Inferring Performance from Code: A Review

Emilio Incerto(✉), Annalisa Napolitano(✉), and Mirco Tribastone(✉)

IMT School for Advanced Studies, 55100 Lucca, Italy
{emilio.incerto,annalisa.napolitano,mirco.tribastone}@imtlucca.it

Abstract. Performance is an important non-functional property of software that has a direct impact on the end-user's perception of quality of service since it is related to metrics such as response time, throughput, and utilization. *Performance-by-construction* can be defined as a development paradigm where executable code carries some kind of guarantee on its performance, as opposed to the current practice in software engineering where performance concerns are left to the later stages of the development process by means of profiling or testing. In this paper we argue that performance-by-construction techniques need to be probabilistic in nature, leveraging accurate models for the analysis. In support of this idea, here we carry out a literature review on methods that can be used as the basis of performance-by-construction development approaches. There has been significant research—reviewed elsewhere—on performance models derived from high-level software specifications such as UML diagrams or other domain-specific languages. This review, instead, focuses on methods where performance information is extracted directly from the code, a line of research that has arguably been less explored by the software performance engineering community.

1 Introduction

Non-functional (also called *extra-functional*) properties of software are related to issues concerning *how* a system works, as opposed to functional properties which establish what it does. Among many relevant such properties including security, dependability, and reliability is software performance. Briefly, it can be understood as a property analyzable through a number of quantitative metrics related to *how fast* the software system can yield the desired output. Typical performance metrics of interest are *response time*, i.e., how long it takes to obtain a reply since a request has been issued; *throughput*, i.e., how many requests can be

This work has been partially supported by the Italian Ministry for Education under grant SEDUCE no. 2017TWRCNB.

Electronic supplementary material The online version of this chapter (https://doi.org/10.1007/978-3-030-61362-4_17) contains supplementary material, which is available to authorized users.

T. Margaria and B. Steffen (Eds.): ISoLA 2020, LNCS 12476, pp. 307–322, 2020.
https://doi.org/10.1007/978-3-030-61362-4_17

served per unit time; and *utilization*, i.e., the percentage of time that a software resource is busy servicing some request.

Performance metrics can be defined mathematically (e.g., [11]), in which case it is possible to easily see how they can be formally related to each other. They can also be related to other metrics such as energy consumption (via appropriate models [41]) and availability (e.g., excessively long response times of a web application causing server crashes).

Performance is a key property that directly affects the end-user's perception of quality of a software system. It is such an important aspect that, as claimed by Harman and O'Hearn [29], "in many practical deployment scenarios, particularly mobile, **performance is the new correctness**." Despite its relevance, however, the practice of software engineering does not seem to make use of principled criteria to reason about performance. For example, the Android developers' guide suggests a rule of thumb for improving the performance of an app by means of multithreading [1]:

> *"You can use trial-and-error to discover the minimum number of threads you can use without running into problems."*

Such state of affairs is unsatisfactory for at least two reasons. First, conducting performance analysis only through testing or runtime profiling raises several issues about the cost and the degree of coverage of the experiments. Indeed, it implies a software development process where performance issues are left to the latest stages, which may make serious flaws too expensive to fix, such as in the notable case of halting a NASA space mission due to on-board software producing unacceptably large response times [2]. The second limitation is that testing approaches can detect the presence of performance issues, but they do not carry explanatory and generalization power on their own. Specifically, they do not provide a *model* of the software system under investigation that can be used for the analysis of further *what-if* scenarios or for formal verification.

A paradigm based on *performance-by-construction* principles aims instead at the development of software with guarantees of achieving given performance objectives [57]. In order to achieve this, it appears inescapable that this paradigm leverage appropriate models of software systems that can yield (accurate) performance predictions. Traditionally, performance models of computer and communication systems are probabilistic [11,56]. Essentially, this can be motivated by two orthogonal modeling choices to capture *external* and *internal* uncertainty, respectively [45]. With the former we refer to the typical use of stochastic processes to model the workload of software system (i.e., the pattern of arrivals of requests) as well as to abstract from the details of other environmental features such as the hardware on which the software system runs. With the latter we refer to the explicit use of programming primitives that generate samples from given probability distributions, as in the context of probabilistic programming [27].

A model capable of predicting performance properties is necessarily a distinct artifact than the software system under consideration. Of course, it could be built by hand by the software architect/engineer. However this would not fit with the need of providing automated support within a development process. Moreover,

in general it is likely difficult to find engineers who have competences both in the problem domain and in the performance modeling techniques—recognized as a main obstacle to model-based performance analysis in software engineering [63].

More automated support to model building can be offered by model-driven development techniques where the performance model is algorithmically derived from software specifications such as behavioral UML diagrams (e.g., [58,59,64]) or domain-specific languages (e.g., [10]) annotated with quantitative information. Some of the vast literature on this topic has already been reviewed [7,35].

