PERFORMANCE MODELS FOR ANALYSIS AND CONTROL OF IT SYSTEMS

Asparuh Ventsislavov Hristov

Performance Models for Analysis and Control of IT Systems

Asparuh Ventsislavov Hristov

The research presented in this thesis was done as part a Public-Private Partnership between The Center for Mathematics and Computer Science (CWI) and ING Bank.

Cover artwork by: Jasmin Schiffer and Kristina Kalpaklieva
Printing by: Ipskamp Printing B.V.

VRIJE UNIVERSITEIT

PERFORMANCE MODELS FOR ANALYSIS AND CONTROL OF IT SYSTEMS

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad Doctor
aan de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof.dr. V. Subramaniam,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de Faculteit der Bètawetenschappen
op dinsdag 18 september 2018 om 9.45 uur
in de aula van de universiteit,
De Boelelaan 1105

door

Asparuh Ventsislavov Hristov

geboren te Sofia, Bulgaria

promotoren:	prof.dr. R.D. van der Mei
	prof.dr. S. Bhulai
copromotor:	dr. J.W. Bosman

Acknowledgements

The past four years will certainly remain some of the most memorable years of my life. One of the reasons is my decision to pursue a PhD degree. This choice helped me expand my knowledge through learning, researching and experimenting. I look forward to applying the acquired skills in future cooperations with the people that accompanied me throughout the years as well as new acquaintances and friends that I meet along the way.

First of all, I would like to express my sincere gratitude to my promotors Rob van der Mei and Sandjai Bhulai and my copromotor Joost Bosman for all of the assistance and expertise that they provided during these four years. Their wisdom, motivation, and immense knowledge is truly inspiring. I believe they form the best group of mentors one can wish for. I would like to thank them for encouraging my research and for allowing me to develop not only as a research scientist but also as a better person. Rob's ability to balance the great amount of responsibilities he has, projects and his private life has set an incredibly high bar to strive towards to. To me Sandjai is the most sincere, open-minded and approachable professor I have ever known. I highly admire Joosts motivation, dedication and curiosity.

Besides my promotors, I would like to thank my thesis committee: Floske Spieksma, Mathisca de Gunst, Ger Koole, Guszti Eiben, Hans van den Berg and Maurits de Graaf for their time, effort and insightful feedback on the thesis. Furthermore, I would like to highlight the contribution of Bert Zwart in considerably improving this thesis.

I am deeply indebted to Ger Koole for his fundamental role in my professional development. I still clearly remember being inspired by his lectures and his success in applying mathematics to health care. He provided me with guidelines and support in two events crucial for my career: starting my first job in a company where I could apply my expertise, and finding my PhD project. I feel very privileged to have received so much help from Ger.

For me it was very important that the research is applicable and based on real-world challenges, and therefore, I would like to thank ING for being part of this project and for providing numerous interesting topics for research. In particular, I would like to thank Joost Bosman. His vision, support and way of reasoning was a source of inspiration to me. The team at ING has been a source of good advice and collaborations as well as of friendships. It has been

a great pleasure to work with the people there.

The time spent at CWI and at the VU has always been enjoyable greatly due to my colleagues there. I already miss the insightful brainstorming sessions with them and the countless opportunities to talk about research, mathematics, computer science and simply interesting innovations.

For practical reasons, unfortunately I cannot list the names of all the friends that have supported me throughout the last years. I highly value the moments we shared together! I hope each one of them knows how much I trust, respect and admire their unique way of being!

I am deeply thankful to my family. Words cannot express how grateful I am to my parents Natasha and Ventsislav, my brother Petar and my wife Jeanette for the sacrifices that they have made on my behalf. Their love, support and believe shaped and continue shaping me as the person I am.

Contents

1 Introduction **5**
- 1.1 Performance evaluation and optimal control 6
- 1.2 Queueing Theory and Markov Decision Processes 8
- 1.3 Evolutionary algorithms 10
- 1.4 Practical solutions 12
- 1.5 Outline of the thesis 13

2 Control of a Tandem Queue with Start-up Costs **17**
- 2.1 Introduction 18
- 2.2 Model 19
- 2.3 Optimal policy 22
- 2.4 Approximation technique 24
  - 2.4.1 Second server switched off 25
  - 2.4.2 Second server switched on 27
- 2.5 Results 28
- 2.6 Conclusion 30

3 Analysis and Control of a Single Server Queue with Backlog Processing **33**
- 3.1 Introduction 34
- 3.2 Model 35
- 3.3 Analysis 37
  - 3.3.1 Analytic solution for $K = 2$ 38
  - 3.3.2 Fluid approximation for large K 42
- 3.4 Optimization of the batch size K 44
- 3.5 Results 46
- 3.6 Conclusion 50
- 3.7 Appendix 51

