IMT School for Advanced Studies, Lucca
Lucca, Italy

Automated Learning of Quantitative Software Models from System Traces

PhD Program in PhD in Systems Science - Track in Computer Science and Systems Engineering (CSSE)

XXXIII Cycle

By

Annalisa Napolitano

2022

The dissertation of Annalisa Napolitano is approved.

PhD Program Coordinator: Prof. Rocco De Nicola, IMT School for Advanced Studies Lucca

Advisor: Prof. Mirco Tribastone, IMT School For Advanced Studies Lucca

Co-Advisor: Prof. Emilio Incerto, IMT School For Advanced Studies Lucca

The dissertation of Annalisa Napolitano has been reviewed by:

Prof. Andrea Marin, University "Ca' Foscari" of Venice

Prof. Laura Carnevali, University of Florence

Dott. Fabio Pinelli, IMT School For Advanced Studies Lucca

IMT School for Advanced Studies Lucca
2022

Contents

List of Figures vii

List of Tables x

Vita and Publications xi

Abstract xiii

1 Introduction 1

2 Related Work 7

2.1 Overview 7

2.1.1 Exploration techniques 8

2.1.2 Output models 9

2.2 Learning low-level informative models 10

2.3 Learning high-level informative models 12

2.4 Learning memoryless models 17

3 Background 20

3.1 Markov Chains 20

3.1.1 Discrete-time Markov chains 21

3.1.2 Continuous-time Markov chains 21

3.2 Queuing Networks 24

3.3 Mean-field Approximation 27

3.4 Variable length Markov chains 31

3.4.1 Higher-order Markov chains 31

3.4.2 VLMC and Context function 31
3.4.3 Minimal Markov Hull 33

4 Memoryless Process Modeling 35
4.1 Discrete-time QN model 36
4.2 Non-linear estimators 36
4.3 Service demand estimator 37
4.3.1 Quadratic programming formulation 38
4.3.2 Moving horizon estimation 39
4.3.3 Application to the load balancer case study 40
4.3.4 Numerical evaluation 41
4.4 Queuing network estimator 45
4.4.1 Linear programming formulation 46
4.4.2 Numerical evaluation 48

5 Memoryfull Process Modeling 64
5.1 Learning the VLMC of a program 64
5.1.1 Input programs definition 65
5.1.2 VLMC of a program 66
5.1.3 The algorithm: *ProgramToVLMC* 66
5.1.4 Computational complexity of *ProgramToVLMC* . . . 72
5.2 Numerical evaluation 73
5.2.1 Accuracy evaluation 74
5.2.2 Sensitivity evaluation 79
5.2.3 Comparison with state-of-the-art 81
5.2.4 Tuning the granularity level 83

6 Conclusion 91

7 Bibliography 94

List of Figures

1 Multi-server queue 24

2 Load balancer QN example 26

3 Comparison between the simulated queue lengths (solid lines) of the CTMC simulations of the QN of Figure 2, with those obtained by the solution of the ODEs (dashed lines) with parameters as in (3.7) [71]. 29

4 (a) Context tree enriched with examples of next-symbol probability distributions (in blue). (b) Minimal Markov hull, obtained from (a); the red elements in (b) are the states that need to be added to (a) in order to represent the process with a first-order MC (see Section 3.4.3). (c) First-order MC of the example with context in Eq. (3.11). For simplicity, the probability distributions for the next state are omitted. 33

5 Simulation and mean-field solution for the queue lengths of the load-balancer running example [68]. 41

6 Queuing network topology used in the comparison experiments against QMLE. 42

7 Prediction error of the topology estimator of the what-if analysis with unseen population vectors 52

8 (a) Summary statistics on the prediction error for the experiments of Fig. 7, and (b) for the experiments of Fig. 9. In each box-plot, the line inside the box represents the median error, the upper and lower side of the box represent the first and the third quartiles of the observed error distribution, while the upper and lower limit of the dashed line represent the extreme points not to be considered outliers. The red dots depict the outliers. 54
9 Prediction error of the topology estimator of the what-if analysis with different server multiplicities 58
10 a) Summary statistics on the computation time needed for learning all the 40 QN models (i.e., 10 for each model size) used for the experimentation in Figures 7 and 9. 59
11 Comparison between the predictions errors of the RNN-learned models and the LP-learned, in the what-if over populations. 60
12 Comparison between the predictions errors of the RNN-leaned models and the LP-learned, in the what-if over concurrency levels 61
13 Summary statistics on the prediction error for the experiments of Fig. 11 and Fig. 12 with learning methods RNN and LP, respectively. 63
14 A simple synthetic program (a) with its control flow (b) . . 65
15 Learned PST for the Program 14a. The red crossed nodes identify pruned nodes in the location PST of l_2. 71
16 Comparison between the empirical (ground-truth) and simulated (VLMC-learned) probability distributions of the number of steps until termination for the benchmark programs. 79
17 Comparison between *BSort* and *BSort Group* with input size 10: (a) PDFs comparison, (b) mean values and variance comparison with also the empirical distribution (ground-truth). 86

18 Comparison between the empirical and *BSort Group* with input size 19: (a) PDFs comparison, (b) mean values and variance comparison. 87
19 Comparison between *QSort*, *QSort Group* and *QSort 2Groups* PDFs with input array size of 9: (a) PDFs comparison, (b) mean values and variance comparison between the empirical (ground-truth), *QSort*, *QSort Group* and *QSort 2Groups*. 89
20 Comparison between the empirical and *QSort 2Groups* PDFs with input array size of 19: (a) PDFs comparison, (b) mean values and variance comparison. 90

List of Tables

1 Model parameters for the load balancer case study example 40

2 Comparison between QMLE and MHE service demand estimators. 43

3 Scalability analysis of MHE service demand estimator. . . 45

4 Sensitivity analysis with respect of pruning factor α and the parameter n_{min}. For each benchmark we used the same runs of the corresponding ground-truth models (see Figure 16). 80

5 Comparison of runtimes (column T) and peak memory usage (column M) of *ProgramToVLMC* and the algorithms [30] and [110] by means of their implementations in [91] and [47], respectively. 82

6 Scalability analysis of the Bubble Sort with respect to two different granularity levels: the unaggregated *Bsort* and the aggregated *BSort Group*; the 'x' values represent experiments not performed due to time and memory limits. . . . 85

7 Scalability analysis of the Quick Sort with respect of three different granularity levels; the 'x' values represent experiments not performed due to time and memory limits. . . . 88

Vita

Mach 14, 1991	Born, Frascati (Rome), Italy
2015	Bachelor Degree in Informatics Engineering Final mark: 110/110 University of Rome "Tor Vergata"
2017	Master Degree in Informatics Engineering Final mark: 110/110 cum laude University of Rome "Tor Vergata"
2017	PhD in Computer Science: Analytical Performance Models of Computer Systems Imt School for Advanced Studies Lucca, Italy

Publications

1. M. D'Angelo, M. Caporuscio and A. Napolitano, "Model-Driven Engineering of Decentralized Control in Cyber-Physical Systems," *2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems (FAS*W)*, 2017, pp. 7-12, doi: 10.1109/FAS-W.2017.113.
2. M. D'Angelo, A. Napolitano and M. Caporuscio, "CyPhEF: A Model-Driven Engineering Framework for Self-Adaptive Cyber-Physical Systems," *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, 2018, pp. 101-104.
3. E. Incerto, A. Napolitano and M. Tribastone, "Moving Horizon Estimation of Service Demands in Queuing Networks," *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2018, pp. 348-354, doi: 10.1109/MASCOTS.2018.00040.
4. E. Incerto, A. Napolitano and M. Tribastone, "Inferring Performance from Code: A Review," *International Symposium on Leveraging Applications of Formal Methods*, 2020, pp. 307–322. Springer
5. E. Incerto, A. Napolitano and M. Tribastone, "Statistical Learning of Markov Chains of Programs," *2020 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2020, pp. 1-8, doi: 10.1109/MASCOTS50786.2020.9285947.
6. E. Incerto, A. Napolitano and M. Tribastone, "Learning Queuing Networks via Linear Optimization," *2020 Proceedings of the ACM/SPEC International Conference on Performance Engineering*, 2021, pp. pp. 51–60.

Abstract

Models are primary artifacts in software system development. In particular, performance models allow us to evaluate and reason about extra-functional properties, such as the average response time and throughput, for which meeting adequate quality levels is increasingly important. Indeed, performance quality is considered as essential as correctness in many practical development scenarios. Markov processes are valuable models for the qualitative analysis of performance. In this thesis, we will present statistical methods that learn Markov models directly from the running software system traces. We will focus on two classes of processes: with and without memory. In the first scenario, we aim to learn fundamental performance metrics, i.e., service demands and routing probabilities, using queuing networks (QN). For processes with memory, instead, we will exploit variable length Markov chains (VLMC) to capture data dependencies throughout the traces of system executions. The conducted numerical evaluations, the presented in-depth study of the literature, and the performed appropriate comparisons with similar tools allow us to demonstrate how the approaches presented in this work constitute a significant step forward concerning state of the art.

Chapter 1

Introduction

The quantitative evaluation of computer systems, using performance metrics such as throughput and response time, provides a means to analyze software quality. Those indeed, are important extra-functional properties that, as much as the functional ones, influence the way the user perceives the system, the degree of satisfaction, and the willingness to pay for it. As a consequence, predicting performance has become essential during all phases of computer systems realization, from design to implementation and testing.

Performance impacts business in a massive way and is considered one of the most pressing concerns, as claimed in a survey among IT administrators [128]. Google Search is now ranking faster mobile pages higher [101], and US retailer Walmart, for example, reported incremental revenues by up to 1% for every 100 milliseconds of page load improvement in their website [44]. Furthermore, several cloud operators that provide solutions as infrastructure-as-a-service (IaaS), platform-as-a-service (Paas), or software-as-a-service (Saas) [141], focus their business on the ability to provide excellent performance at reasonable costs; they stipulate service level agreements with the users by specifying certain quality-of-service (QoS) requirements regarding availability and performance metrics (e.g., average response times, throughput, and maximum response time), whose violations lead them to economic penalties [5]. These and

several other examples could justify the recent trend according to which “in many practical deployment scenarios, particularly mobile, performance is the new correctness” [62].

Accurately and quantitatively predicting the performance of a computer system hinges on the availability of a powerful and highly-predictive model with well-calibrated parameters, which is able to reproduce the system’s dynamics correctly. Performance metrics, e.g., the number of procedure calls [37] and average runtimes [14], are models easier to obtain and lower-informative. To obtain the average response time, for example, it is sufficient to average process runtimes in the log file; yet, the information level and predictive power are low and those metrics give no clues about why the program execution shows that performance. Generally, these quantities are extracted by profilers, whose lack of predictive power prevents the complete understanding of the underlying process and its dynamics (see also [139]).

Markov chains (MC) are a powerful model to describe computer systems’ dynamics with stochastic processes [22, 122] and are at the basis of well-known approaches to formal verification based on probabilistic model checking [82]. They allow quantitative analysis of many extra-functional properties [85,104,126], including performance [21,123]. When the system is modeled with a Markov chain, we can visualize its state in any future time instant, in many different scenarios, and thus perform what-if analysis of the performance by varying the configuration parameters. Unfortunately, Markov models are difficult to obtain because they depend upon considerable craftsmanship in both the problem domain and in the required analytical techniques, hindering their adoption in practice [70, 137].

To overcome this issue, we intend to extract Markov models *automatically*, from traces of the system executions. In this way, the developer can use the tool we provide to analyze the performance without necessarily knowing how the process that statistically describes the system matches with the resulting Markov model.

Concurrent systems with all mutually independent functional blocks, as it should happen in software with a microservices paradigm [43] or

hardware with numerous interconnected components that perform different independent functions, e.g., CPU, hard disk, RAM, IO devices, can be modeled with a first-order Markov chain and are usually represented in the literature using well-known quantitative performance models known as queuing networks (QN) [124]. In a queuing network, numerous users that access shared resources are routed, queued when they find that the requested resource is busy, and complete the service when the resource is idle. The parameters that uniquely characterize the distributions of the service times of each node of the network, in case they are exponential, are the service demands [68]. Thus, they are the key quantities to estimate, together with routing probabilities when the topology is unknown, which define statistically the route of a job in the system.

There is constant research for efficient techniques that can provide continuously up-to-date service demands estimators in a non-intrusive manner; this is due to the increasing usage in software engineering of quantitative models to perform self-adaptation at runtime, i.e. change the behavior of the system depending on the variations of the surrounding environment, to continuously meet the performance quality-of-service requirements [4,33,66,68,72,73,84].

We thus provide an estimator of the service demands of a queuing network that can answer such a call [68], addressing two main challenges, which, to the best of our knowledge, have not been fully answered in the literature yet. First of all, we relaxed the steady-state assumption that has been essential in previous approaches to exploit many well-known relationships and/or analytical results for QNs (see, e.g., [22]), as the Utilization Law [121], which allows determining service demands from the knowledge of the steady-state throughput and utilization at a station [27]. We provide an approach that works well even during the transient-state. This is crucial in self-adapting control, where the system parameters can change quickly and continuously, possibly preventing the system from reaching the steady-state.

Secondly, since the learning could be coupled to the self-adaptation, it could be performed during system runtime; thus, it should be non-

intrusive in the instrumentation and not interfere in any way with its performance. Therefore, it is not possible, for example, to perform *active probing*, inserting extra traffic into the system to evaluate the response with load-tests at different utilization levels (e.g., [7]). Our technique [68, 71], instead, should be placed among those that make use of queue-length information only [132, 133], which can be usually accessed externally (i.e., from the operating system) without the need for direct instrumentation of the application.

The key intuition was to consider the mean-field approximation [25] of the QN as a dynamical model of the system under consideration, which holds both during transient and steady-state. This approximation describes, with a compact set of ordinary differential equations (ODEs), an accurate estimation of the evolution of the queue lengths when the population of the system and the number of servers (i.e., the scaling factor) is sufficiently high [81]. The mean-field ODEs exploit the idea of considering the average behavior of the population instead of tracking the behavior of any single job individually, allowing us to avoid the well-known state space explosion problem that would occur in the case of an exact representation based on the forward equations of the continuous-time Markov chain (CTMC, see, e.g., [22]).

The discretization in time of the set of ODE equations resulting from the mean-field approximation represents the constraints of the optimization problem we constructed, with the initial condition given by the starting queue lengths at each station [68, 71]. The decision variables are the service demands and the objective is to minimize the difference between measured queue lengths and those predicted by the mean-field ODE model.

Furthermore, focusing on the two key aspects of being not-intrusive and fast in producing service demands estimations, we will strengthen our approach with a topology estimator, trying to recover not only service demands but also the entire network with routing probabilities [71].

In [50] routing probabilities and service demands are obtained by encoding such nonlinear equations represented by the mean-field ODEs with a recurrent neural network. Here we show how the same problem

can be mastered by linear optimization, thus considerably reducing the computational cost of the estimation [71]. In this dissertation, we will provide evidence of the effectiveness and the accuracy of our method with numerical evaluations and comparisons with the previous state-of-the-art solutions.

In the case of software systems in the form of source codes, if we hypothesize to represent each instruction as a state, to track program locations, and to abstract from variable values, the program could be interpreted statistically by assigning a probability distribution to the input [35,119], according to the well-know formalism of *probabilistic semantics* of programs, see [52,89]; this notion, however, has not yet been used to synthesize probabilistic models for the whole program without focusing only on the quantification of the probability to satisfy specific path formulas [35,46,70].

Unfortunately, rarely the programs' source codes could be represented with a first-order Markov chain. In programs, indeed, the memoryless assumption often does not hold because of data and control dependencies between instructions and functional blocks. The code could be memoryless only if non-nested program locations were all uncorrelated, and hence if, in every sequence of non-nested conditions, the variables representing the conditions' evaluations were all mutually independent. This fact does not hold, in general, for real programs.

When the next-outcome probabilities do not depend only on the current state but on a certain finite number (i.e. the order k) of previous states, the process is with memory and it can be represented with a k-order Markov model. These models have the drawback of requiring a huge amount of memory, which grows exponentially with the order k, hindering their adoption for software modeling, where the data and control dependencies among variables could be plentiful and the order consequently very high.

The mitigation we propose consists of considering only a set of states for the current program history, i.e., the *context*, that is relevant in order to compute the next-state probability distribution. Thus, for any possible history, the order $h \leq k$ will be a variable number up to the maximum

order k of all the possible previous histories. This compact representation is called variable-length Markov chain (VLMC) and is still correct and with no information losses, improving considerably the amount of memory required when the process shows variable length, i.e., when h has high variability.

Furthermore, our objective is to minimize the amount of memory required, by exploiting approximations based on the trade-off between precision and the amount of history to incorporate in the retained contexts, and to develop an algorithm *ad hoc* for the learning that exploits the typical properties of a program [70], as the sparsity of the transition matrix, and the knowledge of the static control-flow graph (CFG), which could be effortlessly obtained using a static code analysis tool such as *JavaParser* [65]. This static information knowledge allows us to obtain a considerable improvement on memory allocations and execution time needed with respect to the *general purpose* VLMC learning algorithms [47, 90], whose massive cost prevents their usage even for programs of small complexity.

In section 5.2.3, through a numerical evaluation with several benchmarks, we provide evidence of how the prototype implementation of our solution, i.e., *ProgramToVLMC* [70], outperforms the previous solutions when applied to software systems in both memory consumption and time needed for the learning process, making it possible to learn VLMC from the source code of many benchmarks, where it was previously practically infeasible.

Thesis Organization: Chapter 2 presents the literature review of this dissertation about profilers and techniques for the extraction of performance models from both *memoryless* and *memoryfull* processes, and the reason why the proposed techniques outperform the state-of-the-art solutions; Chapter 3 explains the basics about Markov chains, variable order Markov chains and queuing network, whose knowledge is essential to fully understand the dissertation's results; Chapter 4 provides our findings in terms of memoryless processes performance predictions, by exploiting queuing networks and optimization techniques; Chapter 5 shows our findings about memoryfull processes modeling, with VLMC.

Chapter 2

Related Work

In this chapter, the literature review of this dissertation will be presented based on our previous analysis in [69].

Models are separated artifacts from the system, and they can be extracted manually by the architect/engineer. Still, an automatic mechanism to extract these quantitative models from the system will mitigate the problem of the necessity of considerable craftsmanship in the knowledge of both the problem domain and mathematical performance model theories underlying, which is generally difficult, and recognized as the main obstacle to model-based performance analysis in software engineering [137].

2.1 Overview

When dealing with software systems, model-driven development technique could automatize model extraction from software artifacts such as behavioral UML diagrams (e.g., [129, 130, 138]), or domain-specific languages (e.g., [16]), annotated with quantitative information; a comprehensive summary of these techniques can be found in the Balsamo et al. review [11, 79]. Unfortunately, model-driven techniques could not always be applicable, for example in a continuously changing development process, where source-code is modified more rapidly than

the artifacts belonging to the requirement specifications, which will be at a certain point only a representation of an old version of the software [51]. In fast-pacing development, the only viable solution could be learning directly performance models from code, i.e., code-driven techniques. In [69], we reviewed several code-driven techniques for generating performance models, consisting of 24 research papers published in the period 1982-2019, and we detected the dimensions that mostly characterize their differences: the learning techniques, the exploration techniques, the type of the output model or performance metrics, and the scalability level. Code-driven learning techniques could exploit static [108, 111, 135] and/or dynamic [58, 140] analysis to extract the model. In the former case, the code is read and analyzed offline, while in the latter case the instrumented program is executed numerous times producing traces that describe its behavior; by analyzing them, the necessary information to build the performance model is gathered.

2.1.1 Exploration techniques

The considered approaches rely on 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 [10], i.e., analyzing program's traces produced during the runtime on the real developed application, load testing [57], i.e., stressing the software by evaluating the program at runtime with increasing workloads, random sampling, i.e., testing the program under a randomly distributed input [119], and symbolic execution [9, 53, 76], i.e., exhaustively exploring the execution tree using symbolic values for input instead of concrete ones. Each of the listed techniques has its strengths and weaknesses that limit its use, which we will describe exhaustively.

Runtime monitoring is simple to use and provides a reliable analysis of the typical behavior of the system; unfortunately, it does not provide the characterization of performance in particular scenarios such as congestion or low-level utilization, and it may be not easy to instrument the system correctly [20]. In this type of technique, indeed, there is partic-

ular attention not to interfere with the software operation through the instrumentation, not to be intrusive, and not to ingest extra traffic into the system by doing active probing [67].

Load testing, instead, could account for the worst-case scenario, which is an interesting evaluation in performance engineering, yet, it cannot account for the typical scenario. *Random sampling* is the easiest to implement, it might be the only viable option when the program is too complex, or some source-code portions are unknown [36], however, without any heuristic, it could be extremely unlikely to observe interesting but rare system behaviors [29]. *Probabilistic symbolic execution* could be exhaustive in analyzing all the program's paths; unfortunately, the complexity of the symbolic execution grows too fast with the program's number of paths. This issue limits the application of the symbolic execution to some notable behavior such as average-, worst-, and best-case scenarios [35].

2.1.2 Output models

The resulting performance models differ in various aspects: from applicability and easiness of use to information content and predictive power. *Enriched call graphs* and *control-flow graphs* are compact data structures to store paths or edges probabilities [10]. The totality of the program's paths cannot be always represented when the program under consideration is too large or too complex since the number of paths is exponential, and thus the techniques that produce call graphs or enriched control-flow graphs limit the exploration of the software to the *hottest* paths, i.e., those that are highly frequent and/or those that have the greatest impact on the performance. These kinds of models are medium/high informative, depending on the number of program paths that represent and how precisely they measure the frequency of every path.

Both *performance metrics* and the number of procedure calls [37] and average runtimes [14] are low-level informative models since they do not explain the reason behind the resulting values of these metrics. Similarly to *bottleneck detection* models, which provide insight on the worst-case ex-

ecutions of the program, that could be in terms of *hot paths* detection [31], or input workloads that trigger performance bottlenecks [1].

Although *target event probabilities* [89, 113] are typically used for bug finding and they do not represent a direct measure of performance, they can give insight on performance by providing probabilities of costly functions or inefficient blocks of code. *Cost-function models* provide a closed-form expression that describes how some performance metrics of choice vary according to the input size and thus they are considered medium-level informative. The function could represent the average behavior [140] as well as the asymptotic one [57]; thus, these techniques do not distinguish among a set of functionally and asymptotically equivalent algorithms. Finally, Markov processes [21] are a powerful high-predictive and high-informative model to analyze performance [108] of computer and communication systems. Furthermore, if the system is memoryless we can represent it with a compact (first-order) Markov chain (see first-order Markov models in Section 3.1). Unfortunately, programs are rarely memoryless since transition probabilities among instructions depend on the previous history (see higher order models in Section 3.4.1).

2.2 Learning low-level informative models

This section will present the related works that extract low-level informative models, such as performance metrics and call graphs.

Sarkar et al. [114] developed a framework to derive mean and variance values of the runtime of the program's procedures, by counter-based executions of the application, and afterward these values are stored in an enriched control-flow graph. This approach provides learning that is context-insensitive: measurements are taken and then multiplied with path frequencies, independently of the call site. This introduces some approximation since measurements of runtimes could depend on process history and context, intended as the values of all the variables.

Magpie [13, 14] is a tool for end-to-end online performance analysis from traces collected at the user endpoint, that construct a probabilistic model of the requests and the system' behaviors. The instrumentation

is conducted through black-box instruments such as kernel-level tracing for Windows [115] or WinPcap packet capture library [109]. The results are clustered by performance and/or requests' features, and these clusters provide a means to detect anomalous requests and system malfunctions.

JinsightEx [118] measures performance metrics, e.g. runtimes and memory usage, of a Java program, and organizes them through *execution slices*, i.e., portions of the program execution that are user-defined with dynamic or static criteria. These slides provide a view for the developer to analyze and inspect performance, and they can be grouped in *workloads* to facilitate the analysis of larger portions/procedures.

Gprof [58] produces a call-graph enriched with procedure's runtimes, by post-processing samples of program counter periodically measured through runtime monitoring in a single real execution of the program.

PerfXRL [1] uses reinforcement learning [75] to guide the dynamic analysis of a program in execution with changing workloads, in order to find input values that trigger performance bottlenecks; starting from an input set that could possibly be extremely large.

Ammons et al. [2] address the bottleneck detection problem, i.e., identify the worst-case execution, by comparing their cost metric in a summarized model obtained through heuristics from a given profile of the program execution (e.g., call-tree). They develop two algorithms: one that finds expensive paths and the other that computes how the cost of a path differs among similar executions.