Model-driven approaches may not always be applicable, for instance when the code that is automatically generated from the higher-level specification is likely to undergo manual modifications. Indeed, after these, the related performance model may not be a faithful representation of the actual system under consideration any longer [23]. In order to avoid this problem, another approach might be to use the code as the model of the software system itself, thus inferring performance models directly from the code. Of course, this rules out the possibility of conducting performance analysis at the very early stages of the software development. However, it fits well with agile processes based on successive iterations, where changes in the codebase can be reflected onto changes in the associated performance model.

In this paper we present a literature review on the state of the art of techniques which produce performance predictions for code analysis in order to evaluate their feasibility as tools to be used within a performance-by-construction development framework. The literature analyzed, consisting of 24 research papers published with the period 1982–2019, is mostly located in the sub-fields of computer science regarding programming languages and software engineering. In addition to a brief description of each method, we provide a classification in terms of the assumptions on the input program, the type of technique employed (i.e., whether it uses static or dynamic analysis), and the output provided (i.e., if it yields a model or directly a performance prediction). We conclude the paper with a discussion of the main limitations of the state of the art for their use in realistic development processes based on performance-by-construction.

Search and Selection. The research was conducted by selecting from a search engine a set of representative and highly cited papers from the literature, which deal with the issue of extracting performance from the code, specifically [13,18,21,24,25,38,51,65]. We then evaluated the ongoing and outgoing citation links of these papers and the most interesting, as well as the most cited, were selected for analysis in this semi-systematic review.

2 Analysis Dimensions

In this section, we analyze the several aspects that distinguish the analyzed methods of performance generation from code. They concern learning techniques, exploration techniques, the type of the output model or performance metrics, and the scalability level.

2.1 Learning Techniques

Although they differ greatly from each other, all performance learning techniques from software can be condensed into two categories:

- **Static analysis:** the source-code of the program is systematically inspected to infer performance. Often, an intermediate model is created, e.g. differential equations [61], Markov processes [47], or a step-counting function [50] which simplifies the software by focusing only on performance evaluation.
- **Dynamic analysis:** an instrumented version of the program is executed, and by analyzing traces, the necessary information to build the performance model is gathered (e.g., which parts of the code have been actually explored, the number of calls issued to a particular functions). While some approaches are based on one single run to inspect one specific profile [28], others perform several runs with different workloads to obtain metrics, models and trend functions of different profiles such as worst-, average-, best-case scenarios [65]. This could be costly if the program needs to be executed numerous times with different input sizes. There are several techniques to select the workloads (e.g., load-testing, probabilistic symbolic execution, random sampling), which we will analyze later in Subsect. 2.2.

Notice that often these techniques are combined together for the definition of hybrid approaches. For example, static analysis is used to create an instrumented version of the program that is then executed with dynamic analysis [21].

2.2 Exploration Techniques

In this subsection, we will describe the different techniques that can be used both for exploring the program's paths in the static analysis and to generate the workload input sequences that guide the dynamic analysis.

- **Runtime monitoring** implies analyzing the logs or execution traces of the real (instrumented) system. Approaches that exploit runtime monitoring care about instrumenting the program as efficiently as possible, so that to leave the system performance unchanged [6].
  This kind of exploration technique does not make any assumption on the input features and thus the resulting performance models show the typical system behavior and not a peculiar case.
- **Load testing** is an input generation technique that tries to stress the software by evaluating it with a workload of increasing size [25]. This may imply evaluating a particular scenario for a given size e.g., worst and best cases, trying to find out the right workload based on user-specified features, heuristics on the complexity of the data structures, or observations.
- **Random sampling** implies testing the program under an input randomly distributed according to some probability distribution [55]. Random sampling is efficient and easy to implement; in addition, sometimes it might be the

only viable option when the program is too complex or some source-code portions are unknown [19]. However, the main limitation is that without any heuristic it could be extremely unlikely to observe interesting but rare system behaviors [14].

– **Symbolic execution** exhaustively explores the execution tree of a program using symbolic values for the input instead of concrete ones [5,33]. Each execution edge could be described by a condition formula on the input variables. A path is described with the conjunction of all conditional formulas of its edges, called the *path condition*. The execution tree can be explored with any algorithms for traversing trees, such as breadth-first search. The search is done by trying to symbolically satisfy the (partial) path condition: if the set of solutions is not empty the search continues, by evaluating also child edges conditions; otherwise, the path condition is impossible to satisfy and thus that branch of the tree is marked as unreachable. Probabilistic symbolic execution [24] arises when symbolic execution is combined with model counting [26] in order to obtain not only reachability/unreachability information but also path probabilities, by comparing the number of path solutions, i.e., the cardinality of the path condition admissibility set, with the cardinality of the input set [18]. While in random testing the input distribution can be arbitrarily chosen, probabilistic symbolic execution works only for uniformly distributed input.

2.3 Output Model

The output models of the surveyed methods differ in many aspects, such as the amount of information they encode, the predictive power or the efficiency of the analysis techniques. They can be categorized as follows.

