham's et al. [28] efforts in profiling covers the essential cost developers usually think about when analyzing program performance. Moreover, it provides different levels of abstractions for the developer to examine. The essential limitation of *gprof* [28] is that it is highly dependent on the given developer tests to drive the profiling analysis. Developers are not usually performance testing experts. Moreover, given unit tests for applications are usually written to assert functional requirement soundness, it is highly unlikely the given tests will be helpful for performance testing. Hence, *gprof* would highlight bottlenecks in the programs that are trivial and miss potential threats to program performance. A question comes to mind is would more detailed profiling help mitigate such limitation?

Ball and Larus [5] focused on the efficiency of the profiling tool itself for fine-grained performance analysis. Path profiling, where a profiler measures how many times a path is executed within a method [4, 1, 6, 47, 22, 25, 55], is much more precise than a block or edge profiling. Moreover, it provides much more detailed information about the method's internal cost compared to *gprof* [28] by breaking a method into paths instead of possible method calls from the profiled method. However, detailed look into a method introduces a significantly higher overhead. It could even be sometimes infeasible for a broad set of applications. Thus, it remained infeasible for a while.

Ball and Larus [5], introduced an algorithm to enhance the overhead issue that assigns unique IDs to each path within a method to keep a counter of how many times the path was executed. More precisely, they first convert CFG (Control-Flow Graph) of methods to DAGs (Directed Acyclic Graph). The transformation of CFG creates dummy paths from graph entry to a loop head and from the loop end to the graph exit for each existing loop (backedges). Loop transformation would reduce the size of the graph and misrepresent loops, but it is necessary for instrumentation. Given the DAG, each edge assigned an integer value such that the sum of any path values in the DAG is a unique value. Such an assignment allows for the unique identification of each path by calculating its ID instead of storing it in memory. Each time a path is taken, the algorithm calculates the path's ID and increment its counter. Hence, it is possible to collect detailed information on large applications.

Duesterwald et al. [25] focus even more on profiling efficiency. They argue that in some profiling cases (e.g. just-in-time compilation), there is an even higher demand for lower overhead. A solution is to impose less profiling to gain more knowledge within a smaller space and time. The key idea is to identify a threshold to determine a path head is hot. Once a path head is identified as hot no more profiling is made to it and a prediction is made about its tail using dynamic optimization system. The argument is that shorter intervals of path evaluations help reduce the overhead while maintaining the same hot path predictions.

The work established by Ball and Larus [5] determines bottlenecks in applications in terms of the execution complexity. However, it does not take into account the application usage of memory. Mudduluru et al. [55] consider such problem and established

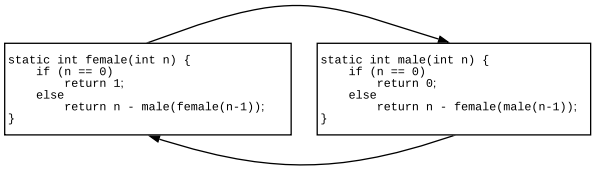

Figure 2: Java example of mutually recursive methods forming a loop based on Hofstadter female and male sequences.

a control-flow profile (called it object-flow profile) to track object (data) creation and access based on Ball and Larus numbering. For each allocation, Mudduluru et al. [55] will preserve a control-flow graph from the allocation site to the locations where the object is used (maintaining a count for each edge). The intuition is that hot paths in such flow profile will help locate inefficiently used memory spaces.

Efficient and highly-detailed profiling techniques [5, 25, 55], made it possible to collect counts information of path profiles regardless of the application size or complexity. However, a more detailed view makes it evident that designated performance testing is necessary compared to *gprof* [28] as more precise paths need to be exercised. The efficiency and preciseness of performance analysis tools do not ensure the fruitfulness of the profiling results nor the understanding. If different, it can help highlight where additional testing might be necessary.

2.1.2 Loop Focused Profiling

In addition to the need to overcome limitations in interpreting loops based on control-flow profiling [22], the common knowledge in the field and few study papers [38, 59] asserts that most of the complex and hard to fixed performance issues happened within special forms of loops. Loops can be in different forms. For example, loops can be in the simple form of language provided keywords such as `for` or `while` loops, as a recursive method or even less clear as mutually recursive methods (see Figure-2). Studies [38, 59] assert that performance issues are even more severe within nested loops. Such conclusion prompted different research efforts to focus the program analysis on loops where most of the performance issues occur.

Focusing on loops when analyzing program performance can be beneficial from different perspectives. First, it reduces the number of instrumented instructions to those within interesting loops. Thus, reduc-

ing the overhead. Second, it guides the studies of program performance toward the actual symptom of performance issues. In this section, we go over unique efforts [61, 82, 24, 76] that specifically studies the effect of special loop cases on performance.

Nistor et al. [61] established the general idea of monitoring instructions behavior within loops. In particular, they look for nested loops that do redundant work. For instance, the code shown in Figure-3 illustrates a severe computation redundancy that is hard to find [61]. As the outer loop iterate over all items in a data set (line 3), it calls the method `drawItem` which in turn calls the method `drawVerticalItem`. The inner loop within `drawVerticalItem` (line 10) also iterate over all items in the data set to find the one with maximum volume. As the volume in the data set does not change over the different loops such computation is redundant and found to be causing the rendering to freeze. The performance issue is fixed by caching the maximum volume value within the outer loop to avoid redundant work.

```
1  // Simplified from the XYPlot class in JFreeChart
2  public void render(...) {
3    for (int item = 0; item < itemCount; item++) { // Outer Loop
4      renderer.drawItem(...item...); // Calls drawVerticalItem
5    }
6  }
7  // Simplified from the CandlestickRenderer class in JFreeChart
8  public void drawVerticalItem(...) {
9    int maxVolume = 1;
10   for (int i = 0; i < maxCount; i++) { // Inner Loop
11     int thisVolume = highLowData.getVolumeValue(series, i).intValue();
12     if (thisVolume > maxVolume) {
13       maxVolume = thisVolume;
14     }
15   }
16   ... = maxVolume;
17 }
```

Figure 3: Computation redundancy performance issue found on `JFreeChart` as shown in [61].

To automatically find redundant computations, Nistor et al. [61] monitored the memory access within the identified nested loops. If a group of instructions access similar memory values across iterations, then those instructions are probably computing similar results. Thus, there is a performance issue. Nistor et al. [61] introduced a tool called Toddler that implements loops monitoring. Toddler first statically analyzes the code searching for loops and assigning unique IDs to each loop. Then, Toddler instruments the code by inserting triggers to identify loops at three major stages: loop starts, loop iteration starts and loop end. Toddler identifies a read instruction by both the static occurrence of the instruction in the code and the dynamic context (call stack) in which the instruction is executed and call it IPCS (Instruction Pointer + Call Stack). For each loop within a nested loop structure (outer or inner) a sequence of IPCS is collected. IPCS sequences are eventually compared given a threshold looking for *read* values similarities across loop iterations. If there is any such IPCS sequence, Toddler reports a performance issue.

Slightly different from Toddler, Song and Lu [76] tackled the problem based on prior knowledge of the performance issues symptoms within loops. Initially, Song and Lu [76] studied known performance issues that occurred within loops to provide a taxonomy of the root causes of the inefficiencies. Their study resulted in two major classifications of loop inefficiency resultless loops and redundant loops. Resultless loops are mainly the type of loops that does a lot of computations but then does not show any side effect. Redundant loops are the types that does repetitive computations (same inputs and outputs on some of the iterations). Song and Lu [76] argue that using the taxonomy to look for suspicious loops helps focus the search on a smaller set of loops. To validate their hypothesis, they developed a tool called `LDoctor` that uses static analysis techniques to identify potential loops. It also uses dynamic analysis techniques along with sampling to analyze applications under a given workload. The tool proves to be efficient and accurate, but only for the given limited search scope.

The solution Toddler [61] and `LDoctor` [76] represents, provide a focused look at program performance analysis. However, they are both passive techniques. That is, they do not explore instrumented application beyond what the developers have written in terms of test cases. Thus, similar to old basic techniques [28, 5, 25, 55] they inherit the limitation of the developer's assumptions during analysis. As a mitigation, the same idea of exploring loops has been explored with the attempt to stress loops given new inputs.

In order to overcome the limitation of finding new unanticipated performance issues within loops, Xiao et al. [82] analyzed application given a set of test cases. The test cases provided are assumed to be expressing the application functionality. Xiao et al. [82] call such test cases, scenarios. For example, for a compression algorithm, a test case or a scenario can be a task a user can complete and a set of parameters to manipulate. This restriction makes it easy for the author to create new inputs by manipulating the parameters and recording the ones that affect the performance the most. However, even with such restriction, the approach is limited in finding scalable inputs only. Scalable inputs are those that increase the size of the test case but does not manipulate the

logic of the input. For example, when compressing files they only are capable of increasing the number of files to compress, not the nature of the files. Thus, they evaluate only one aspect of input manipulation.

Dhok and Ramanathan [24] presented another technique that identifies the main limitation of Toddler [61] and others. They attempt to generate more tests based on what the developers or existing random test generators [65] provide. First, they generate methods summaries based on the given tests. The methods summaries are composed of information pertaining to the presence of a loop, the objects traversed in each loop, and the methods invoked within the loop. Second, they identify methods with potential nested loops. Method detection is based on looking at the call graph for symptoms of known patterns [81] that lead to bad performance using similar techniques of the ones presented in [76]. Finally, they generate performance focused tests for the given methods with emphasis on the scale of the inputs. This approach overcomes the manual parameter identification presented by Xiao et al. [82]. However, in addition to finding scalable inputs only, the introduced approach main limitation is it assumes that initial given tests are going to lead to interesting performance issues. Moreover, it is limited to known symptoms of bad performance within the code. Thus, it can reveal some performance issues, but at the same time skip others.

The loops focused technique provides a more valuable understanding of the performance issues in generals. However, they either are not exploring unanticipated issues or mainly provide scalable inputs to a small set of known performance issues patterns [81]. These efforts [61, 76, 82, 24] along with basic ones [28, 5, 25, 55] are limited in asserting developers understanding of the program by profiling applications based on developer's test cases. If an unanticipated performance issue exists, neither passive nor active presented performance analysis tools would help to capture them.

2.1.3 Model Based Profiling

Even more than functional requirements, non-functional requirements (e.g. performance) are hard to understand and test. Such nature of performance requirements made it necessary to explore methods that would assist in establishing some boundaries that define the system performance goals. For example, in a word processing application, it is expected for a letter to appear instantly on the screen as soon as the user hits the letter key in the keyboard. However, in addition to the difficulty of thinking about every possible scenario in the application in terms of performance, it is hard to put a number that defines the performance needs in such cases. Moreover, if a task becomes the focus of the developers, it is usually easier to understand if the current performance is acceptable given some prior knowledge than writing specifications that define the expected performance. Performance modeling is one of the formal ways to mitigate the issue and establish rules about program expected performance (oracles) [7]. Performance models define precise performance boundaries of an application or module of the application. Given the help it would provide to the developers, many [14, 36, 7, 44] attempted to automatically find such models.