The approaches in [89,113] produce target events probabilities of probabilistic programs, i.e. both those whose variables assume uncertain values during execution, and non-deterministic program, e.g., multithreaded or distributed applications, respectively. The former solution solves nondeterminism by sampling the value of the variables from probability distributions assigned to them, while the latter generates a tree using bounded symbolic execution and implement over it a scheduler, based on either an exact algorithm using Markov decision processes [105], or an approximated solution using Monte Carlo sampling, that is iteratively improved with reinforcement learning. Sankaranarayanan et al. [113]

introduce a heuristic on the model counting phase of the probabilistic symbolic execution, namely *volume bound computation*, and exploit this technique in conjunction with Monte Carlo sampling, random simulation, and statistical tests to provide bounds on the probability a certain event happens. They claim, moreover, that only an adequate subset of the execution paths are needed to compute those bounds.

Filieri et al. [45] developed an algorithm for computing target events probabilities, that is based on Monte Carlo sampling to improve the Bayesian estimates. Furthermore, they introduce the *informed sampling* technique on the symbolic tree to firstly explore paths with high statistical significance, and afterward improve convergency of the symbolic execution.

Borges et al. [24] deal with the automatic learning of a target event probability in a program with a given continuous probability distribution over the floating-point domain of the input profile. They evaluated three different strategies based on gradient descent optimization [100], and they implement heuristics for the learning phase to improve the scalability of the probabilistic symbolic execution in their solution. The idea under the improvement is to rank the edge condition constraints of the symbolic tree according to their impact on the convergence of the statistical learning and the model counting.

2.3 Learning high-level informative models

This section will describe works learning medium and high informative models and noteworthy approaches for this dissertation.

Buse et al. [31] developed a methodology for *hot paths* identification, which relies on the idea that the most likely hot paths are those that have the smallest impact (i.e., many different variables' values modifications) on the program's state, intended as the set of all variables values in the stock and global ones. Any machine learning algorithm could then be trained over the set obtained through the static paths enumeration, to identify input features characterizing hot paths (Weka [64] is used here). Since this approach relies on machine learning, it suffers from overfitting and it also has the limitation of considering only intra-procedural

paths within one single class, ignoring procedure calls that cross the class boundaries. Thus, it may reveal unuseful in case the program behavior is not fully captured by one single class.

Speedoo [37] is an approach that provides optimization suggestions to the developer by identifying the program's portions and groups of methods that mostly affect performance (i.e., hot paths), and by searching for performance antipatterns, e.g., cyclic invocation, expensive recursion, which represent improvement opportunities. The hot methods are ranked according to several factors: the architectural importance defined by the Design Rule Hierarchy algorithm (DRH), by Cai et al. and Wong et al. [32, 136], and it is related to the size of the sub-calls tree, dynamic execution metrics such as CPU time and memory allocations and static complexity metrics, e.g., the number of loops in the procedure.

Zaparanuks et al. [140] provide an approximation of a descriptive cost function for the program performance in terms of several cost metrics, such as algorithmic steps, number and size of read/write operations on data structures, number of object allocations. A set of representative program traces are given as input, while the tool infers the input size and type, e.g., recursive data structures, and arrays, by computing the number of elements and their memory occupation. The two main limitations concern the fact that only a descriptive model is returned and not a proper quantitative evaluation, and also that the approach is unable to infer input sizes of different elements than a data structure, for example, primitive types.

Wang et al. [131] derive performance metrics in applications developed on the cloud. The approach tests the cloud infrastructure using micro-benchmarks and resulting in performance distributions of each cloud resource, e.g., memory and CPU; while the application is analyzed stand-alone, separately from the infrastructure with a given input of a typical workload, producing the *resource usage* profile. Moreover, with the given input the approach analyzes the application deployed in the cloud producing the *baseline performance metrics*, and finally by merging the resource usage, the cloud infrastructure's metrics, and the baseline profile results in the final performance model of the application.

Trend-prof [57] computes the program's empirical complexity as a function of the workload size and user-specified features, e.g., the sizes of input files. The evaluation is done using load testing, by executing the program many times with variable input sizes, and finally, the measured execution times of the program's blocks are fitted against linear or power-law functions.

Another approach that extracts an asymptotic cost function is the one of Coppa et al. [38], which aims to find asymptotic inefficiencies from program traces. They rely on the read memory size (RMS) metric, counting the number of accessed memory cells to relate the collected cost value to the input size. The supporting tool exploits Valgrind [99] for the instrumentation, and by measuring the minimum and the maximum costs of executing routines and by using curve fitting and curve bounding techniques, they estimate the function that best describes the asymptotic behavior of the procedure.

Ball et al. [10] compute path frequencies and performance metrics, starting from the principle that since control-flow edge transactions' probabilities are dependent on each other, to obtain a path probability it is not sufficient to multiply the probabilities of all the edges of that path. The limitation of path profiling is that the number of paths is exponential with respect to the program size. To mitigate the issue they estimate only intra-procedural dynamic paths -those executing during the (non-intrusive) runtime monitoring- of an acyclic version of the program.

Whole Program Paths (WPP) [83] relies on the previous work [10] for instrumentation and path discovery, and by developing a novel compression algorithm that finds regularities, i.e., repeated code portions, it transforms the traces to the directed acyclic graph storing the dynamic control-flow. This resulting graph is also able to find hot subpaths.

Geldenhuys et al. [53] exploit probabilistic symbolic execution by extending Java Symbolic PathFinder [103] and trying to retrieve a high-informative model with probabilities of reaching any program location of a sequence of operations on a data structure, as insertions and deletions from a BinomialHeap, TreeMap, and Binary tree. Because of the expensiveness and scarce scalability of the probabilistic symbolic execu-

tion, the approach is limited to sequences of 5 operations, and it explicitly expresses the impossibility of analyzing with their techniques sequences of 14 or more operations. For this reason, we still consider their resulting model medium/low-informative.

Interesting approaches are those that extract the probability density function of the performance metric of interest, which is a compact, high-informative model since it both encapsulates path probabilities and the cost metric.

PT4Cloud [63] learns the probability density function of the execution cost of applications on the Cloud. They test the application with some pre-selected benchmarks and pre-specified workloads and they stop their non-parametric statistical approach when the two resulting distributions are statistically equivalent.

PerfPlotter [35] exploits probabilistic symbolic execution with Java Symbolic PathFinder [103] to determine paths with either high or low probabilities that also have the property of terminating under a certain threshold number of steps, under a usage-profile given as input. The resulting paths are executed, and the effective runtime is measured, combined, and weighted with path probabilities to compute the probabilistic density function. The limitation of this work is intrinsic in the choice of probabilistic symbolic execution, whose scalability is a concerning issue. *PerfPlotter* is indeed able to model only a subset of the program's executions: the average-, best- and worst-cases.

Another relevant approach is the Ramalingam et al. [108], which is the only approach producing a high-informative, high-predictive model as a Markov chain. The input of the algorithm is the program control-flow graph, already annotated with edges probabilities, which will become transition probabilities of the (first-order) MC; then, a path probability is simply computed by multiplying the probabilities of the edges of that path, by assuming all edges probabilities mutually independent. This assumption does not hold true, in general, for real programs, and transition probabilities among edges depend on the values of the variables due to data and control dependencies, i.e., the previous history.

Also noteworthy are the techniques that extract context-sensitive per-

formance models, considering the impact on the performance of the program's state, intended as the entire set of all variable values or the program's history.

Luckow et al. [88] propose a technique to determine the worst-case complexity function with *guided* symbolic execution, run firstly to small inputs and iteratively increasing its size. During iterations, only paths that account for the worst-case are selected, and historic information is considered to select the next branch to symbolically execute. Afterward, the program is dynamically explored in worst-case paths, and the measured metrics of interest (e.g., execution time and memory usage) are merged with probabilities and fitted against the asymptotic complexity function.

Brünink et al. [28] provide a methodology to dynamically analyze a program during runtime producing performance assertions. An intermediate runtime model is constructed, in the form of a graph that describes the hot functions of the program's expected behavior, enriched with path probabilities computed either in context-sensitive or context-insensitive settings. Context information is evaluated for paths belonging to different clusters of estimated performance metrics.

Explicitly considering context information will provide a more accurate performance model and an interesting and peculiar deeper view of the program's behavior, by knowing the amount of memory among program locations and/or procedures and how this memory is distributed. This perspective is valuable also in parallelizing the program, placing separately independent blocks of code. This literature on code-driven performance learning hints at a future direction on context-sensitive profiling methodologies with high-predictive and high-informative modeling, such as, for example, (higher-order) Markov chains. Exploiting variable-length Markov chains (VLMC) could be convenient and visionary. Indeed VLMCs have the advantage of maximal accuracy, even exactness in some circumstances, with minimal memory occupation. These models, mostly used for DNA sequencing [42, 90] and pattern recognition [23, 49], were recently used in software modeling to perform intrusion detection [94, 95], but never been used for quantitative performance

modeling.

2.4 Learning memoryless models

Unlike the previous sections, here we will be confronted with memoryless performance models, of which we can find references in the related work section of the publications of Incerto et al. [68, 71].

When the process is memoryless it can be modeled with a first-order Markov chain and, in the case of multitenancy, with a queuing network (QN). Thus, it is more correct to compare with profilers that, assuming the independence among states, estimate service demands and/or transition probabilities. Indeed, profilers as *PerfPlotter* [35], *Gprof* [58] and the others cited above, have the advantage of not making assumptions on the distribution of the service rates, while we are assuming them to be exponentially distributed, yet the models that they generate lack descriptive and predicting power, while the QN with well-estimated parameters can predict all future time instants queue lengths at each station.

We aim at finding an estimator that runs at runtime, minimizing the amount of instrumentation needed, and allowing the learning of parameters also during the transient regime. Most related work on service demands estimation relies on a corollary of the well-known Little's Law unfortunately working only in the stationary regime, i.e., the Utilization Law [121], using different statistical inference approaches such as linear regression [102], non-linear optimization [96], clustering regression [40], independent component analysis [120], pattern matching [41] and Gibbs sampling [125, 132] based on measured steady-state values of utilization and/or throughput. The Utilization Law relates utilization and throughput means with the average service demands, requiring quantities that may be difficult to measure. It is not always possible, indeed, to measure the utilization value of a station, especially in a cloud environment, such as Platform-as-a-Service (PaaS), in which the developer does not have complete control over the underlying architecture.

Furthermore, many approaches require *active probing*, i.e., measuring the values of interest at different utilization levels, stressing with load

testing, and injecting extra traffic into the running system, aimed to inspect the system from the inside. For example, the technique presented by Liu et al. in [87] is related to our service demand estimator [68] because it is based on a quadratic programming optimization. They measure every station's utilization under different load configurations and estimate demands by relying on a steady-state closed equation for the QN. Active probing will necessarily be intrusive, hindering system usage during runtime, and will induce extra traffic that may interfere with the one generated by the actual users, making it difficult to reconstruct the original metric, i.e., the one generated by the regular traffic only, from the measurement (see [8]).

Instead, the works of Awad et al. and Wang et al [7, 132, 133] are non-intrusive in the system instrumentation, since the former measures only end-to-end response time and throughput of the transactions, while the latter two, similarly to our approaches [68,71] monitor queue lengths only. The former estimates service demands by means of a fitting problem with nonlinear optimization and has the limitation of requiring *active probing*, with load testing at different utilization levels, and many observations of the system in the steady-state regime. Thus even if these are end-to-end measurements only, they may interfere with the system. Furthermore, the work of Menasce et al. can obtain only a feasible assignation of the service demands, indistinguishable up to permutations in series of queuing stations.

The authors of [132] develop an estimator based on Gibbs sampling with a computational cost already high for networks of small/moderate size, inapplicable for a real system at runtime. In [133] it is presented a closed-form expression to evaluate all the network's stations service demands for a multiclass application and a load-independent scenario. The load-dependent case is treated approximately by appropriate scaling of the computed service demands of the load-independent case. A detailed comparison against these techniques, presented in [68], will be exhibited in Section 4.3.4.

Moreover, with the paper [71], which extends the one presented in the previous work [68], we do not settle only for finding service rates

but we estimate also the entire network topology, i.e., the connections among stations with the corresponding routing probabilities. In this regard, the work by Garbi et al. [50] focuses on the same problem from another perspective: instead of using the ODEs to solve a linear optimization problem, they train with real traces a recurrent neural network that learns the values of rates and routing probabilities. We numerically evaluate the accuracy of our approach compared to the one of Garbi et al. [50] in Section 4.4.2.

Chapter 3

Background

3.1 Markov Chains

A stochastic process $X(t) = \{X(t) \mid t \geq 0\}$ is a family of random variables $X(t)$, each indexed by a value of $t \in T$ belonging to the *index set* T, and each random variable $X(t)$ assuming values in a sample space Ω. The index t is often considered as the *time* in which the process evolves, referring to a *discete-time* stochastic process if the index set is discrete, having the same cardinality of the set of natural numbers $\mathbb{N}$, i.e., $T = \{0, 1, 2, ...\}$, instead we refer to a *continuous-time* process when the index set T is continue, with the same cardinality of the real numbers set. A stochastic process $X(t)$ is a chain when the sample space Ω is discrete and thus $X(t)$ assumes values that can be represented using natural numbers $\{1, 2, 3, 4, ...\}$. An example of chain is the stochastic process quantifying the number of users waiting for a service at a server facility.

A Markov process is a stochastic process satisfying the *memoryless* property, which asserts that the next-state probability distribution does not depend on the previous history but only on the current state of the process itself. In the next subsections, we will formally define the memoryless property of *discrete-time* Markov chains (DTMC) and *continuous-time* Markov chains (CTMC). For an in-depth complete discussion on MCs, see, e.g., [123].

3.1.1 Discrete-time Markov chains

A discrete-time MC is a stochastic process $\{X_n : n \in \mathbb{N}\}$, where the random variables X_n take values in a discrete set S, ranged over by variables x_i. Its conditional probability distribution satisfies the *memoryless* property, which, in the case of discrete-time MC can be formally defined as follows:

$$\Pr(X_{n+1}{=}x_{n+1} \mid X_n{=}x_n, X_{n-1}{=}x_{n-1}, \ldots, X_0{=}x_0) \\ = \Pr(X_{n+1}{=}x_{n+1} \mid X_n{=}x_n)\ \forall\, n \in \mathbb{N}. \quad (3.1)$$

That is, the probability of being in state x_{n+1} at time $n+1$ only depends on the state at the previous time step n and not on the whole history $x_n x_{n-1} \ldots x_0$ (from most to least recent). Whenever convenient, for simplifying notation we shall omit to explicitly write the random variables and denote the history as a string of states; for instance, (3.1) will be more concisely written as $\Pr(x_{n+1} \mid x_n x_{n-1} \cdots x_0) = \Pr(x_{n+1} \mid x_n)$.

Here, we will work with time-homogeneous MCs, where the conditional probabilities do not depend on n; together with the memoryless property, this allows us to represent the MC with only the set of states S, and the $|S| \times |S|$ conditional probabilities:

$$p_{i,j} = \Pr(j \mid i), \qquad \forall n \in \mathbb{N}, i, j \in S.$$

representing the probability of the transition from state i to state j in one step. The matrix resulting from placing the $|S|$ probabilities $p_{i,j}$, each in the row i and column j is called transition probability matrix.

3.1.2 Continuous-time Markov chains

The difference between discrete and continuous Markov chains is that in the latter case the index t ranges in a continuous index-set, meaning that transitions could occur in any possible time instant. The continuous-time Markov chain $\{X(t) : t \geq 0\}$ satisfies the memoryless property, i.e. the following relation holds:

$$\Pr(X(t+s){=}x_{t+s} \mid X(t){=}x_t, X(t-h){=}x_{t-h}, \ldots, X(0){=}x_0)$$
$$= Pr(X(t+s){=}x_{t+s} \mid X(t){=}x_t).$$

$$\forall\, t, s, h \geq 0 \quad \forall\, x_{t+s}, x_t, x_{t-h}, x_0 \in S$$

Notice that this equation does not imply only that transition probabilities are independent of the previous history, but also that the time instants at which transition occur are independent of the amount of time already elapsed in the current state of the Markov chain, i.e., the *sojourn time*. This property is typical of exponential distributions; indeed it can be proved that the amount of time before a transition occur follows an exponential probability distribution, which is also the only continuous distribution having the memoryless property, i.e., $\Pr\left(\mathcal{T} > s+t \mid \mathcal{T} > s\right) = \Pr(\mathcal{T} > t).$, where $\mathcal{T}$ is the random variable representing the sojourn time.

When the MC is homogeneous the probability of seeing a transition from the current state $X(t) = x_t$ to the next state $X(s) = x_s$, for any possible real $s \geq t$, only depends on the difference between t and s, namely, $\tau = s - t$. The notation of transition probabilities quantifying the probability of the transition from the current state $i \in S$ to the next state $j \in S$, after a time interval of τ, can be simplified as follows:

$$p_{i,j}(\tau) = \Pr(X(t+\tau) = j \mid X(t) = i), \qquad \forall\, t \geq 0.$$

Transition rates

In the case of continuous-time MC, we use the notion of transition rates instead of transition probabilities to simplify the notation by removing the dependency of the probability from the transition time. Let us consider the probability of the MC to transit from the state i to the state j in the interval $[t, \Delta t)$, i.e., $p_{i,j}(t, t+\Delta t)$, which is neglectable for small values of Δt, while tending to 1 if Δt grows to infinity, and let us consider an observation window little enough that the probability of observing more than one transition for the MC can be considered equal to zero.

Moreover, we deal with homogeneous MC, by assuming $p_{i,j}(t, t+\Delta t) = p_{i,j}(\Delta t)$ constant $\forall\, t \geq 0$. Transition rates are defined as follows:

$$q_{i,j} = \lim_{\Delta t \to +0} \left\{ \frac{p_{i,j}(\Delta t)}{\Delta t} \right\} \ \forall\, i \neq j$$

Thus, the probabilities:

$$p_{i,j}(\Delta t) = q_{i,j}\Delta t + o(\Delta t) \ \ \forall\, i \neq j$$

As previous formulas suggest, transitions rate could be seen as *how fast* transitions occur, which is independent of the time interval Δt. Finally, to determine the value q_{ii}, we rely on the property according to which probabilities sum up to 1 (see the details in [123]), finding that $q_{ii} = -\sum_{i \neq j} q_{i,j}$. The matrix whose elements are $q_{i,j}$ is called *generator* or *transition rates* matrix Q, and the sum over the rows is equal to 0.

If we put together the concept of transition rates with the sojourn time, when the former can be related to the velocity of the transitions, we can easily infer that the rate of the exponential distribution of the sojourn time is $\mu_i = -q_{ii}$.

The Chapman-Kolmogorov Equations

The Chapman-Kolmogorov equations for a homogeneous continuous-time Markov chain can be explained with the following:

$$p_{i,j}(t+\Delta t) = \sum_{all\ k} p_{ik}(t)p_{kj}(\Delta t) \quad \forall\, t, \Delta t \geq 0 \ \ \forall\, i,j \in S. \tag{3.2}$$

which implies that the probability of reaching the state j, from the current state i, in a period of time of length $t+\Delta t$, is equal to the sum of probabilities of going from i to any intermediate state k in a period t and from k to j in δt. This can be rewritten, equivalently, as:

$$p_{i,j}(t+\Delta t) = \sum_{all\ k} p_{ik}(\Delta t)p_{kj}(t) \quad \forall\, t, \Delta t \geq 0 \ \ \forall\, i,j \in S. \tag{3.3}$$

If we compute the derivative of the quantity $p_{i,j}(t)$ in these two formulas (3.2 and 3.3), we find respectively the forward 3.4 and backward 3.5

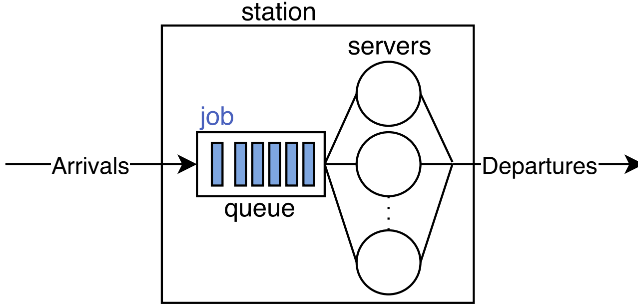


Figure 1: Multi-server queue

Chapman-Kolmogorov equations:

$$\frac{dp_{i,j}(t)}{dt} = \sum_{all\ k} p_{ik}(t) q_{kj} \quad \forall\, t, \Delta t \geq 0 \ \forall\, i, j \in S. \tag{3.4}$$

$$\frac{dp_{i,j}(t)}{dt} = \sum_{all\ k} q_{ik} p_{kj}(t) \quad \forall\, t, \Delta t \geq 0 \ \forall\, i, j \in S. \tag{3.5}$$

Notice that, the Chapman-Kolmogorov equations are also defined for DTMCs. Yet, we decided not to include them in this recall chapter because we do not exploit them in the dissertation.

3.2 Queuing Networks

Queues in computers systems arise whenever there are many users, also called *clients* or *jobs*, competing for a service in a shared resource, which most of the times is a *service station*, called also *node*. It is worth pointing out that even if we refer only to computer systems, the queuing theory could be applied to every practical socio-economic scenario.

Figure 1 represents a multi-server facility with jobs arriving at the station requiring a certain workload, i.e., *service demand*, to the server. If the arriving job finds a server idle it immediately starts to be served for an amount of time necessary to satisfy its service demand, otherwise, it is placed in the waiting queue until its turn comes, i.e., at least one server

becomes idle and there are no preceding jobs in the queue, according to the scheduling policy of the queue itself, e.g., first-in, first-out (FIFO), also known as first come, first served (FCFS). When a job completes its service demand, it then departures from the station.

Since interarrival times and service demands are random variables and server capacity as well as server multiplicity are costly resources, queues arise; and the system engineer is in charge of optimizing the trade-off between costs and performance, typically measured with metrics such as maximum and average queuing time a user perceives, station throughput, and so on. A service station is completely determined once it is specified:

- The probability distribution of the users' interarrival time, i.e., the time between two consecutive arrivals at the station. Since we assume interarrivals times are exponentially distributed (i.e., we are placing in a CTMC scenario), the only parameter to know is the arrival rate λ, i.e., how many jobs arrive on average in a time unit.

- The probability distribution of the service demands, i.e. the time needed for a job to depart once it has started service at the server. We also assume service demands to be exponentially distributed, thus the only parameter needed is the service rate μ.

- The scheduling policy, i.e., which is the policy according to which a job is chosen among the others in the queue to start service. In this dissertation, we will always consider only the first-come-first-served queue, FCFS, also called FIFO.

- The server multiplicity, or *concurrency levels*, is the parallelization level and characterize how many jobs can be executing at the same time, often referred to as the number of the cores of a processor.

A queuing network (QN) is a network of many nodes, each of them being a station like the one in Figure 1 providing different kinds of service, e.g., in a computer system, IO devices, disks, RAM memory, CPU. The jobs circulate in the network going from one station to another according to the routing matrix, i.e., the matrix whose element at row i and column j,

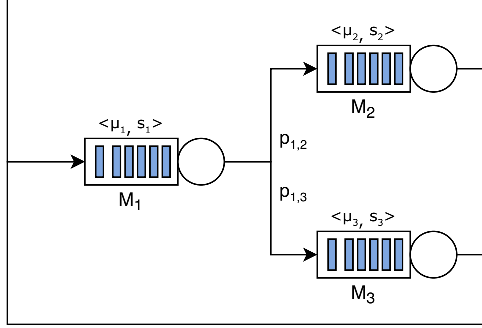


Figure 2: Load balancer QN example

$p_{i,j}$, provides the probability that a job, finishing service at the station i, goes to the station j.

We limit our study to closed queueing networks, where a fixed population circulates in the system, and there are no new arrivals neither departures of jobs. Furthermore, we often model the think time of the customer between one session request and the next one with a reference station M_1, that is considered as an infinite server, i.e., with a server multiplicity greater or equal to the population size N.

A closed queueing network is completely characterized with the following key quantities:

- The population size N, i.e., the fixed number of jobs circulating in the network
- The network size M, i.e., the number of stations in the network
- The server multiplicity vector, i.e., $(s_1, s_2, \ldots, s_M)$, where each s_i is the concurrency level of M_i, also called station i, with $1 \leq i \leq M$.
- The vector of the stations service rates $(\mu_1, \mu_2 \ldots \mu_M)$, each μ_i being the service rate of the station i, with $1 \leq i \leq M$.
- The routing probability matrix $P = (p_{i,j})_{1 \leq i,j \leq M}$

- $x(0) = (x_1(0), \ldots, x_M(0))$ is the *initial condition*, i.e., the number of jobs assigned to each station at time 0.