– **Enriched call graphs and control-flow graphs.** Mostly path or edges probabilities obtained through the code analysis are stored in a compact form in (enriched) control flow graphs or call graphs [6]. Since the total amount of program's paths is often exponential with respect to the number of visited branches, these techniques typically limit the exploration of the *hottest* ones, i.e., those that have the greatest impact on performance.
– **Performance metrics.** Often profiling approaches deal with discovering some static or dynamic performance metrics, e.g., number of procedure calls [20] and average runtimes [9]. The information level of this kind of model is low since it has no predictive power and it gives no indication as to the reason why the program execution shows those performance metrics.
– **Bottlenecks detection** provides insight on the worst case of the program execution, which can be given in terms of *hot paths* detection [16], or input values that trigger performance bottlenecks [3]. We consider this model to have a low information level since the worst-case scenario does not capture exhaustively the whole program's behavior.
– **Cost functions.** All the approaches that provide some kind of cost-function in terms of the size of input belong to this category. This function could

represent the average-case [65] as the asymptotic one [25]. Cost functions provide insight on how the program behaves as the input grows and thus they are considered medium-level informative. For instance, these techniques do not allow to select the best alternative of an algorithm among a set of functionally and asymptotically equivalent ones.

- **Markov processes** [47]**.** Markov processes are a fundamental model for software systems [11]. To build a Markov model that is compact and has an analytical solution in a closed form (i.e., a Markov chain) it is necessary that the analyzed program is memoryless. This implies that the probabilities of the edges are all mutually independent.
- **Target events probabilities** [38,51]**.** These approaches aim to evaluate the probability that certain target events happen. Even if these techniques are typical of bug finding and do not give directly a performance measure, they can provide insight on performance, since the target events could be previously selected as costly functions or inefficient blocks of code.

3 Model Construction Methods

In this section, we briefly describe all the methods that infer performance from code, presented in chronological order.

Gprof [28] periodically samples the program counter in a single program run with a certain workload, and counts the number of calls and execution times of each procedure. A post-processing step then propagates the sampled values to the program call-graph to estimate the total running time in each procedure.

Sarkar et al. [52] propose a framework for obtaining the mean and variance of the execution times for program's procedures. These values are obtained by a counter-based execution profile of the program and then inserted in the program's extended control flow graph. The proposed solution assumes that the average execution time of a procedure call is independent of the call site and thus the observed time value is multiplied by the frequency of that procedure call, without any concern about the program history and data flow.

Ramalingam et al. [47] study the problem of determining *how often*, i.e., with *what probability*, a fact holds true during program execution. The input is the program control flow graph whose edges are labeled with a probability. The program is simply modeled as a first-order Markov chain, by assuming the probability of the program execution following a particular branch is independent of the execution history, which does not hold in general for real programs.

Ball et al. [6] focus on path profiling, i.e., computing paths' frequencies and performance metrics. They claim that since edges probabilities are not independent, it is impossible to obtain paths frequencies by simply combining edges frequencies. In many cases the next visited program instructions are dependent on the execution history, thus making path profiling essential for finding accurate performance models of programs. Unfortunately while edge profiling is linear respect to the program size, path profiling is exponential. In order to tame such issue, the authors provide a solution for runtime estimation of intra-procedural

path frequencies of an acyclic version of the program, by minimizing the overhead of the instrumentation. To further mitigate the scalability issues they only consider dynamic paths, i.e., those that have been actually executed during the program runtime monitoring.

Whole Program Paths (WPP) [36] is an approach to learn and represent the program's dynamic control flow, i.e., the set of executed paths. Differently from previous approaches, it considers loop iterations as well as interprocedural paths. The work shows also how to compute *hot subpaths*. The instrumentation and path discovery phase is done relying on the published work [6]; the novelty of WPP is the compression algorithm, which, by finding regularities (i.e., repeated code), transforms the traces more compactly into the directed acyclic graph.

JinsightEx [54] samples performance metrics (i.e., execution time, memory and other resource usage) of a Java program's *execution slices*, which are user-defined through dynamic or static criteria. The slices represent the primary view of the performance models and they can be grouped in workloads to facilitate larger analysis procedures. JinsightEx allows the user to browse this data to evaluate the number of called objects, allocations, method calls; and to find performance and memory problems in many industrial applications.

Magpie is an online performance modeling service that collects detailed end-to-end traces from users on the running system and constructs probabilistic models of its behavior [8,9]. It instruments the system using black-box approaches such as kernel-level tracing for Windows [53] or WinPcap packet capture library [48]. *Magpie* constructs a model of the observed behavior, by clustering requests features and performance. Using these behavioral clusters it is possible to detect anomalous requests and system malfunctions.

Ammons et al. [4] find bottlenecks, given some kind of profiles (e.g. call tree) of the system execution. There are two algorithms: one that finds expensive paths of a program and another that computes how the path cost differs from similar execution runs. They build a summarized model of the program that is based on heuristics, by collecting cost metrics of execution paths, and they provide an interface for querying this model and comparing paths cost metrics to find the worst-case. The approach is evaluated on a real-world case study.