4 Closed-Form Control Policies of Markov Decision Processes Using Symbolic Regression **53**
- 4.1 Introduction 53
- 4.2 Method 55
- 4.3 Running example 1: The ‘write behind’ cache 58

4.3.1 Model . . . 58
4.3.2 Data generation . . . 59
4.3.3 Algorithm settings . . . 61
4.3.4 Results . . . 62
4.4 Running example 2: The M/M/1 queue with customer rejection 65
4.4.1 Model . . . 65
4.4.2 Data generation . . . 66
4.4.3 Algorithm settings . . . 66
4.4.4 Results . . . 67
4.5 Conclusion . . . 68

5 Closed-Form Performance Evaluation of Markov Decision Processes Using Symbolic Regression 69
5.1 Introduction . . . 70
5.2 Method . . . 71
5.3 Running example: Two-stream blending system . . . 72
5.3.1 Model . . . 72
5.3.2 Data generation . . . 74
5.3.3 Algorithm settings . . . 76
5.3.4 Results . . . 77
5.4 Conclusion . . . 79
5.5 Appendix . . . 80

6 Performance Evaluation of Nested Systems 83
6.1 Introduction . . . 83
6.2 Model . . . 85
6.2.1 Saturated tandem queues with nested sessions . . . 85
6.2.2 Performance metrics . . . 87
6.2.3 Bottleneck analysis . . . 88
6.3 Performance analysis . . . 88
6.3.1 Exact solution for $N = 2$ . . . 89
6.3.2 Recursive scheme for $N > 2$ . . . 90
6.3.3 Bottleneck identification technique . . . 92
6.4 Results . . . 93
6.4.1 Results for three-layered networks . . . 93
6.4.2 Results for ten-layered networks . . . 94
6.5 Bottleneck identification results . . . 96
6.6 Conclusion . . . 98

Bibliography 101

Summary **111**

Samenvatting **115**

Chapter 1

Introduction

Nowadays, IT services are expected to operate in a continuous, 24/7 fashion. In our globally connected world, smartphones and tablets, among other devices, are used as means to ensure access to all kind of services at any time and place. The ever-growing demand for mobile communications and online services increases both the traffic volumes and the Quality of Service (QoS) expectations from the end users. In response, companies migrate their services to online environments where they can meet and satisfy a larger group of people. Furthermore, financial institutions implement an increasing number of ICT solutions, e.g., (mobile) Internet banking, iDeal in the context of e-commerce, etc. These new technologies greatly facilitate our everyday lives, but at the same time, make our society dependent on their availability and quality. Therefore, at any time, even the slightest disruptions or malfunctions are immediately being noticed and may cause a great impact on the user experience, and consequently, on the service providers' reputations.

In the last two decades, we have been witnessing a paradigm shift from the traditional information-oriented Internet into an Internet of Services (IoS). This shift brought the existence and the importance of concepts like software as a service, shared platforms and cloud computing. Examples as self-driving cars, online language translation and Amazons cashierless shops indicate the extent to which digital technologies interact with the physical world. Next to that, companies are able to create online marketplaces that allow both suppliers and consumers to flourish, resulting in a tremendous growth of shared platforms for booking hotels, renting cars, etc. This has opened up virtually unbounded possibilities to introduce new services facilitating business processes and improving the quality of life. However, in most of the cases, the same innovations introduce complicated dependencies between various services offered by a multitude of third parties, each having its own QoS requirements.

The increasing complexity of IT infrastructures poses significant challenges in managing computer systems. Next to that, the growing dependence on IT

services has increased the need for reliable implementations which are resilient to malfunctions, cyber attacks, high-load traffic, etc. Therefore, studying the performance of ICT service chains and their optimal control is of great importance. Ensuring efficient usage of the available resources while preserving the desired QoS in such complex environments requires one to go beyond ad-hoc solutions and develop quantitative QoS models and methods for controlling QoS. Motivated by this, in this thesis we study techniques to derive fundamental insight into the performance and the optimal control of two common system design paradigms in ICT service chains. More specifically, we analyze models of applications connected in either a (1) *sequential* or (2) *nested* fashion.

1.1 Performance evaluation and optimal control

Typically, there is a trade-off between the performance on the one hand and the cost of a given computer system on the other hand. The goal to properly balance this trade-off makes performance analysis crucial. A performance evaluation is required when a number of alternative designs have to be compared to find the one that best meets the operational demands. As the field of computer design matures, the computer industry is becoming more competitive, and it is more important than ever to ensure that the alternative selected provides the best cost-performance trade-off. For example, a system administrator would have to evaluate the performance of a number of systems to decide which one is best for a given set of applications. Even if there are no alternatives, performance evaluation of the current system helps in determining how well the system is operating and whether any improvements need to be made. Furthermore, performance evaluation enables service providers to answer 'what-if' questions regarding the predicted performance for different system configurations. Unfortunately, the types of applications are so numerous that it is not possible to have a standard measure of performance, a standard measurement environment, or a standard technique for all cases. Therefore, the first step in performance evaluation is to select the right measures of performance, the right measurement environments, and the right techniques.