Figure 2 reports an illustrative example of a closed QN representing a load balancer system with three stations: the load balancer station M_1 and the two distributed replicas M_2, M_3, providing the same kind of service to arriving jobs. Jobs are distributed among the replicas according to the routing probabilities $p_{1,2}$ and $p_{1,3}$; after service completion, they return back to M_1. The load balancer is a means to equally distribute requests between the two replicas, with the aim of maintaining both the queues and the utilization of the two stations balanced. It is a well-known technique in performance engineering to build scalable distributed systems [127].

Finally, it is worth clarifying that while service demand in closed networks is defined with the total service time of a job in a station, and thus it can be seen as the product of the expected visit ratio and the expected service time at the node (see Section 6.10 in [61]), in the dissertation we refer at the service demand μ_i of the station i as the mean service time of a single visit. This could be seen as a service demand of a virtual network with the expected visit ratio equal to one for all stations.

3.3 Mean-field Approximation

In queuing network theory, the *mean-field*, or *fluid* approximation, either called *fluid limit*, approximates the dynamics of the discrete process represented by the CTMC modeling the queue lengths of a queuing network, with those of a deterministic real-valued process, thorough a system of ordinary differential equations (ODEs) [25].

The underlying idea, giving the name to the approximation itself, is to think about the users waiting at the queues no more as individual identities, but as a fluid flowing between the stations of the network, thus forgetting about tracing the individual behavior of any single user, which would have been infeasible even for small populations, and consider the

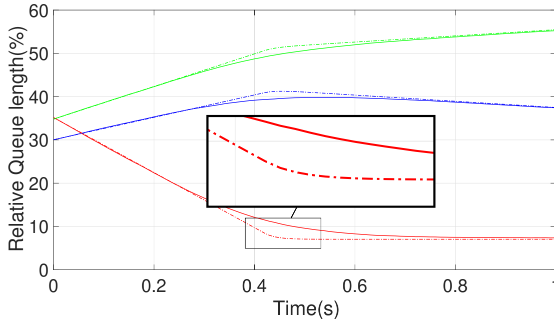
average behavior, according to the following equation:

$$\dot{x}_i(t) = -\mu_i \min\{x_i(t), s_i\} + \sum_{j=1}^{M} p_{j,i}\mu_j \min\{x_j(t), s_j\}, \ \ \forall\, t \geq 0. \tag{3.6}$$

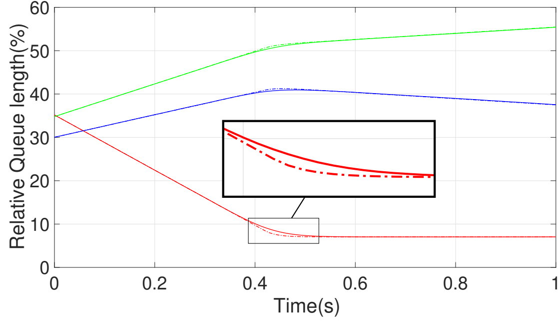
with i varying between 1 and M representing the index of the station, $x_i(t)$ represents the random variable representing the number of jobs queuing in the station i, and the dot notation used to denote time derivative.

The explanation of this equation is intuitive: the first term on the left side represents the average decrease in the queue length of the station i due to the number of jobs that are finishing work, i.e., the server capacity (rate) μ multiplied by the number of jobs on service, i.e., $\min\{x_i(t), s_i\}$. Indeed if the station is actually idle it happens that $s_i > x_i(t)$ and there are $x_i(t)$ jobs executing, otherwise, there are s_i jobs executing, i.e., the parallelism level; instead the second term on the left side represent the increment due to jobs arriving from the other stations after service completion, i.e., the sum of the number of jobs that are actually completing service at any station $1 \leq j \leq M$, i.e., $\mu_j \min\{x_j(t), s_j\}$, multiplied with the probability of going from the station j to i.

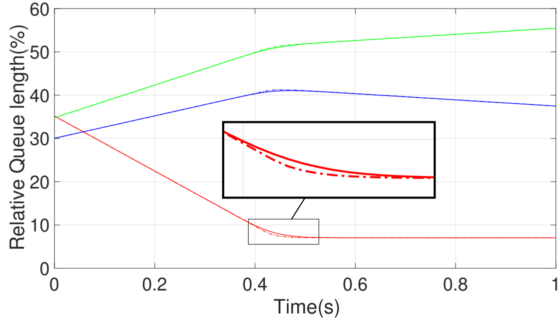
For example, The following system of ODEs represents the mean-



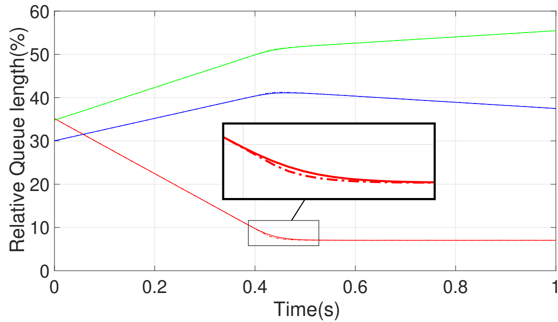
(a) k=1



(b) k=10



(c) k=20



(d) k=50

Figure 3: Comparison between the simulated queue lengths (solid lines) of the CTMC simulations of the QN of Figure 2, with those obtained by the solution of the ODEs (dashed lines) with parameters as in (3.7) [71].

field approximation of the load-balancer queuing network in Figure 2:

$$\begin{aligned}
\dot{x}_1(t) &= -\mu_1 \min\{x_1(t), s_1\} + \mu_2 \min\{x_2(t), s_2\} + \mu_3 \min\{x_3(t), s_3\} \\
\dot{x}_2(t) &= -\mu_2 \min\{x_2(t), s_2\} + p_{1,2}\mu_1 \min\{x_1(t), s_1\} \\
\dot{x}_3(t) &= -\mu_3 \min\{x_3(t), s_3\} + p_{1,3}\mu_1 \min\{x_1(t), s_1\}
\end{aligned}$$
$\forall\, t \geq 0.$

The estimation of the average queue lengths tends to the exact value as the *system size* K, intended as the populations of jobs and server multiplicities, grows to infinity, according to Kurtz's theorem [81], being instead rough for small values of K. A numerical evaluation of the accuracy of the ODEs numerical solutions varying the values of the system's size K, against the real measured values of 2000 statistically independent simulations of the evolution of a network similar to the load balancer represented in Figure 2, but with connections also between the stations M_2 and M_3, has been conducted and it is presented in Figure 3 [71]. The network is configured with the following parameters with the aim of showing the most influential approximation error:

$$\mu = \big(21.91, 57.20, 10.49\big) \qquad s = k\,\big(33, 24, 44\big) \qquad x_0 = k\,\big(82, 96, 95\big)$$
$$P = \begin{pmatrix} 0.38, & 0.50, & 0.12 \\ 0.33, & 0.36, & 0.31 \\ 0.15, & 0.73, & 0.12 \end{pmatrix} \tag{3.7}$$

with scaling factor $K \in \{1, 10, 20, 50\}$.

The source of approximation derives from considering the average of the minimum between two random variables as the minimum value of the averages. Indeed, the average value of the queue length varies according to the following equation:

$$\mathbb{E}[\dot{X_i}(t)] = -\mu_i \mathbb{E}[\min\{X_i(t), s_i\}] + \sum_{j=1}^{M} p_{j,i}\mu_j \mathbb{E}[\min\{X_j(t), s_j\}] \;\; \forall\, t \geq 0. \tag{3.8}$$

Yet, it is not consistent that $\mathbb{E}[\min\{X_i(t), s_i\}] = \min\{\mathbb{E}[X_i(t)], s_i\}$.

3.4 Variable length Markov chains

This section will present the basics of processes with memory, i.e., higher-order Markov chains and variable length Markov chains (VLMC), and extends the background analysis in [70]. VLMC is a compact but exact representation of processes showing memory that varies according to the observed context.

3.4.1 Higher-order Markov chains

While MCs that we just defined, also called first-order MCs, follow the memoryless property 3.1, and they only depend on the current state, higher-order MCs encode stochastic processes which depend on the historical past their current state [107], up for a maximum number of previous states k. We denote a k-order MC model, a process with a memory of at most k. That is, there exists an integer $k > 0$ such that the discrete stochastic process $\{X_n : n \in \mathbb{N}\}$ satisfies the property:

$$\Pr(x_{n+1} \mid x_n x_{n-1} \cdots x_0) = \Pr(x_{n+1} \mid x_n x_{n-1} \cdots x_{n-k+1}), \tag{3.9}$$

for all steps $n \geq k$ and for all states. Notice that this equation is analogous to the memoryless property and that since k is a finite fixed value, for any $k > 1$, it is possible to consider the stochastic process $\{Y_n : n \in N\}$ with $Y_n = (X_n, X_{n+1}, \ldots, X_{n+k-1})$ over the state space S^k, which is a first-order MC because every state encodes all the memory information necessary for its evolution, and thus, Y_n satisfies the memoryless property. The drawback of this solution is that the number of states needed to represent the model grows exponentially with the order of the Markov chain (i.e., the memory length) [30]. This massive use of memory makes them generally impractical to analyze.

3.4.2 VLMC and Context function

A process showing a variable length memory satisfies "h-memoryless" Equation 3.9 for finite value $1 \leq h \leq k$ varying between 1 and the max-order memory length, depending on the specific history that has been

observed, that is

$$\Pr(x_{n+1} \mid x_n x_{n-1} \cdots x_0) = \Pr(x_{n+1} \mid x_n x_{n-1} \cdots x_{n-h+1}), \, \forall \, n \in \mathbb{N}, n \geq h. \tag{3.10}$$

The *context function*, denoted by $\mathcal{C}$, is the key concept to understand the memory evolution of a variable length Markov chain, containing the information about which part of the previous history is relevant to determine the conditional probability distribution for the next state. For any complete process history of all past visited states, $x_n x_{n-1} \cdots x_0$, the function $\mathcal{C}(x_n x_{n-1} \cdots x_0)$ returns its context, that is the longest prefix $x_n \cdots x_{n-k+1}$ such that $\Pr(x_i \mid x_n x_{n-1} \cdots x_0) = \Pr(x_i \mid x_n \cdots x_{n-h+1})$ for all states x_i, and $1 \leq h \leq k$.

To better illustrate the concept of the context function, let us consider a stochastic process with state space $L = \{l_0, l_1\}$, and assume that the conditional probability distribution for the next state depends only on the current state l_j if $j = 0$, but on the previous history if $j = 1$, according to the following context function:

$$\mathcal{C}(x_n x_{n-1} \cdots x_0) = \begin{cases} l_0 & \text{if } x_n = l_0, \\ l_1 l_1 & \text{if } x_n = l_1 \wedge x_{n-1} = l_1, \\ l_1 l_0 l_0 & \text{if } x_n = l_1 \wedge x_{n-1} = l_0 \wedge x_{n-2} = l_0, \\ l_1 l_0 l_1 & \text{if } x_n = l_1 \wedge x_{n-1} = l_0 \wedge x_{n-2} = l_1. \end{cases} \tag{3.11}$$

The *context tree* is a compact representation of the context function, where each context, i.e., the codomain of $\mathcal{C}$, is encoded as a path ordered from the most to the least recently observed state when traversed from the root to the leaves [110]. In this representation, first-level nodes represent 1-length knowledge; thus, a context tree with first-level nodes only is a first-order MC. On the contrary, a complete k-height context tree implies a k-order full MC. Figure 4a shows the context tree of the simple example just described.

A VLMC is completely encoded by turning the context tree into a *probabilistic suffix tree* (PST) [110], which associates each context tree node with a probability distribution: the *next symbol distribution*, representing the conditional distribution for the next state given the context of the

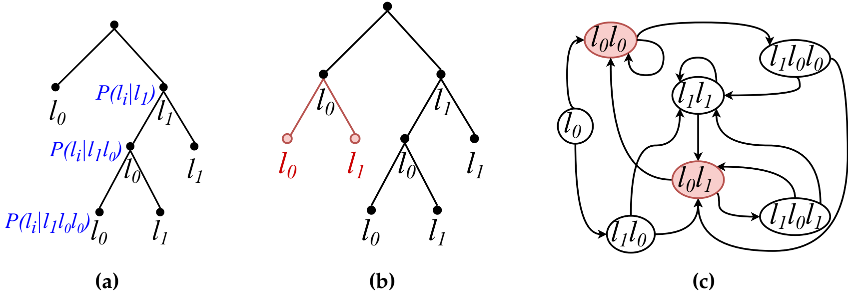


Figure 4: (a) Context tree enriched with examples of next-symbol probability distributions (in blue). (b) Minimal Markov hull, obtained from (a); the red elements in (b) are the states that need to be added to (a) in order to represent the process with a first-order MC (see Section 3.4.3). (c) First-order MC of the example with context in Eq. (3.11). For simplicity, the probability distributions for the next state are omitted.

node itself. Suppose, for instance, that we wish to find $\Pr(l_i \mid l_1 l_0)$ in the tree of Figure 4a: this is encoded in the probability distribution of the second-level node l_0 that is a direct child of the first-level node l_1.Notice that we used the symbology $l_0, l_1, l_2 \ldots$ in order to suggest that the states we consider are program locations of a software system; this notation will be further explained and used in all this dissertation.

3.4.3 Minimal Markov Hull

In this subsection, we discuss how to move from a PST representation of a VLMC to a first-order chain, using the algorithm that was first proposed by Ron et al. [110] and then described by Magarick [92]. With the help of an example, we show that a PST is not necessarily memoryless in general. Suppose that our traces are generated according to the context function of Eq. (3.11). In state l_0, the context is determined by the leftmost leaf of the tree. If the next state is again l_0, then the context stays unchanged because $C(l_0, l_0) = l_0$ (i.e., we forget the first occurrence of l_0). However, if the next state is l_1, then the previously forgotten state l_0 needs to be taken into consideration because $C(l_1 l_0 l_0) = l_1 l_0 l_0$. This

violates the memoryless property, a problem that would be solved by adding the extra contexts $l_0 l_0$ and $l_0 l_1$ to the tree, to keep track of which states were executed. The next-state probability distribution of the added nodes is obtained simply by copying that of their father's. In the illustrative example (see Figures 4a, 4b), the contexts $l_0 l_0$, $l_0 l_1$ are included in the tree, by adding l_0 and l_1 as l_0's offspring. In [92] it is proved that, in general, we can create a memoryless tree by adding all suffixes of all contexts in the context tree that have not been already considered. The number of leaves in the resulting context tree is no more than $O(|\tau|^2)$ where $|\tau|$ is the length of the deepest path from the root to a leaf of the original context tree.

The tree obtained by this "Markovianization" is called *minimal Markov hull*, and although the worst-case memory is still exponential, it often has considerably fewer states in practice than the complete tree, which encodes the full k-order Markov chain. Figure 4b shows the minimal Markov hull of the illustrative example, and it is composed of 9 nodes, rather than the 14 of the complete tree. Figure 4c shows the first-order MC of the illustrative example.

Chapter 4

Memoryless Process Modeling

In this chapter, basing on the publications in [68,71], we will present the formulation of the service demands/queuing networks estimators in the forms of non-linear, quadratic, and linear programming problems.

Our research evolved from only searching service demands values in QNs with routing probabilities known *a priori* [68], using a naive quadratic formulation, to searching the entire unknown topology [71], i.e. both routing probabilities and service demands, with a more elaborate linear programming formulation. All the experiments about mixed-integer optimization presented in this chapter are conducted by using the Python interface of the CPLEX optimization tool[1]. The dataset of queue length traces is collected by simulating the corresponding continuous-time Markov chain, using Gillespie exact algorithm [55]. The learning processes have been conducted on a laptop equipped with a 2.8 GHz Intel i7 quad-core processor and 16GB RAM.

[1] https://www.ibm.com/products/ilog-cplex-optimization-studio

4.1 Discrete-time QN model

We build our estimators basing on a discretization of the ODEs 3.6 in time using the well-known Euler numerical integration method, according to which $\dot{x}_i(t) \approx (x_i(t+\Delta t) - x_i(t))/\Delta t$ with a certain fixed little enought time step Δ. Denoting with $\overline{x}_i(k)$ the approximation of the k-th value of the variable x, the ODEs 3.6, representing the estimation of the average queue lengths dynamics, will become the following system of difference equations:

$$\overline{x}_i(k+1) = \overline{x}_i(k) - \Delta t \mu_i \min\{\overline{x}_i(k), s_i\} + \Delta t \sum_{j=1}^{M} p_{j,i} \mu_j \min\{\overline{x}_j(k), s_j\} \tag{4.1}$$

with $k \geq 0$ and $1 \leq i \leq M$ and with initial conditions $\overline{x}_i(0) = x_i(0)$.

4.2 Non-linear estimators

The discretization of the ODEs representing the approximate evolution of the QN's queue lengths could be used to constrain an optimization problem with the objective of minimizing the error between the measured values of queue lengths, i.e., $\tilde{x}_i(k)$, and those predicted by the ODE model, i.e., $\overline{x}_i(k)$, over the observation window H. The problem is initialized with the measured values of the queue lengths at the initial time step $\tilde{x}(0)$.

$$\begin{aligned}
&\text{minimize} \sum_{i=1}^{M} \sum_{k=0}^{H-1} e(\overline{x}_i(k) - \tilde{x}_i(k)) \\
&\text{subject to:} \\
&\qquad \text{Eq. } (4.1), \quad \text{for } 0 \leq k \leq H-1,\ 1 \leq i \leq M, \\
&\qquad \overline{x}_i(0) = \tilde{x}_i(0), \quad 1 \leq i \leq M.
\end{aligned}$$

The error e is a positive definite function of the difference between the measured and the predicted value; examples used in the proposed solution are the absolute error and the square error. The advantage of

the square error is that, since it is differentiable, many heuristic solving algorithms could be used, yet it has the disadvantage of tending to prioritize only the big error values, minimizing the error only based on them while neglecting over small error values. This issue makes the square error particularly unsuitable for the entire topology prediction when the problem is less constrained, and small prediction errors must be taken into account to obtain an accurate estimation.

The non-linearity is due to the presence of the minimum function in the Equation (4.1) and, in the case of topology estimation, also to the presence of the bilinear factors $p_{j,i}\mu_j$for $1 \leq j \leq M$, since both the probabilities and the service demands are decision variables that have to be determined by the optimization problem. In the following sections, estimators of both service demands and topologies are described and from time to time the mechanisms to avoid non-linearities and lead back to quadratic and linear formulations are presented.

4.3 Service demand estimator

In the first scenario, we seek at estimating the value of the service demands of the stations, which will become the decision variables of the problem, together with the predicted values of the queue lengths, while given both queue lengths measurements and routing probabilities.

$$
\begin{aligned}
&\underset{x,\mu}{\text{minimize}} \sum_{i=1}^{M} \sum_{k=0}^{H-1} (\overline{x}_i(k) - \tilde{x}_i(k))^2 \\
&\text{subject to:} \\
&\qquad \text{Eq. (4.1)}, \quad \text{for } 0 \leq k \leq H-1,\ 1 \leq i \leq M, \\
&\qquad \overline{x}_i(0) = \tilde{x}_i(0), \quad 1 \leq i \leq M.
\end{aligned}
$$

Here, we have used the square error function, minimizing the square of the difference between the predicted, i.e., $\overline{x}_i(k)$, and the measured queue lengths, i.e., $\tilde{x}_i(k)$, for each station i, $1 \leq i \leq M$.

4.3.1 Quadratic programming formulation

Regarding service demands estimation, the non-linearity caused by the minimum function in the Equation (4.1) is solved with an appropriate variable change, resulting in a quadratic programming formulation. The auxiliary variable $T_i(k)$ is set as follows:

$$T_i(k) := \Delta t \mu_i \min\{\overline{x}_i(k), s_i\}, k \geq 0,\ 1 \leq i \leq M,$$

$T_i(k)$ essentially represent the instantaneous discretized throughput of the station i. Thus, the quadratic programming problem is the following:

$$\begin{aligned}
&\underset{x,T}{\text{minimize}} \sum_{i=1}^{M} \sum_{k=0}^{H-1} (\overline{x}_i(k) - \tilde{x}_i(k))^2 \\
&\text{subject to:} \\
&\qquad \overline{x}_i(k+1) = \overline{x}_i(k) - T_i(k) + \sum_{j=1}^{M} p_{j,i} T_j(k), \quad \text{for } 0 \leq k \leq H-1, \\
&\qquad \qquad 1 \leq i \leq M, \\
&\qquad \overline{x}_i(0) = \tilde{x}_i(0), \quad 1 \leq i \leq M.
\end{aligned} \tag{4.2}$$

The problem above finds the optimal values of predicted queue lengths and instant throughput for each station and time-instant, $\overline{x}_i^*(k)$ and $T_i^*(k)$, with $0 \leq k \leq H-1$ and $1 \leq i \leq M$, the corrisponding values of the service rates can then be computed as follows:

$$\mu_i^* := \frac{\sum_{k=0}^{H-1} T_i^*(k)}{\sum_{k=0}^{H-1} \min\{\overline{x}_i^*(k), s_i\}}, \qquad 1 \leq i \leq M. \tag{4.3}$$

The average service demand of the station i is then the mean value of the corresponding exponential distribution, i.e., $1/\mu_i^*$.

Since predicted values of queue lengths are used in the equation of the next-step predicted ones and incorporated in $T_i(k)$, the error of the mean-field approximation propagates among future estimations of queue length, altering the resulting optimal value of the service demand.

4.3.2 Moving horizon estimation

We aim to apply the service demands evaluation during system execution, possibly together with a self-adaptation control strategy, according to the new paradigm of the evaluation at runtime already presented in Chapter (2). Thus, in this case, we assume to have only measurements of previous time instants. The moving horizon estimation strategy is based on a window of a fixed size of previous measurements. Thus, when starting to inspect the system, H time instants are needed to collect observations. The moving horizon estimation is based on iteratively repeating the following steps:

1. setting the QP formulation with the observed values of queue lenghts $\tilde{x}_i(k)$ at any station for all the previous H time instants, i.e., $0 \leq k \leq H-1$, the given (fixed) values of the routing probabilities $p_{i,j}$ among all stations $1 \leq i \leq M, 1 \leq j \leq M$, and the given (fixed) values of the server multiplicity s_i at each station i, with $1 \leq i \leq M$.

2. solve the optimization problem (4.2) finding the optimal values of the queue lengths $\overline{x}_i^*(k)$ and instant throughput $T_i^*(k)$ for every H time instants of the window, $0 \leq k \leq H-1$ and every stations $1 \leq i \leq M$, and computing the corresponding estimation of the service demand $1/\mu_i^*$ (see Equation 4.3)

3. shifting the observation window by discarding the oldest values of the queue lengths samples at each stations, i.e., $\tilde{x}_i(k)$, and including the newly measured ones.

4. return to step 1.

By applying this technique, one could promptly detect changes in the value of the service demands (possibly due to malfunctions, upgrades, components' deteriorations, etc,...), regardless of the assumption of the system being in the steady-state regime. This set up is similar to the model predictive control (MPC) approach developed in the Emilio et al. work in [73]. In this case, the purpose of the optimization problem is to find out the unknown values of the service demands, while in [73]

was to find the optimal configuration of the network topology in terms of either routing probabilities (e.g., load balancer case study) or service demands (e.g., elastic cloud case study) to obtain the desired threshold of some desirable properties (e.g., high throughput or short/little queue lengths).

4.3.3 Application to the load balancer case study

In this section, we will provide an example of our moving horizon modeling technique with a load balancer case study, showing the accuracy of the mean-field ODE estimator of the QN's dynamics [68]. The service demand value of station 1 is known and fixed to 1.0 and the ones of the stations 2-3 must be estimated. We model station 1 as a delay by setting its multiplicity $s_0 \geq N$. In order to simulate and validate the estimation, the network is parametrized with the values (randomly chosen) in the Table 1.

Table 1: Model parameters for the load balancer case study example

N	μ_1	μ_2	μ_3	s_0	s_1, s_2	$p_{1,2}, p_{1,3}$	$p_{2,1}, p_{3,1}$
5500	1.0	68.97	73.80	∞	20	0.5	1.0

Figure 5 depicts the queue length traces of the three stations, obtained from a simulation of a continuous-time Markov chain underlying the QN with parameters in Table 1. The traces are generated assuming exponential distributions at each station, in a period of 20 time units, sampled every $\Delta t = 0.01$; thus, obtaining a total of 2000 time steps. The plot evinces that the numerical solutions of the ODE model (dotted lines) represent an accurate estimation of the QN ground truth simulation values (solid lines).