Trend-prof [25] derives the asymptotic behavior of a program by computing its empirical computational complexity. This is done by executing the program on workloads of different sizes and user-specified numerical features, for example the number of bytes of the input file. Measured execution times of program blocks are fit against linear or power law models. *Trend-prof* is evaluated on several large programs; the authors report cases in which the program meets its expected bounds, performs better than its worst-case, or shows performance bugs.

Buse et al. [16] provide a descriptive statistical model of paths frequencies that is obtained by static analysis of the source code with *path enumeration*. The approach is validated on several benchmarks. The qualitative analysis provides insights on which source code features characterize hot paths. Since the number of program paths could be exponential, only interprocedural paths within one single class are considered; calls across class boundaries are ignored. The idea

that underlines the approach is that the most likely hot paths are those that have little impacts on the program state, intended in terms of changes of global variables and stack. With this idea in mind, any machine learning algorithm could be trained to select the source-level features that identify hot paths; in the paper, Weka [31] is used. This approach suffers from overfitting, and in case the behavior of the program is not fully captured by a single class, it may reveal unuseful.

Zaparanuks et al. [65] exploit heuristics to determine a program's approximated cost functions from traces of representative program executions. This approach automatically determines the input size, measures the program's cost for each input, and fits a cost function. Several cost measures are supported such as algorithmic steps, number and size of reads/writes on data structures and the number of objects creations. The program input type (e.g., recursive data structures, arrays, and so on) and the input size are obtained by computing the number of elements of the structure or its memory occupation. A limitation of this approach is that it cannot infer the input size of programs that do not work with data structures but on primitive types, and that since it is based on heuristics, it returns an approximate cost-function rather than an exact one.

Geldenhuys et al. [24] propose an extension of Java Symbolic PathFinder [44] that estimates probabilities of each particular program locations using probabilistic symbolic execution. Although the cost of symbolic execution is mitigated by implementing some heuristics, the scalability of this technique is still a concerning issue. In the paper, the authors present case studies involving 4 and 5 operations on a data structure, i.e., insertions and deletions from BinomialHeap, TreeMap, and BinaryTree. They claim the infeasibility of the analysis for programs having a sequence of 14 operations.

Coppa et al. [21] present a profiling methodology to discover hidden asymptotic inefficiencies from program traces. Grow rates of routines as a function of the input size is dynamically measured with a metric—the read memory size (RMS)—that counts the accessed numbers of memory cells. Thee supporting tool, named *aprof*, builds upon Valgrind [42] for the instrumentation. It determines the RMS and the minimum and maximum cost of executing routines and exploits curve fitting and curve bounding to obtain the functions that best describe their asymptotic behavior.

Sankaranarayanan et al. [51] statically analyze probabilistic programs, characterized by variables that assume uncertain values during execution, by assigning them probability distributions. They provide bounds on the probability that a certain event happens and claim that to determine those bounds only an adequate subset of program's execution paths is needed. The initial set of paths is obtained using random simulations and statistical tests, while probability bounds are obtained using symbolic execution, a heuristic they implement for the problem of computing the volume of an n-dimensional convex polyhedron, namely *probabilistic volume bound computation* and Monte Carlo sampling.

Like the previous work, Luckow et al. [38] consider the probability of a target event in case of nondeterministic programs, e.g., multithreaded or distributed programs. They firstly implement a symbolic tree scheduler to handle uncertainties using Markov decision processes [46] (exact algorithm) or Monte Carlo sampling (approximated algorithm) on the symbolic tree generated with a bounded symbolic execution of the program. Then they exploit reinforcement learning [32] to iteratively improve the tree of the approximate algorithm. Finally, model counting techniques and some heuristics are used to compute branch probabilities until reaching the target event.

Filieri et al. [22] propose a method for computing the probability of a target event for a program. The method is based on Monte Carlo sampling to improve Bayesian estimates of the sought probability. To speed up convergency they propose the *informed sampling* technique, with which paths with high statistical significance are explored first.

Borges et al. [12] describe a methodology for the automatic estimation of the probability of a target event given an input profile described via continuous probability distribution over the floating-point domain. The method supports three strategies, based on gradient descent optimization [43] and on heuristics, to improve the learning phase (hence, the scalability of the approach) that are based on ranking the edge condition constraints of the symbolic execution according to their impact on the convergence of the statistical analysis and counting.

Brünink et al. [13] present an approach to infer the performance specification of a running system by creating runtime models and subsequently producing performance assertions. These models are graphs that describe the expected behavior of the system in its hot functions, tracing probabilities in a context-sensitive or insensitive way, as needed. The context information is inserted when the performance metrics (i.e., the runtimes) of the procedures, evaluated for different contexts, belong to different clusters of values. Although they do not exploit analytical rigorous models they succeed to obtain accurate performance.