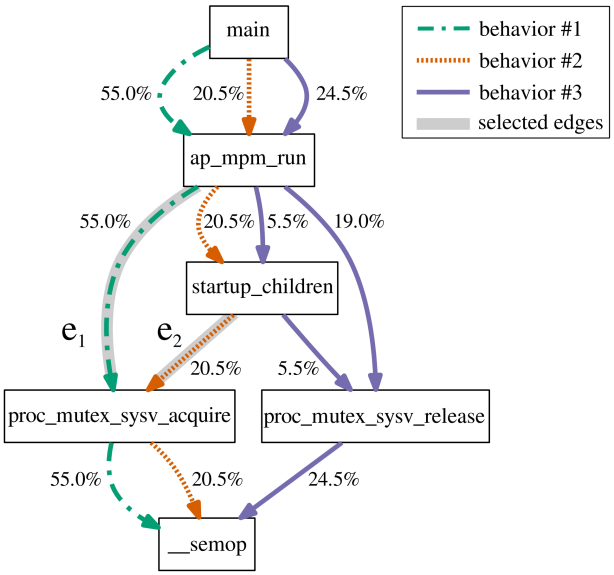

Figure 4: Performance model of calls to method `__semop` in Apache 2.0.64 [14].

Brünink and Rosenblum [14] made a notable effort in this area. They automate the finding of performance models based on actual runs of the program then summarize these models to maintain performance assertions about the program. The intuition is that such assertions would help monitor the performance of the application and triggered at any performance deviations. To find these performance models, Brünink and Rosenblum [14] monitor an application during a given run (usually a deployed application) to obtain runtime insight of given methods (usually hot methods). Then they analyze the collected runtime data to check if it fits different clusters of runtimes and stable (i.e. no new unclassifiable data is further

showing up). If such data for a given method exists, then they collect call stack information to relate them to these different runtimes' clusters. The process then repeats for the given method callers until no further interesting methods are introduced. The generated performance models are usually large and hard to understand. Therefore, Brünink and Rosenblum [14] introduced the idea of finding performance assertions. These performance assertions are the shortest possible descriptive paths in the form of an expression to the given method and the time it took relative to the path. The resulting set of assertions is maintained for future tests or actual use monitoring. Any execution that breaks a given assertion is a performance issue.

An example of the generated model given Brünink and Rosenblum [14] approach is illustrated in Figure-4. The model shows the different summarized context to the method `__semop` in Apache v2.0.64. This automatically learned model state that calls that does not involve edge e_1 or e_2 take less than 70ms. Otherwise, executions would take less than 70ms in 41.9% of the cases if it includes edge e_1 and in 35.5% of the cases if it includes edge e_2. Hence, assertion about the method `__semop` for newer versions can be as simple as if $e_1 \wedge \neg e_2$ then $t < 70ms$ in 41.9% of the cases, if $\neg e_1 \wedge e_2$ then $t < 70ms$ in 35.5% of the cases and if $\neg e_1 \wedge \neg e_2$ then $t < 70ms$ in 99% of the cases where t is the time spent to call the method `__semop`. Using the expression to capture the system behavior given new changes to the code is precise and meaningful to developers.

Brünink and Rosenblum [14] approach provide a method that would generate test oracles. However, obtaining useful values for the test oracles occurs only if the tool used with deployed applications. Valuable workloads are only present when actual users are using the system. Therefore, actual performance insights appear only if the application is in use. And as stated before, at this stage of the application life cycle, performance issues are usually unaffordable.

Hoefler et al. [36] presented a method to build performance models for parallel applications. Nevertheless, the essence of their work is also applicable to non-parallel applications. Hoefler et al. [36] offer to introduce performance-modeling techniques in every software development stages (e.g., design, implementation, testing, etc.). Although such an approach is unrealistic for more agile[2] project management methodology as design efforts are minimal, the guidelines they introduced from their study apply to user-based applications. They obtained the guidelines from experimenting with performance modeling on a set of subject applications. These guidelines can differ based on the point of view when looking at an application. For example, identifying input parameters that influence the runtime considers application workload, but determining communication patterns considers applications structure. Although no automation proposed to generate performance models, they established a foundational systematic approach of performance modeling for others to use (e.g., the work developed by Brünink and Rosenblum [14]).

[2]Agile is a widely adapted software development approach under which requirements and artifacts grow and change together.

```
(? i :( j |(&# x ?0*((74) |(4 A ) |(106) |(6 A ) ) ;?) )
    ([\ t ]|(&((# x ?0*(9|(13) |(10) | A | D ) ;?) | (
    tab ;) |( newline ;) ) ) ) *( a |(&# x ?0*((65) |
    (41) |(97) |(61) ) ;?) ) ([\ t ]|(&((# x ?0*(9|(13)
     |(10) | A | D ) ;?) |( tab ;) |( newline ;) )) )
    *( v |(&# x ?0*((86) |(56) |(118) |(76) ) ;?)) ([\
    t ]|(&((# x ?0*(9|(13) |(10) | A | D ) ;?) | ( tab
    ;) |( newline ;) ) ) ) *( a |(&# x ?0*((65) | (41)
    |(97) |(61) ) ;?) ) ([\ t ]|(&((# x ?0*(9|(13)
    |(10) | A | D ) ;?) |( tab ;) |( newline ;) ) ) ) *
     ( s |(&# x ?0*((83) |(53) |(115) |(73) ) ;?) ) ([\
     t ]|(&((# x ?0*(9|(13) |(10) | A | D ) ;?) | ( tab
     ;) |( newline ;) ) ) ) *( c |(&# x ?0*((67) | (43)
     |(99) |(63) ) ;?) ) ([\ t ]|(&((# x ?0*(9|(13)
    |(10) | A | D ) ;?) |( tab ;) |( newline ;) ) ) ) *
     ( r |(&# x ?0*((82) |(52) |(114) |(72) ) ;?) )([\
    t ]|(&((# x ?0*(9|(13) |(10) | A | D ) ;?) |( tab
    ;) |( newline ;) ) ) ) *( i |(&# x ?0*((73) |(49)
    |(105) |(69) ) ;?) ) ([\ t ]|(&((# x ?0*(9|(13)
    |(10) | A | D ) ;?) |( tab ;) |( newline ;) ) ) )
    *( p |(&# x ?0*((80) |(50) |(112) |(70) ) ;?) )([\
    t ]|(&((# x ?0*(9|(13) |(10) | A | D ) ;?) |( tab
    ;) |( newline ;) ) ) ) *( t |(&# x ?0*((84) |(54)
    |(116) |(74) ) ;?) ) ([\ t ]|(&((# x ?0*(9|(13)
    |(10) | A | D ) ;?) |( tab ;) |( newline ;) ) ) )
    *(:|(&((# x ?0*((58) |(3 A ) ) ;?) |( colon ;)) ) )
     .)
```

Listing 1: Input generated by *SlowFuzz* [67] to demonstrate a special input that causes a slowdown in PCRE [66] regular expression matching library.

In general performance modeling approaches [14, 36, 7, 44] are essential in identifying boundaries that would help in testing the performance of the applications. However, those boundaries can be vague or difficult to understand. Hence, difficult to communicate them with developers to solve a performance issue. Moreover, identifying those boundaries is difficult. Automating such task require thorough unit tests and workloads that would lead to potential performance issues within a given application.

2.1.4 Actionable Profilers

An essential strength in any given profiling technique is its ability to simplify its results to facilitate its comprehensibility by developers. Simplifying the results is not an easy task since it requires predicting the developer's needs. Moreover, results need to be at a level of abstraction where the developer is thinking. For example, Listing-1 shows a special input

that exhibits a 20% slowdown in the PCRE [66] regular expression matching library. From the example, it is clear how it is hard to link the input to the root cause of the performance issue. Lack of improving the result understandability could lead to losses in performance optimization opportunities. Therefore, efforts [60, 71, 23] to analyze applications were made with the goal of understandability in mind.

A simple method to measure the understandability of the results is to make them actionable. Actionable results are automatic syntactically valid suggestions of performance issues fixes a developer can approve or reject. If a developer, based on his longtime understanding of the code, agrees that the change does not break any functional requirements, then he can approve the fix without worrying about completely understanding the performance issue. Such recommendations take out the barrier of explaining the performance issue by providing a code change that guarantees performance improvement. In this section, we discuss essential efforts in making performance analysis tools more actionable and the limitations of such efforts.

Nistor et al. [60], are the first to ask the question of *how likely are developers to fix a detected performance issue?* They study the question in relation to many attributes such as how likely is fixing a performance bug would introduce a functional bug or break a good coding practice. Most importantly, they study the behavior of the developers given the effort and time needed to understand the performance issue as well as understanding the trade-off on other modules of the software if any. They found that it is less likely to fix performance issues if it is hard to understand the issues or relate them to other modules in the software. Given this understanding, they settle on statically finding bugs that have non-intrusive fixes. Primarily, they focus on loops that waste computation after a certain condition is satisfied. The fix for such performance issues is simple and a developer needs only to check if a condition is satisfied to break off the loop. Nistor et al. [60] showed that their reported issues have a high fixing rate given that they are easy to understand. However, even if the found performance issue would introduce a significant speedup in the software, the search scope is very limited. Thus, it is clear that there is a trade-off between the performance analysis tool comprehensibility and the result understandability.

Performance analysis tools that does not sacrifice the potential of finding the most significant and most complicated performance issues, usually have an embedded understandability limitation. Thus, the value of using the developed technique could be lost. To mitigate such limitations, there have been performance analysis techniques that focused on improving the comprehensibility of their results. The primary method in which these techniques solve this issue is by making their findings actionable. Actionable profilers simplify the results and increase the probability of adopting it.

An important question comes to mind when provided with a solution that makes change suggestions is why would not the profilers make the changes without even going back to the developer? As the profiler ensures performance enhancement, they should make the change. To the contrary of compiler optimization, profilers cannot ensure preserving the program integrity if a change is applied. The suggested changes are always syntactical sound but not necessarily semantically. Preserving semantic safety means the performance analysis technique would be too limited. Performance analysis techniques targets changes that are beyond the compiler ability to optimize. Therefore, human judgment is necessary to approve the changes.

Selakovic et al. [71] present a notable effort in providing actionable results. The focus of the technique is on finding the optimal order of logical expressions. Given a logical expression (e.g. `if` or `switch` statements), what would be the optimal order of the expressions to evaluate and reach a decision. For example, if we have the statement `if (a > 0 && b == 1)`, based on the program executions which expression "`a > 0`" or "`b == 1`" would be `false` most of the times. Whatever expression that usually yields `false` more frequently, should be evaluated first. The intuition is that if a low-cost expression evaluates mostly to `false`, you would want to check it first to avoid wasting calculations on other expressions that regardless of their results the branch will not be taken. The given example is trivial and changing its order would not affect the performance. However, other logical expressions could be large and have a tangible opportunity. In addition, the cumulative gain from these small changes is what the authors [71] are looking for.

Selakovic et al. [71] first instrument a program to capture logical expressions. For each logical expression, the profiler will run all available checks (expressions) after preserving the program state. Given the available test suite, the profiler will collect data about the expressions' cost as well as results for each traced execution. The cost is measured by the number of executed branching points within each expression rather

than time. Measuring the time for such expression is hard as these are usually fast operations. Given the expressions' common results (`True` or `False`) and the execution cost, the profiler will compute the computational cost of all possible orders of the expressions. Once they chose a given order and it proved to be making the program faster by executing it, the author will suggest the new order to the developer as an actionable suggestion.