For contrasting design alternatives, it is too time consuming to build these different systems and test their performance in a practical setting. Even for

a given design, large multiple-client testing is very time consuming if at all feasible. Therefore, we develop models to assess the performance of the design alternatives quickly and quantitatively. The development of efficient QoS mechanisms is complicated by the omnipresence of the phenomenon of uncertainty. *Stochastic models* are instrumental to capture such uncertainties and provide a basis for educated control of systems with uncertainty. In the next paragraphs, we study performance problems where the stochastic nature of certain processes complicates the analysis.

The term *capacity management* is used to denote the problem of ensuring that the currently available computing resources are used to provide the highest performance. The process of adjusting system parameters to optimize the performance is also called *performance tuning*. While the alternatives for capacity management consist of analyzing usage patterns and changing system configurations to maximize the *present* performance, capacity planning is concerned with the *future*. One of the important problems for managers of data processing installations is to ensure that adequate computer resources will be available to meet the future workload demands in a cost-effective manner while meeting the performance objectives. However, in most of the cases this future demand is uncertain. Therefore, stochastic models play a key role in capacity planning.

Next to optimal usage of available resources, in this thesis we consider a general class of *dynamic resource allocation* problems within a stochastic control framework. This class of problems arises in a wide variety of applications, each of which intrinsically involves resources of different types and demand with uncertainty and/or variability. The goal is to dynamically allocate capacity for each resource type in order to serve the uncertain/variable demand and maximize the expected net-benefit over a time horizon of interest based on the rewards and costs associated with the different resources.

In addition to variability in the service quality due to uncertainty in the demand over time, the system might be prone to chaotic behavior or network failures which would greatly influence resource availability. Chaotic behavior may for example be caused by unexpected interactions between systems, often due to configuration errors. In worst cases, a wrong configuration causes network or system failures. To exemplify the implications of such a failure, one could consider the online payment system of a given bank. A disruption in this system during a few days is likely to cause bankruptcy of the corresponding financial institution. Evidently, techniques to analyze 'what-if' scenarios and

to identify the best decision in a given situation are of utmost importance.

1.2 Queueing Theory and Markov Decision Processes

One way to evaluate (or predict) the performance of an existing computer system is to create a complete virtual prototype of the system and simulate the possible effect of altering system parameters. However, even a single simulation for a specific instance of the analyzed system may require a considerable amount of running time. Given that the search space of possible workloads and input parameters is often huge, vast numbers of simulations are needed to properly cover the range of alternative designs. A method to tackle such a challenge would be to characterize the workload stochastically and create a mathematical model. This way, by formal analysis of the parameter space one can study the cases under which the computer system is likely to perform well versus those under which it is likely to perform poorly.

In computer systems, many jobs require various system resources. In case only one job can use the specific resource at a given time, all other jobs requiring this resource would have to either wait in a queue or get canceled. Therefore, queues are at the heart of any computer system. The CPU uses a time-sharing scheduler to serve a queue of jobs waiting for CPU time. A computer disk serves a queue of jobs waiting to read or write blocks. A router in a network serves a queue of packets waiting to be routed. The router queue is a finite capacity queue, in which packets are dropped when demand exceeds the buffer space. Memory banks serve queues of threads requesting memory blocks.

Queueing theory [2, 17, 44] provides a powerful means to determine the time that the jobs spend in various queues in the system by modeling the uncertain system dynamics (e.g., arrival and service patterns) as stochastic processes. This way, by analyzing the corresponding mathematical models it becomes possible to predict the performance metrics of interest. Therefore, it is not surprising that queueing theory is one of the key analytical modeling techniques used for computer systems performance analysis [31, 73].

The items moving through a queueing system are often referred to as *cus-*

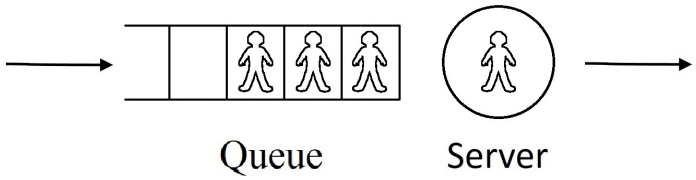


Figure 1.1: Illustration of a queue, in which customers wait to be served, and a server. The picture shows one customer being served at the server and three others waiting in the queue.

tomers as real human waiting line may be envisioned (see Figure 1.1). On the other hand, when a computer scientist writes about queues, he or she may often refer to the items moving through the queueing system as *jobs* since the application is often jobs circulating through a computer system which can be modeled as a network of queues. Therefore, in our thesis we will use the latter term. To analyze the behavior of the system one needs to keep track of the state of the queueing network. However, even small queueing networks may have a number of states so large as to produce computational problems. A great deal of research has gone into devising techniques to overcome this challenge.