PerfPlotter [18] is a framework for performance analysis of a program that takes as input the source code and a usage profile and generates a probability density cost function. *PerfPlotter* extends Java Symbolic PathFinder [44] using probabilistic symbolic execution to detect paths with low and high probabilities under the given usage profile, and the resulting set of paths are executed to measure the effective runtime (precisely the subset chosen is that of the paths with high or low probability whose termination within a certain number of steps has been established). Finally, these results are combined and weighted with paths' probabilities to obtain the probability density function (PDF). This approach can infer the PDF, still having a scalability limitation due to the usage of probabilistic symbolic execution.

Luckow et al. [37] propose a technique based on *guided* symbolic execution to generate the worst-case complexity function of the input size. First, symbolic execution is run with a small value of input size, which is subsequently increased. The symbolic execution is guided by selecting only the paths that account for the worst cases. To be more accurate, during path selection the history of choices

is taken into account when deciding which branch to execute next. Thus, the method produces a context-sensitive model of worst-case paths that are analyzed to fit the cost function using some resource consumption metrics (e.g., execution time or memory usage).

Wang et al. [60] present an approach to analyze the performance of applications deployed on Cloud. The approach first tests the Cloud infrastructure with typical micro-benchmarks and evaluates the performance distribution of each resource, e.g., memory and CPU. Then it tests the user-defined application with a given input that characterizes the program's typical workload, resulting in the *resource usage profile* of the target application. Finally, it conducts the same tests on the application deployed in the cloud producing the *baseline performance*. By combining these models the approach provides statistics that allow the developer to understand which kind of performance specification the application meets.

Speedoo [20] is an approach to identify groups of methods that are crucial to the program's performance and whose optimization would lead to the best speed-up possible. It suggests optimization opportunities for these methods based on performance (anti-)patterns detection, e.g., cyclic invocation, expensive recursion. *Speedoo* ranks the methods based on metrics of architectural importance (e.g. the size of the sub-calls tree) according to the Design Rule Hierarchy algorithm (DRH), defined by Cai et al. and Wong et al. [17,62], dynamic execution metrics (e.g., CPU time), and static complexity metrics (e.g., the number of loops).

PT4Cloud [30] is concerned about obtaining performance models of application developed on the cloud, addressing the issue of performance uncertainty due to IaaS resource managing. Their purpose is to find reliable stop conditions to test runs to cut down the cost of performance testing. They test the selected benchmarks with their pre-specified workloads and compute the performance distribution of the deployed application. By using a non-parametric statistical approach, they stop testing when they find that two subsequent distribution are statistically equivalent.

PerfXRL [3] presents an approach to find input values that trigger the performance bottlenecks of the system. Given an input space, possibly very large with multiple possible combinations, *PerfXRL* dynamically analyzes the system by executing it with a certain input and then guiding the analysis with the resulting cost reward value, using reinforcement learning.

4 Conclusion and Future Lines of Reseach

Performance is a crucial non-functional property that affects the user's perception of the software's quality. While it could be useful to know performance from early development stages, model-driven approaches may not always be applicable. When the code is continuously developed the real software source-code may differ considerably from model artifacts. In this scenario, performance models should be inferred directly from the deployed system. In this work we present a

Table 1. Summary of the analyzed methods

Method	Learning Techn.	Exploration Techn.	Output Model	Info. Level	Scalability
[28]	Dynamic analysis	Runtime monitoring	Enriched call-graph	Low	Medium
[52]	Static and dynamic analysis	Offline monitoring with given input	Performance metrics (sub-routines execution times and variance)	Low	Low
[47]	Static analysis	CFG sequential exploration	Markov Chain	Medium	Medium-low
[6]	Dynamic analysis	Runtime monitoring	Enriched CFG with acyclic intraprocedural path frequencies DAG (Directed Acyclic Graph)	Medium	Medium-high
[36]	Dynamic analysis	Uses [6]	Whole program paths and hot subpaths detection	Medium	Medium-high
[54]	Dynamic analysis	Realistic traces as input	Performance metrics organized in execution slices	Low	Medium
[9]	Dynamic analysis	Runtime monitoring	Performance metrics organized in clusters of request features	Low	High
[4]	Dynamic analysis	Profiles navigation searching the longest path	Bottlenecks detection	Medium-low	Medium-high
[25]	Dynamic analysis	Offline monitoring of chosen workloads described with numerical features	Computational complexity function of user-specified features	Medium	Medium
[16]	Static analysis	Loop bounded static path enumeration and counting with machine learning	Hot paths identification	Medium-low	Medium
[65]	Dynamic analysis	Realistic traces as input	Approximate descriptive cost function	Medium-low	Medium
[24]	Static analysis	Symbolic execution	Paths probabilities	Medium	Low
[21]	Dynamic analysis	Traces as input	Asymptotic cost function	Medium	Medium
[51]	Simulation + Static analysis	Random sampling with Monte Carlo + symbolic execution	Target events probabilities	Medium	Medium
[38]	Static analysis + simulation	Symbolic execution + Monte Carlo sampling and reinforcement learning	Target event probability	Low	Medium-high
[22]	Static analysis + simulation	Symbolic execution + Monte Carlo sampling and Hypothesis testing (i.e. *Importance Sampling*)	Target event probability	Low	Medium-high
[12]	Static analysis	Symbolic execution	Target event probability	Low	Medium
[13]	Dynamic analysis	Runtime monitoring	Performance metrics of hot functions (context sensitive profiling)	Medium-high	Medium
[18]	Static analysis	Symbolic execution	Probability density function of program runtime	High	Medium-low
[37]	Static analysis	Symbolic execution + policy guided exploration	Asymptotic cost-function	Medium	High
[60]	Dynamic analysis	Offline monitoring with typical benchmarks (Cloud) and typical input (stand-alone)	Cloud application performance statistics	Medium	High (Cloud)
[20]	Static and dynamic analysis	Design Rule Hierarchy algorithm + profiling tools	Optimization suggestions	Medium-low	High
[30]	Dynamic analysis	Testing with given inputs + non-parametric statistical approach for stop conditions	Cloud application performance distributions	High	High (Cloud)
[3]	Dynamic analysis	Reinforcement learning guided testing	Input values that trigger performance bottlenecks	Low	High