The work introduced by Selakovic et al. [71] achieves a high degree of result simplification and understandability. However, the trade-off between the understandability of the results and efficiency is significant. Based on the authors' evaluation of the technique, the enhancement when applying all the semantics preserving changes is between 2.5% and 6.5% at the application-level. Given the significant size of the evaluated applications (e.g. Apache Struts), this is a very low improvement. Moreover, these changes are highly dependent on the given workload in the unit tests. As we know, a significant issue is that it is hard for developers to anticipate real-world workloads. Thus, any change in the workload at deployment might render the performance changes obsolete.

Another distinguishable work that attempts to be actionable is established by Della Toffola et al. [23]. In addition to being an actionable profiler, Della Toffola et al. [23] introduced a technique called *MemoizeIt* to look for memoization opportunities. As code might suffer from redundant computations that lead to program performance degradation, these are important opportunities to optimize the code. Traditional profilers might miss or low-rank such opportunities. Locating such redundant computations based on the given inputs and outputs of methods could reveal memoization opportunities that enhance the programs overall performance [23, 32, 83].

MemoizeIt narrows-down the tracked elements into the target object, parameters, and return results. Nevertheless, such profiling technique can be significantly expensive. Therefore, the authors introduced an iterative approach to monitoring the program. First, run the program with light profiling that records the execution time of each method. A method that does not have an expensive computation, is discarded as less likely to be optimized even further. Second, increase the depth of object exploration gradually looking for structure inconsistencies based on a flattened object representation (flattened representation is nothing but representing objects' types and values in nested arrays). For example, if a method `foo()` is called twice wherein the first call returned an object with two fields but in the second an object with three fields, then the method is dropped from the list for further analysis. Such iterative trimming and depth increasing allows *MemoizeIt* to maintain efficiency while increasing accuracy gradually. It is important to note that as the depth can be unbounded, for *MemoizeIt* the authors observed that object exploration of depth-2 is sufficient to be accurate. Third, *MemoizeIt* computes a cache-hit rate as it iteratively monitors candidate methods. *MemoizeIt* discards methods that go below a user-defined hit-rate threshold (defaulted to 50%) for each iteration. Such computation helps to maintain a list of potential methods that is more likely to benefit from memoization.

In addition to profiling methods based on their input and output, cluster them, and rank the method based on their potential performance gain, *MemoizeIt* suggests a fix to the developer to simplify the result and be actionable. In order to provide a suggestion on how to fix a method, *MemoizeIt* simulates different ways of fixes while validating the program integrity after applying the fix based on the given unit tests. Memoization fixes usually happen globally or locally (instance level) and are of multi (storing more than one input and output value) or single cached input-output pairs. Combining these possibilities required the authors [23] to simulate four different fixes. Given the result with the highest hit-rate, *MemoizeIt* considers it as the best possible change and suggests it to the developer.

There are other memoization techniques [58, 83, 37], but taking only *MemoizeIt* [23] as a representative technique we can identify the major limitations in memoization. First, given the conducted tests by Della Toffola et al. [23], it is clear that the number of such memoization opportunities is very small. From eight different applications from the DeCapo benchmarks [11], Della Toffola et al. [23] found only nine distinct memoization opportunities. Second, most of the found memoization opportunities are workload dependent. Thus, the fix, regardless of its performance gain, might not be of significance when deploying the applications. We can generalize these limitations to all memoization techniques.

The general observation of actionable profilers [60, 71, 23] is that there is a clear trade-off between the result understandability and performance optimization opportunities. To make results actionable, they had to be simple. Moreover, simple fixes are less likely to capture significant or unanticipated performance issues.

2.1.5 Other non-related but Distinguishable Performance Analysis Techniques

Some performance analysis techniques have been targeting specific domains such as mobile devices or supercomputing. Nevertheless, the essential challenges are usually the same. In this section, we are going to cover techniques that target different but tightly related systems to our work. The two main targeted performance analysis platforms are those that focus on mobile applications and parallel programs or scientific computing.

Given that smartphones are widely popular nowadays, the challenges smartphone applications developers face in tuning the performance of their applications is fundamentally similar to known performance analysis challenges [49]. However, the focus of the developer in parts of the applications where performance issues might appear is slightly different. For example, many of the performance issues found on smartphone applications are UI related. Because the UI actions trigger asynchronous executions in many different ways, Kang et al. [40, 41] introduced a method to track and profile these executions by categorizing them into small sets. Similarly, Brocanelli et al. [13] focus their effort on the expensive operation that happens on threads other than the applications main thread but cause performance issues. These techniques are different in directing the analysis to specific areas of the application but are fundamentally similar to all other performance analysis techniques in data collection and analysis.

Performance analysis of supercomputers or more generally distributed systems has its own different emphasis on what to profile. However, techniques that target supercomputers are also fundamentally similar to techniques that target sequential applications. Frameworks like TAU [73], in addition to providing a performance analysis of distributed systems, they target providing an architecture specific insight about the application performance [62, 12]. There are techniques that try to measure the software performance given the supercomputer capabilities [62] and highlight if the systems have been used to its full extent or not. Other [12] tackle the issue of collecting and unifying the knowledge about the data in a more modular supercomputer environment. Some techniques like the one presented by Herodotou and Babu [35] focus on providing an insight to the developer about the system performance under a given workload by manipulating multiple configuration options.

More focused on parallel programs, Curtsinger and Berger [21] attempt to provide developers with information about the expected performance gain if a particular method is fixed. They show that by slowing other threads whenever the method executed on one of the tracked threads. The slowdown of threads simulates the performance gain of fixing the targeted part of the code.

A fundamental difference in performance analysis techniques for supercomputer or closely related applications is that the results' understandability is not a significant concern. Users of supercomputer analysis tools (e.g. [46]) are usually more experienced and knowledgeable about their applications. Moreover, the use case for those systems like what Herodotou and Babu [35] presented is usually task oriented. That is, the user has a given location in mind about the software and needs comprehensive insight about its interactions and cost. Thus, there is less to no emphasis on understandability in such techniques.

The work focused on mobile devices or supercomputers is essentially similar to other performance analysis techniques designed for sequential computers. They are distinguishable in that they are domain specific. Given the knowledge about a given domain and the probability of where performance issues might appear, they tailor the fundamental methods of profiling to exploit such performance issues. However, these established works does not provide a solution to overcome the essential challenge of identifying problematic workloads. Moreover, they do not provide a method for communicating the profiler's findings with developers. In fact, the work established for supercomputing does not look at these two challenges as major issues. They usually know the expected workloads with high accuracy. In addition, users are usually having a goal in mind that leverage the result communication issue. Thus, we do not see our work highly related or specific to these fields.

2.2 Goal for Software Engineers

Software performance has been a dominating quality of software compared to other non-functional attributes during design (e.g., usability, modifiability, security, etc.). Software engineers design their system with performance in mind regardless of the requirements. For example, in file management applications, the performance requirements might not be formally stated. However, there is a general understanding that it is not acceptable for a file creation to take minutes or even seconds. Software engineers

usually relay on formal software architecture tactics [8] or even on the experience of the developers to meet the performance constraints.

Developers make decisions early on the application development cycle about software performance. According to the most cost-efficient, the design might include hardware or software solutions. Solutions that involve hardware are usually limited to financial constraints. Moreover, the hardware solutions have their own limitations and often are applicable if the requirements are guaranteed to exceed individual hardware capabilities [1, 63]. For example, a web-based application that processes tens of millions of requests in a few seconds might replicate the software on multiple machines to handle more requests on time. However, many performance issues are fixable within the software design. In fact, some performance issues are not solvable even if replicated over many pieces of hardware.

Established design tactics based on accumulated experiences help mitigate the severity of possible software performance issues [8]. For example, process time and blocked items are the two major contributors to web-application performance. The processing time of software might involve excessive use of hardware (e.g., CPU or memory) that needs to be carefully designed to ensure efficiency. Blocked items might emerge from resource contention, resource availability, or computation dependency. Software engineers need to consider each of these cases at the software level design to avoid introducing performance issues.

How to decide on what is acceptable as performance is another factor software engineers need to tackle. Most frequently, it is clear what would be the satisfactory execution or response time. However, as applications become functionally complete, previous measures might not hold. For example, developers need to anticipate the user's short-term memory in completing a task [54]. If a given system provides multi-steps to complete a job, a user has a short-term memory about the tasks they are carrying from one-step to another. If the accumulative time exceeds a defined threshold of the user's short-term memory, then the application's usability gets lower. Thus, previously identified acceptable responses change. Consequently, finding and fixing those tasks becomes harder.

An even more complicated problem is the developer anticipation of workload. Most of the reported performance issues arise from unanticipated workload or library use misunderstanding [38, 84]. Regardless of the developer's experience, it is hard to know how users will use the application. Any unanticipated use of the system could create severe performance issues. Even harder is finding these performance issues on deployed applications.

Software engineers need (and looking for) tools that help them identify what has been done, when, and by whom as applications evolve [75]. Widely used continuous integration tools proved their usefulness for software engineers. These tools provide different perspectives on how the application is performing. Moreover, they introduce the opportunity for the developers to fix any highlighted issue as they appear. Similar to the code coverage measure of functional testing, for example, a performance analysis tool that continuously monitors and clearly report the application performance is necessary.

The techniques we covered in the previous section (Section 2.1) provide different methods to explore the system given existing workloads. However, these tools are either does not explore unanticipated performance issues or lack providing an easy way of understanding the results.

Understandability of a performance analysis tool results is a major barrier [39, 2, 33]. From the efforts we presented, it is clear that there is usually a tradeoff between how valuable the findings of a performance analysis tool and its results understandability. There have been efforts to bridge such gap by relating the found performance issues to the architectural representation of the system [19]. Nevertheless, even if such a tool is bridging the gap, perceiving the results at a later point in the development process (e.g., before going live) could make such effort obsolete as the architectural state might have already deviated from how they initially designed. A tool that brings clarity and continues insight about the performance of the applications to software engineers is necessary to preserve desired application performance.

A more significant limitation found in performance analysis tools is their lack of reporting unanticipated performance issues. The tools presented in the last section assume developers are performance-testing experts, and they have a well-established set of performance test cases. Hence, they mostly focus on how to measure performance or where to look for the issues. However, as stated before a significant gap is in providing developers with unanticipated workloads. There is a need for tools that generate new workloads given the application's invariants to expose possible performance issues. Moreover, tools that allow developers to match their understanding of the application

performance with the new findings to either prevent such cases (through validation) or assert the performance state.

In the next section, we go through a set of efforts that either try to provide an insight about the application's performance or provide new exploratory workloads. These efforts do not focus on the understandability of the results. However, they only tackle the workload issue as we see it the more significant of the two problems.

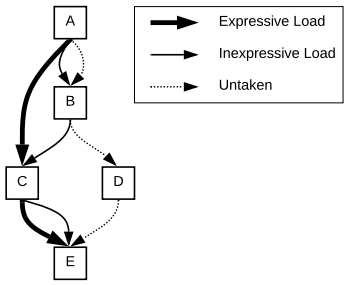

Figure 5: Example of different paths traversed based on testing data.