With the chosen values of the parameters, the system does not reach the steady-state regime within the entire time course of 20 seconds. We applied iteratively the just described moving horizon estimation, with a window $H = 100$, solving 1901 optimization problems (4.2), and obtaining accurate estimations of the service demands of stations 2 and 3 with

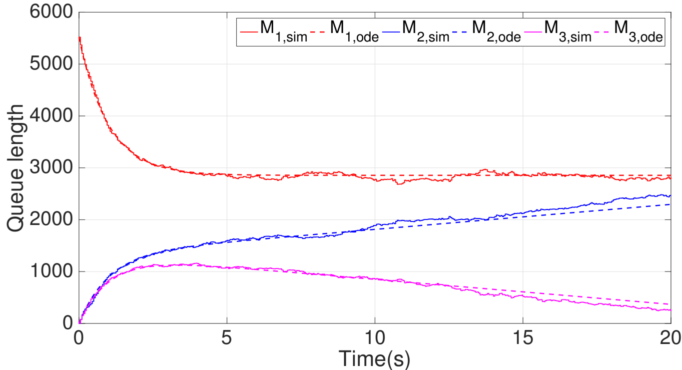


Figure 5: Simulation and mean-field solution for the queue lengths of the load-balancer running example [68].

average percentage errors across all iterations of 1.4%, and 2.6%, respectively, in a neglectable computing time on an ordinary laptop, i.e. 0.01 seconds.

4.3.4 Numerical evaluation

In this entire subsection, we refer to the numerical evaluation we conducted in [68], about the effectiveness and scalability of our moving horizon estimation (MHE) approach with traces from stochastic simulations of networks of different sizes and topologies with randomly generated parameters. Concerning accuracy and computational time, we strengthen our validation by comparing MHE with a previous recent approach that estimates service demands from queue lengths measurements only, namely, the Queue Length Maximum Likelihood Estimation (QMLE), in [133]. The scalability analysis is conducted on networks of increasing sizes and random topologies. The replication package of the experiments is available at `https://goo.gl/zNdr5f`.

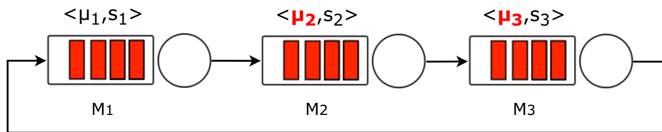


Figure 6: Queuing network topology used in the comparison experiments against QMLE.

Comparison with QMLE

The first variant of the QMLE algorithm presented in [133] uses nonlinear optimization on an almost exact closed-form expression of the QN evolution that is based on the BCMP theorem [15]; the second proposed technique relies on an approximated formula using the Bard-Schweitzer mean value analysis (BS-AMVA) [12, 116]. We compare our MHE solution with the latter variant of QMLE. Indeed, the exact solution is not suitable for online estimation because of its computational complexity, requiring execution times of the order of 10^4 seconds, even for small tandem networks. For a fair comparison, we decided to consider a tandem network with three stations, like the one in Figure 6, with a delay station M_1 and two stations with unknown values of the service rates μ_i, which must be found by the estimators.

The accuracy of the MHE strongly depends on the precision of the mean-field formula, which is more accurate as the system's size grows, i.e., the scaling factor K multiplying both the initial job population $x(0)$ and the vector of server multiplicities (see Background chapter, Section 3.3). The precision of QMLE depends on the accuracy of the BS-AMVA and the scaling factor they use to treat the load-dependent scenario (i.e., multi-server). The precision of the BS-ATVA approximation tends to increase with the network's population [12]. The scaling for the load-dependent case essentially consists of multiplying the value obtained by the load-independent closed-form expression with the factor approximating the minimum between the number of servers and the average queue length at the station. This introduces an error especially at low-utilization regimes when queue lengths are often less than the number of servers.

We take into account and consider in our analysis all those sources of approximation error by generating synthetic experiments varying both the *system size*, i.e. $k = 1, 2, 5, 10, 20, 50$, and the *utilization level* of the bottleneck station M_2, varying in $U_2 \in \{0.1, 0.3, 0.4, 0.6, 0.8\}$.[2] Service demand values are fixed arbitrarily with $\mu_1 = 1.0$, $\mu_2 = 27.0$, and $\mu_3 = 48.0$. The initial condition $x(0)$ was also fixed in each experiment to achieve the desired values of steady-state utilizations of the stations. We reproduced the experimental environment of the work [133], requiring 10^5 steady-state queue length samples, while for MHE we consider 100 non-overlapping intervals of length $H\Delta t$, using a different value of the window H for each experiment in order to obtain a fixed number of observed events, approximately 1500, maintaining a discretization step suitable to ensure an accurate enough ODE numerical solution, i.e., $\Delta t = 0.1$.

	$x(0) = (3,0,0)$ $H = 2347, U_2 \approx 0.10$		$x(0) = (9,0,0)$ $H = 688, U_2 \approx 0.30$		$x(0) = (12,0,0)$ $H = 521, U_2 \approx 0.40$		$x(0) = (19,0,0)$ $H = 353, U_2 \approx 0.60$		$x(0) = (26,0,0)$ $H = 262, U_2 \approx 0.80$	
K	QMLE	MHE	QMLE	MHE	QMLE	MHE	QMLE	MHE	QMLE	MHE
1	0.52	9.25 ± 1.03	1.37	9.63 ± 1.06	2.07	7.90 ± 1.01	3.40	6.58 ± 0.81	5.15	4.89 ± 0.69
2	448.30	4.13 ± 0.62	126.54	3.93 ± 0.58	67.18	4.20 ± 0.63	5.46	3.90 ± 0.56	2.33	3.59 ± 0.54
5	184.02	2.26 ± 0.33	60.41	3.02 ± 0.43	42.09	2.76 ± 0.38	8.78	2.07 ± 0.33	1.65	2.06 ± 0.34
10	92.29	1.65 ± 0.27	30.53	1.99 ± 0.31	23.18	1.82 ± 0.31	9.50	2.09 ± 0.30	3.89	1.50 ± 0.24
20	45.18	1.37 ± 0.21	15.01	1.13 ± 0.19	11.32	1.36 ± 0.18	6.41	1.36 ± 0.19	5.81	1.17 ± 0.18
50	18.67	0.74 ± 0.10	6.08	0.81 ± 0.14	4.57	0.78 ± 0.11	2.72	0.81 ± 0.12	5.17	0.73 ± 0.10

Table 2: Comparison between QMLE and MHE service demand estimators.

Table 2 reports the results of the comparison for each of the five experiments, with the corresponding utilization values of the stations U_2 and U_3 and the initial condition $x(0) = (x_1(0), x_2(0), x_3(0))$, to be multiplied by the corresponding scaling factor K. For each experiment, each station, and each value of the scaling factor K, we measured and report the accuracy of both QMLE and MHE, by computing the mean absolute percentage error between the estimated and the true value of the service demand. The values reported for the MHE are the 95% confidence intervals over the 100 independent samples of the moving horizon estimation technique.

The considerations arising from these results are the following:

[2] detailed results reporting statistics about station M_3 can be found in the replication package of this thesis

i) In the single-server, load-independent case, i.e., $K = 1$ QMLE tends to outperform MHE, especially at low-utilization regimes, becoming comparable for high utilizations, i.e., $U_2 = 0.6$ and $U_2 = 0.8$. Indeed when $K = 1$, we are far away from the deterministic regime approximation (see Background Chapter, Section 3.3). QMLE instead does not suffer from the multi-server scaling factor approximation of the BS-AMVA equation. Still, the worse solution provided by the MHE, i.e., a range of confidence of lenght ca. 21% around the estimated value of the service demands, is an acceptable performance for small queuing networks. This approximated value of 21% is obtained considering that in the worse case, the MHE estimate, say μ_{est}, is in the range $[\mu_{est} - 9.63 - 1.06, \mu_{est} + 9.63 + 1.06]$ with a probability of 95%.

ii) In the multi-server scenario $K > 1$, i.e., load-dependent, MHE considerably outperforms QMLE, especially at a low-utilization regime, when the correction of the BS-AMVA single-server to the multi-server using the scaling factor tends to be worse. For high utilization, indeed, the queues are mostly full and the servers are almost always busy; that is the reason why the approximation from dynamics of the multi-server, each serving μ_i requests per second, with one server serving $s_i \cdot \mu_1$ requests per second, where s_i is the multiplicity of the station i, is neglectable.

iii) As expected, larger values of K lead to more accurate estimates for the MHE method. The results show that this also holds for QMLE.

Scalability analysis

Here, we further analyze the scalability of the proposed approach by presenting the effectiveness and accuracy of our MHE method with queuing networks of increasing sizes and various topologies. For each network's size, intended as the number of its stations $M = 5, 10, 15, 20$, we generated 20 QNs with random parameters. The routing matrix was generated as a random stochastic matrix, ensuring a closed QN workload, the servers' multiplicities were picked uniformly in $\{20, \ldots, 50\}$, while the service rates were drawn from the interval $[10, 50]$. For each such QN,

	Errors				*Runtimes* (s)			
M	min	avg	95-th	max	min	avg	95-th	max
5	1.60	2.53	4.12	4.50	0.03	0.03	0.03	0.04
10	1.63	2.46	3.28	3.56	0.08	0.08	0.09	0.09
15	1.59	2.63	3.48	4.56	0.18	0.18	0.19	0.19
20	1.62	2.52	3.19	3.82	0.34	0.38	0.48	0.52

Table 3: Scalability analysis of MHE service demand estimator.

the initial number of jobs was chosen to obtain a steady-state utilization of the bottleneck station of about 0.8, using the approximate formulas presented in [78]; we computed the average service demand estimate across all stations and with 100 non-overlapping observation windows from a sample path, with $H = 200$.

Table 3 shows, for each value of M, the minimum, average, the 95-th quantile, and the maximum error across the 20 random QNs. The MHE method reveals to be accurate for any considered number of stations M without considerable differences in precision. The execution time, instead, grows almost linearly with the increasing number of stations in the network.

Notice that, even for very large systems (i.e., $M = 20$), the average time needed for obtaining a new estimation by solving one QP optimization does not exceed the fraction of a second in the worst case (i.e., 0.52s), making the method particularly suitable for online parameter estimation.

4.4 Queuing network estimator

In this section, based on [71], we will provide a topology estimator in terms of a non-linear programming problem and we present its numerical evaluation. Decision variables are routing probabilities as well as service demands and predicted values of the queue lengths, known parameters are the server multiplicities and the measurements of the queue lengths.

$$\begin{aligned}
&\underset{x,\mu,P}{\text{minimize}} \sum_{i=1}^{M} \sum_{k=0}^{H-1} \mid \overline{x}_i(k) - \tilde{x}_i(k) \mid \\
&\text{subject to:} \\
&\qquad \text{Eq. (4.1)}, \quad \text{for } 0 \leq k \leq H-1,\ 1 \leq i \leq M, \\
&\qquad \overline{x}_i(0) = \tilde{x}_i(0), \quad 1 \leq i \leq M.
\end{aligned} \tag{4.4}$$

In this case, we use the absolute error function, searching for the optimal vector of service rates μ and routing probability matrix P that minimize the absolute value of the differences between the predicted, i.e., $\overline{x}_i(k)$, and the measured queue lengths, i.e., $\tilde{x}_i(k)$. These absolute errors are summered up for each station i, $1 \leq i \leq M$, and over the discrete-time instants of the observation window H; the sum is then minimized. The optimization problem is initialized with given measured queue lengths at the initial time instant, i.e., $\tilde{x}(0)$.

4.4.1 Linear programming formulation

The non-linear QN optimization problem just presented could be extremely difficult to solve, hindering its usage in practice. It was already presented in the work of Incerto et al. [73] that the runtime of finding the solution of a globally optimal non-linear program similar to the one in (4.4) could be considerable respect to finding the solution to the corresponding mixed-integer formulation. The non-linearity in Equation (4.1), is due to the presence of both the minimum functions and the bilinear terms $p_{j,i}\mu_j$, with $1 \leq j \leq M$.

Here, we will provide an exact linear programming formulation of the problem (4.4), considering that both the queue length measurements and server multiplicity values are known for each station at any time

instant of the observation window.

$$\underset{\mu_i,\, g_{i,j},\, E_{i,k},\, \hat{E}_{i,k}}{\text{minimize}} \sum_{i=1}^{M}\sum_{k=0}^{H-1} \hat{E}_{i,k} \tag{4.5}$$

subject to:

$$E_{i,k} = -\mu_i \gamma_i(k) + \sum_{j=1}^{M} g_{j,i}\gamma_j(k) - \Delta\tilde{x}(k), \tag{4.6}$$

$$\mu_i = \sum_{j=1}^{M} g_{i,j}, \quad g_{i,j} \geq 0, \tag{4.7}$$

$$\hat{E}_{i,k} \geq E_{i,k}, \quad \hat{E}_{i,k} \geq -E_{i,k}, \tag{4.8}$$

$$1 \leq i,j \leq M, 0 \leq k \leq H-1,$$

sostituing $\Delta\tilde{x}(k) = \frac{\tilde{x}_i(k+1)-\tilde{x}_i(k)}{\Delta t}$, that is just an abbreviation for a compact rewriting of the equation, and $\gamma_i(k) = \mathbb{E}[\min\{s_i, \tilde{x}_i(k)\}]$, that exploits the measurements available in order to remove the non linearity due to the minimum function. The trick of $\gamma_i(k)$ is to consider the observed value of the queue length instead of the predicted one, removing the decision variable as an argument of the minimum and allowing $\gamma_i(k)$ to be computed and considered as an LP parameter.

Another key aspect of this formulation, making the optimization faster and more accurate, is no more integrating Equation (4.1) across the entire observation window but constructing the entire trajectory of the QN evolution by considering the sum of the prediction errors of a single step $E_{i,k}$ (see 4.6).

The last, most interesting, linearization solution is the variable change substituing the bilinear term $g_{j,i} = p_{j,i}\mu_j$ with a new variable, constrained such that the sum over all stations $1 \leq j \leq M$ of $g_{j,i}$ is μ_i, since P is a stochastic matrix (see 4.7). After the optimization, the estimated routing probabilities can be computed as $p^*_{i,j} = g^*_{i,j}/\mu^*_i$, with $1 \leq i,j \leq M$ where $g^*_{i,j}, \mu^*_i$ are the optimal $g_{i,j}$ and $\mu_{i,}$, respectively. Finally, (4.8) are the standard linear programming constraints for the minimization of the absolute value of a decision variable [26].

4.4.2 Numerical evaluation

For the numerical evaluation of our topology estimator, we generated random QNs of increasing size and complexity $M \in \{5, 10, 15, 20\}$ using the stochastic simulation framework Stochkit [112]. For each case, we generated 10 different queuing networks by drawing the entries of the routing probability matrix from a uniform distribution, service rates in the interval [1.0,100.0], concurrency levels of the M_i station, with $i > 1$, in the interval [1,64]. The station M_1 is instead the reference station, representing a think time of the user between one operation and the next one; its concurrency level must be greater or equal to the total population N, i.e., is the same as considering the reference station as an infinite server.

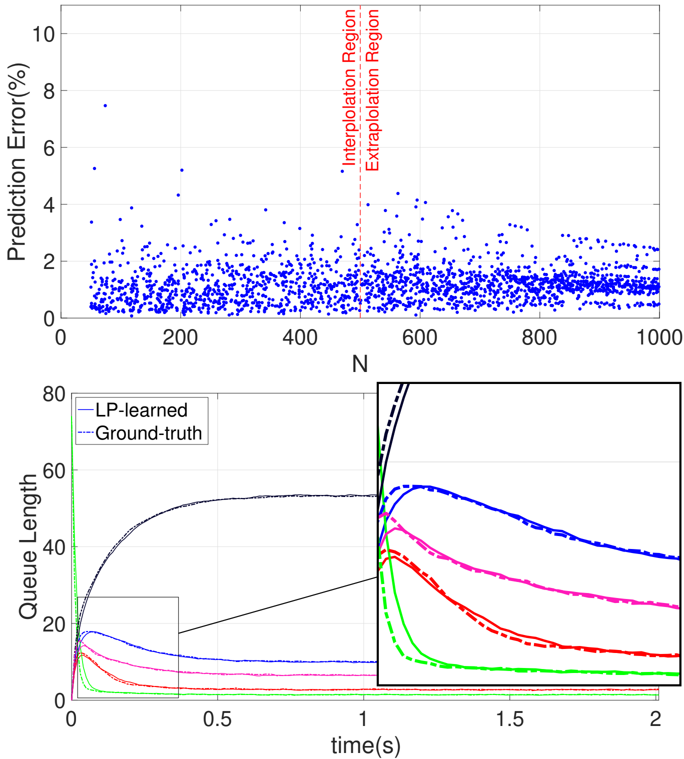
We generate the training set of each QN starting from 100 different initial population vectors, by choosing the initial number of jobs at each station randomly in [0,100], resulting in a total number of jobs circulating in the QN uniformly distributed between 50 and $100M$. Thus, for each of these initial conditions, we simulated the network 2000 times and collected the average queue length trajectories.

Furthermore, while the time horizon T of the simulations must be chosen big enough to be able to completely capture the dynamics of the system reaching the steady-state regime, and not too big to have simulation data too much large to be collected, the discretization step interval, i.e. Δt must be chosen small enough in order to avoid to loose relevant events among two observations [6]. We set $T = 2s$ and $\Delta t = 0.01$, collecting $H = 200$ points.

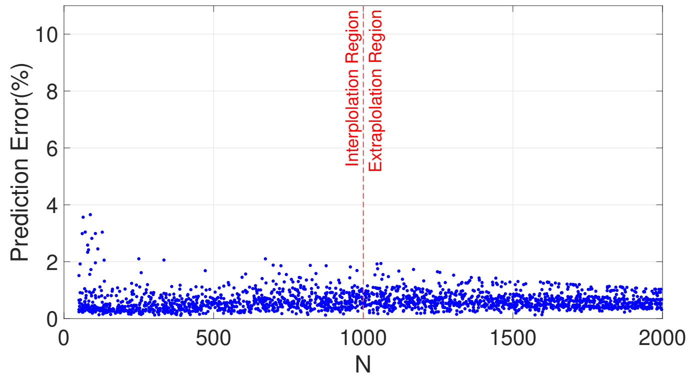
To this aim, we conduct two distinct *what-if* analyses, by first learning the QNs of the learning dataset presented above, then changing the initial vector, the size of the population, and the concurrency levels of the stations. We simulate both the learned model and the original one (i.e., ground-truth) with these changed parameters, and finally, we evaluate the difference between their dynamics, exploiting the following error function:

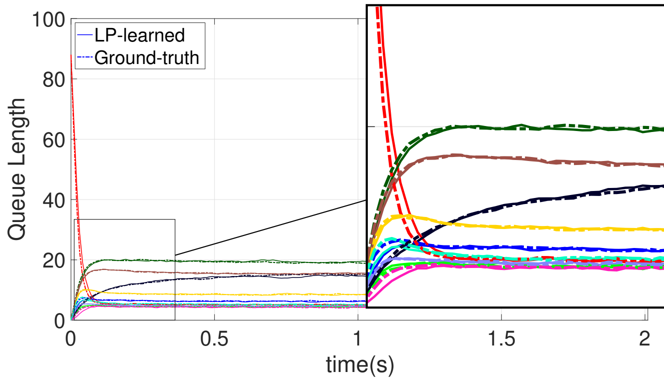
$$err_i = \frac{\max_{h=1}^{H} |\tilde{x}_i(h) - \overline{x}_i(h)|}{2N} \cdot 100. \tag{4.9}$$

This equation represents the absolute value of the maximum relative error of the station i among all the observation windows, with respect to the total number of jobs circulating the network (notice that since we are dealing with closed QNs N is to be considered constant). We refer the reader to [50] for a further detailed discussion on (4.9).

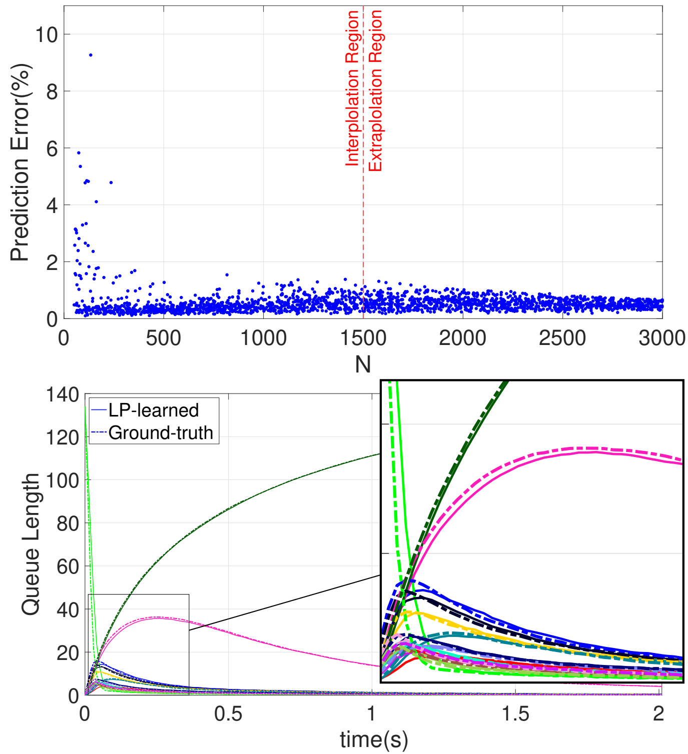


(a) M=5

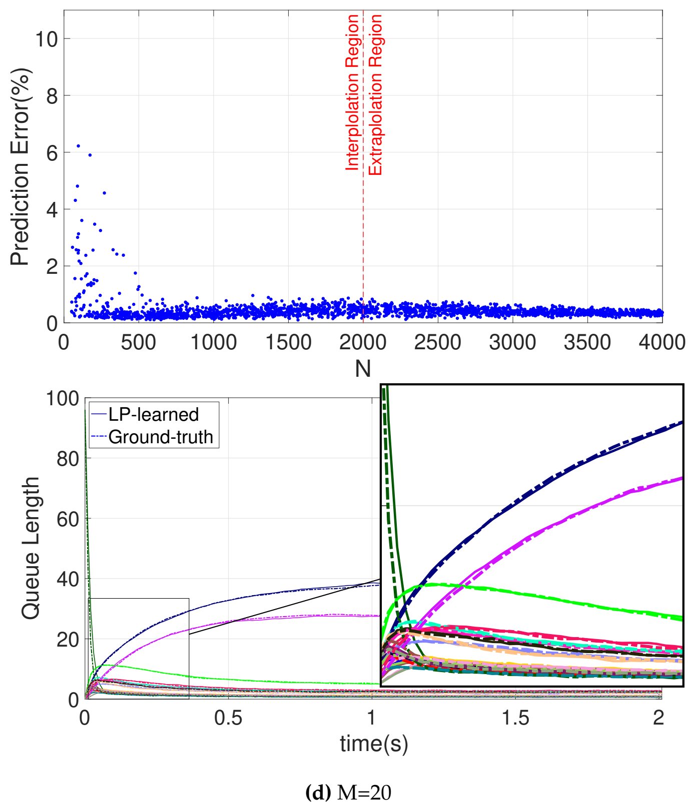




(b) M=10



(c) M=15



(d) M=20

Figure 7: Prediction error of the topology estimator of the what-if analysis with unseen population vectors

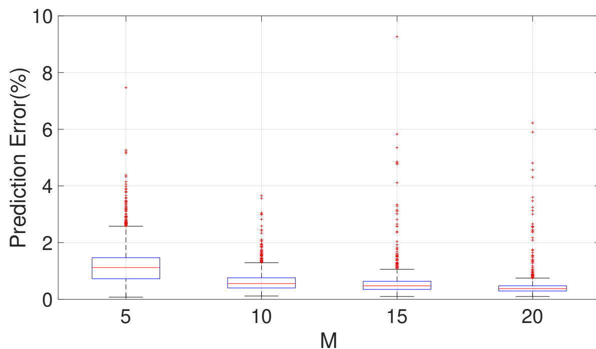
What-if analysis over client population

Here, we tested the learned QN with 200 new vectors of the initial population, different from those used for the learning phase. The initial number of jobs of each station was chosen between 0 and 200, with the total population N uniformly distributed between 50 and $200M$, discarding initial conditions used for learning. The comparison is then executed among the averages of the queue length dynamics of all these 2000 independent stochastic simulations of the QN learned by our approach with those produced by the ground-truth QN.

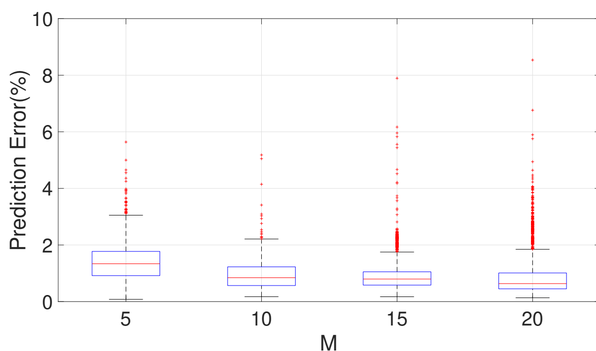
The first and the third rows of the Figure 7 report the scatter plots of the prediction errors for each considered QN and for each network size M. Prediction errors are mostly below 2% and even in worst cases, they are still below 10%, proving the high predictive accuracy of the proposed solution. Moreover, there is no significant difference between the interpolation region, where there are similar population vectors of those used for learning, and the extrapolation region, i.e., outside the range of initial conditions used for learning, remarking the high flexibility of our model to be simulated with unseen scenarios and predict the corresponding system's evolution.