literature review of methods that produce performance information from code, trying to underline typical inefficiencies and future lines of research. Table 1 presents a summary of the evaluated methodologies and a comparison according to the proposed analysis dimensions.

Initially, the focus of the literature was on system profiling (mainly through dynamic analysis), using runtime monitoring [6,9,36] or offline monitoring starting from some realistic representative traces of program executions [25,34,54]. Recently, efforts have moved toward improving the applicability and scalability of symbolic execution (mainly with static analysis) [22,37,38]. Heuristics tried to speed up learning by approximating the paths probabilities [51] or by limiting the set of paths considered by the analysis to the most representative ones, i.e., worst-case, best-case, average-case [18].

In addition, most of the methodologies analyzed are able to learn low or medium *information content* models, such as performance metrics [13,54] or identification of hot paths [4,16]. The work presented by Ramalingam and Ganesan [47] is the only approach that extracts a model with a high predictive power like a Markov chain. Unfortunately, their model needs the memoryless assumption, i.e., the probability of the program execution following a particular branch is independent of the execution history, which obviously does not hold true in many cases. Also noteworthy are all the approaches that learn the probability density functions of the execution cost of the program [18,30], a compact but at the same time informative performance model, as it encapsulates the execution probabilities and the runtime.

Another interesting consideration, present in works of Brünink et al. and Luckow et al. [13,37], is to consider the impact of the context information on the probabilities of execution of the path, creating a context-aware model. It is evident, indeed, that the future behavior of the program is highly dependent on the state (i.e., the values of the variables) and therefore on past history. Explicitly considering this information in the performance model can provide a new and interesting view and allow the developer to better understand the reasons behind the performance behavior of a program. One could envisage the use of models with high predictive and implicitly context-sensitive content such as variable-length Markov chains [15,49], typically used for text analysis and pattern recognition. These techniques, never used for performance, have been used by Mazeroff et al. [39,40] to describe the behavior of the system in order to identify anomalies and malicious behaviors.

References

1. Android Developers' Guide: Threading performance. https://developer.android.com/topic/performance/threads.html. Accessed 23 July 2020
2. NASA delays satellite launch after finding bugs in software program. https://fcw.com/Articles/1998/04/19/NASA-delays-satellite-launch-after-finding-bugs-in-software-program.aspx. Accessed 4 Feb 2018
3. Ahmad, T., Ashraf, A., Truscan, D., Porres, I.: Exploratory performance testing using reinforcement learning. In: 2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), pp. 156–163. IEEE (2019)