3 Input Based Performance Analysis

Inputs are an essential factor in any software analysis tool. However, the issue of expressive inputs absence within the performance analysis field is even more complicated. Different sets, sizes, and orders of inputs can express different limitations of a method. Moreover, the performance of a given method based on a given input can significantly differ according to the whole program state. For instance, `Mozilla Bug #490742` [38] illustrates such a performance issue. The reported (and fixed) performance issue appeared only when users tried to bookmark 20 or more pages at once using the *Bookmark-All* functionality on Mozilla Firefox. Without going into details of how the performance issue was introduced and fixed, it is easy to see how a given case could escape the software testing designed toward testing the functionality of the method. The given example shows how performance issues with a relatively low number of inputs could escape testing. Load testing [86, 15, 16] could be a probable solution for such simple performance issues. However, the dimensionality of inputs is only assumed to be larger in most cases. Also, there exists performance issues that does not occur because of the size of the inputs. Thus, the issue of having a meaningful input that expresses the application performance is significantly difficult.

The number of different possible paths within an application can be significantly large. Static and dynamic analysis tools provide insightful feedback [79] and module classification [30] about the application under test. For example, code coverage tools [79] can easily report that the path `ABDE` in Figure-5 has never been taken. Such insight allows the developers to write tests that express the functionality of such a path. However, presented performance analysis tools do not help the user distinguish how differently tested paths `ACE` and `ABCE` from Figure-5. Because of the given input data by the user, a passive performance analysis tool could mislead the developer into believing that path `ACE` has a performance issue and thereby miss actual performance optimization opportunities.

In this section, we present proposed performance analysis tools that look into the influence of the input data on the performance of applications under test. There are two major types of input-based performance analysis tools. One tries to provide insight on how the inputs influence the application performance based on the given inputs by the developer (e.g. [51, 85]). While the other tries to explore new inputs that would express the performance of the application beyond what the developer anticipated [31, 72, 48].

3.1 Input Influenced Insight

There is a number of performance analysis techniques that realize the significant influence of the inputs on how applications are performing [51, 50, 20, 3, 72, 45, 85, 87, 74]. These approaches differ in how they define the inputs and analyze them. For example, some efforts [20, 85, 18] try to define the cost functions of the analyzed methods based on the given inputs. Some others [74, 87], narrowly define input as the possible configuration a user could change for a given application and provide insight about which set of combinations maximizes the performance. Others [45, 3] try to find application bottlenecks based on the given inputs and provide the developer with insightful feedback to manually provide better inputs. Despite their differences, all of these techniques are either passive where they do not look for new interesting inputs or rely on the developer to drive the input search.

3.1.1 Defining Cost Functions

Algorithm researchers and software engineers for scientific applications usually use cost functions when analyzing algorithms complexity. Cost functions are usually asymptotic lower and upper bound representations of the algorithm time or space cost. Such insight can be useful to understand what inputs could lead the application under test into an unsatisfying performance. Using cost functions, software engineers overcome the input anticipation issue by confirming whether the inputs they expect fall within a bad performance portion of the cost function or not. Cost functions are not necessarily trivial to obtain. Moreover, they are even harder when the number of inputs is unlimited for complicated applications. Hence, performance analysis researchers [27, 20, 85, 18] attempted to find these cost functions automatically based on the program runs.

Automating the construction of cost functions is not trivial. One of the most essential problems is how to determine the inputs of an algorithm. Zaparanuks and Hauswirth [85] established a technique that identifies all loops in a control-flow graph and recursions in the program's call graph to locate areas where performance issues might occur. Moreover, they use the execution count of loops as the cost instead of measuring time to avoid large overhead. Zaparanuks and Hauswirth [85] essential shortcoming is in determining the algorithm/method inputs. Either they limit what they considered inputs on field references to data structures accessed within the method execution or external input files. Even more, their approach has issues with algorithm inputs based on primitive data. Because the cost interpretation of primitive data types can differ (e.g. an integer can be seen as a *number of digits* on the schoolbook multiplication algorithm or a *value* on a factorial algorithm), their approach focuses on one aspect only.

Conceiving primitive inputs is not the only essential limitation on Zaparanuks and Hauswirth [85] work. Experimental algorithmic techniques such as Zaparanuks and Hauswirth [85] take portions of the code and test it extensively on different input sizes. While such approaches can provide valuable insight about the code portion, it suffers from studying those small portions out of their context (the system as a whole) [78, 88]. Much of the performance issues that escape testing are those that combine multiple and complex interactions between different portions of the system. Taking a method out of its context or looking at it within a static context does not provide complete insight into the application performance.

Coppa et al. [20] understood the context issues for performance analysis in general and for approaches that try to generate cost functions in particular. They mitigated the limitations of existing approaches that automatically measure the performance of the routines as a function on their input size by looking at it within the actual context of the software.

Before looking into the context issue, it is important to look into how Coppa et al. [20] approached the essential challenge of defining the inputs to track for performance analysis. To solve this problem Coppa et al. [20] introduced a metric called Read Memory Size (RMS). They define RMS as the number of distinct memory cells first accessed by a method (call it `foo()`), or by a descendant of the method in the call tree, with a read operation. They obtain such information using tools such as Valgrind [57]. The authors argue that calls to memory by a function for the first time (never accessed before) with a read operation contain the input values of the routine. Conversely, if a cell value is first written and then read by the routine, the value is not part of the input as it was determined by the routine itself.

Although the definition of the RMS can limit the number of tracked inputs, we think this is an important contribution on how to define inputs in the field. This approach in defining inputs solves the issue of understanding complex user-defined objects by representing them in their simplest form. The overhead of the technique is significantly high, but that is an expected trade-off between the details and cost of any software analysis tool.

Having the RMS defined, Coppa et al. [20] define the performance analysis technique as the collection of RMS for each method encountered in the program. For example, for method `foo()`, they find the set $N_{foo} = \{n_1, n_2, ...\}$ of distinct RMS values on which `foo()` is called during the execution of the program. For each estimate of the input size $n_i \in N_{foo()}$, they collect the number of times the method is called under that input, the maximum and minimum cost for the routine to be executed based on the observation regardless of the definition of the cost, the sum of all the costs observed for the given method and $RMS(n_i)$ and the sum of the costs' square.

This definition of how Coppa et al. [20] collect inputs makes their approach context sensitive. Because they separated how they collect input information from where the method under observation is located, they were able to capture the whole program context. As the ultimate goal of the complexity analysis of an algorithm (or a method) is to find a closed

form expression for the cost (e.g. running time) on the input size, Coppa et al. [20] used curve fitting and curve bounding to generate cost functions.

In line with expectations, the overhead of Coppa's et al. [20] approach is significantly high. Compared to other tools, their approach requires an average of 30 times the normal run (the peak was 78.3x). For space requirements, their approach on average requires 2 times the normal space. It is normal to have such high overhead given the fine granularity of collected information.

Coppa et al. [20] state that a single run of the system under test is mostly sufficient to use their tool. The key observation they highlight is that the number of distinct RMSs for each method will not increase under the same input. Distinct RMSs are more important than RMSs with different values because they expose different paths within the same method. Nevertheless, we think this is the essential limitation on such approaches [20, 85], as low distinct RMSs are possible given that developer's inputs usually target functional testing.

Closely related to these two major techniques [20, 85], Chen et al. [18] generate cost functions for selected paths. Based on given inputs, they classify paths into high-probability and low-probability paths. The high-probability paths are those that execute under most of the given inputs. Respectively, low-probability paths represent the corner cases of the program that found based on a small number of inputs. Chen et al. [18] use symbolic execution [68] to classify the given paths and use loops unfolding to ensure the scalability of the technique. The high-probability paths are a representation of the program normal execution. Understanding the program performance under these cases helps developers understand the state of the program normal behavior. Low-probability paths, on the other hand, represent the usually untested cases by developers. Highlighting these cases, bring developers' attention to unanticipated issues. Given the symbolic inputs and the high-probability and low-probability paths, Chen et al. [18] generate cost functions.

Symbolic inputs are efficiently translatable to actual inputs in theory [80]. However, given how Chen et al. [18] treated loops to prevent an explosion on the number of possible paths, interesting performance insights are not explored since the majority of performance issues occur within loops as discussed in Section-2.1.2. Moreover, the inputs generated by the symbolic inputs are not pathological inputs. Meaning they express the change in the input scale rather than the inputs' nature. Thus, also missing important performance analysis opportunities.

The approaches presented [20, 85, 18] regardless of other limitations, they suffer from the essential issues of how to make sure that given inputs are sufficient for driving the analysis tools into interesting performance issues. For techniques such as Coppa et al. [20], this can be mitigated by incorporating the code coverage insight to ensure that the highest number of paths is taken. However, this is beyond the problem of input generation.

3.1.2 Inputs as Configurations

A common root cause of introducing performance issues is the misuse of off-shelf software [38, 59, 34]. Developers usually do not clearly understand how to use an API (Application Programming Interface) or APIs behavior changes across different versions of the same system but no update is applied to their calls. Whether these off-shelf software are libraries or standalone programs, the configuration passed considered as inputs. From this perspective few techniques [87, 74] were proposed to analyze how different configurations (inputs) combinations might influence the application's performance.

Performance issues introduced on applications that were working as expected can be hard to understand for software engineers. These performance issues usually arise by introducing wrong configurations over different versions of the software. Zhang and Ernst [87] introduced a recommender system to choose the right configurations for the desired performance.

To identify and report configuration changes that cause performance issues, Zhang and Ernst [87] require two different versions of the same artifacts. Instrumenting the code and using the same configuration, they use user's usage (test cases or actual usage of the system) to record traces. The instrumentation follows simple techniques that identify predicates on branching control-flows and count their executions. Given traces, Zhang and Ernst [87] match predicates of the traces on the old version of the system to the ones from the newer version. They then compute the behavioral deviation for the matched predicates from different versions within a given method. This gives an indication of how similar or different these two predicates are within different versions. Using thin slicing, following the dependency between a slicing criterion (e.g. statement initialization) and predicate using only the data flow dependency, to identify the relationship between a given behavioral change and

a configuration option, Zhang and Ernst [87] recommend which configuration is most likely the cause of the behavioral deviation.

A significant limitation in Zhang and Ernst [87] approach is that they require two different versions to measure the influence of configurations. Access to older versions of the same software might sometimes be difficult if not impossible. However, even if older versions are available, Zhang and Ernst [87] approach does not look into how the given configurations influence the system compared to other configuration for the same version. Thus, missing performance optimization opportunities that are actual to the software. To mitigate this Siegmund et. al. [74] proposed an approach that looks at how a user can select the optimal configurations of a system while maintaining the desired performance.

The work presented by Siegmund et al. [74] target large systems where configurations changes by the user could significantly affect the performance. For example, measuring the performance of a database management system with *indexing* turned on or off can provide useful insight about the *indexing* effect. The number of such features can be significant in large systems and it is hard to predict their effect on performance by users. Even in Siegmund et al. [74], the number of features can be an obstacle because the number of interaction possibilities is exponential on the number of features. To avoid such issue they compose features that cannot be measured in isolation into a single feature to reduce the number of possibilities. In addition to that, Siegmund et al. [74] focused on test heuristics to trim the search space. Given these guidelines, Siegmund et al. [74] predict the system performance and report it to the users.