The second and the fourth rows, instead, show the entire queue length evolutions of both the ground-truth model (dashed lines) and the QN predicted by the linear programming optimization (solid lines). The queue length dynamics plotted for each network size are those causing the maximum prediction errors in the what-if analysis, among all the different initial population vectors, i.e., 7.46% when $M = 5$, 3.65% when $M = 10$, 9.26% and 6.22%, when $M = 15$ and $M = 20$, respectively.

Finally, the box-plot in Figure 8a reports the summary statistics, in terms of the prediction errors, of the four scatter plots of Figure 7 and shows that there is no significant difference in terms of accuracy among the four different considered network's sizes; the average error decreases, possibly due to the fact that the optimization error becomes more constrained allowing for more precise estimation.



(a)



(b)

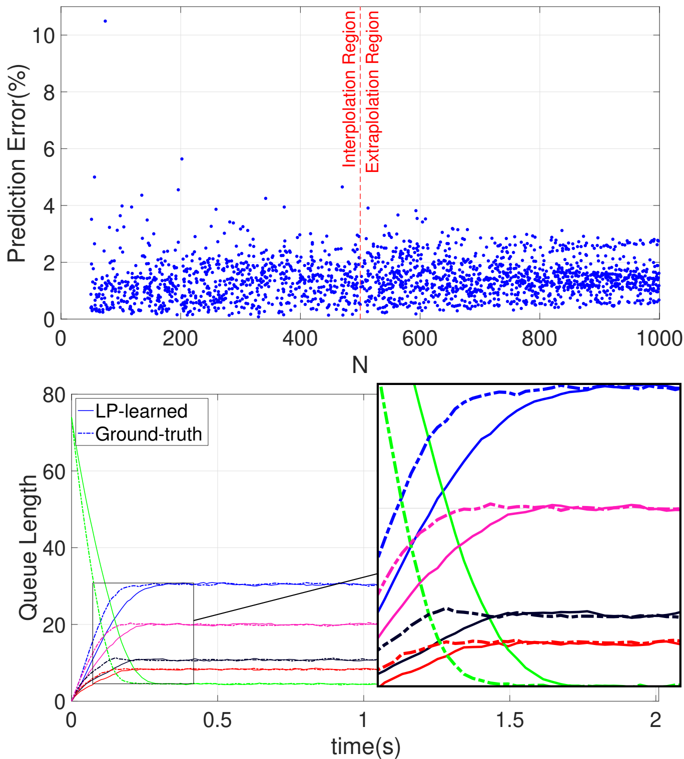
Figure 8: (a) Summary statistics on the prediction error for the experiments of Fig. 7, and (b) for the experiments of Fig. 9. In each box-plot, the line inside the box represents the median error, the upper and lower side of the box represent the first and the third quartiles of the observed error distribution, while the upper and lower limit of the dashed line represent the extreme points not to be considered outliers. The red dots depict the outliers.

What-if analysis over concurrency level

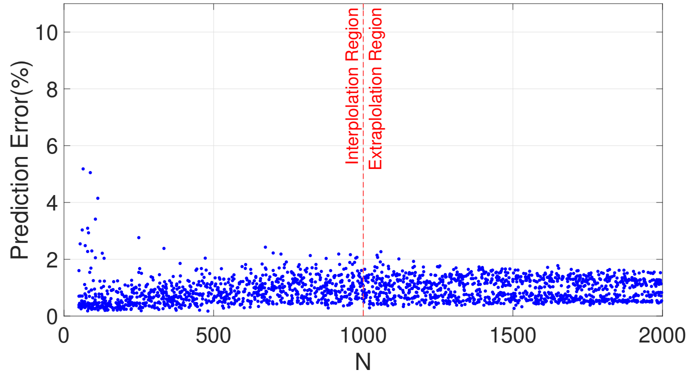
In this what-if analysis, we tested the prediction power over modification of the stations' concurrency levels. For each generated QN and for each station, except the reference $M1$, we reassigned a new value of concurrency level, picked at random uniformly in $[1, 64]$. Then we compare

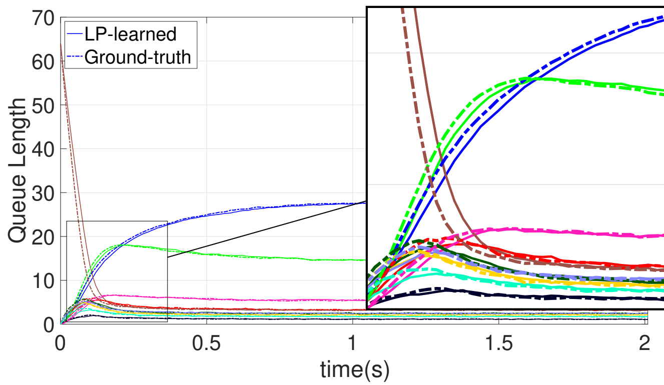
the resulting queue length dynamics resulting from the simulation of the LP-learned QN, with those resulting from the ground-truth model, both with these new values of the stations' concurrency levels. In this case, we consider as initial conditions the same ones of the what-if analysis over client population just presented. This basically means that this second what-if analysis is conducted on top of the other one, possibly combining the sources of errors of both.

The first and the third rows of the Figure 9 report the scatter plots of the prediction errors for each considered QN, and for each network size M. Even in this case, prediction errors are considerably low, mostly

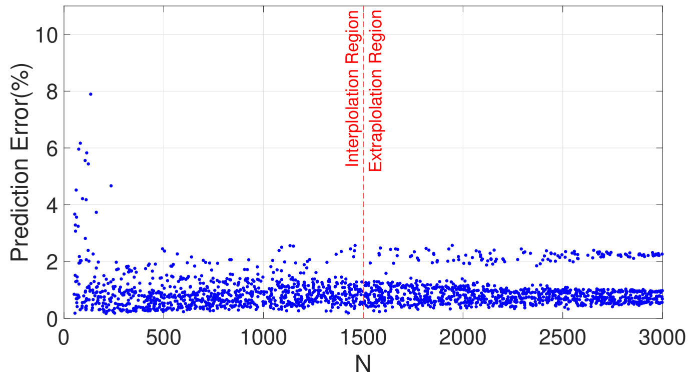


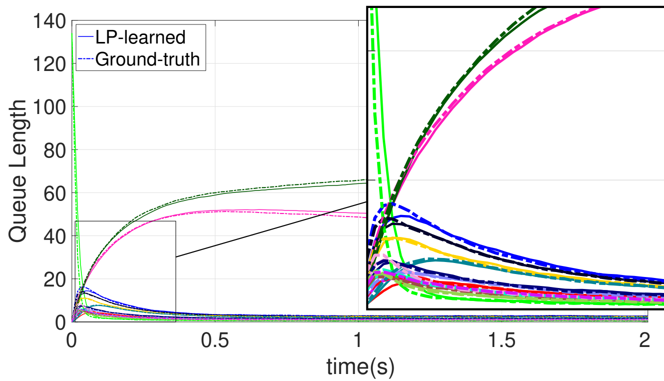
(a) M=5



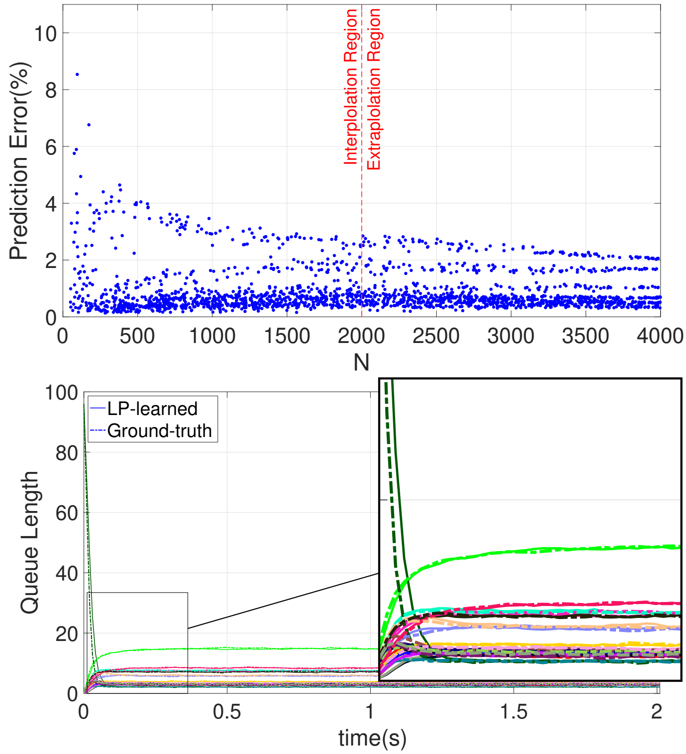


(b) M=10





(c) M=15



(d) M=20

Figure 9: Prediction error of the topology estimator of the what-if analysis with different server multiplicities

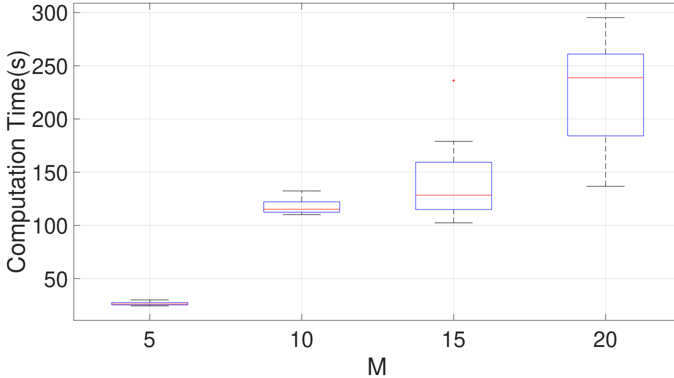


Figure 10: a) Summary statistics on the computation time needed for learning all the 40 QN models (i.e., 10 for each model size) used for the experimentation in Figures 7 and 9.

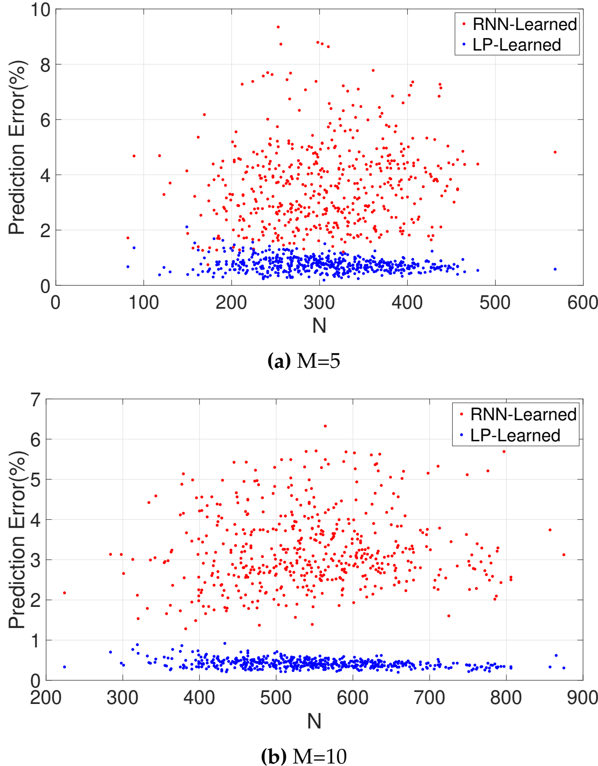
below 4% and always below 10%. In the second and fourth rows of the Figure 9, instead, are reported the comparisons in single evolutions of the queue lengths for each network size-the ones with worst prediction errors, i.e., 10.49%, 5.18%, 7.89%, 8.53, for $M = 5$, $M = 10$, $M = 15$, $M = 20$, respectively.

Although in this case there is a slightly bigger deviation between the LP-learned queue-length and the ground-truth evolutions, the dynamics are well predicted since the LP-learned curves well follow the ground-truth ones. This small increase in the errors can be justified by both the fact that here the variations in the conditions are twofold, i.e., both the initial vector of the population and in server multiplicities; and by the fact that changing the number of servers, which was constant in all the traces used for learning, is a greater structural change than the previous one on the population.

The box-plot in Figure 8b reports the summary prediction errors of the four scatter-plots of the Figure 9. Also in this case, there is no significant difference in terms of accuracy among the four different considered network sizes, and the average error decreases.

Figure 10 presents the summary statistics of the solution time of the

40 optimization problems of this experimentation, grouped by the size of the corresponding QN. Thanks to the linear formulation, the average solution time grows almost linearly with the size of the learned model.



(a) M=5

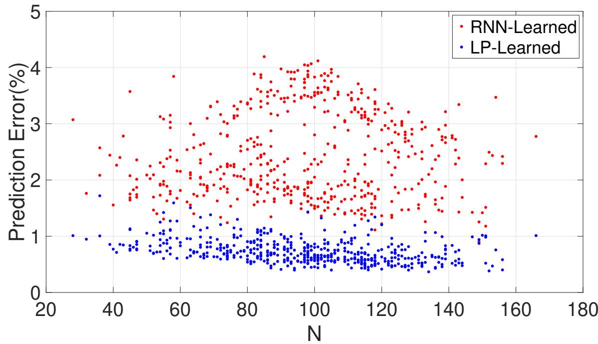
(b) M=10

Figure 11: Comparison between the predictions errors of the RNN-learned models and the LP-learned, in the what-if over populations.

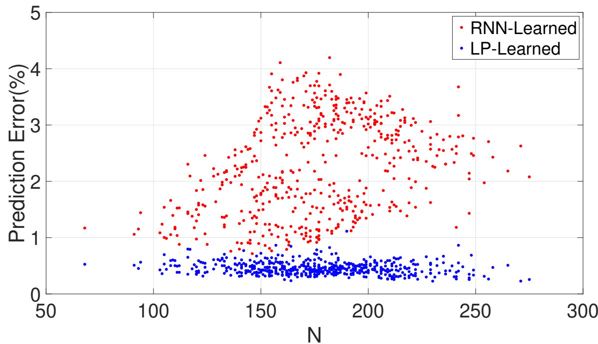
Comparison with state-of-the-art

In this section, we conduct a comparison with the recursive neural network (RNN) estimator recently presented by Garbi et al. [50]. We specifically refer to the RNN approach [50] since is the unique topology estimator of the literature, i.e., that estimates both the service demands

and routing probabilities, based only on the knowledge of queue length measurements and server multiplicities. To be as fair as possible, we decided to repeat exactly the same experiments conducted by the RNN method, using their replication package, with 100 randomly generated initial conditions, $T = 10s$ and $\Delta t = 0.01$. Yet, each QN is simulated again to obtain the traces and the necessary values to punctually compute each $\mathbb{E}[\min\{s_i, \tilde{x}_i(k)\}]$ that cannot be derived from the already averaged queue data in the replication package.



(a) M=5



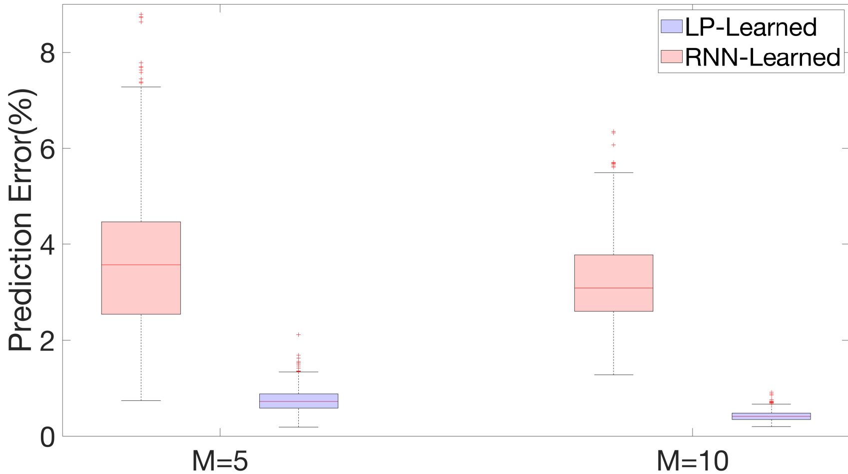
(b) M=10

Figure 12: Comparison between the predictions errors of the RNN-leaned models and the LP-learned, in the what-if over concurrency levels

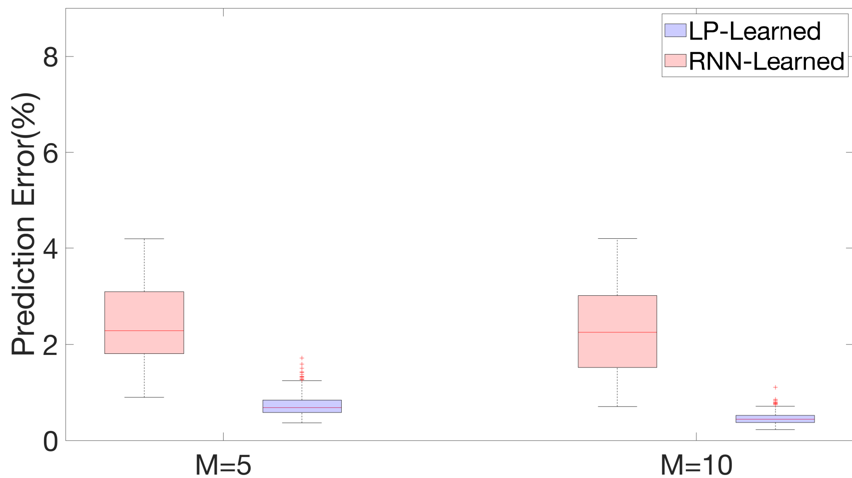
Figure 11 and Figure 12 present the resulting prediction errors of both

the approaches in the what-if analyses over initial population vectors and concurrency levels, respectively, with network size $M = 5$ (a), and $M = 10$ (b). The results show that the LP-based approach (blue points) outperforms the RNN-based one (red points) for all the evaluated scenarios, reporting consistently smaller prediction errors.

This is also confirmed by the summary statistics of Figure 13. These results strengthen the effectiveness and generality of the proposed approach that has proved able to produce more accurate QNs models, with a higher prediction power, despite having been learned in a less constrained setting. Indeed, in [50], the main diagonal of the routing probabilities matrix is assumed to be known, while in our tests we do not need this assumption.



(a) RNN-Learned vs LP-Learned accuracy in the same exact what-if over population (See Fig. 11)



(b) RNN-Learned vs LP-Learned accuracy in the same exact what-if over concurrency levels (See Fig. 12)

Figure 13: Summary statistics on the prediction error for the experiments of Fig. 11 and Fig. 12 with learning methods RNN and LP, respectively.

Chapter 5

Memoryfull Process Modeling

This chapter is based on the pubblication of Incerto et Al. [70].

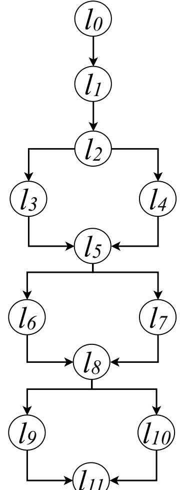
Here, we present the modeling process of those software systems showing variable-length memory, as programs source code executions in particular. These processes are learned from traces of their execution, captured from the runtime of an instrumented version of the source-code with inputs sampled by given probability distributions, resulting in a variable-length Markov chain (VLMC). The solution we propose, i.e., *ProgramToVLMC* [70], is a domain-specific adaptation of popular general-purpose algorithms that exploits static knowledge of the program's control flow graph to optimize the runtime and the memory allocation of the learning procedure. We demonstrate the validity of the proposed techniques with a numerical evaluation on several benchmarks.

5.1 Learning the VLMC of a program

In this section, it is presented how to learn a VLMC model from sample traces of program execution, using our specialized and optimized algorithm *ProgramToVLMC*.

```
    public void syntheticP(){
l0          Random r=new Random();
l1          int[] b=new int[3];
l2          if(r.nextInt(100)< 20)
l3              b[0]=1;
            else
l4              b[0]=0;
l5          if(r.nextInt(100)< 80)
l6              b[1]=1;
            else
l7              b[1]=0;
l8          if((b[0]==0 && r.nextInt(100)<40)||
            (b[0]==1 && r.nextInt(100)<90))
l9              b[2]=1;
            else
l10             b[2]=0;
l11     }
```

(a)



(b)

Figure 14: A simple synthetic program (a) with its control flow (b)

5.1.1 Input programs definition

The input of the learning algorithm must be a bounded program, i.e. a finite program whose termination is guaranteed and thus whose loops iterations can be bounded up to a certain number, written in an imperative programming language. Moreover, we assume the program has undergone standard transformation techniques, such as loops unwinding and function inlining, resulting in an equivalent loop-free [59] version made up of only basic statements, sequential compositions, and conditional statements [48]. We also consider as input for the learning process the program's *full* control-flow graph[1] (CFG) $G = (L, E)$, where L is the set of program locations and E the set of transitions.

Figure 14a presents a synthetic simple program that will be used through this section as an illustrative example to help the reader understand the entire learning algorithm, and Figure 14b shows its control-

[1]The graph whose nodes are all the program's instructions and the edges are control flow transitions among them [54].

flow graph. The program consists of a simple Java method with three conditional branches in sequence, whose evaluations are stored in the b array. Capturing the stochastic behavior of this program could be challenging due to main difficulties:

- the program is not memoryless, in the sense that `b[2]` depends on the evaluation of `b[0]`, which is statements behind;
- the memory has variable length, i.e., `b[0]` and `b[1]` are memoryless conditions while `b[2]` is not.

Thus, it is clear that the behavior of this program cannot be fully captured by a first-order Markov chain, since the next-symbol probability of the third condition l_8 depends on the previous history, and it would be equal to 0.4 if the first condition l_2 was evaluated true and l_3 occurs while would be 0.9 if l_2 was evaluated false and l_4 occurs.

5.1.2 VLMC of a program

The VLMC of a program P with *n* PLs, as the output of the learning algorithm, can be formally defined as the tuple $(\mathcal{C}, \mathcal{P})$ where:

- $\mathcal{C} : \bigcup_{i=1}^{n} L^i \to \bigcup_{i=1}^{k} L^i$ is the context function, which associates a context of length at most k (the maximal order of the VLMC) with each sequence of PLs (which have length at most n).
- $\mathcal{P} : Im\{\mathcal{C}\} \to \mathbb{R}^L$ is the next-symbol probability distribution, associating each context with a conditional probability distribution over all the possible next PLs.

5.1.3 The algorithm: *ProgramToVLMC*

Overview

Algorithm 1 describes the overall process of learning VLMC of a program from sample traces of its execution, collected from independent runs of an instrumented version of the program; each trace representing the effectively executed path of the full control flow graph G. Inputs

Algorithm 1: Fit a VLMC for program P

```
1  ProgramToVLMC (Traces, G, n_min, α)
     inputs: The set of traces; the control-flow of P; the minimal
             occurrences of a context; the pruning factor
     output: A VLMC for P
2    Global sa ← buildSuffixArray(Traces)
3    PST ← ∅
4    foreach PL l_i ∈ G do
5        LocationPST ← growPST(l_i)
6        prune(LocationPST)
7        PST.addChild(LocationPST)
8    end
9    return PST
```

of the algorithm are also two configuration parameters, which influence both the accuracy and the memory allocation, in terms of the order of the resulting VLMC: the minimal number of occurrences n_{min} of a context in the traces in order to be considered for the generation of the PST, and the pruning factor α. These are detailed later.

In line 2, the algorithm builds the suffix array [86] of the traces, which will be used in the next stages to compute the number of occurrences of each considered context, and thus the probability distributions. Then in line 5, for each PL l_i, the algorithm constructs the complete sub-PST rooted at l_i, namely the *location PST* (See the function `growPST 2` explained in the next section). This *location PST* contains all the prefixes of the CFG paths ending in l_i, thus representing all the possible memory dependencies for that PL.

The pruning phase in line 6 will reduce the complete location PST, by retaining only its statistically significant nodes (see the pruning phase explained in Section 9). Afterward, this tree is added to the final PST output of the whole learning algorithm (line 7).