One of the key concepts in queueing theory is the *Markov property*. A stochastic process has the Markov property if the future states of this process are independent of the past and depend only on the present. Such processes are called *Markov processes*. The Markov property makes a process easier to analyze since we do not have to keep track of the complete past trajectory to make a statement about the future. That is, it is not necessary to include in the state description the elapsed times in service or the time since the last arrival. This implies that knowing the present state of the process is sufficient to describe a queueing network at an instant of time, and furthermore, predict the future behavior of the Markov process.

Therefore, analysis of Markov processes allows predicting the system performance. Depending on the application, the performance might be measured by the mean delay in the system, the probability that the delay exceeds some threshold value, the maximum possible throughput of the system, the mean

number of servers being utilized (e.g., total power needs), or any other such metric. In some cases, it is required to predict more than one of these metrics to evaluate the implications of alternative system designs. Hence, although prediction is important, an even more important goal is finding how to upgrade the system to improve performance (e.g., is it better to buy a faster disk or a faster CPU). Typically to determine which resources would be best to improve one needs to identify the limiting factor of the system often referred to as the *bottleneck*. Note, however, that often without purchasing any additional resources, one can improve performance by deploying a smarter control policy for the existing resources.

Markov Decision Processes (MDPs) provide a mathematical framework for modeling control policies in a stochastic environment. MDPs are an extension of Markov processes [69, 84]. The difference is that in addition to the state space one defines a set of *actions* (allowing control of the system) and a set of *rewards* (providing incentive in good decision making). More precisely, at each time step, the decision maker may choose any action that is available in the current system state. The process responds at the next time step by moving into a new state with a probability that depends on the current state and the taken action. Furthermore, the process acquires the reward corresponding to this transition.

The core problem of MDPs is to find a decision policy maximizing the obtained rewards and given in the form of a function mapping *optimal* actions to each possible state of the system. Therefore, to ensure that the *optimal control policy* resembles the one which optimizes the performance metric of interest, one has to capture the relationship between the system states and the desired performance metric when defining the MDP rewards. As an example, to minimize the mean waiting time at a given queue in the system, one can define negative rewards, i.e., costs, for each waiting job at the corresponding server. Note that in this case the reward function depends only on the number of jobs in the queue, which is captured solely by the system state.

1.3 Evolutionary algorithms

Next to the queueing theory, the *machine learning* field also provides powerful techniques to evaluate the performance of a system and study its optimal control. More specifically, machine learning is a method of data analysis that

automates analytical model building. There are numerous machine learning algorithms designed to tackle challenges like computer vision, text and speech recognition, self-driving cars, etc. In this thesis, we focus on the so-called *evolutionary algorithms* [23,40] as they are suitable for analyzing the performance of a given system and finding an optimal decision strategy.

The general underlying idea behind evolutionary algorithms is that a *population* of candidate solutions to an optimization problem is gradually evolved toward better solutions via mechanisms inspired by biological evolution, such as *reproduction*, *mutation*, *recombination*, and *selection*. The candidate solutions to the optimization problem play the role of *individuals* in a population, and the *fitness function* determines the quality of the solutions. The environmental pressure causes natural selection (i.e., survival of the fittest) and this results in a rise in the overall fitness of the population. Therefore, a typical evolutionary algorithm requires (1) a *genetic representation* of the solution domain, and (2) a *fitness function* to evaluate the solution domain. Based on the fitness function, some of the better individuals are selected to seed the next generation by applying a reproduction scheme, i.e., crossing, and/or mutation to them. This procedure creates a set of new candidates, whose fitness is in turn also evaluated and compared. The process can be iterated until a solution with sufficient quality (i.e., fitness) is found or a predefined computational limit is reached. The following steps summarize the described procedure:

Step 1:
Randomly generate the initial population of individuals.

Step 2:
Evaluate the fitness of each individual in that population.

Step 3:
Repeat the following steps until termination:

Step 3.1: Select the 'best-fit' individuals for reproduction.
Step 3.2: Recombine and mutate to generate new individuals.
Step 3.3: Evaluate the fitness of new individuals.
Step 3.4: Replace least-fit part of the population with new individuals.