Such techniques have a different definition of what we considered as input. Nevertheless, these are still passive techniques (no new inputs are generated). Also, even if they generated new configuration (i.e. for scalable configuration), they are manipulating systems at a very high level. Such high-level observation could lead to missing some interesting performance behaviors as well as mismatching the developer's levels of abstractions.

3.1.3 Input Driven Analysis

The need for input driven performance analysis was grasped by a few established techniques [45, 3], but the amount of automation is very limited to nonexistent. Such established ideas, recognize the importance of the inputs to drive the software engineers understanding of the software's performance. Moreover, they understand the number of iterations and deep understanding needed to find inputs that influence performance. Thus, these techniques provided tools that assist in the performance analysis process based on the give inputs.

Küstner et al. [45] present the simplest form of the techniques. Given that inputs highly influence the application under test performance, Küstner et al. [45] provided a tool that highlights methods based on their inputs. There is no automation on the tested inputs. Rather they provide an input based perspective of the application performance. For example, given the selected set of methods a developer would like to understand, the developer defines input ranges (e.g. $x < 5; 5 <= x <= 10; 10 < x$). Using the different inputs ranges, the tool generates three different profiles for each input range based on the provided test suite. Küstner et al. [45] put a lot of emphasis on the input when analyzing the application under test. However, the needed manual interference by the developers is an obvious limitation.

Ayala-Rivera et al. [3] also focus on the workload to assist developers improve their productivity. However, compared to Küstner et al. [45] they provide some notion of automation of the workload. Instead of providing only performance feedback and wait for new inputs, Ayala-Rivera et al. [3] allow the developers to identify a set of important input parameters and their characteristics. The given inputs are then automatically stressed (e.g. quadratically increasing an array size for each new execution) based on predefined policies to inspect possible performance issues. Although such an approach might have some automation to manipulate the workload, it does not actually explore any unanticipated issues by providing scalable inputs, as scalable inputs are not necessarily interesting inputs.

3.2 Input Generation

Finding a solution that would drive all the presented performance analysis techniques to actual performance issues requires searching the space of all possible special inputs given the whole program context [43]. Special inputs are not only a stress of some input size (e.g. increasing a size of an array for a sorting algorithm [15]), random generation of load inputs then passively select the most diverse [86], or focus on increasing the coverage of the tests [16], rather it is a deeper understanding of the given method or

algorithm functionality.

Interesting performance inputs usually defined as *pathological inputs* [48]. Simply, *pathological inputs* are inputs that maximize the cost of software execution given different combinations of inputs but of a fixed length. For example, on sorting algorithms *pathological inputs* are these inputs that maximize the execution time without providing longer arrays of inputs (i.e. *scalable inputs*). The length of the inputs is important and can escape the developers understanding of the software complexity. However, *pathological inputs* are the most difficult to generate and understand.

Although such performance information is valuable, sometimes it is missing from the most well-documented libraries. For example, Java Development Kit (JDK) documentation of array sorting states that the given algorithm "offers $n * log(n)$ performance on many data sets that cause other *quicksorts* to degrade to quadratic performance" [64]. Reaching a high detailed state of performance analysis automatically is a challenging task. Moreover, finding special inputs for a software application that is not as self-contained as a sorting algorithm is even more challenging.

In this section, we present the best attempts we are aware of to automatically generate new special inputs for the purpose of performance analysis. Distinguishable input generation techniques use machine learning methods [31], genetic algorithm [72] or fuzzing [48] to search for special inputs.

3.2.1 Machine Learning Driven

The earliest found technique to use machine learning as a driver for special inputs finding presented by Grechanik et al. [31]. They created a tool called FOREPOST that takes initial inputs and their associated execution times to generate new possible special inputs.

FOREPOST [31, 53, 50, 51] execute the application under test on a small set of randomly chosen test inputs. Then it infers rules with high precision for selecting test input data automatically to drive the application toward possible performance issues. Rules are in a form of `if-then` statements. For example, based on the loaning system presented in the paper, a rule could be "if inputs `convictedFraud` is `true` and `deadboltInstalled` is `false` then the test case is good." The given example indicates that using the given inputs leads to an expensive performance (more computation time), thus it is a good test case as it exposes performance issues. As input data are clustered into expensive and cheap tests, FOREPOST report methods that are specific to expensive test cases, which is most likely to contribute to a bottleneck.

It is important to understand how FOREPOST obtains performance rules to understand the usage of a machine learning technique. FOREPOST uses the set of values of the application under test as input to a machine-learning algorithm. Such input can be represented as $V_{I_1}, \ldots, V_{I_k} \rightarrow T$ where V_{I_m} is the value of the input I_m and $T \in \{Good, Bad\}$. The machine learning classification algorithm learns the model and outputs rules of the form $I_p \odot V_{I_p} \bullet I_q \odot V_{I_q} \bullet \ldots \bullet I_k \odot V_{I_k} \rightarrow T$, where $\odot$ is one of the relational operators and $\bullet$ stands for logical connector `and` or `or`. Such learned rule is feedback to the testing script to automatically collect execution costs and guide the selection of new input data. Repeating the process will partition the input data and generate newly learned rules. The algorithm reaches a high degree of probability of expensive input values if no new rules are learned.

Grechanik et al. [31] argue that frequently invoked methods, which appear in cheap and expensive test cases, can be of no significance to performance insight. Rather, FOREPOST reports less frequently invoked methods that appear within expensive test cases or have little to no significant impact in cheap test cases. It is clear that such argument does not always hold as for simple example (e.g. sorting algorithms) we could have good and cheap test cases over many invocations and still be of performance significance. However, such observation can be domain specific [10] as Grechanik et al. [31] mainly evaluate their tool on closed-source loaning application and select only Boolean inputs to manipulate.

Evaluating FOREPOST shows that the technique does actually generate inputs that worsen the performance. For example, under random testing of *JPetStore*, a widely used java benchmark, it takes an average 576.7 seconds to execute 125,000 transactions. With FOREPOST, executing the same number of transactions takes an average 6,494.8 seconds. However, FOREPOST is less efficient in finding bottlenecks. Examining FOREPOST's top 30 possible bottlenecks methods for a Renter application results in finding a single performance issue of wasted computations.

We believe that the limitation in ranking interesting performance issues is not a significant issue. In fact, we argue that reporting bottlenecks should not

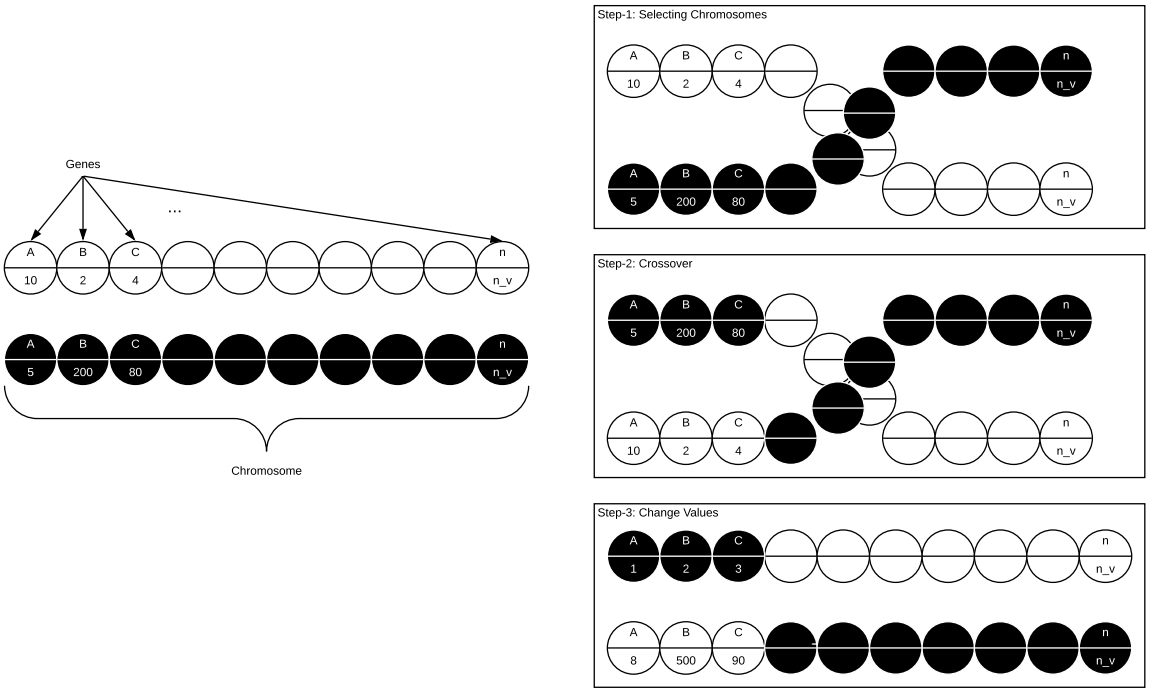

Figure 6: An abstract representation of using genetic algorithms for input generation.

be examined using an input generation techniques (see Section-4). Rather it should only feed such data to specifically built profilers (e.g. *gprof*). However, an actual limitation with FOREPOST is its consideration of high-level inputs only. FOREPOST does not manipulate inputs at a finer level (e.g. method level). Rather it only looks to the application's inputs. Some would argue that such high-level inputs are the only inputs a user of the system can manipulate. Nevertheless, it is not the level at which software engineers are examining the system. Usually, different developers develop and test different modules of the software. Generating inputs that encompass all modules can decrease results understandability. Moreover, any change to the internals of the system could expose interfaces to previously unexamined inputs by FOREPOST.

3.2.2 Genetic Algorithms Based

Another distinguishable approach was proposed by Shen et al.[72, 50, 51]. Their definition of the input generation for performance problems is similar to the one given by Grechanik et al. [31]. However, toward the evaluation of their approach as we will explain later, their definition lacked a demonstration of generality. Shen et al. [72] define the input generation problem as a search and optimization problem. Moreover, they suggest using a genetic algorithm to drive the search task. Shen et al. [72] argue that machine learning based techniques [31] are fit for pattern recognition rather than a search and optimization problem. Thus, because the genetic algorithm core idea is to find new fitter "*individuals*" based on existing ones, they think it is a good fit for inputs for performance testing.

Although we think that a good classifier (machine learning algorithm) is suitable for the problem on input generation if combined with a good input selection method for testing, a genetic algorithm is also a good fit if combined with a non-random input selection method. An important limitation in the genetic approach, as we will describe it, is the possibility of falling into less important local-minimas when searching for special inputs.

Shen et al. [72] major contribution is on explaining how to represent inputs using genetic algorithm. In the genetic algorithm, they have what is known as an *individual* who is essentially a *chromosome*. *Chromosomes* in their part are made of a set of *genes*. The goal of genetic algorithms is to generate new *individuals* by crossing-over fit *chromosomes*. Calculating the fitness of a *chromosome* is based on a predefined fitness function that considers each *gene*. Figure-6

shows a representation of the *chromosomes* tailored for the input generation problem.