4. Ammons, G., Choi, J.-D., Gupta, M., Swamy, N.: Finding and removing performance bottlenecks in large systems. In: Odersky, M. (ed.) ECOOP 2004. LNCS, vol. 3086, pp. 172–196. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24851-4_8
5. Baldoni, R., Coppa, E., D'elia, D.C., Demetrescu, C., Finocchi, I.: A survey of symbolic execution techniques. ACM Comput. Surv. (CSUR) **51**(3), 1–39 (2018)
6. Ball, T., Larus, J.R.: Efficient path profiling. In: Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 29, pp. 46–57. IEEE (1996)
7. Balsamo, S., Di Marco, A., Inverardi, P., Simeoni, M.: Model-based performance prediction in software development: a survey. IEEE Trans. Softw. Eng. **30**(5), 295–310 (2004)
8. Barham, P., Donnelly, A., Isaacs, R., Mortier, R.: Using magpie for request extraction and workload modelling. In: OSDI, vol. 4, p. 18 (2004)
9. Barham, P., Isaacs, R., Mortier, R., Narayanan, D.: Magpie: online modelling and performance-aware systems. In: HotOS, pp. 85–90 (2003)
10. Becker, S., Koziolek, H., Reussner, R.: Model-based performance prediction with the palladio component model. In: Proceedings of the 6th International Workshop on Software and Performance (WOSP), pp. 54–65 (2007)
11. Bolch, G., Greiner, S., De Meer, H., Trivedi, K.S.: Queueing Networks and Markov Chains: Modeling and Performance Evaluation with Computer Science Applications. Wiley, Hoboken (2006)
12. Borges, M., Filieri, A., d'Amorim, M., Păsăreanu, C.S.: Iterative distribution-aware sampling for probabilistic symbolic execution. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, pp. 866–877 (2015)
13. Brünink, M., Rosenblum, D.S.: Mining performance specifications. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 39–49 (2016)
14. Bucklew, J.: Introduction to Rare Event Simulation. Springer, New York (2013). https://doi.org/10.1007/978-1-4757-4078-3
15. Bühlmann, P., Wyner, A.J., et al.: Variable length Markov chains. Ann. Stat. **27**(2), 480–513 (1999)
16. Buse, R.P., Weimer, W.: The road not taken: estimating path execution frequency statically. In: 2009 IEEE 31st International Conference on Software Engineering, pp. 144–154. IEEE (2009)
17. Cai, Y., Sullivan, K.J.: Modularity analysis of logical design models. In: 21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006), pp. 91–102. IEEE (2006)
18. Chen, B., Liu, Y., Le, W.: Generating performance distributions via probabilistic symbolic execution. In: Proceedings of the 38th International Conference on Software Engineering, pp. 49–60 (2016)
19. Chen, T.Y., Kuo, F.C., Merkel, R.G., Tse, T.: Adaptive random testing: the art of test case diversity. J. Syst. Softw. **83**(1), 60–66 (2010)
20. Chen, Z., et al.: Speedoo: prioritizing performance optimization opportunities. In: Proceedings of the 40th International Conference on Software Engineering, pp. 811–821 (2018)
21. Coppa, E., Demetrescu, C., Finocchi, I.: Input-sensitive profiling. ACM SIGPLAN Not. **47**(6), 89–98 (2012)
22. Filieri, A., Păsăreanu, C.S., Visser, W., Geldenhuys, J.: Statistical symbolic execution with informed sampling. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 437–448 (2014)

23. Garcia, J., Krka, I., Mattmann, C., Medvidovic, N.: Obtaining ground-truth software architectures. In: Proceedings of the 35th International Conference on Software Engineering (ICSE), pp. 901–910 (2013)
24. Geldenhuys, J., Dwyer, M.B., Visser, W.: Probabilistic symbolic execution. In: Proceedings of the 2012 International Symposium on Software Testing and Analysis, pp. 166–176 (2012)
25. Goldsmith, S.F., Aiken, A.S., Wilkerson, D.S.: Measuring empirical computational complexity. In: Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 395–404 (2007)
26. Gomes, C.P., Sabharwal, A., Selman, B.: Model counting (2008)
27. Gordon, A.D., Henzinger, T.A., Nori, A.V., Rajamani, S.K.: Probabilistic programming. In: Proceedings of the Future of Software Engineering (FOSE), pp. 167–181 (2014)
28. Graham, S.L., Kessler, P.B., Mckusick, M.K.: Gprof: a call graph execution profiler. ACM Sigplan Not. **17**(6), 120–126 (1982)
29. Harman, M., O'Hearn, P.: From start-ups to scale-ups: opportunities and open problems for static and dynamic program analysis. In: SCAM (2018)
30. He, S., Manns, G., Saunders, J., Wang, W., Pollock, L., Soffa, M.L.: A statistics-based performance testing methodology for cloud applications. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 188–199 (2019)
31. Holmes, G., Donkin, A., Witten, I.H.: WEKA: a machine learning workbench. In: Proceedings of ANZIIS 1994-Australian New Zealnd Intelligent Information Systems Conference, pp. 357–361. IEEE (1994)
32. Kaelbling, L.P., Littman, M.L., Moore, A.W.: Reinforcement learning: a survey. J. Artif. Intell. Res. **4**, 237–285 (1996)
33. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976)
34. Kluge, M., Knüpfer, A., Nagel, W.E.: Knowledge based automatic scalability analysis and extrapolation for MPI programs. In: Cunha, J.C., Medeiros, P.D. (eds.) Euro-Par 2005. LNCS, vol. 3648, pp. 176–184. Springer, Heidelberg (2005). https://doi.org/10.1007/11549468_22
35. Koziolek, H.: Performance evaluation of component-based software systems: a survey. Perform. Eval. **67**(8), 634–658 (2010)
36. Larus, J.R.: Whole program paths. ACM SIGPLAN Not. **34**(5), 259–269 (1999)
37. Luckow, K., Kersten, R., Păsăreanu, C.: Symbolic complexity analysis using context-preserving histories. In: 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST), pp. 58–68. IEEE (2017)
38. Luckow, K., Păsăreanu, C.S., Dwyer, M.B., Filieri, A., Visser, W.: Exact and approximate probabilistic symbolic execution for nondeterministic programs. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, pp. 575–586 (2014)
39. Mazeroff, G., De, V., Jens, C., Michael, G., Thomason, G.: Probabilistic trees and automata for application behavior modeling. In: 41st ACM Southeast Regional Conference Proceedings (2003)
40. Mazeroff, G., Gregor, J., Thomason, M., Ford, R.: Probabilistic suffix models for API sequence analysis of windows XP applications. Pattern Recogn. **41**(1), 90–101 (2008)