We emphasize that many parts of the evolutionary process are inherently stochastic. For example, in Step 3.1 fitter individuals have a higher probability to be selected than less fit ones, but this does not imply that the weak

individuals have no probability to become a parent or to survive. Next to that, when crossing candidate solutions in Step 3.2 the choice of which parts will be recombined is to certain extent random. Similarly for mutation, the parts that will be mutated within an individual, and also the new parts replacing them, are both chosen in a stochastic manner. Therefore, there are two fundamental forces that drive the population to evolve and produce solutions of better quality in consecutive generations:

- **Variation**, creating the necessary diversity and thereby facilitating novelty.
- **Selection**, acting as a force pushing quality.

1.4 Practical solutions

In practice, the usefulness of a given algorithm is evaluated based on two metrics: *accuracy* and *computational time*. In real-world applications, the cost of the method in terms of solution time is often as important as the method's accuracy in analyzing the performance or in identifying the optimal decision policy. Certain techniques become computationally prohibitive, even for systems of moderate complexity.

This implies that the concept of *scalability* is particularly important when analyzing a given system. Scalable solutions are capable to cope with an increase in the system size. Therefore, an extremely valuable type of techniques is the class that obtains *closed-form* results (examples of such techniques include [9,47,63,89]). Closed-form expressions allow instant calculation of a new solution for any change in the parameters. In addition, algebraic formulas give the opportunity to study the *performance sensitivity* in the system parameters and assess the *robustness* in the parameter estimation.

In this context, there is a trade-off when modeling the analyzed system. Naturally, less *complex* models allow for more *accurate* and more *computationally efficient* solutions. One conceptional difference between queueing theory and machine learning is in the way the two fields approach this trade-off. Typically using machine learning techniques one would analyze data describing the past behavior of the system. Therefore, the obtained insights are rarely transferable to other systems with different past realizations. Next to that, most of

the algorithms for solving the system (e.g., predicting the performance) involve approximations and do not allow sensitivity analysis or efficient recalculation procedure in case of a change in the system specifics. On the other hand, queueing theory generally deals with the analysis of a simplified model of the system. The goal is to develop a model that (1) captures the most important aspects of the system in question, and (2) can be solved with great precision by an *efficient* numerical algorithm. Therefore, the results of the queueing analysis are applicable to any system that shares the modeled system dynamics.

A common way to combine these two approaches is to use machine learning algorithms on a past data to uncover important patterns in the system behavior and predict certain system parameters (e.g., the arrival rates). Next, these patterns are modeled within a queueing framework and the optimal decision policy is derived from an MDP analysis.

In this thesis, we present a novel technique to bring machine learning and queueing theory together. Contrary to the former method, we start with queueing theory and generate data by solving the corresponding queueing model. Consequently, we use a specific type of evolutionary algorithm on this data to produce a closed-form solution of the given model. In addition, we enhance the *accuracy* and the *efficiency* of the evolutionary algorithm by incorporating insights derived from queueing theory analysis.

1.5 Outline of the thesis

The main contribution of this thesis is the development of new methods to evaluate the performance, and furthermore, to optimally control various ICT service chains. Therefore, our research focuses on both the theoretical results and the practical applications of these techniques. The explored concept of combining the fields of queueing theory and machine learning proves to be a highly promising idea. We believe that the pioneering studies in this thesis are setting only the beginning of a new research direction that offers both remarkably accurate and easily scalable solutions to real-world problems.

One of the most common ways to model two dependent (e.g., connected in a chain) services is by a tandem queue. Namely, the output of one server becomes the input of the second node. As an example of a system with such a feature,

in Chapter 2 we analyze a simple *database caching mechanism*. We model the system as a two-node tandem of single-server queues with holding costs and a start-up cost for the second server. To optimize the long-term average costs associated with the system, both of the servers can be switched on or off at any point in time. For this model we develop an algorithm approximating the optimal control of the servers that minimizes the long-term average costs. The presented technique proves to be simple, intuitive and at the same time highly accurate and computationally inexpensive.

In Chapter 3, we further explore methods to evaluate the performance of database caching mechanisms and subsequently optimize these mechanisms. In contrast to the preceding chapter, this one focuses on the so-called 'write-behind' caching mechanism. Due to the dependency between the application level and the cache, we model this system as a single server queue with a buffer (instead of a tandem network). In the analyzed setting, requests arriving in the system require *pre-processing* (i.e., being written in the cache) before being *post-processed* (i.e., being written in the database). This mechanism allows storing a number of requests in the buffer before serving them all at once as a *batch*. We analyze the system with a fixed batch size and derive the corresponding steady-state behavior. Furthermore, we present a method to obtain a function that captures the relationship between the size of the batch and the mean waiting time in the queue. Using our technique, we show how to determine in a highly efficient and scalable way the 'optimal' batch size, i.e., the one that minimizes the mean waiting time.