For each set of inputs, Shen et al.[72] consider these as *chromosomes*. Within each given *chromosome*, we have a set of *genes* that represent an individual parameter and its value. For example, if a method accepts an array `x = [1, 2, 3]` and a boolean `y = True`, then a *chromosomes* is the sequence of *genes* $\{x_1 = 1, x_2 = 2, x_3 = 3, y = True\}$. Using inputs with different values for the same method, the fittest sequences of inputs (i.e. *chromosomes*) are crossed-over to generate a potentially new fitter sequence of inputs.

The fitness function simply maps the input values to the elapsed execution time. Inputs that maximizes the fitness function are fit inputs. Given a run of the application under test based on randomly generated inputs, the fitness function will have few candidate sequences of inputs. As shown in Figure-6 each pair of the candidate sequence of inputs (i.e. *chromosome*) will be crossed-over. The crossover phase simply consists of selecting a subset of the inputs (i.e. a set of *genes*) and exchange them between the sequence of inputs. Finally, for each crossed subset of inputs the values are randomly changed (authors did not provide details on how to select new values thus we assume it is random). In the last step of each iteration, the application under test is run again using the newly generated sequence of inputs to calculate its fitness.

The presented evaluation by Shen et al. [72] does not provide clear results to assess the potential of the technique. The evaluation uses URLs as inputs, which does not correlate clearly on how the approach is applicable to widely available non-web-based applications. Moreover, the experiments highlight the technique's results on injected performance issues. The essential goal of input generation techniques is to generate unanticipated inputs that reveal actual performance issues. Injecting performance issues does not help in understanding the possible limitation of the technique. Thus, it is hard to draw conclusions about the approach.

Regardless of the evaluation methodology, we can identify some possible limitations. First, the crossover step between two sequences of inputs (i.e. *chromosomes*) by itself does not necessarily generate new inputs. Trying different combinations of the same set of values can lead to some interesting results, but no actual new inputs generated. Thus, we fall into the essential issue of the developer's given inputs that does not necessarily cover all the performance possibilities. Second, based on the previous

```
function quicksort(array):
    /* initialize three arrays to hold
    elements smaller, equal and greater
    than the pivot */
    smaller, equal, greater = [], [], []
    if len(array) <= 1:
        return
    pivot = array[0]
    for x in array:
        if x > pivot:
            greater.append(x)
        else if x == pivot:
            equal.append(x)
        else if x < pivot:
            smaller.append(x)
    quicksort(greater)
    quicksort(smaller)
    array = concat(smaller, equal, greater)
```

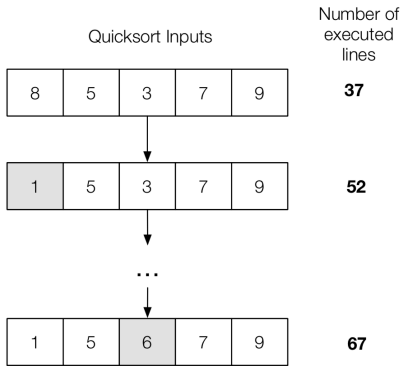

Figure 7: [67] Steps taken by *SlowFuzz* to find inputs that maximize the execution cost of *quicksort*.

limitation, Shen et al. [72] randomly generated new values for each exchange input (i.e. *gene*). This solution does introduce an actual new input. However, because the given genetic algorithm does not calculate how each input (i.e. *gene*) is contributing to the fitness of each sequence of inputs, the new values are not necessarily selected with high potential of generating a new fit sequence of inputs that drives the application's performance into performance issues. The absence of a link between the newly selected values and the new combinations of inputs is an essential limitation of such approaches.

3.2.3 Fuzzing Driven Inputs

To our knowledge, Lemieux et al. [48] presented the most general and focused approach to generate input that leads to performance issues and target the comprehensive definition of performance inputs. They

precisely target *pathological inputs* by always fixing the size of the manipulated set of inputs.

Lemieux et al. [48] use fuzz testing as an engine to drive the input generation. Fuzzing is widely used in functional requirements testing where the application under test is barraged with randomly generated tests. For functional testing, the goal of fuzzing is to use feedback-directed mutational fuzzing to increase the code coverage. Petsios et al. [67] are the first to use fuzzing for performance inputs generation. Their intuition is to iteratively use evolutionary search techniques to maximize a program execution cost.

As shown in Figure-7, Petsios et al. [67] developed a technique called *SlowFuzz* that uses fuzzing engine to generate inputs. In addition, *SlowFuzz* defines a cost function to rank the inputs based on their execution cost. The example shown explains how the fuzzing algorithm iteratively finds a sorted array that maximizes the cost of executing the given *quicksort* algorithm (i.e. increasing the length of execution paths).

Lemieux et al. [48] adopt the same methodology but instead of targeting inputs that only maximize the execution cost of a given path (e.g. *SlowFuzz* [67]), they also considered inputs that hit new locations (i.e. increasing the coverage). *SlowFuzz* is greedy in that it looks for inputs that maximize the count of edges on the control-flow graph over exploring new paths in favor of achieving a worsen performance in a shorter time.

The implementation of Lemieux et al. [48] approach called *PerfFuzz*. Initializing *PerfFuzz* requires a seed input that is known to run the program. *PerfFuzz* adds the seed input to a set called `ParentInputs` that maintains known special inputs. The set holds inputs that are known to either maximize the test coverage (finding new paths on a control-flow graph) or maximize the execution cost of the given control-flow graph. Lemieux et al. [48] argue that such setup allows them to avoid local maximums by having a multi-dimensions objective. In fact, they argue that it helps them not necessarily finding the global maximum, but potentially many different near global maximums.

If we look at a single iteration of *PerfFuzz*'s algorithm, we find that the algorithm first generates new inputs by randomly permuting the bytes of inputs from the `ParentInputs` that has potential. Bytes permutation simplifies the input problem. However, might not be a good approach for complicated inputs such as widely used data structures. Potential inputs, nevertheless, are the ones from the `ParentInputs` set that maximizes performance value for some cost measurement in the application. The cost definition can differ based on the developer needs. For example, it could be the number of bytes allocated at `malloc` statement or cache misses. For *PerfFuzz* they defined cost as the execution counts of control-flow graph edges.

```
(1) "tvÇ1PFEj??A4A+v!^?^AE!§^?MPttò8dg80ÿ(8mrÿÿÿÿ"
(2) "t t t t i nv t X t 1 9 t l t l t t t t t"
(3) "t <81>v ^?@t <80>!^?@t <80>!t t^Rn t t t t t t t t t"
```

Figure 8: Different inputs generated by *PerfFuzz* [48] for the WordFrequency application.

All newly found inputs are `ChildInputs`. As *PerfFuzz* finds a collection of new `ChildInputs`, the application under test is run again given the new collection. *PerfFuzz* adds inputs to the `ParentInputs` set if they show a maximization in the coverage (newly executed edges) or cost (count of edge execution in this case). The number of inputs in the `ParentInputs` set cannot exceed the number of control-flow graph edges at any given point in time. Thus, *PerfFuzz* reevaluates all inputs within the set of `ParentInputs` each time a new input shows a performance maximization. The algorithm repeats this process of finding new inputs and running the application against them until it hits a given threshold (e.g. 1 hour).

Lemieux et al. [48] evaluate their findings compared with the ones generated by *SlowFuzz* [67]. *PerfFuzz* clearly outperformers *SlowFuzz* in finding inputs that are more pathological. However, an essential issue in fuzzing based approaches, in general, is the difficulty of understanding their generated inputs.

For instance, applying *PerfFuzz* on a WordFrequency application would generate results similar to the ones shown in Figure-8. The shown three inputs are all revealing different performance issues. Input (1) in Figure-8 depicts a single long word issue, which maximizes the time taken by the application to compute a hash of the word. The second input (2) in Figure-8 exercises a case where a repeated execution of the method `add_word()` occurs because of the many short words. The last given input (3) presents a case where many hash collisions are occurring, thus longer execution time in traversing a linked list.

All the given inputs in Figure-8 are valid and valuable examples. However, relating them to the root causes of the performance issue requires a deep un-

derstanding of the algorithm. In addition, as the size of the application grows the problem becomes even worse.

A more significant issue with *PerfFuzz* is its focus on a single high-level input at a time. As presented before high-level inputs are not the level at which software engineers are thinking. Especially for large complex applications, manipulating the high-level inputs would not be in line with a sub-module or a function a developer is writing. Nevertheless, the issue with *PerfFuzz* is not in manipulating high-level inputs, but also manipulating a single input at a time. Limiting the number of inputs interacting with a system to a single input reduces the problem space significantly. For example, in a basic database system, it is important to understand the relation of indexing on different operations such as insertion and querying. Examining a single variable at a time could lead to missing interesting performance insights. Also, it makes *PerfFuzz* simplifies the problem beyond applicability for real-world applications.

Another limitation is in the time needed to find valuable input. Because *PerfFuzz* [48] mutate input randomly, the effort needed to find local maximum is costly. A more educated permutation of input values could cut the time needed (6 hours for the WordFrequency application) to find interesting inputs significantly.

4 Efficiency Measurement

An apparent issue in the field of performance analysis, in general, is the lack of a unified method for efficiency measurement [69]. For us, efficiency measurements mean how effectively a technique helps software engineers in finding actual and previously unknown performance issues. Mytkowicz et al. [56] illustrate the severity of the efficiency measurement problem. In their work, they show how a set of Java profilers (*xprof*, *hprof*, *jprofile*, and *yourkit*) do not agree on a *hot* method under a unified benchmark and the testing data. Such disagreements indicate that there must be at least three wrong profilers as Mytkowicz et al. [56] state. In examining the core issues between the given profilers, they found them violating a fundamental sampling attribute. None of the profilers collected samples randomly. Therefore, each profiler would serve a different purpose.

When presenting their evaluation, authors of performance analysis tools follow different methods for evaluation. Some authors present their findings and confirm them with the application's developers as a method of measuring efficiency [29, 28]. Either they provide analysis of their findings or assisting the developers to understand the results. Others evaluate their efforts against tools with similar general goals but focus on other strengths such as overhead or coverage [20]. A more prominent efficiency measurement methodology [21, 31] is comparing results with other performance analysis techniques given a unified application under test but one that express the presented goal better. This approach is usually unfair and unrealistic because the goals of the two performance analysis techniques are unmatchable. For example, Curtsinger and Berger [21], who develop an application for finding performance improvement opportunities within a parallel program, compares its results to the ones produced by *gprof* [28]. The goal for *gprof* was never to profile parallel programs. Thus, the comparison does not hold.

Grouping applications benchmarks for performance testing such as the DeCapo [11] benchmark is also insufficient efficiency measurement. These benchmarks are not necessarily a realistic representation of real-world applications. In addition, these benchmarks could lead to more tailored solutions, as the goal becomes to find *new* performance issues from the same application. Most importantly, the goal is not only finding performance issues but also assisting software engineers throughout the software evolution.