To this end, building the VLMC by exploiting the structural information of G has one practical advantage: it allows pruning each location PST earlier than general-purpose algorithms, thus reducing the peak

Algorithm 2: Recursively create the location PST of PL s_c by backwardly enlarging the considered context

```
1 growPST (s_c s_{c-1} ... s_0)
      inputs: The input context s_c s_{c-1} ... s_0
      output: The candidate location PST of the PL s_c
2     pstNode ← new Node(s_0)
3     pstNode ← Pr(s_i|s_c s_{c-1} ... s_0), ∀s_i|s_c → s_i ∈ E
4     foreach PL s_j|s_j → s_0 ∈ E do
5         if (isObserved(s_c s_{c-1} ... s_0 ∪ s_j, n_min)) then
6             pstNode.addChild(growPST(s_c s_{c-1} ... s_0 ∪ s_j))
7         end
8     end
9     return pstNode
```

memory requirement. The other approaches of Maechler et al. and Gabaldinho et al. [47,91] explore all contexts, producing the entire complete PST and then prune only at the end. See Section 5.2 for a numerical evaluation of this aspect.

Growing the PST

Here, we will describe the core-function of the PST construction, namely `growPST`, see Algorithm 2. Denoting a general context with $s_c s_{c-1} \dots s_0$, where s_c is the most recent and s_0 the least recent PLs, the algorithm creates the location PST for s_0 by recursively visiting each possible previous *ancestors* in G, i.e., each PL s_i for which it exists a path from s_i to s_0; and computes the path's corresponding next-symbol probability distribution.

We describe the basic and the recursive step separately, to better illustrate the `growPST` function 2, and we apply all the algorithmic steps to our illustrative example, generating the Location PST of the node l_5 of the CFG in Figure 14b.

The *base step* represents the first call to the function `growPST`, made by the Algorithm 1 at line 5, where the *current* sub-tree "in construction" is initialized. In our example, we have as initial argument the context

l_5, and at line 2 of this base step is allocated the node corresponding to $s_0 = l_5$, as the root of the Location PST of l_5. In line 3 the next-symbol probability distribution of the actual context, i.e. l_5 is computed, namely $Pr(s_i \mid l_5) \, \forall s_i \; s.t. \; l_5 \rightarrow s_i \in E$, and associated with the just allocated node.

The *recursive step* starts at line 4 and it is called for every predecessor of the actual context, enlarging the contest itself with these ancestor PLs. In our example, for the first recursion call these ancestor Pls would be l_3 and l_4, and thus the arguments $l_5 l_3$ and $l_5 l_4$. Moreover, in line 5, we limit the exploration to only the contexts observed at least n_{min} times in the traces, according to the algorithm's configurations. Once all the recursions are closed, the function `growPST` returns in line 9 the Location PST related to the initial calling context, i.e., l_5, to the Algorithm 2.

Let us illustrate the recursive step in more detail by assuming that it operates on the context $s_c s_{c-1} \ldots s_0 = l_5 l_4$. As a first step, on line 2, the node corresponding to l_4 is created. In line 3, the next-symbol probability distribution $Pr(s_i \mid l_5 l_4)$ is calculated and stored in the node of the location PST just created. Once again, in line 4, the recursion starts for all the PLs directly connected to $s_c = l_4$ and whose context has been observed at least n_{min} times in the collected traces: in this case, we process l_2 only (see the CFG of Figure 14b). This implements the backward visit of the prefix of G ending in l_5 since, at each recursion, the considered context is enlarged and older dependencies are examined. The recursion ends with the initial node of the CFG, i.e., l_0, because it has no incoming arcs. Thus, the control returns to the previous recursion which adds the location PST rooted in l_4 as its direct child (line 6).

Learning the next-symbol distribution

The next-symbol probability distribution is computed in line 3 of the Algorithm 2 with the classical frequence approach: given the context $s_c s_{c-1} \ldots s_0$, the probability of visiting the location s_i, i.e., $Pr(s_i \mid s_c s_{c-1} \ldots s_0)$, can be estimated with the number of sequences containing $s_i s_c s_{c-1} \ldots s_0$ divided by the total number of sequences containing $s_c s_{c-1} \ldots s_0$.

It is worth notice that even in this step we can benefit from the structural information about the program by limiting the computation of the next-symbol distribution only on the outgoing labels from s_c (of size at most 2), instead of the entire state space L, as it should have been done in the case of general-purpose learning [92].

Pruning the tree

During the pruning phase, the complete Location PST is reduced to eliminate contexts that do not bring a statistically significant contribution to the memory [30]. This is done by comparing the next-symbol probability distributions of each leaf nodes of the PST with the one of its father, pruning the leaf when the two distributions are considered statistically equivalent, i.e., when their *Kullback-Leibler (KL) divergence* [80] is less than the user-specified cutoff $\mathcal{X}^2_{|L|-1;1-\alpha}/2$, where α is the pruning parameter of Algorithm 1.

Small α values produce smaller PSTs, less memory occupation, and coarser approximations, while high α values lead to more accurate reproductions of the program behavior, with a greater memory occupation. This pruning process goes on with the tree leaves until all of them will be already evaluated; notice that an internal node might become an unevaluated leaf if all its children are pruned. The pseudo-code describing this algorithmic step is not reported since there are no substantial modifications with respect to the state-of-the-art pruning algorithms [92].

ProgramToVLMC example

In order to see how this algorithm works in practice, Figure 15 depicts the PST learned for the Program 14a with CFG of Figure 14b. In particular, below each node is reported the next-symbol probability distribution of its corresponding context. Although for the sake of readability in Figure 14b the values are truncated to the first decimal digit, during the learning process they have been considered as 64-bit floating-point numbers. For collecting the input traces, the program ran 10^5 independent times. The red-crossed nodes identify the elements of that location

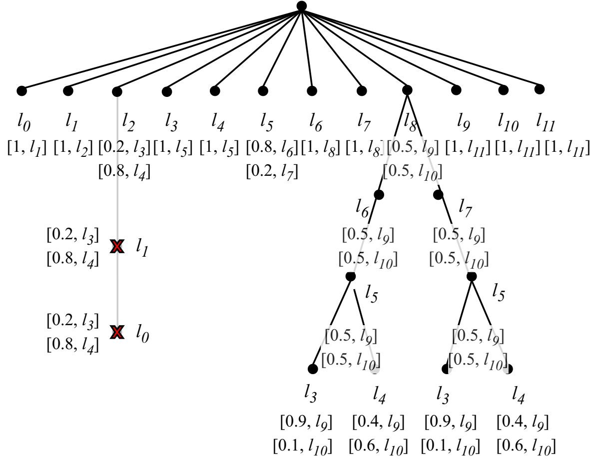


Figure 15: Learned PST for the Program 14a. The red crossed nodes identify pruned nodes in the location PST of l_2.

PST that are eliminated during the pruning phase in the location PST of l_2. In particular, in this example, starting from the leaf l_0 the next-symbol probability distribution of the context $l_2 l_1 l_0$ (shown as a label of that node) is compared with that of its father context $l_2 l_1$. Since they are equal, the corresponding Kullback-Leibler divergence is zero and l_0 is removed from the tree. Then, the same process is recursively iterated by comparing the probability distribution of the context $l_2 l_1$ with that of the context l_2, which similarly causes the removal of l_1 from this location PST.

The resulting PST, as expected from Subsection 5.1.1, indicates that the program exhibits variable-length memory; e.g., PL l_2, corresponding to the first branching condition, has a memoryless distribution towards its reachable locations, l_3 and l_4, with numerical values that are consistent with the one induced by the program. Conversely, l_8, corresponding to the condition of the last branch $b[2]$, has a next-symbol distribution of memory of length 3, since it depends on the evaluation of $b[0]$. Indeed, by traversing the PST we see, for example, that $\mathcal{P}(l_9 \mid l_6 l_5 l_3) = 0.8997 \approx 0.9$ and $\mathcal{P}(l_9 \mid l_6 l_5 l_4) = 0.3997 \approx 0.4$.

5.1.4 Computational complexity of *ProgramToVLMC*

To study the complexity of *ProgramToVLMC*, we take $\alpha = 1.0$ and $n_{min} = 1$, since this leads to the largest possible PST. In addition, we focus only on Algorithm 2 because it is the computational bottleneck. For this analysis, we let r and n denote the number of sequences and maximal length of a path of the CGF, respectively (i.e., we have at most n PLs). The creation of the suffix array takes $O(rn)$ time [77]. The complexity of Algorithm 2 is $O(n^2)$ for each path of length n, since it resembles the basic algorithm for the suffix tree construction [60]. In addition, considering that in the worst case the algorithm generates a new path for each simulation run, we have a worst-case cost of $O(rn^2)$ for the creation of the whole context tree. For each node, the cost for computing the next-symbol probability distribution is $O(\log(rn) + n)$ for counting through the suffix array [86]. Hence, Algorithm 2 takes $O(rn^2(\log(rn) + n))$ time. The pruning phase does not worsen the computational complexity because it only involves a backward visit of the PST, paying a constant price for each node (i.e., the comparison of the corresponding next-symbol probability distributions involve at most 2 outgoing PLs). Therefore, overall, *ProgramToVLMC* takes $\boldsymbol{O(rn^2(\log(rn) + n))}$ time.

The state-of-the-art approaches in [30] and [110] take time $O(rn\log(rn))$ and $O((rn)^2n)$, respectively.[2] Thus, we have:

$$O(\text{Alg. [30]}) < O(\textit{ProgramToVLMC}) \leq O(\text{Alg. [110]}) \tag{5.1}$$

The leftmost inequality is implied by the relationship

$$rn\log(rn) < rn^2\log(rn) < rn^2(\log(rn) + n).$$

The rightmost inequality derives from $O(rn^2(\log(rn)+n)) < O(rn^2(rn+rn)) = O((rn)^2n)$. The experimental evaluation conducted in Section 5.2 numerically supports this asymptotic relationship. Moreover, it shows that although *ProgramToVLMC* does not improve the theoretical complexity of the learning process, thanks to the heuristics defined in this

[2] For the latter, the complexity bound is not explicitly stated in the original paper but can be derived from [17].

section, we are able to extract VLMCs generated by benchmark programs that would not be analyzable through the currently available state-of-the-art implementations.

5.2 Numerical evaluation

In this section, we numerically evaluate the accuracy of *ProgramToVLMC* on the 9 considered benchmark programs. We conduct a sensitivity analysis with respect to the configuration hyper-parameters, and we compare *ProgramToVLMC* with the state-of-the-art general-purpose VLMC learning techniques. All the presented experiments are conducted with a prototype Java implementation of both the *ProgramToVLMC* algorithm and the benchmarks, on a Linux machine with 48 cores and 60GB of memory. The benchmarks are the following:

- **Double Loop [35]**: It is composed of two loops executed in sequence, each iterating a number of times random between 0 and 30. This is considered as our base case because it can be formally proven as memoryless and hence with this benchmark we can evaluate the ability of our approach to producing a VLMC of proper order.

- **Nested Loop [35]**: It is composed by two nested loops. For each loop, the number of iterations is random between 0 and 20.

- **Bubble Sort, Insertion Sort and Quick Sort [39]**: For these, the input is an array of size 9 with random 32-bit integer values.

- **Binary Search [39]**: It implements the binary search looking for a random generated 32-bit integer y in an array of 100 elements uniformly distributed in the 32-bit integer set. The value of y is chosen such that the probability of finding it in the array is 20%.

- **Square root Newton [117]**: It computes the square root value of a random variable between 0 and 100000 in the double-precision floating-point set, using Newton's method.

- **Euclid Algorithm [117]**: It computes the greatest common divisor, using Euclid's algorithm, of two variables, uniformly distributed between 0 and 200.
- **Miller-Rabin [97]**: It implements the Miller-Rabin primality test. Given a uniformly distributed integer variable x between 0 and 1000 it checks if x is prime or not. Unlike the other programs, it is a randomized algorithm, i.e., it employs probabilistic statements as part of its logic.

We generated the learning traces of each benchmark using uniform distributions of the inputs. However, it is also allowed to use any other probability distributions within the approach. Furthermore, the models learned with our approach are compared with the ground-truth, i.e., the empirical behavior of the program exercised many independent times with input values drawn from a uniform distribution, using as a measure of performance the density function of the number of visited program locations until reaching the program termination.

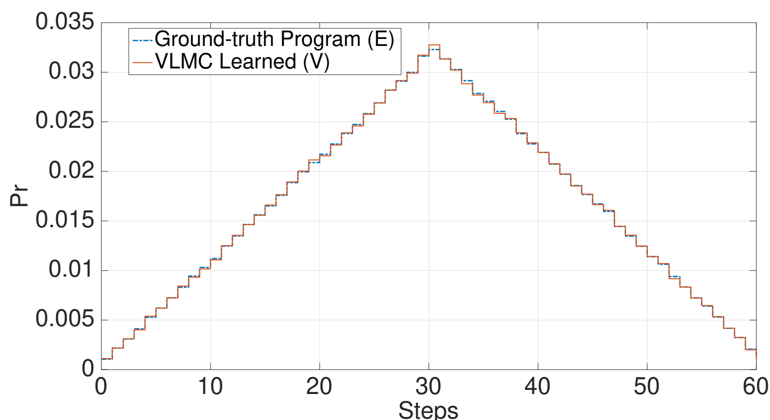
To obtain statistically robust estimates, while executing independent samples from each benchmark and obtaining independent batches, we stop to collect traces when the KL divergence of two successive distributions was less than a threshold (i.e, 10^{-5}). The KL test is meaningful since it is commonly employed to quantify the information loss when one distribution is used in place of another [18]. We remark that a divergence tending to zero indicates high fidelity of the approximating distribution.

5.2.1 Accuracy evaluation

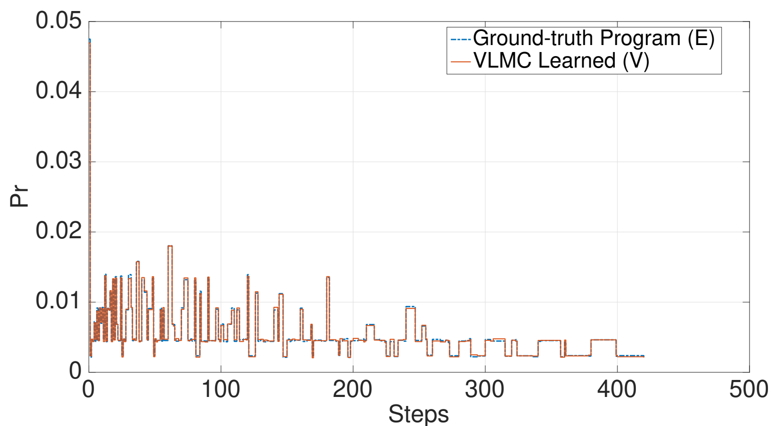
In this subsection, we evaluate the accuracy of *ProgramToVLMC* with configuration parameters $\alpha = 1$ and $n_{min} = 1$, comparing the simulation of the learned PST, i.e., V, with the ground-truth distributions,i.e., E, of the number of steps to the termination. The number of runs of both the simulations of E and V is the same and it is determined at least to statistically well capture the program's dynamics, as explained previously.

In the Figure 16 numerical evaluation of the accuracy of the proposed approach is reported. The results show that with a sufficient number of

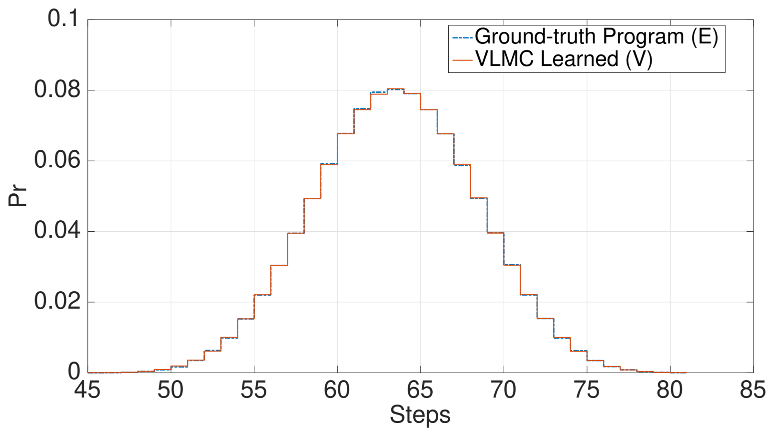
samples, i.e., traces generated by the adequate number of runs, *ProgramToVLMC*follows the ground-truth evolutions with extremely high accuracy, resulting in overlapping functions of the number of steps to termination in all the considered cases, and KL divergence values always below 10^{-4}.



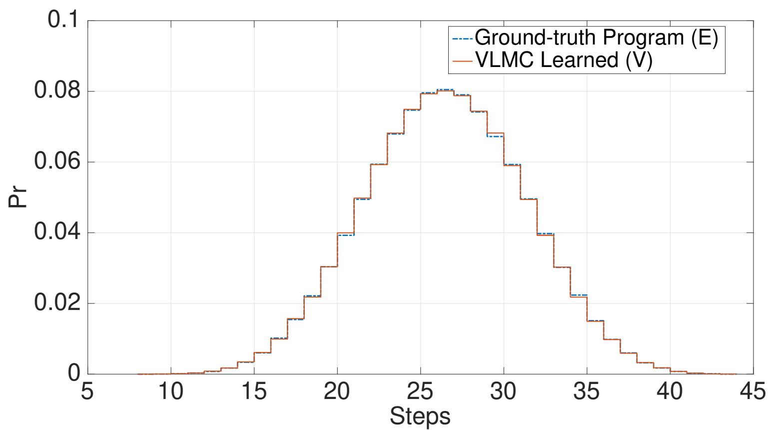
(a) Double Loop [35], $D_{KL}(E \parallel V) = 3.63 \times 10^{-5}$, VLMC order 31, number of runs 10^6



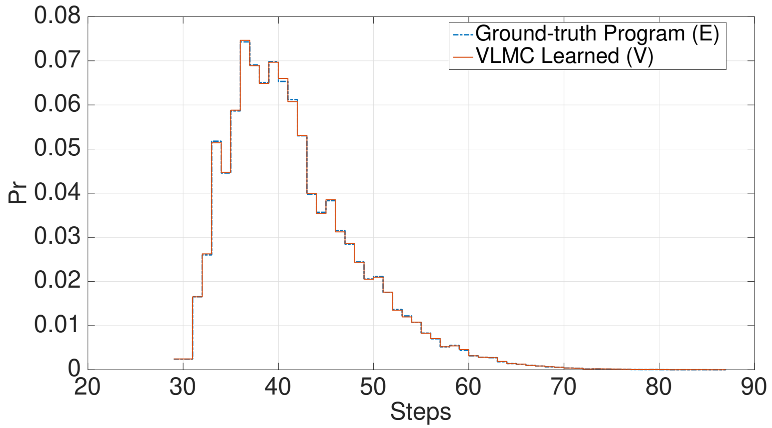
(b) Nested Loop [35], $D_{KL}(E \parallel V) = 7.58 \times 10^{-4}$, VLMC order 22, number of runs 6×10^5



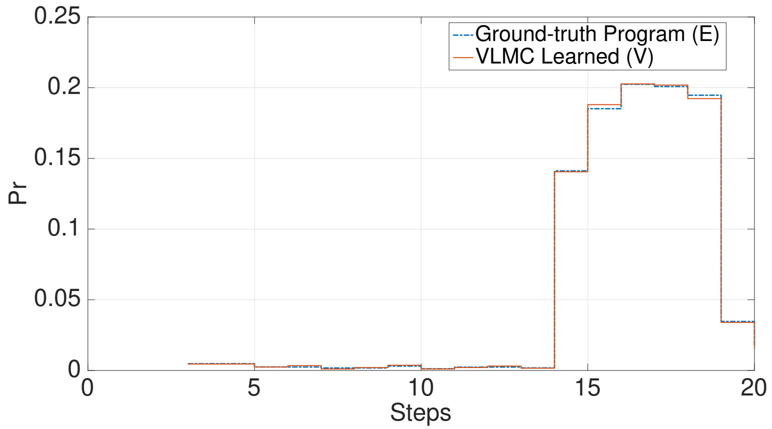
(c) Bubble sort [39], $D_{KL}(E \parallel V) = 6.45 \times 10^{-5}$, VLMC order 84, number of runs 10^6



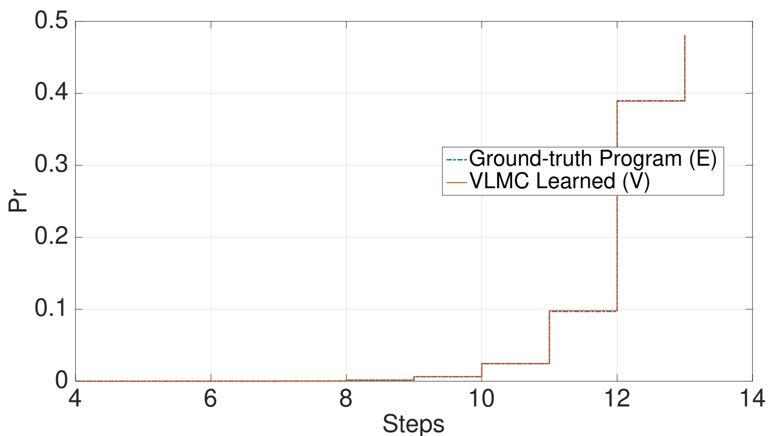
(d) Insertion Sort [39], $D_{KL}(E \parallel V) = 1.01 \times 10^{-4}$, VLMC order 48, number of runs 5×10^5



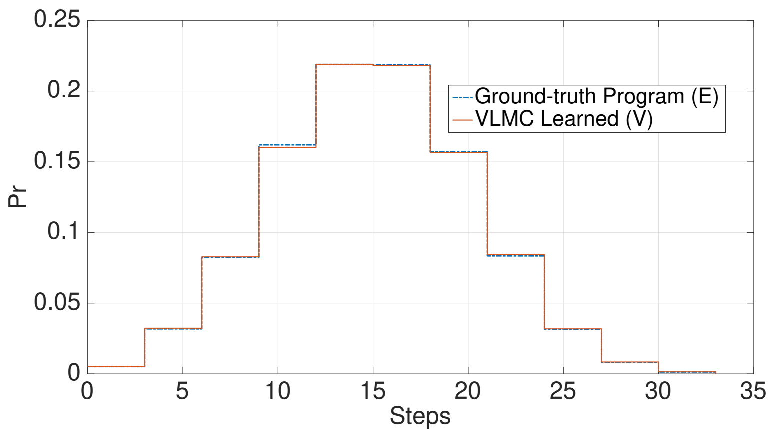
(e) Quick Sort [39], $D_{KL}(E \parallel V) = 5.35 \times 10^{-5}$, VLMC order 81, number of runs 8×10^5



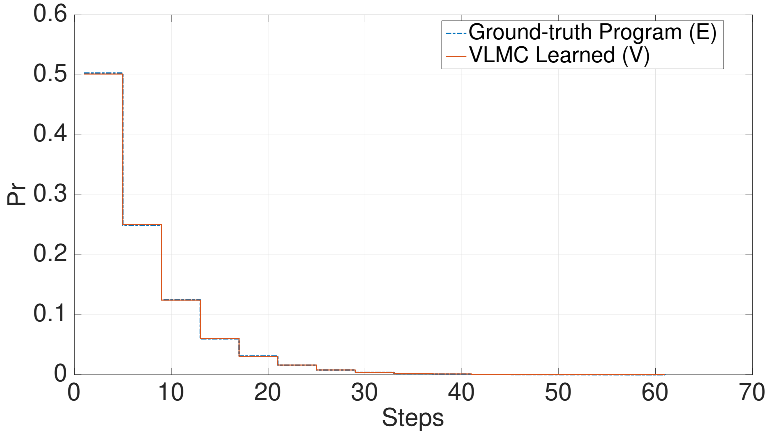
(f) Binary Search [39], $D_{KL}(E \parallel V) = 8.76 \times 10^{-4}$, VLMC order 17, number of runs 2×10^5



(g) Square root using Newton's method [117], $D_{KL}(E \parallel V) = 1.21 \times 10^{-5}$, VLMC order 1, number of runs 5×10^5



(h) Euclid's Algorithm [117], $D_{KL}(E \parallel V) = 5.95 \times 10^{-5}$, VLMC order 1, number of runs 2×10^5



(i) Miller-Rabin primality test [97], $D_{KL}(E \parallel V) = 1.30 \times 10^{-4}$, VLMC order 46, number of runs 10^5

Figure 16: Comparison between the empirical (ground-truth) and simulated (VLMC-learned) probability distributions of the number of steps until termination for the benchmark programs.

Yet, it is worth notice that *ProgramToVLMC* learns a VLMC of order 31 for the Double Loop program, which was instead supposed to be memoryless by construction. This discrepancy is due to the choice of $\alpha = 1$, which prevent pruning nodes that are almost statistically equivalent to their ancestors, but whose next-symbol probability distributions slightly differ each other for a quantity that should be neglectable. Indeed, the choice of $\alpha = 1$ makes the comparison highly sensitive to the noise generated by empirically estimating such distributions by repeating independent runs, and leads to an over-sized model. By using an infinitesimally less stringent value of α (i.e., $\alpha = 0.999999$), we would have obtained a memoryless VLMC with $D_{KL}(E \parallel V) = 7.72 \times 10^{-5}$.