In Chapter 4, we introduce a novel approach to solving MDPs with an optimal policy that represents a threshold function. The method is based on a specific type of evolutionary algorithms, the so-called Symbolic Regression. Our technique results in an approximation of the optimal decision policy given as a *closed-form* expression. Furthermore, we explore how the performance of the evolutionary algorithm can be improved by taking into account specifics of the corresponding MDP framework. To illustrate the technique we consider two running examples. The first one is a generalization of the 'write behind' cache studied in the preceding chapter. The second one is an $M/M/1$ queueing model with customer rejection. Applying our approach on these two running examples results in analytic expressions that approximate the optimal control policy with great accuracy. Next to that, we show that the obtained mathematical formulas allow sensitivity analysis of the system parameters.

The potential of this technique is further studied in Chapter 5, where we

extend it to evaluate the relevant performance metrics of a given queueing network rather than the optimal decision policy. We show that the method is able to benefit from insights and results derived from analysis of the queueing model in specific cases (e.g., high or low traffic regimes, fluid or diffusion analysis). Namely, we present a way to incorporate the vast body of research on queueing theory into the evolutionary algorithm. By expanding on these results, our technique generates an expression that is applicable to a broader range of parameters.

Finally, Chapter 6 abstracts over the models considered to this point and adopts a high-level view of ICT service chains. Namely, we analyze a layered queueing network without restricting the possible number of servers in the system. The nodes are organized in a nested fashion and therefore influence each others service rate. This strong dependency between the servers has proven to make such networks complicated and difficult to study. Moreover, even when the utilization rates and the saturation throughput of the servers are known, determining the limiting factor in the network is far from trivial. We present a simple, computationally tractable and nevertheless highly accurate method for approximating the above-mentioned performance measurements. In addition, we propose an extension to the intuitive *'slowest server rule'* for identification of the bottleneck, and show through extensive numerical experiments that this method works very well.

Chapter 2

Control of a Tandem Queue with Start-up Costs

Many systems across a broad range of applications involve sequential processing steps and can be modeled as tandem queues. Next to performance metrics such as waiting time and sojourn time distributions, most often there are also costs associated with running such systems (e.g., holding costs at the queues, operational costs for the servers). Therefore, identifying a control policy that minimizes the running costs of the system is of great interest both from a theoretical and a practical point of view. In the present chapter, we study database caching mechanisms as a specific example of a tandem queue.

The majority of modern data centers are using shared storage solutions to improve performance. A main challenge is coordination of data persistence between the application and the storage layer. In this context, there is a delicate *balance* between the *achievable throughput* on the one hand and *contention* on the other hand. The contention at the application layer can be highly reduced by first writing the data on a cache, and subsequently synchronizing the cached data with the storage layer. The question is how to properly configure the *cache-synchronization* parameters, e.g., when to perform the cache-synchronization. To study this decision problem we model the caching mechanism as a two-node tandem queue with linear holding costs and a start-up cost for the second server, i.e., the cache. We present an intuitive, easy to understand, and at the same time accurate, algorithm to approximate the optimal control policy of the cache. Extensive numerical experimentation shows that the approximation works extremely well for a wide range of parameter combinations.

The work in this chapter is based on A.V. Hristov, S. Bhulai, J.W. Bosman and R.D. van der Mei. Control of a tandem queue with a start-up cost for the second server. To appear in *Stochastic Models* [35].

2.1 Introduction

Queueing systems in which the departures from one server become the arrivals to a downstream server are often modeled as tandem queues. These models have proven to be very challenging to analyze [18, 53, 61, 88], and despite decades of research, there are still many open problems without an analytic solution [3, 11, 50, 58]. At the same time, tandem queues abound in applications, and therefore, the study of these systems is also important for practice. Moreover, in cases when one has a certain control over the network, the analysis of the model becomes crucial for optimal management of the application [56, 90]. For example, in some practical situations, it is possible to temporarily switch off certain nodes in order to protect servers further down the network from overflow. Systems with this feature can be found in traffic control [29], transportation, manufacturing, and many other fields [34, 45, 54, 98].

One particular application that can be modeled as a tandem network and uses such control techniques is the caching in computer databases (illustrated in Figure 2.1). By implementing the cache mechanism, the contention at the application level might be strongly reduced by first accumulating write operations in the cache and only afterwards processing them in the database. Furthermore, to avoid overload, new writes to the cache are blocked whenever there are already a certain number of requests in it. The corresponding threshold value is generally referred to as the *high water mark*. Motivated by this application, in this chapter we study two-node tandem queueing networks where one can switch on/off any of the nodes.

Note that switching off a server results in reducing the service capacity of the system. Therefore, there is a trade-off for such tandem queues in balancing the requirement for 'good' performance while minimizing the resource usage. One common technique to capture this trade-off is to introduce different penalties (i.e., holding costs) for jobs waiting in the various queues [16, 75]. This way, one combines both metrics (e.g., the sojourn time spent in the system and the required resources) in a single indicator. The problem is, therefore, reduced to the following question: "How to control the system in order to minimize the costs associated with it?". However, this challenge is still analytically intractable, in spite of the considerable amount of research and the great importance from a practical point of view.