Performance analysis tools do not apply fixes automatically. This is a major aspect of performance analysis tools due to the tradeoff between program soundness and the potential of finding performance issues. Using the performance improvement percentage based on automatically applied fixes is a precise measurement for compiler-level optimization techniques. It shows the benefits of automatically applied fixes. Limiting performance analysis tools to find and apply valid fixes of performance issues restricts the potential of reporting interesting performance issues as discussed before. Therefore, it is hard to adopt the same approach by performance analysis techniques.

As fixes are manual, the performance improvement measurement cannot be ever precise for the performance analysis techniques. Humans' experiences and understanding can vary significantly. Thus, fixes applied for the reported performance issues could provide different speedup values depending on the developer's experience. Nevertheless, we think that fixing the top k number of reported performance issues is the best available way for measuring performance analysis techniques. Authors of any technique can

put the needed effort and consultation into fixing the reported k performance issues to maximize the benefits. This could hopefully bridge the gap in the performance efficiency measurement limitation, as the value gained from fixing reported performance issues is essentially what software engineers would consider in using one tool or another.

An even more compelling solution for the input generation focused approach is to use existing performance analysis techniques to measure efficiency. The three major input generation efforts presented in this report [72, 31, 48] should have a unified evaluation method. For example, using tools such as *gprof* [28] to monitor the ranking change in performance issues between different methods. Performance analysis tools might not serve a precise goal an input generator is targeting, but different ones should cover and highlight different goals.

5 Machine Learning for Input Generation

Machine learning found to be effective in many different problems but barely touched on for performance analysis. In fact, our comprehensive search for techniques that explicitly adapt machine learning to generate input for performance testing results only in the work presented by Grechanik et al. [31]. As presented before Grechanik et al. [31] approach did not generalize enough to identify inputs automatically.

As machine learning proves itself as a valid or promising solution for different applications, we thought it is important to explore what established machine learning modules are applicable to the input generation problem. Finding an arrangement of inputs that maximizes a method execution is a very high-dimensional problem. Machine learning is widely known to be suitable for high-dimensional problems.

Machine learning is classified into supervised, unsupervised and reinforced learning. We first evaluate the applicability of each learning method then discuss the most relevant method to us.

Supervised machine learning approaches are applicable in cases where labeled data are available. In software analysis, data is not an issue as inputs can be paired with the runtime cost based on a given traced run. In fact, the sheer amount of data where slight changes in input could produce a slight uninteresting change in output is one of the problems. Another is that supervised learning models must use some sort of random data generation method to generate actual new inputs. Otherwise, the model would be passive in that it looks only at what user provided as inputs rather than finding new ones.

We think that supervised learning methods are more applicable in performance driven source code learning than input generation. For example, using regression-testing techniques that identify the source code change causing a performance change [52, 70], can create labeled data of source code fixes for performance improvements. A challenge here is to automate the deployment, run of the application and traces collection of not only heterogeneous artifacts but also many different versions of the same artifact.

Unsupervised machine learning techniques, where no labels are available, are suitable for clustering, anomaly detection, finding associations, etc. Similar to supervised learning, unsupervised learning can cluster expensive inputs from cheap inputs. However, also similar to supervised learning, there must be an input generation used along with unsupervised learning. Depending on user-created tests would lead to limitation such as the ones presented in different passive performance analysis approaches (e.g. [28, 5, 61]).

Reinforcement learning maintains a reward function that increases as a learning model gets closer to the optimal solution by trying different inputs at different stages of the learning process. In our case, the reward function can be seen as a fitness function that increases as a method's execution cost increases and decreases otherwise. In addition, the learning stages as runs of the application under test with different inputs. Assuming method inputs can be automatically identified, we think that reinforcement learning methods are the most applicable to our problem.

Stanley and Miikkulainen [77] established a notable work for reinforcement learning. They focus on examining the efficiency of using fixed-topology models (e.g. fixed number of hidden units, node, and connections) versus evolving-topology for reinforcement learning tasks (see Figure-9). The NeuroEvolution of Augmented Topologies (NEAT) presented by Stanley and Miikkulainen [77], can be seen as a method that searches for behavior instead of a value function that is more suited to continuous and high-dimensional problems. Although a fully connected model that uses backpropagation can in principle approximate any continuous function, the model topology can affect the speed and accuracy of learning. Moreover, deciding on a topology that is applicable for a given

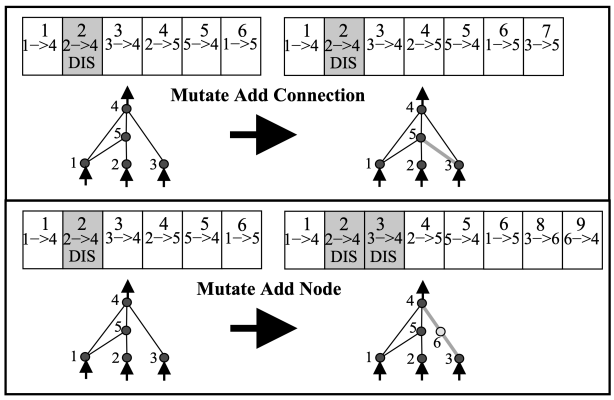

Figure 9: Mutation of a neural network topology by adding a connection or node [77].

problem is a time-consuming task as well as it could emphasize some inputs over others.

Because the input generation problem is a high-dimensional problem, where even the flattened representation of a single complex object can result in many different primitive inputs to permute, machine learning is a promising fit for the problem. Also, because the different input have different emphases on a method performance (e.g. the case of indexing effect on database operation compared with the actual data), solutions such as NEAT are an even better fit. We think NEAT [77] is a promising methodology for the search of performance inputs. The ability to iteratively decide on the emphasis each input has (evolving the topology) for different methods and different high-dimensional inputs is a solution that has not been explored within the performance analysis field.

Recent input generation work [31, 72, 67, 48]; either reduce the dimensionality of the inputs by permuting a single input at a time or builds a fixed-model targeting different types of inputs. We believe that the use of a neural evolution reinforcement model such as NEAT [77], could help overcome existing limitations.

6 Future Work

For the input generation problem alone, there are still many open problems. Automatically identifying complex methods input and finding an expensive combination of these inputs are the two major open problems.

Coppa et al. [20], as described before, showed a technique of identifying method inputs by recording the read operation to memory for the given method. Grouping the set of inputs into a meaningful way to developers before using them is an issue. Another is to reduce the excessive overhead of identifying such inputs, as identifying the input is only a first step in any input generation methodology.

Given Coppa et al. [20] technique, we would like to examine its applicability to work with fuzzing techniques [67, 48]. Inputs from the Coppa et al. [20], if identified successfully would increase the dimensionality and nature of the inputs beyond existing evaluations of the fuzzing techniques [67, 48]. Hence, it could highlight easy fixes to the existing fuzzing techniques or shows that fuzzing is limited to such a problem.

Furthermore, using the same inputs, we would like to examine the different in efficiency between the available fuzzing techniques [67, 48] and an applicable reinforcement model such as NEAT [77]. We would like to look into different forms of efficiency. For example, the understandability between the two, time of search and expressiveness of the performance issue.

Finally, low-prioritized, we also would like to further explore the idea of composing a data-set of source code level performance fixes. Using regression testing techniques such as *PerImpact* [52], we would like to identify source code fragments that are proven to be fixing performance issues. Using the two versions, supervised machine learning algorithms can be trained to suggest changes at a source code level to avoid common performance issues. If possible, this solution would have a high understandability rate as well as avoid the need to generate inputs of common performance issues.

References

[1] Glenn Ammons, Thomas Ball, and James R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. *ACM SIGPLAN Notices*, 32(5):85–96, May 1997.

[2] Mona Attariyan, Michael Chow, and Jason Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 307–320, Berkeley, CA, USA, 2012. USENIX Association.

[3] Vanessa Ayala-Rivera, Maciej Kaczmarski, John Murphy, Amarendra Darisa, and A. Omar Portillo-Dominguez. One size does not fit all. *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering - ICPE '18*, 2018.

[4] Thomas Ball and James R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1319–1360, Jul 1994.

[5] Thomas Ball and James R. Larus. Efficient path profiling. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 29, pages 46–57, Washington, DC, USA, 1996. IEEE Computer Society.

[6] Thomas Ball, Peter Mataga, and Mooly Sagiv. Edge profiling versus path profiling. *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '98*, 1998.

[7] Simonetta Balsamo, Antinisca Di Marco, Paola Inverardi, and Marta Simeoni. Model-based performance prediction in software development: A survey. *IEEE Trans. Softw. Eng.*, 30(5):295–310, may 2004.

[8] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley Professional, 3rd edition, 2012.

[9] Jon Louis Bentley. *Writing Efficient Programs*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1982.

[10] Alexandre Bergel, Oscar Nierstrasz, Lukas Renggli, and Jorge Ressia. Domain-specific profiling. In Judith Bishop and Antonio Vallecillo, editors, *Objects, Models, Components, Patterns*, pages 68–82, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[11] Stephen M. Blackburn, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot, B. Moss, Aashish Phansalkar, Darko Stefanović, and et al. The dacapo benchmarks: java benchmarking development and analysis. *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications - OOPSLA '06*, 2006.

[12] D. Boehme, T. Gamblin, D. Beckingsale, P. Bremer, A. Gimenez, M. LeGendre, O. Pearce, and M. Schulz. Caliper: Performance introspection for hpc software stacks. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 550–560, 2016.

[13] Marco Brocanelli and Xiaorui Wang. Hang doctor: runtime detection and diagnosis of soft hangs for smartphone apps. *Proceedings of the Thirteenth EuroSys Conference on - EuroSys '18*, 2018.

[14] Marc Brünink and David S. Rosenblum. Mining performance specifications. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 39–49, New York, NY, USA, 2016. ACM.

[15] Jacob Burnim, Sudeep Juvekar, and Koushik Sen. Wise: Automated test generation for worst-case complexity. *2009 IEEE 31st International Conference on Software Engineering*, 2009.

[16] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[17] Sudipta Chattopadhyay, Lee Kee Chong, and Abhik Roychoudhury. Program performance spectrum. *Proceedings of the 14th ACM SIGPLAN/SIGBED conference on Languages, compilers and tools for embedded systems - LCTES '13*, 2013.

[18] Bihuan Chen, Yang Liu, and Wei Le. Generating performance distributions via probabilistic symbolic execution. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 49–60, New York, NY, USA, 2016. ACM.

[19] Zhifei Chen, Bihuan Chen, Lu Xiao, Xiao Wang, Lin Chen, Yang Liu, and Baowen Xu. Speedoo: prioritizing performance optimization opportunities. *Proceedings of the 40th International Conference on Software Engineering - ICSE '18*, 2018.

[20] Emilio Coppa, Camil Demetrescu, and Irene Finocchi. Input-sensitive profiling. In *Proceedings of the 33rd ACM SIGPLAN Conference on*

Programming Language Design and Implementation, PLDI '12, pages 89–98, New York, NY, USA, 2012. ACM.

[21] Charlie Curtsinger and Emery D. Berger. Coz: Finding code that counts with causal profiling. *Proceedings of the 25th Symposium on Operating Systems Principles - SOSP '15*, 2015.

[22] Daniele Cono D'Elia and Camil Demetrescu. Ball-larus path profiling across multiple loop iterations. *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications - OOPSLA '13*, 2013.