5.2.2 Sensitivity evaluation

Here it is presented the sensitivity evaluation of *ProgramToVLMC* with respect to its hyper-parameter α and n_{min}. Differently from the previous analysis, to evaluate the accuracy we rely on the notion of the log-

Table 4: Sensitivity analysis with respect of pruning factor α and the parameter n_{min}. For each benchmark we used the same runs of the corresponding ground-truth models (see Figure 16).

		Double Loop			Nested Loop			Bubble Sort			Insertion Sort			Quick Sort		
n_{min}	α	*Order*	$\mathcal{L}$	*#Nodes*	*Order*	$\mathcal{L}$	*#Nodes*	*Order*	$\mathcal{L}$	*#Nodes*	*Order*	$\mathcal{L}$	*#Nodes*	*Order*	$\mathcal{L}$	*#Nodes*
1	**1.0**	**31**	**77.37**	**1427**	**22**	**407.50**	**9200**	**84**	**717.21**	**1823996**	**48**	**220.59**	**1116394**	**81**	**464.25**	**1459552**
1	0.9	1	77.37	62	22	408.55	8481	16	726.48	4736	1	221.91	53	40	479.49	20877
1	0.8	1	77.37	62	22	408.75	8456	16	726.53	4722	1	221.91	53	40	479.98	20515
1	0.7	1	77.37	62	22	408.88	8438	16	726.62	4704	1	221.91	53	40	480.40	20130
10	1.0	31	77.37	1427	22	407.50	9200	84	721.01	1305398	46	221.05	816131	80	466.00	930804
100	1.0	31	77.37	1427	22	407.50	9200	75	725.26	271741	38	221.68	139598	57	477.02	202583
1000	1.0	31	77.37	1427	22	407.50	9200	61	740.32	41580	29	221.87	21987	39	524.94	32388

likelihood function $\mathcal{L}$ of a VLMC, as defined in [91]. This metric is indeed commonly employed for measuring the goodness of fit of statistical models to a dataset [98]: lower the value of $\mathcal{L}$ greater the accuracy. Furthermore, the log-likelihood metric allows the comparison also between distributions that have different supports, which is required instead for KL divergence. A metric with the possibility of having different supports was needed because the choice of hyper-parameters can have a significant impact on the shape and the support of the learned model.

Table 4 reports the sensitivity analysis with respect to the pruning factor α and to n_{min}; each model was learned using the same number of runs of the corresponding ground-truth distribution (see Figure 16). Columns *Order* and *#Nodes* represent the order and the total number of nodes, respectively, of the learned VLMC (before Markovianization, see Section 3.4.3). In the first row of the table, we find the reference values of the order, the number of nodes, and log-likelihood when we learn the vlmc as faithfully as possible, with $\alpha = 1.0$ and $n_{min} = 1$.

As expected, decreasing values of α tend to considerably smaller size of the learned models, but to larger prediction errors, i.e., higher log-likelihoods. We observe the same trend but with a worse compression by increasing n_{min}. Furthermore, when the process is particularly complex and has a higher variance, e.g., Bubble Sort and Quick Sort, the worsening in the accuracy appears to be significantly greater by decreasing n_{min} then decreasing α. There are yet cases where smaller α does not impact on precision but causes significant compression of the learned model,e.g., Double Loop and Insertion Sort. For Double Loop, the re-

sults confirm that values of α less than 1.0 yield a memoryless model, and with inputs uniformly distributed also the Insertion Sort is memoryless. The same fact does not hold for higher n_{min} values that cannot detect memoryless behaviors. The other examples show variable memory length.

5.2.3 Comparison with state-of-the-art

In this subsection, we evaluate the scalability, in terms of runtime and memory allocation, of our *code-specialized* algorithm, against general-purpose VLMC learning algorithms originally introduced in [30] and [110], by comparing our prototype with the implementations available in [91] and [47], respectively.

We limit this analysis only to the two benchmarks Double Loop and Nested Loop, due to technical limitations of the available implementations leading to the impossibility of executing them with other more complex programs in a reasonable runtime or in feasible memory requirements, e.g., using [91] requires more than 60 GB of RAM for learning the VLMC of Quick Sort on an array of size 3. Other limitations concern also the maximum number of allowed symbols (i.e., the PLs in our case). Double Loop and Nested Loop, instead, were the only suitable benchmarks for a meaningful assessment of scalability with respect to increasing program sizes.

Furthermore, these programs are representative cases because they are opposite in memory requirements and thus in the resulting order of the learned VLMCs. While the Double Loop is memoryless the Nested Loop presents high-impact, long memory effects. For the sake of fairness, the learning was conducted using the same value for the two hyperparameters (i.e., $\alpha = 1.00$, $n_{min} = 1$) and with the same number of statistically independent runs (see Figure 16). We verified by manual inspection that all algorithms provide the same precision, equivalent VLMCs; up to small numerical differences, which are traceable back to the use of different statistical tests in the pruning phase.

Table 5 reports the results of our comparison about the execution time

Table 5: Comparison of runtimes (column T) and peak memory usage (column M) of *ProgramToVLMC* and the algorithms [30] and [110] by means of their implementations in [91] and [47], respectively.

	Double Loop							*Nested Loop*					
	ProgramToVLMC		[91]		[47]			*ProgramToVLMC*		[91]		[47]	
$\#I$	T (s)	M (GB)	T (s)	M (GB)	T (s)	M (GB)	$\#I$	T (s)	M (GB)	T (s)	M (GB)	T (s)	M (GB)
10	5	0.62	46	8.97	168	0.87	4	5	0.49	1	0.27	229	0.92
15	8	0.70	92	16.00	454	1.15	5	7	0.60	100	12.98	626	1.16
20	12	0.87	152	21.90	1024	1.43	6	9	0.84	161	19.56	1499	1.49
30	24	1.68	373	40.06	3530	1.90	8	16	0.93	617	45.49	7392	2.19

and peak memory occupation of the algorithms for learning VLMCs of the two benchmarks with increasing complexity, represented by the column $\#I$, i.e., the number of the loop iterations. Thanks to the fact that we efficiently exploit structural information about the program, and consider only the next-symbol probabilities between PLs that are connected by transitions in the CFG, we considerably outperform both approaches in the considered metrics. For example, *ProgramToVLMC* shows considerably less memory consumption respect the approach of Maechler et al. [91]. Indeed, they [91] do not limit the maximal order of the learned VLMC and for each context in the collected traces, they analyze all the preceding symbols to discover statistical dependencies; with a potentially huge number of spurious paths being explored, including those across two consecutive statistically independent runs. Moreover, the pruning phase in [91] is invoked only after the complete creation of the PST. Instead, Algorithm 1, *ProgramToVLMC* prunes each location PST, leading to earlier memory release.

We observe that instead [47] consumes much less memory than [91]. They indeed avoid considering conditional probabilities among two consecutive independent runs, by asking for the specification of the maximal order of the resulting VLMC as an input parameter. We still outperform their approach in terms of execution times, although the worst-case computational complexities are similar (cf. Section 5.1.4). This can be explained by the fact that without considering the CFG information, they ignore the sparse nature of the transition matrix of a program, where

each state (i.e., PL) has at most two possible transitions, assuming instead that the state space is fully connected and computing for each node of the PST the frequencies of next-symbol with respect of all existent PLs. As a final remark, we remember that also albeit [91] works on a fully connected model, its computation time does not explode with the dimension of the input because of its lower computational complexity.

5.2.4 Tuning the granularity level

In this subsection, we will present the preliminary experiments that we perform in order to improve the scalability of the proposed approach to learn models of increasing size, e.g., Bubble Sort with more than 9 elements. The key idea is to consider a different degree of granularity rather than the single instruction, still considering the number of steps to completion as performance metric. We focus on Bubble Sort and Quick Sort because they are the most complex problems among the benchmarks considered in the previous analyses.

Listing 5.1 shows an example of the Xtend [19] generator we use for unrolling of Bubble Sort, with the inner loop grouped to be considered as a whole. The state representing the group takes values in the natural numbers, in this example between 0 and 40, representing the sum of PLs executed within the group.

It is worth noticing that this compression implies information loss: by grouping several conditions inside a block and considering only the sum of PLs executed, we are losing the information concerning all probabilities of specific paths inside the block, obtaining an aggregate measure. Yet, it allows the developer to apply the methodology even to larger programs.

Table 6 shows the scalability improvement obtained with aggregate measures, where *BSort* represents the original unaggregated bubble sort while *BSort Group* the one considering the inner loop as a group with the aggregated number of steps; the fit time represents the time necessary to build the VLMC in milliseconds, and $\#Nodes$ the number of nodes of the resulting pruned VLMC. With fitting times below 10^5 seconds, we

```
def static bubbleSortGroup(){
'''
/*@Block:{name:=bubbleSortGroup3;param:=(int [] arr);return:=int
    [];}*/
  int n = arr.length;
  int temp = 0;

  /*@Loop:{unroll:=20;}*/
  for(int i=0; i < n; i++){/*@Edge:{cost:=1;}*/
    /*@Group:{min:=0;max:=40;}*/{
      /*@Loop:{unroll:=20;}*/
      for(int j=1; j < (n-i); j++){/*@Edge:{cost:=1;}*/
        if(arr[j-1] > arr[j]){/*@Edge:{cost:=1;}*/
          //swap elements
          temp = arr[j-1];
          arr[j-1] = arr[j];
          arr[j] = temp;
        }
      }
    }
  }
  return arr;
  '''
}
```

Listing 5.1: Example of the Xtend generator we use for the Bubble Sort Group experiment

were able to model the *BSort* until input size of 10 elements, while the *BSort Group* until 19. We used for this comparison $8 \cdot 10^5$ runs.

The results of the metric the number of steps to completion remain accurate, as we can notice from the comparison between the probability density functions of *BSort* and *BSort Group* with input array of size 10, in Figure 17a and from the mean and the variance in Table 17b. We validate the accuracy of the *BSort Group* even in the largest experiment with an input array size of 19; in this case comparing the *BSort Group* with the empirical distribution, i.e., ground-truth. Figure 18a shows the comparison between the two probability distribution functions, while Table 18b the mean and variance values.

For the Quick Sort experiment, we detect two different granularity levels, the unaggregated *QSort*, the one aggregating the first of the two

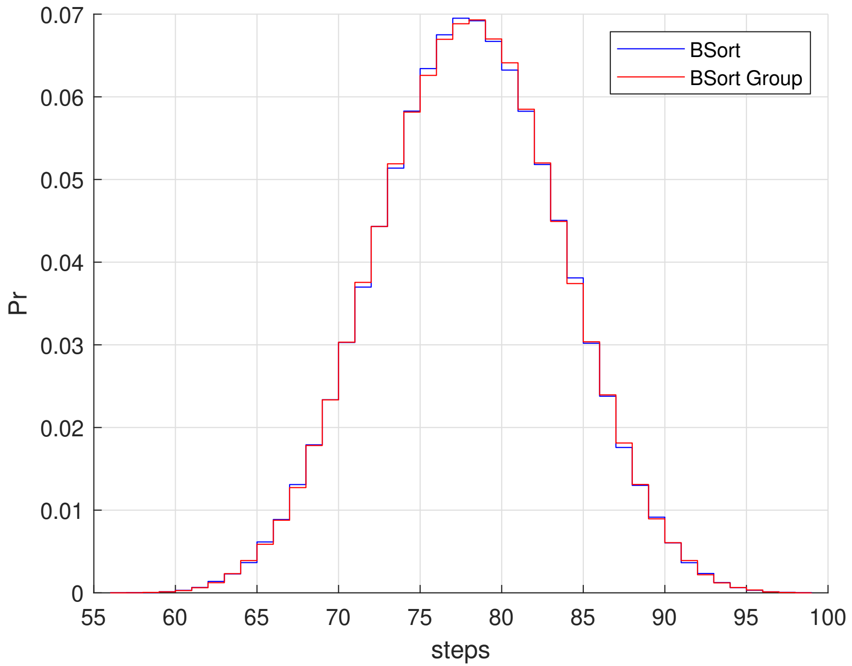
Table 6: Scalability analysis of the Bubble Sort with respect to two different granularity levels: the unaggregated *Bsort* and the aggregated *BSort Group*; the 'x' values represent experiments not performed due to time and memory limits.

	Fit time (ms)		$\#Nodes$	
$\#El$	*BSort*	*BSort Group*	*BSort*	*BSort Group*
7	$4.5 \cdot 10^4$	$4.7 \cdot 10^2$	$2.5 \cdot 10^4$	$5.9 \cdot 10^2$
8	$5.8 \cdot 10^5$	$1.2 \cdot 10^3$	$2.1 \cdot 10^5$	$1.9 \cdot 10^3$
9	$4.1 \cdot 10^6$	$4.1 \cdot 10^3$	$1.8 \cdot 10^6$	$6.3 \cdot 10^3$
10	$2.7 \cdot 10^7$	$1.3 \cdot 10^4$	$1.1 \cdot 10^7$	$2.1 \cdot 10^4$
...	...	...	...	...
15	x	$9.1 \cdot 10^5$	x	$1.5 \cdot 10^6$
16	x	$1.8 \cdot 10^6$	x	$2.5 \cdot 10^6$
17	x	$3.9 \cdot 10^6$	x	$3.7 \cdot 10^6$
18	x	$5.2 \cdot 10^6$	x	$5.1 \cdot 10^6$
19	x	$6.5 \cdot 10^6$	x	$6.6 \cdot 10^6$

loops in sequence, denoted by *QSort Group*, and the one considering both the loops as two aggregated values for the number of steps, denoted by *QSort 2Groups*.

Table 7 shows the scalability improvement obtained with aggregate measures. With fitting times below 10^8 we were able to model the *QSort* until input size of 10 elements, while the *QSort Group* and the *QSort 2Groups* until 19. We used for this comparison $8 \cdot 10^5$ runs. Although we can observe a great compression between the *QSort* and both the others, there is not a great difference in size between *QSort Group* and the *QSort 2Groups*.

In the accuracy evaluation of the Quick Sort with input arrays of different sizes, in Figure 19 and 20 we can observe that the three experiments, i.e. *QSort*, *Qsort Group* and *QSort 2Groups*, show comparable precision. The most compressed *QSort 2Groups* PDF, for input arrays of 19 elements, faithfully follows the empirical curve of observed behavior for the number of steps to completion metric.

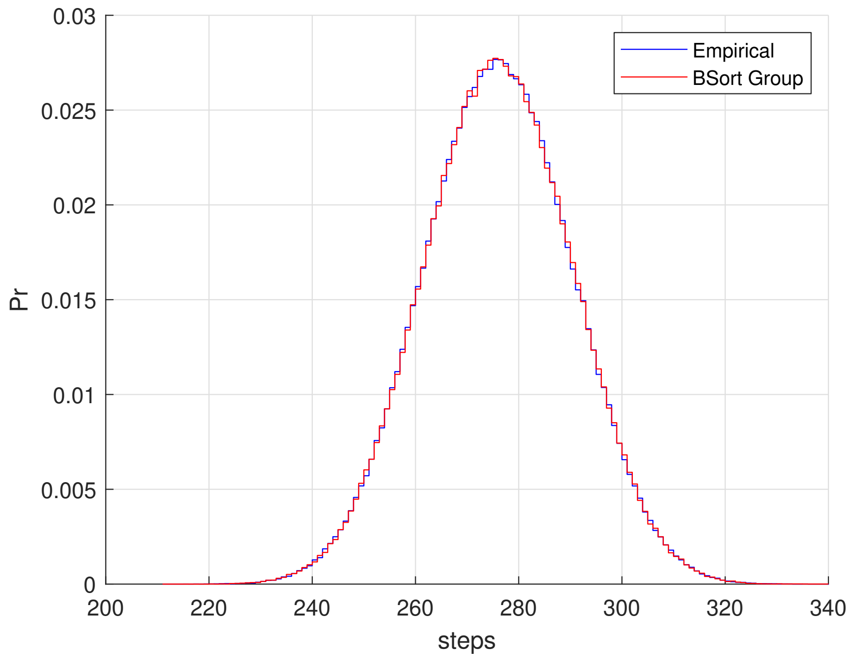


(a)

	Empirical	*Bsort*	*BSort Group*
avg	77.499	77.510	77.523
var	31.219	31.307	31.298

(b)

Figure 17: Comparison between *BSort* and *BSort Group* with input size 10: (a) PDFs comparison, (b) mean values and variance comparison with also the empirical distribution (ground-truth).



(a)

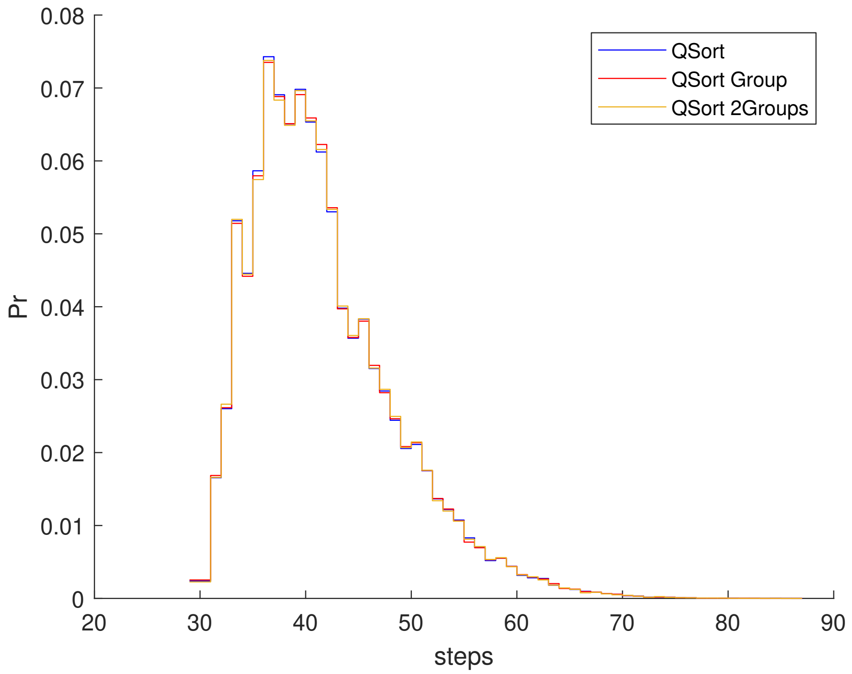
	Empirical	*BSort Group*
avg	275.490	275.514
var	204.045	203.671

(b)

Figure 18: Comparison between the empirical and *BSort Group* with input size 19: (a) PDFs comparison, (b) mean values and variance comparison.

Table 7: Scalability analysis of the Quick Sort with respect of three different granularity levels; the 'x' values represent experiments not performed due to time and memory limits.

	Fit time (ms)			$\#Nodes$		
$\#El$	*QSort*	*QSort Group*	*QSort 2 Groups*	*QSort*	*QSort Group*	*QSort 2 Groups*
7	$5.8 \cdot 10^4$	$1.7 \cdot 10^3$	$5.8 \cdot 10^2$	$2.5 \cdot 10^4$	$1.3 \cdot 10^3$	$1.1 \cdot 10^3$
8	$2.1 \cdot 10^6$	$3.2 \cdot 10^4$	$1.9 \cdot 10^3$	$1.8 \cdot 10^5$	$4.1 \cdot 10^3$	$3.6 \cdot 10^3$
9	$5.4 \cdot 10^6$	$1.2 \cdot 10^4$	$7.0 \cdot 10^3$	$1.5 \cdot 10^6$	$2.3 \cdot 10^5$	$1.2 \cdot 10^4$
10	$1.6 \cdot 10^7$	$6.6 \cdot 10^4$	$2.7 \cdot 10^4$	$7.8 \cdot 10^6$	$7.7 \cdot 10^5$	$3.9 \cdot 10^4$
...	...	...	...	...	...	...
13	x	$1.3 \cdot 10^6$	$7.8 \cdot 10^5$	x	$8.8 \cdot 10^5$	$7.8 \cdot 10^5$
14	x	$2.6 \cdot 10^6$	$1.7 \cdot 10^6$	x	$1.8 \cdot 10^6$	$1.6 \cdot 10^6$
15	x	$7.4 \cdot 10^6$	$2.6 \cdot 10^6$	x	$3.3 \cdot 10^6$	$2.8 \cdot 10^6$
16	x	$1.1 \cdot 10^7$	$4.3 \cdot 10^6$	x	$5.1 \cdot 10^6$	$4.3 \cdot 10^6$
17	x	$1.7 \cdot 10^7$	$6.9 \cdot 10^6$	x	$7.1 \cdot 10^6$	$6.0 \cdot 10^6$
18	x	$3.1 \cdot 10^7$	$1.1 \cdot 10^7$	x	$9.2 \cdot 10^6$	$7.6 \cdot 10^6$
19	x	$3.8 \cdot 10^7$	$1.4 \cdot 10^7$	x	$1.1 \cdot 10^7$	$9.3 \cdot 10^6$

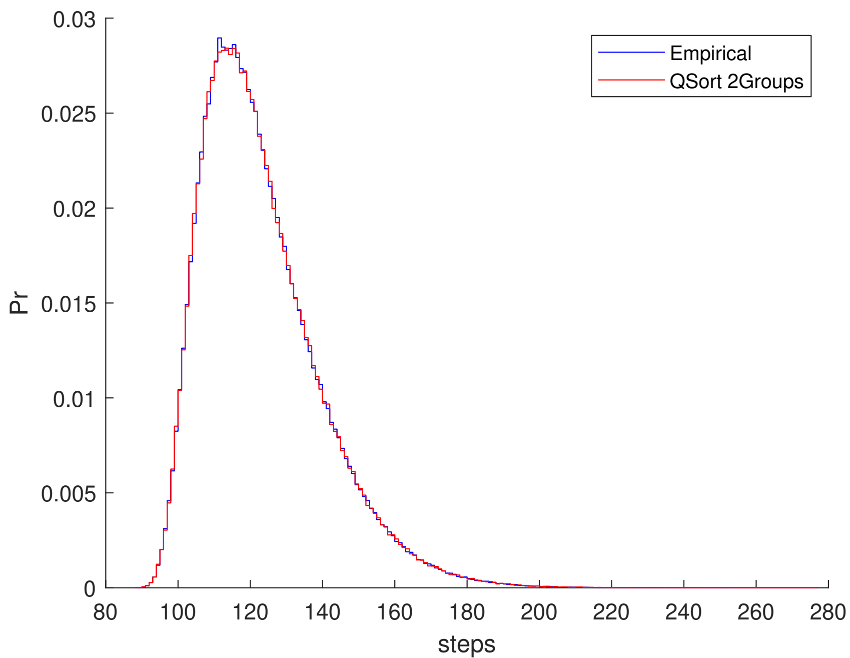


(a)

	Empirical	*Qsort*	*QSort Group*	*QSort 2Groups*
avg	41.207	41.200	41.201	41.199
var	46.931	46.872	46.834	46.732

(b)

Figure 19: Comparison between *QSort*, *QSort Group* and *QSort 2Groups* PDFs with input array size of 9: (a) PDFs comparison, (b) mean values and variance comparison between the empirical (ground-truth), *QSort*, *QSort Group* and *QSort 2Groups*.



(a)

	Empirical	*QSort 2Groups*
avg	122.514	122.523
var	269.780	270.829

(b)

Figure 20: Comparison between the empirical and *QSort 2Groups* PDFs with input array size of 19: (a) PDFs comparison, (b) mean values and variance comparison.

Chapter 6

Conclusion

In this dissertation, we addressed the problem of software system modeling for both *memoryless* and *memoryfull* processes. In the former scenario, the key idea was to interpret the stochastic process of a network of many stations providing different services, i.e., a queuing network, as a deterministic system of ordinary differential equations. Thanks to the mean-field approximation we are able to analyze systems in both transient and stationary regimes [68, 71], differently from most of previous approaches [40, 41, 96, 102, 120, 121, 125, 132], which rely on an equation that holds only with measured *steady-state* values of utilization and/or throughput.

Another strength of our approach is being non-intrusive in the system instrumentation, requiring only queue-length measurements, easily available at the operative system layer, instead of many others requiring *active probing*, i.e., stressing with load testing and/or injecting extra traffic into the running system, causing performance degradation of the system under inspection [87]. With a numerical evaluation, we provide the evidence of the effectiveness of our service demands estimator with networks of increasing sizes, see Section 4.3.4, and comparing in Section 4.3.4 with the Queue Length Maximum Likelihood Estimation (QMLE) in [133]; most similar to our in considering only queue-length observations.