Next to that, changing the 'state' of a server in such tandem networks (i.e.,

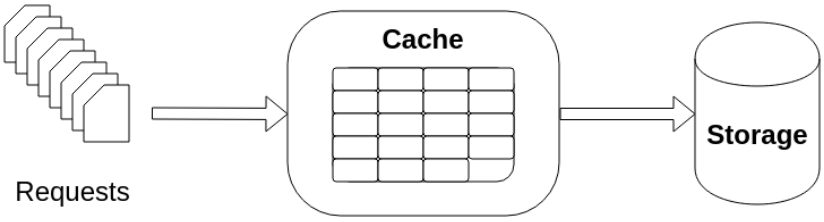


Figure 2.1: Cache used for databases.

shutting it down or starting it up) might also require resources on its own. For instance, there are certain costs associated with establishing a data transfer between the cache and the corresponding database. Motivated by this, we study a system with two servers that has a start-up cost associated with the second node. Our framework can be considered as a generalization of the ones researched in [28, 75].

In the following section, we introduce the model. In Section 2.3 we illustrate what the optimal decision policy for such systems looks like. Subsequently, in Section 2.4 we propose an approximation algorithm for obtaining the optimal control policy. The accuracy of the presented technique is discussed in Section 2.5. Finally, in Section 2.6 we conclude with a summary and discuss ideas for possible further research.

2.2 Model

We consider a two-node tandem queue with single servers at both queues. Jobs arrive at the first node and after receiving service there, they are transferred to the second one. Subsequently, jobs are served at the second node and leave the system. We assume Poisson arrivals with rate λ at queue 1 and exponential service times with mean $\beta_i = 1/\mu_i$ at server $i \in \{1, 2\}$. Moreover, both nodes are serving at maximum one job at a time according to the First In First Out (FIFO) regime. The queues are taken to be of a finite size N and K for queue 1 and queue 2, respectively.

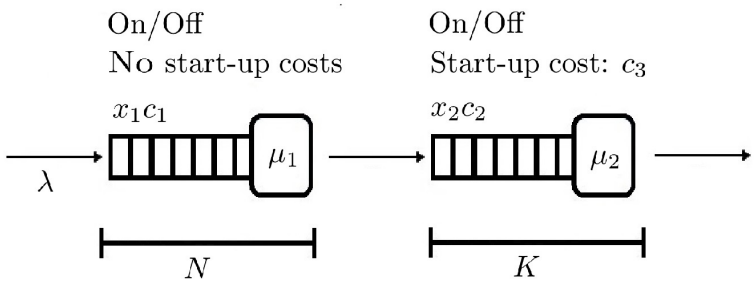


Figure 2.2: Illustration of the model.

Each job in queue $i \in \{1, 2\}$ generates a holding cost $c_i \geq 0$ per time unit. Moreover, there is a start-up cost for the second server, denoted by $c_3 \geq 0$. To optimize the cost, one can control the system by switching on or off any of the two servers at any point in time with the exception of the following two cases:

- the first server cannot be working when the buffer space at the second node is full (i.e., when there are K jobs in the second queue);
- the second server switches off whenever it becomes idle.

Furthermore, to avoid a trivial solution of never switching off the first server, we assume that $c_2 > c_1$. This way, for certain system states when the second server is not operating it might be optimal to keep the jobs in the first queue. Next to that, the start-up cost of server 2 creates an additional trade-off between serving jobs at the second node as soon as possible and waiting for enough jobs before switching on the server. Therefore, the goal is to identify the decision policy minimizing the average costs per time unit. Figure 2.2 gives an illustration of the model.

To calculate the optimal policy, we formulate the system as a Markov Decision Process (MDP) with state space $\mathcal{S} := \{0, 1, \ldots, N\} \times \{0, 1, \ldots, K\} \times \{0, 1\}$, where state (x_1, x_2, s_2) corresponds to having x_i number of jobs at node $i \in \{1, 2\}$ and server 2 being *off* for $s_2 = 0$ or *on* for $s_2 = 1$. Note that we do not have to explicitly include the state of node 1 in our model due to the following two model properties:

- The service times are exponentially distributed and exhibit the memoryless property.
- The first server can be switched on/off instantaneously at no cost.

The action space consists of four possible actions $a \in \mathcal{A} = \{1, 2, 3, 4\}$, defined as follows:

- 1 - switch off both servers;
- 2 - switch on server 1 and switch off server 2;
- 3 - switch on server 2 and switch off server 1;
- 4 - switch on both of the servers.