41. Möbius, C., Dargie, W., Schill, A.: Power consumption estimation models for processors, virtual machines, and servers. IEEE Trans. Parallel Distrib. Syst. **25**(6), 1600–1614 (2014)
42. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. ACM Sigplan Not. **42**(6), 89–100 (2007)
43. Nocedal, J., Wright, S.: Numerical Optimization. Springer, Heidelberg (2006). https://doi.org/10.1007/978-0-387-40065-5
44. Păsăreanu, C.S., Rungta, N.: Symbolic pathfinder: symbolic execution of Java bytecode. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, pp. 179–180 (2010)
45. Perez-Palacin, D., Mirandola, R.: Uncertainties in the modeling of self-adaptive systems: a taxonomy and an example of availability evaluation. In: Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering, pp. 3–14 (2014)
46. Puterman, M.L.: Markov decision processes. Handb. Oper. Res. Manag. Sci. **2**, 331–434 (1990)
47. Ramalingam, G.: Data flow frequency analysis. ACM SIGPLAN Not. **31**(5), 267–277 (1996)
48. Risso, F., Degioanni, L.: An architecture for high performance network analysis. In: Proceedings of the Sixth IEEE Symposium on Computers and Communications, pp. 686–693. IEEE (2001)
49. Ron, D., Singer, Y., Tishby, N.: The power of amnesia: learning probabilistic automata with variable memory length. Mach. Learn. **25**(2–3), 117–149 (1996). https://doi.org/10.1023/A:1026490906255
50. Rosendahl, M.: Automatic complexity analysis. In: Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture, pp. 144–156 (1989)
51. Sankaranarayanan, S., Chakarov, A., Gulwani, S.: Static analysis for probabilistic programs: inferring whole program properties from finitely many paths. In: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 447–458 (2013)
52. Sarkar, V.: Determining average program execution times and their variance. In: Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation, pp. 298–312 (1989)
53. Schlabach, T.: Insight into event tracing for windows (2019)
54. Sevitsky, G., De Pauw, W., Konuru, R.: An information exploration tool for performance analysis of Java programs. In: Proceedings Technology of Object-Oriented Languages and Systems, TOOLS 38, pp. 85–101. IEEE (2001)
55. Sharir, M., Pnueli, A., Hart, S.: Verification of probabilistic programs. SIAM J. Comput. **13**(2), 292–314 (1984)
56. Stewart, W.J.: Probability, Markov Chains, Queues, and Simulation. Princeton University Press, Princeton (2009)
57. Tribastone, M.: Towards software performance by construction. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018. LNCS, vol. 11244, pp. 466–470. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03418-4_27
58. Tribastone, M., Gilmore, S.: Automatic extraction of PEPA performance models from UML activity diagrams annotated with the MARTE profile. In: Proceedings of the Seventh International Workshop on Software and Performance (WOSP) (2008)

59. Tribastone, M., Gilmore, S.: Automatic translation of UML sequence diagrams into PEPA models. In: Fifth International Conference on the Quantitative Evaluation of Systems (QEST), pp. 205–214 (2008)
60. Wang, W., et al.: Testing cloud applications under cloud-uncertainty performance effects. In: 2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST), pp. 81–92. IEEE (2018)
61. Wegbreit, B.: Mechanical program analysis. Commun. ACM **18**(9), 528–539 (1975)
62. Wong, S., Cai, Y., Valetto, G., Simeonov, G., Sethi, K.: Design rule hierarchies and parallelism in software development tasks. In: 2009 IEEE/ACM International Conference on Automated Software Engineering, pp. 197–208. IEEE (2009)
63. Woodside, M., Franks, G., Petriu, D.C.: The future of software performance engineering. In: Proceedings of the Future of Software Engineering (FOSE), pp. 171–187 (2007)
64. Woodside, M., Petriu, D.C., Petriu, D.B., Shen, H., Israr, T., Merseguer, J.: Performance by unified model analysis (PUMA). In: Proceedings of the 5th International Workshop on Software and Performance, pp. 1–12. ACM, New York (2005)
65. Zaparanuks, D., Hauswirth, M.: Algorithmic profiling. In: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 67–76 (2012)