Moreover, we provide a method to discover the entire topology of the system, comprising of routing probability matrix of connection among stations, with a fast linear programming formulation. We performed various what-if analyses varying system configurations after the model learning, always obtaining accurate predictions, see Section 4.4.2. In this case, we compared with the only approach we knew that could model the system's topology, i.e., Garbi et al. [50]. In Section 4.4.2 there is the numerical evaluation that demonstrates how we outperform [50] and provide more accurate estimates.

For *memoryfull* process modeling, we employ variable-length Markov chains (VLMC), which are a compact representation of the processes showing variable length memory, and it is still correct and without information losses. VLMC allows us to exploit an approximation based on the trade-off between precision and the amount of history to incorporate in the retained contexts, cutting down memory requirements and being able to learn a real program as a Markov model for the first time in literature [70].

We numerically evaluate our approach's ability to generate a model that accurately reproduces the program's dynamics in the considered performance metric, i.e., the number of steps to completion; see Section 5.2.1. And we demonstrate the effectiveness and time and memory efficiency of ourselves, comparing with general-purpose learning VLMC learning algorithms, i.e., [47, 90], whose massive cost prevents their usage even for programs of small/medium sizes; see Section 5.2.3. Furthermore, in Section 5.2.4 we presented a preliminary study of the scalability of the approaches by considering different granularity levels and measuring the time needed for VLMC learning and the number of states of the resulting model, which is a measure of the memory consumption.

Limitations and future work. The limitation of service demands and topology estimators is to consider only single-classes queuing networks with exponentially distributed service times. We aim at extending the approaches also to more complex models as multi-classes and layered QNs; and to consider also non-exponential distributions of the stations,

e.g., by fitting service demands against phase time distributions [68]. This requires addressing the nontrivial problem of variance estimator. Yet, it could be particularly feasible when the routing probability matrix is known by considering queue-length measurements instead of predictions in the ODEs over the entire observation window.

Unfortunately, statistical convergence requires a large number of program runs [70]. As mitigation, it would be possible to parallelize the analysis by running independent samples on separate cores. However, this would require developing an algorithm to merge the several VLMCs learned by the workers, which is not straightforward. Further improvements, which we leave for future work, are the integration with symbolic execution (cf. [76]) and rare event sampling techniques [29]. The main idea could be to use symbolic execution to generate the path conditions enabling the exploration of the least likely regions of the program and then use such conditions as constraints to guide the dynamic evaluation toward these regions. Other more sophisticated techniques that could be exploited are importance sampling [56,74] and concolic testing [93,134].

We demonstrate how our method can improve the performance of existing approaches even without improving the theoretical complexity [70]. Although the performance gain is significant both in memory consumption and computing times, extracting VLMCs from complex programs could still be prohibitive given the intrinsic difficulty of programs. The number of the program's contexts, indeed, grows exponentially with the number of branches [70]. For tackling this issue, we plan to further improve the computational complexity of the algorithm following the approach presented in [3]. We would develop a linear time and space algorithm, i.e., $O(rn)$, with an empirical complexity close to the simulation time. Parallelization could also help to reduce the computational time needed, by learning independently the *location PSTs* (see Section 9), similarly to [106].

Finally, we could envisage integrating the creation of the VLMC with static program analysis, like model counting [34] or probabilistic symbolic execution [52].

Chapter 7

Bibliography

[1] 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)

[2] Ammons, G., Choi, J.D., Gupta, M., Swamy, N.: Finding and removing performance bottlenecks in large systems. In: European Conference on Object-Oriented Programming. pp. 172–196. Springer (2004)

[3] Apostolico, A., Bejerano, G.: Optimal amnesic probabilistic automata or how to learn and classify proteins in linear time and space. Journal of Computational Biology **7**(3-4), 381–393 (2000)

[4] Arcelli, D., Cortellessa, V., Filieri, A., Leva, A.: Control theory for model-based performance-driven software adaptation. In: International Conference on Quality of Software Architectures (QoSA). pp. 11–20 (2015)

[5] Ardagna, D., Panicucci, B., Trubian, M., Zhang, L.: Energy-aware autonomic resource allocation in multitier virtualized environments. IEEE transactions on services computing **5**(1), 2–19 (2010)

[6] Ascher, U.M., Petzold, L.R.: Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations. SIAM (1988)

[7] Awad, M., Menasce, D.A.: Deriving parameters for open and closed qn models of operational systems through black box optimization. In: Proceedings of the International Conference on Performance Engineering (ICPE) (2017)

[8] Baccelli, F., Kauffmann, B., Veitch, D.: Inverse problems in queueing theory and Internet probing. Queueing Systems **63**(1-4), 59 (2009)

[9] Baldoni, R., Coppa, E., D'elia, D.C., Demetrescu, C., Finocchi, I.: A survey of symbolic execution techniques. ACM Computing Surveys (CSUR) **51**(3), 1–39 (2018)

[10] 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)

[11] 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)

[12] Bard, Y.: Some extensions to multiclass queueing network analysis. In: Proceedings of the Third International Symposium on Modelling and Performance Evaluation of Computer Systems: Performance of Computer Systems. pp. 51–62. North-Holland Publishing Co. (1979)

[13] Barham, P., Donnelly, A., Isaacs, R., Mortier, R.: Using magpie for request extraction and workload modelling. In: OSDI. vol. 4, pp. 18–18 (2004)

[14] Barham, P., Isaacs, R., Mortier, R., Narayanan, D.: Magpie: Online modelling and performance-aware systems. In: HotOS. pp. 85–90 (2003)

[15] Baskett, F., Chandy, K.M., Muntz, R.R., Palacios, F.G.: Open, closed, and mixed networks of queues with different classes of customers. Journal of the ACM (JACM) **22**(2), 248–260 (1975)

[16] 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. pp. 54–65 (2007)

[17] Begleiter, R., El-Yaniv, R., Yona, G.: On prediction using variable order markov models. Journal of Artificial Intelligence Research **22**, 385–421 (2004)

[18] Belov, D.I., Armstrong, R.D.: Distributions of the kullback–leibler divergence with applications. British Journal of Mathematical and Statistical Psychology **64**(2), 291–309 (2011)

[19] Bettini, L.: Implementing domain-specific languages with Xtext and Xtend. Packt Publishing Ltd (2016)

[20] Bodden, E., Hendren, L., Lhoták, O.: A staged static program analysis to improve the performance of runtime monitoring. In: European Conference on Object-Oriented Programming. pp. 525–549. Springer (2007)

[21] Bolch, G., Greiner, S., De Meer, H., Trivedi, K.S.: Queueing networks and Markov chains: modeling and performance evaluation with computer science applications. John Wiley & Sons (2006)

[22] Bolch, G., Greiner, S., de Meer, H., Trivedi, K.: Queueing networks and Markov chains: modeling and performance evaluation with computer science applications. Wiley (2005)

[23] Borges, J., Levene, M.: Evaluating variable-length markov chain models for analysis of user web navigation sessions. IEEE Transactions on Knowledge and Data Engineering **19**(4), 441–452 (2007)

[24] 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)

[25] Bortolussi, L., Hillston, J., Latella, D., Massink, M.: Continuous approximation of collective system behaviour: A tutorial. Performance Evaluation **70**(5), 317–349 (2013)

[26] Boyd, S., Boyd, S.P., Vandenberghe, L.: Convex optimization. Cambridge university press (2004)

[27] Brosig, F., Kounev, S., Krogmann, K.: Automated extraction of Palladio component models from running Enterprise Java applications. In: VALUETOOLS. pp. 10:1–10:10 (2009). https://doi.org/10.4108/ICST.VALUETOOLS2009.7981

[28] 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)

[29] Bucklew, J.: Introduction to rare event simulation. Springer Science & Business Media (2013)

[30] Bühlmann, P., Wyner, A.J., et al.: Variable length markov chains. The Annals of Statistics **27**(2), 480–513 (1999)

[31] 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)

[32] Cai, Y., Sullivan, K.J.: Modularity analysis of logical design models. In: 21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06). pp. 91–102. IEEE (2006)

[33] Calinescu, R., Ghezzi, C., Kwiatkowska, M., Mirandola, R.: Self-adaptive software needs quantitative verification at runtime. Commun. ACM **55**(9), 69–77 (2012)

[34] Chavira, M., Darwiche, A.: On probabilistic inference by weighted model counting. Artificial Intelligence **172**(6-7), 772–799 (2008)

[35] 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)

[36] Chen, T.Y., Kuo, F.C., Merkel, R.G., Tse, T.: Adaptive random testing: The art of test case diversity. Journal of Systems and Software **83**(1), 60–66 (2010)

[37] Chen, Z., Chen, B., Xiao, L., Wang, X., Chen, L., Liu, Y., Xu, B.: Speedoo: prioritizing performance optimization opportunities. In: Proceedings of the 40th International Conference on Software Engineering. pp. 811–821 (2018)

[38] Coppa, E., Demetrescu, C., Finocchi, I.: Input-sensitive profiling. ACM SIGPLAN Notices **47**(6), 89–98 (2012)

[39] Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to algorithms. MIT press (2009)

[40] Cremonesi, P., Dhyani, K., Sansottera, A.: Service time estimation with a refinement enhanced hybrid clustering algorithm. In: International Conference on Analytical and Stochastic Modeling Techniques and Applications. pp. 291–305. Springer (2010)

[41] Cremonesi, P., Sansottera, A.: Indirect estimation of service demands in the presence of structural changes. Performance Evaluation **73**, 18–40 (2014)

[42] Dalevi, D., Dubhashi, D.: The peres-shields order estimator for fixed and variable length markov models with applications to dna sequence similarity. In: International Workshop on Algorithms in Bioinformatics. pp. 291–302. Springer (2005)

[43] Dragoni, N., Giallorenzo, S., Lafuente, A.L., Mazzara, M., Montesi, F., Mustafin, R., Safina, L.: Microservices: yesterday, today, and

tomorrow. In: Present and ulterior software engineering, pp. 195–216. Springer (2017)

[44] Everts, T.: Time Is Money: The Business Value of Web Performance. " O'Reilly Media, Inc." (2016)

[45] 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)

[46] Filieri, A., Păsăreanu, C.S., Visser, W.: Reliability analysis in symbolic Pathfinder. In: Proc. Int'l Conf. Software Engineering (ICSE). pp. 622–631 (2013)

[47] Gabadinho, A., Ritschard, G.: Analyzing state sequences with probabilistic suffix trees: The pst r package. Journal of statistical software **72**(3), 1–39 (2016)

[48] Gadelha, M.Y., Ismail, H.I., Cordeiro, L.C.: Handling loops in bounded model checking of c programs via k-induction. International Journal on Software Tools for Technology Transfer **19**(1), 97–114 (2017)

[49] Galata, A., Johnson, N., Hogg, D.: Learning variable-length markov models of behavior. Computer Vision and Image Understanding **81**(3), 398–413 (2001)

[50] Garbi, G., Incerto, E., Tribastone, M.: Learning queuing networks by recurrent neural networks. In: Proceedings of the ACM/SPEC International Conference on Performance Engineering. pp. 56–66 (2020)

[51] Garcia, J., Krka, I., Mattmann, C., Medvidovic, N.: Obtaining ground-truth software architectures. In: Proc. Int'l Conf. Software Engineering (ICSE). pp. 901–910 (2013)

[52] Geldenhuys, J., Dwyer, M.B., Visser, W.: Probabilistic symbolic execution. In: ISSTA. pp. 166–176 (2012)

[53] 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)

[54] Géraud, R., Koscina, M., Lenczner, P., Naccache, D., Saulpic, D.: Generating functionally equivalent programs having non-isomorphic control-flow graphs. In: Nordic Conference on Secure IT Systems. pp. 265–279. Springer (2017)

[55] Gillespie, D.: Exact stochastic simulation of coupled chemical reactions. Journal of Physical Chemistry **81**(25), 2340–2361 (December 1977)

[56] Glynn, P.W., Iglehart, D.L.: Importance sampling for stochastic simulations. Management science **35**(11), 1367–1392 (1989)

[57] Goldsmith, S.F., Aiken, A.S., Wilkerson, D.S.: Measuring empirical computational complexity. In: Proceedings of the 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)

[58] Graham, S.L., Kessler, P.B., Mckusick, M.K.: Gprof: A call graph execution profiler. ACM Sigplan Notices **17**(6), 120–126 (1982)

[59] Gulwani, S., Jha, S., Tiwari, A., Venkatesan, R.: Synthesis of loop-free programs. In: PLDI. vol. 11, pp. 62–73 (2011)

[60] Gusfield, D.: Algorithms on stings, trees, and sequences: Computer science and computational biology. ACM SIGACT News **28**(4), 41–60 (1997)

[61] Harchol-Balter, M.: Performance modeling and design of computer systems: queueing theory in action. Cambridge University Press (2013)

[62] Harman, M., O'Hearn, P.: From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis.

In: 2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM). pp. 1–23. IEEE (2018)

[63] 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)

[64] Holmes, G., Donkin, A., Witten, I.H.: Weka: A machine learning workbench. In: Proceedings of ANZIIS'94-Australian New Zealnd Intelligent Information Systems Conference. pp. 357–361. IEEE (1994)

[65] Hosseini, R., Brusilovsky, P.: Javaparser: A fine-grain concept indexing tool for java problems. In: CEUR Workshop Proceedings. vol. 1009, pp. 60–63. University of Pittsburgh (2013)

[66] Huber, N., Brosig, F., Kounev, S.: Model-based self-adaptive resource allocation in virtualized environments. In: Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. pp. 90–99. SEAMS '11 (2011)

[67] Huffaker, B., Plummer, D., Moore, D., Claffy, K.: Topology discovery by active probing. In: Proceedings 2002 Symposium on Applications and the Internet (SAINT) Workshops. pp. 90–96. IEEE (2002)

[68] Incerto, E., Napolitano, A., Tribastone, M.: Moving horizon estimation of service demands in queuing networks. In: 2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS). pp. 348–354. IEEE (2018)

[69] Incerto, E., Napolitano, A., Tribastone, M.: Inferring performance from code: a review. In: International Symposium on Leveraging Applications of Formal Methods. pp. 307–322. Springer (2020)

[70] Incerto, E., Napolitano, A., Tribastone, M.: Statistical learning of markov chains of programs. In: 2020 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS). pp. 1–8. IEEE (2020)

[71] Incerto, E., Napolitano, A., Tribastone, M.: Learning queuing networks via linear optimization. In: Proceedings of the ACM/SPEC International Conference on Performance Engineering. pp. 51–60 (2021)

[72] Incerto, E., Tribastone, M., Trubiani, C.: Symbolic performance adaptation. In: International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS). pp. 140–150 (2016)

[73] Incerto, E., Tribastone, M., Trubiani, C.: Software performance self-adaptation through efficient model predictive control. In: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 485–496 (2017)

[74] Jégourel, C., Wang, J., Sun, J.: Importance sampling of interval markov chains. In: 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). pp. 303–313. IEEE (2018)

[75] Kaelbling, L.P., Littman, M.L., Moore, A.W.: Reinforcement learning: A survey. Journal of artificial intelligence research **4**, 237–285 (1996)

[76] King, J.C.: Symbolic execution and program testing. Communications of the ACM **19**(7), 385–394 (1976)

[77] Ko, P., Aluru, S.: Space efficient linear time construction of suffix arrays. In: Annual Symposium on Combinatorial Pattern Matching. pp. 200–210. Springer (2003)

[78] Kowal, M., Tschaikowski, M., Tribastone, M., Schaefer, I.: Scaling size and parameter spaces in variability-aware software perfor-

mance models. In: International Conference on Automated Software Engineering (ASE). pp. 407–417 (2015)

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

[80] Kullback, S., Leibler, R.A.: On information and sufficiency. The annals of mathematical statistics **22**(1), 79–86 (1951)

[81] Kurtz, T.G.: Solutions of ordinary differential equations as limits of pure Markov processes. In: J. Appl. Prob. vol. 7, pp. 49–58 (1970)

[82] Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: Proc. Int'l Conf. Computer Aided Verification (CAV). pp. 585–591 (2011)

[83] Larus, J.R.: Whole program paths. ACM SIGPLAN Notices **34**(5), 259–269 (1999)

[84] de Lemos, R., et al.: Software Engineering for Self-Adaptive Systems: A Second Research Roadmap, pp. 1–32. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)

[85] Limnios, N., Oprisan, G.: Semi-Markov processes and reliability. Springer Science & Business Media (2012)

[86] Lin, J., Adjeroh, D., Jiang, B.H.: Probabilistic suffix array: efficient modeling and prediction of protein families. Bioinformatics **28**(10), 1314–1323 (2012)

[87] Liu, Z., Wynter, L., Xia, C.H., Zhang, F.: Parameter inference of queueing models for IT systems using end-to-end measurements. Performance Evaluation **63**(1), 36–60 (2006)

[88] 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)

[89] Luckow, K., Păsăreanu, C.S., Dwyer, M.B., Filieri, A., Visser, W.: Exact and approximate probabilistic symbolic execution for non-deterministic programs. In: Proceedings of the 29th ACM/IEEE international conference on Automated software engineering. pp. 575–586 (2014)

[90] Mächler, M., Bühlmann, P.: Variable length markov chains: methodology, computing, and software. Journal of Computational and Graphical Statistics **13**(2), 435–455 (2004)

[91] Maechler, M.: Vlmc: Variable length markov chains. R package version pp. 1–4 (2015)

[92] Magarick, J.: Sequential learning and variable length Markov chains. Ph.D. thesis, Graduate Group in Managerial Science and Applied Economics (2016)

[93] Majumdar, R., Sen, K.: Hybrid concolic testing. In: 29th International Conference on Software Engineering (ICSE'07). pp. 416–426. IEEE (2007)

[94] 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)

[95] Mazeroff, G., Gregor, J., Thomason, M., Ford, R.: Probabilistic suffix models for api sequence analysis of windows xp applications. Pattern Recognition **41**(1), 90–101 (2008)

[96] Menasce, D.A.: Computing missing service demand parameters for performance models. In: Int. CMG Conference. pp. 241–248 (2008)

[97] Miller, G., Rabin, M.: Miller-Rabin Primality Test. `https://www.geeksforgeeks.org/primality-test-set-3-miller-rabin/`

[98] Myung, I.J.: Tutorial on maximum likelihood estimation. Journal of mathematical Psychology **47**(1), 90–100 (2003)

[99] Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. ACM Sigplan notices **42**(6), 89–100 (2007)

[100] Nocedal, J., Wright, S.: Numerical optimization. Springer Science & Business Media (2006)

[101] Osmani, A., Grigorik, I.: Speed is now a landing page factor for google search and ads. `https://developers.google.com/web/updates/2018/07/search-ads-speed` (July 2018)

[102] Pacifici, G., Segmuller, W., Spreitzer, M., Tantawi, A.: Cpu demand for web serving: Measurement analysis and dynamic estimation. Performance Evaluation **65**(6-7), 531–553 (2008)

[103] 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)

[104] Pukite, P., Pukite, J.: Markov modeling for reliability analysis. Wiley-IEEE Press (1998)

[105] Puterman, M.L.: Markov decision processes. Handbooks in operations research and management science **2**, 331–434 (1990)

[106] Qvick, J.R.: Parallel construction of variable length markov models for dna sequences (2020)

[107] Raftery, A.E.: A model for high-order markov chains. Journal of the Royal Statistical Society: Series B (Methodological) **47**(3), 528–539 (1985)

[108] Ramalingam, G.: Data flow frequency analysis. ACM SIGPLAN Notices **31**(5), 267–277 (1996)

[109] Risso, F., Degioanni, L.: An architecture for high performance network analysis. In: Proceedings. Sixth IEEE Symposium on Computers and Communications. pp. 686–693. IEEE (2001)

[110] Ron, D., Singer, Y., Tishby, N.: The power of amnesia: Learning probabilistic automata with variable memory length. Machine learning **25**(2-3), 117–149 (1996)

[111] Rosendahl, M.: Automatic complexity analysis. In: Proceedings of the fourth international conference on Functional programming languages and computer architecture. pp. 144–156 (1989)

[112] Sanft, K.R., Wu, S., Roh, M., Fu, J., Lim, R.K., Petzold, L.R.: Stochkit2: software for discrete stochastic simulation of biochemical systems with events. Bioinformatics **27**(17), 2457–2458 (2011)

[113] 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)

[114] 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)

[115] Schlabach, T.: Insight into event tracing for windows

[116] Schweitzer, P.: Approxiamate analysis of multiclass closed networks of queues. J. ACM **29**(2) (1981)

[117] Sedgewick, R., Wayne, K.: Introduction to programming in Java: an interdisciplinary approach. Addison-Wesley Professional (2017)

[118] 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)

[119] Sharir, M., Pnueli, A., Hart, S.: Verification of probabilistic programs. SIAM Journal on Computing **13**(2), 292–314 (1984)

[120] Sharma, A.B., Bhagwan, R., Choudhury, M., Golubchik, L., Govindan, R., Voelker, G.M.: Automatic request categorization in internet services. ACM SIGMETRICS Performance Evaluation Review **36**(2), 16–25 (2008)

[121] Spinner, S., Casale, G., Brosig, F., Kounev, S.: Evaluating approaches to resource demand estimation. Performance Evaluation **92**, 51–71 (2015)

[122] Stewart, W.J.: Performance Modelling and Markov Chains. In: SFM. pp. 1–33 (2007)

[123] Stewart, W.J.: Probability, Markov Chains, Queues, and Simulation. Princeton University Press (2009)

[124] Stewart, W.J.: Probability, Markov chains, queues, and simulation: the mathematical basis of performance modeling. Princeton university press (2009)

[125] Sutton, C., Jordan, M.I.: Bayesian inference for queueing networks and modeling of Internet services. The Annals of Applied Statistics pp. 254–282 (2011)

[126] Szeliski, R., Zabih, R., Scharstein, D., Veksler, O., Kolmogorov, V., Agarwala, A., Tappen, M., Rother, C.: A comparative study of energy minimization methods for markov random fields with smoothness-based priors. IEEE Transactions on pattern analysis and machine intelligence **30**(6), 1068–1080 (2008)

[127] Tantawi, A.N., Towsley, D.: Optimal static load balancing in distributed computer systems. J. ACM **32**(2), 445–465 (1985)

[128] Thereska, E., Doebel, B., Zheng, A.X., Nobel, P.: Practical performance models for complex, popular applications. ACM SIGMETRICS Performance Evaluation Review **38**(1), 1–12 (2010)

[129] 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)

[130] Tribastone, M., Gilmore, S.: Automatic Translation of UML Sequence Diagrams into PEPA Models. In: Fifth International Conference on the Quantitative Evaluaiton of Systems (QEST). pp. 205–214 (2008)

[131] Wang, W., Tian, N., Huang, S., He, S., Srivastava, A., Soffa, M.L., Pollock, L.: 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)

[132] Wang, W., Casale, G.: Bayesian service demand estimation using Gibbs sampling. In: 21st International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS) (2013)

[133] Wang, W., Casale, G., Kattepur, A., Nambiar, M.: Maximum likelihood estimation of closed queueing network demands from queue length data. In: Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering. pp. 3–14. ACM (2016)

[134] Wang, X., Sun, J., Chen, Z., Zhang, P., Wang, J., Lin, Y.: Towards optimal concolic testing. In: Proceedings of the 40th International Conference on Software Engineering. pp. 291–302 (2018)

[135] Wegbreit, B.: Mechanical program analysis. Communications of the ACM **18**(9), 528–539 (1975)

[136] 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)

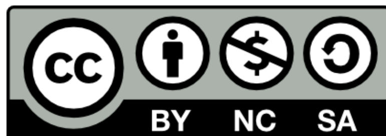
[137] Woodside, M., Franks, G., Petriu, D.C.: The future of software performance engineering. In: Proceedings of the Future of Software Engineering (FOSE). pp. 171–187 (2007)

[138] Woodside, M., Petriu, D.C., Petriu, D.B., Shen, H., Israr, T., Merseguer, J.: Performance by unified model analysis (PUMA). In: WOSP. pp. 1–12 (2005)

[139] Zaparanuks, D., Hauswirth, M.: Algorithmic profiling. In: Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI). pp. 67–76 (2012)

[140] Zaparanuks, D., Hauswirth, M.: Algorithmic profiling. In: Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation. pp. 67–76 (2012)

[141] Zhang, Q., Cheng, L., Boutaba, R.: Cloud computing: state-of-the-art and research challenges. Journal of internet services and applications **1**(1), 7–18 (2010)



Unless otherwise expressly stated, all original material of whatever nature created by Annalisa Napolitano and included in this thesis, is licensed under a Creative Commons Attribution Noncommercial Share Alike 3.0 Italy License.

Check on Creative Commons site:

https://creativecommons.org/licenses/by-nc-sa/3.0/it/legalcode/

https://creativecommons.org/licenses/by-nc-sa/3.0/it/deed.en

Ask the author about other uses.