[23] Luca Della Toffola, Michael Pradel, and Thomas R. Gross. Performance problems you can fix: a dynamic analysis of memoization opportunities. *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications - OOPSLA 2015*, 2015.

[24] Monika Dhok and Murali Krishna Ramanathan. Directed test generation to detect loop inefficiencies. *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016*, 2016.

[25] Evelyn Duesterwald and Vasanth Bala. Software profiling for hot path prediction: Less is more. *SIGARCH Comput. Archit. News*, 28(5):202–211, nov 2000.

[26] Robert F. Dugan. Performance lies my professor told me. *Proceedings of the fourth international workshop on Software and performance - WOSP '04*, 2004.

[27] Simon F. Goldsmith, Alex S. Aiken, and Daniel S. Wilkerson. Measuring empirical computational complexity. *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering - ESEC-FSE '07*, 2007.

[28] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. Gprof: A call graph execution profiler. *SIGPLAN Not.*, 17(6):120–126, jun 1982.

[29] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. gprof: a call graph execution profiler. *ACM SIGPLAN Notices*, 39(4):49, Apr 2004.

[30] Scott Grant, James R. Cordy, and David Skillicorn. Automated concept location using independent component analysis. *2008 15th Working Conference on Reverse Engineering*, Oct 2008.

[31] Mark Grechanik, Chen Fu, and Qing Xie. Automatically finding performance problems with feedback-directed learning software testing. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 156–166, Piscataway, NJ, USA, 2012. IEEE Press.

[32] Philip J. Guo and Dawson Engler. Using automatic persistent memoization to facilitate data analysis scripting. *Proceedings of the 2011 International Symposium on Software Testing and Analysis - ISSTA '11*, 2011.

[33] Shi Han, Yingnong Dang, Song Ge, Dongmei Zhang, and Tao Xie. Performance debugging in the large via mining millions of stack traces. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 145–155, Piscataway, NJ, USA, 2012. IEEE Press.

[34] Xue Han and Tingting Yu. An empirical study on performance bugs for highly configurable software systems. *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement - ESEM '16*, 2016.

[35] Herodotos Herodotou and Shivnath Babu. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. In *Proceedings of the VLDB Endowment, 2011*, volume 4, pages 1111–1122. VLDB, 2011.

[36] Torsten Hoefler, William Gropp, William Kramer, and Marc Snir. Performance modeling for systematic performance tuning. In *State of the Practice Reports*, SC '11, pages 6:1–6:12, New York, NY, USA, 2011. ACM.

[37] Alejandro Infante. Identifying caching opportunities, effortlessly. *Companion Proceedings of the 36th International Conference on Software Engineering - ICSE Companion 2014*, 2014.

[38] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and detecting real-world performance bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 77–88, New York, NY, USA, 2012. ACM.

[39] Milan Jovic, Andrea Adamoli, and Matthias Hauswirth. Catch me if you can: performance bug detection in the wild. *ACM SIGPLAN Notices*, 46(10):155, Oct 2011.

[40] Yu Kang, Yangfan Zhou, Hui Xu, and Michael R. Lyu. Persisdroid: Android performance diagnosis via anatomizing asynchronous executions. *CoRR*, 12 2015.

[41] Yu Kang, Yangfan Zhou, Hui Xu, and Michael R. Lyu. Diagdroid: Android performance diagnosis via anatomizing asynchronous executions. *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016*, 2016.

[42] Chung Hwan Kim, Junghwan Rhee, Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. Perfguard: binary-centric application performance monitoring in production environments. *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016*, 2016.

[43] B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, 1990.

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

[45] Tilman Küstner, Josef Weidendorfer, and Tobias Weinzierl. Argument controlled profiling. In *Proceedings of the 2009 International Conference on Parallel Processing*, Euro-Par'09, pages 177–184, Berlin, Heidelberg, 2010. Springer-Verlag.

[46] Matthew Larsen, Cyrus Harrison, James Kress, David Pugmire, Jeremy S. Meredith, and Hank Childs. Performance modeling of in situ rendering. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '16, pages 24:1–24:12, Piscataway, NJ, USA, 2016. IEEE Press.

[47] James R. Larus. Whole program paths. *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation - PLDI '99*, 1999.

[48] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. Perffuzz: automatically generating pathological inputs. *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis - ISSTA 2018*, 2018.

[49] Yepang Liu, Chang Xu, and Shing-Chi Cheung. Characterizing and detecting performance bugs for smartphone applications. *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*, 2014.

[50] Qi Luo. Automatic performance testing using input-sensitive profiling. *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016*, 2016.

[51] Qi Luo. Input-sensitive performance testing. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 1085–1087, New York, NY, USA, 2016. ACM.

[52] Qi Luo, Denys Poshyvanyk, and Mark Grechanik. Mining performance regression inducing code changes in evolving software. *Proceedings of the 13th International Workshop on Mining Software Repositories - MSR '16*, 2016.

[53] Qi Luo, Denys Poshyvanyk, Aswathy Nair, and Mark Grechanik. Forepost: a tool for detecting performance problems with feedback-driven learning software testing. *Proceedings of the 38th International Conference on Software Engineering Companion - ICSE '16*, 2016.

[54] Ian Molyneaux. *The Art of Application Performance Testing: Help for Programmers and Quality Assurance*. O'Reilly Media, Inc., 1st edition, 2009.

[55] Rashmi Mudduluru and Murali Krishna Ramanathan. Efficient flow profiling for detecting performance bugs. *Proceedings of the 25th International Symposium on Software Testing and Analysis - ISSTA 2016*, 2016.

[56] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Evaluating the accuracy of java profilers. *ACM SIGPLAN Notices*, 45(6):187, May 2010.

[57] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation - PLDI '07*, 2007.

[58] Khanh Nguyen and Guoqing Xu. Cachetor: detecting cacheable data to remove bloat. *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2013*, 2013.

[59] A. Nistor, T. Jiang, and L. Tan. Discovering, reporting, and fixing performance bugs. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 237–246, 2013.

[60] Adrian Nistor, Po-Chun Chang, Cosmin Radoi, and Shan Lu. Caramel: Detecting and fixing performance problems that have non-intrusive fixes. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 902–912, Piscataway, NJ, USA, 2015. IEEE Press.

[61] Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. Toddler: Detecting performance problems via similar memory-access patterns. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 562–571, Piscataway, NJ, USA, 2013. IEEE Press.

[62] G. Ofenbeck, R. Steinmann, V. Caparros, D. G. Spampinato, and M. Püschel. Applying the roofline model. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 76–85, 2014.

[63] Tony Ohmann, Michael Herzberg, Sebastian Fiss, Armand Halbert, Marc Palyart, Ivan Beschastnikh, and Yuriy Brun. Behavioral resource-aware model inference. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 19–30, New York, NY, USA, 2014. ACM.

[64] Oracle. Java development kit se 6 documentation - class array. `https://docs.oracle.com/javase/6/docs/api/java/util/Arrays.html`.

[65] Carlos Pacheco and Michael D. Ernst. Randoop: feedback-directed random testing for java. *Companion to the 22nd ACM SIGPLAN conference on Object oriented programming systems and applications companion - OOPSLA '07*, 2007.

[66] pcre.org. Pcre - perl compatible regular expressions. `http://pcre.org/`.

[67] Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. Slowfuzz. *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security - CCS '17*, 2017.

[68] Corina S. Păsăreanu, Willem Visser, David Bushnell, Jaco Geldenhuys, Peter Mehlitz, and Neha Rungta. Symbolic pathfinder: integrating symbolic execution with model checking for java bytecode analysis. *Automated Software Engineering*, 20(3):391–425, Feb 2013.

[69] Ana B. Sánchez, Pedro Delgado-Pérez, Inmaculada Medina-Bulo, and Sergio Segura. Search-based mutation testing to improve performance tests. *Proceedings of the Genetic and Evolutionary Computation Conference Companion on - GECCO '18*, 2018.

[70] Juan Pablo Sandoval Alcocer, Alexandre Bergel, and Marco Tulio Valente. Learning from source code history to identify performance failures. *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering - ICPE '16*, 2016.

[71] Marija Selakovic, Thomas Glaser, and Michael Pradel. An actionable performance profiler for optimizing the order of evaluations. *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis - ISSTA 2017*, 2017.

[72] Du Shen, Qi Luo, Denys Poshyvanyk, and Mark Grechanik. Automating performance bottleneck detection using search-based application profiling. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 270–281, New York, NY, USA, 2015. ACM.

[73] Sameer S. Shende and Allen D. Malony. The tau parallel performance system. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, may 2006.

[74] Norbert Siegmund, Sergiy S. Kolesnikov, Christian Kästner, Sven Apel, Don Batory, Marko Rosenmüller, and Gunter Saake. Predicting performance via automated feature-interaction detection. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 167–177, Piscataway, NJ, USA, 2012. IEEE Press.

[75] Connie U. Smith. Software performance engineering. *Lecture Notes in Computer Science*, pages 509–536, 2005.

[76] Linhai Song and Shan Lu. Performance diagnosis for inefficient loops. *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, May 2017.

[77] Kenneth O. Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127, Jun 2002.

[78] William N. Sumner, Yunhui Zheng, Dasarath Weeratunge, and Xiangyu Zhang. Precise calling context encoding. *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10*, 2010.

[79] Mustafa M. Tikir and Jeffrey K. Hollingsworth. Efficient instrumentation for code coverage testing. *Proceedings of the international symposium on Software testing and analysis - ISSTA '02*, 2002.

[80] Willem Visser, Corina S. Păsăreanu, and Sarfraz Khurshid. Test input generation with java pathfinder. *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis - ISSTA '04*, 2004.

[81] Alexander Wert, Jens Happe, and Lucia Happe. Supporting swift reaction: Automatically uncovering performance problems by systematic experiments. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 552–561, Piscataway, NJ, USA, 2013. IEEE Press.

[82] Xusheng Xiao, Shi Han, Dongmei Zhang, and Tao Xie. Context-sensitive delta inference for identifying workload-dependent performance bottlenecks. *Proceedings of the 2013 International Symposium on Software Testing and Analysis - ISSTA 2013*, 2013.

[83] Guoqing Xu. Finding reusable data structures. *Proceedings of the ACM international conference on Object oriented programming systems languages and applications - OOPSLA '12*, 2012.

[84] Shahed Zaman, Bram Adams, and Ahmed E. Hassan. A qualitative study on performance bugs. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, MSR '12, pages 199–208, Piscataway, NJ, USA, 2012. IEEE Press.

[85] Dmitrijs Zaparanuks and Matthias Hauswirth. Algorithmic profiling. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 67–76, New York, NY, USA, 2012. ACM.

[86] Pingyu Zhang, Sebastian Elbaum, and Matthew B. Dwyer. Automatic generation of load tests. *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, Nov 2011.

[87] Sai Zhang and Michael D. Ernst. Which configuration option should i change? In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 152–163, New York, NY, USA, 2014. ACM.

[88] Xiaotong Zhuang, Mauricio J. Serrano, Harold W. Cain, and Jong-Deok Choi. Accurate, efficient, and adaptive calling context profiling. *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation - PLDI '06*, 2006.