Now, we can formulate the Bellman equations for this MDP:

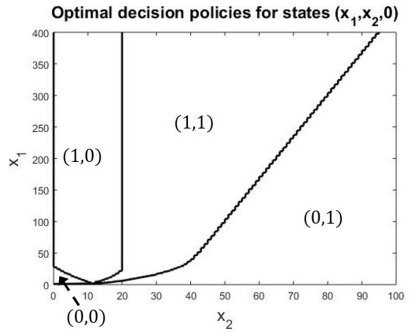
$$g + V(x_1, x_2, s_2) = c_1 x_1 + c_2 x_2 + \min_{a \in \mathcal{A}} T_a(x_1, x_2, s_2),$$

where V denotes the value function. Moreover, g denotes the long-term average costs per time unit and $T_a(x_1, x_2, s_2)$ (for $a \in \mathcal{A}$ and $(x_1, x_2, s_2) \in \mathcal{S}$) are given by:

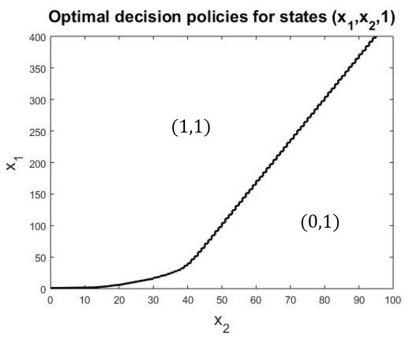
$$\begin{aligned}
T_1(x_1, x_2, s_2) &:= \lambda V\left(\min\{x_1 + 1, N\}, x_2, 0\right) + (1 - \lambda) V(x_1, x_2, 0); \\
T_2(x_1, x_2, s_2) &:= \lambda V(\min\{x_1 + 1, N\}, x_2, 0) + \mu_1 V(x_1 - 1, x_2 + 1, 0) \\
&\quad + \mu_2 V(x_1, x_2, 0); \\
T_3(x_1, x_2, s_2) &:= c_3(1 - s_2) + \lambda V(\min\{x_1 + 1, N\}, x_2, 1) \\
&\quad + \mu_2 V\left(x_1, x_2 - 1, I\left(x_2\right)\right) + \mu_1 V(x_1, x_2, 1); \\
T_4(x_1, x_2, s_2) &:= c_3(1 - s_2) + \lambda V(\min\{x_1 + 1, N\}, x_2, 1) \\
&\quad + \mu_1 V(x_1 - 1, x_2 + 1, 1) + \mu_2 V\left(x_1, x_2 - 1, I\left(x_2\right)\right),
\end{aligned}$$

where the rates are scaled in such a way that $\lambda + \mu_1 + \mu_2 = 1$. Furthermore, we define $I(x_2) = 0$ if $x_2 = 1$, and $I(x_2) = 1$ otherwise. Moreover, recall that some actions are not possible under certain values for x_1 and x_2. More precisely,

- for $x_1 = 0$ or $x_2 = K$, actions 2 and 4 (i.e., T_2 and T_4) are not permitted as it is not possible to switch on server 1 when the first queue is empty, or when the second buffer is full;



(a) $s_2 = 0$, i.e., second server is off



(b) $s_2 = 1$, i.e., second server is on

Figure 2.3: Optimal decision policy for $\mu_1 = 2.2$ and $\mu_2 = 4$.

- for $x_2 = 0$, actions 3 and 4 (i.e., T_3 and T_4) are not permitted as it is not possible to switch on server 2 when the second queue is empty.

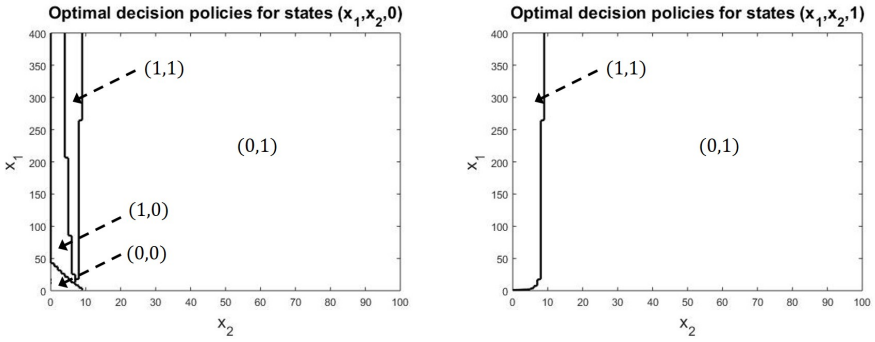
2.3 Optimal policy

In this section, we study the optimal decision policy based on results derived by numerically solving the MDP by applying the value iteration technique. As an example, we use two systems with the following parameters: $N = 750$, $K = 200$, $c_1 = 0.1$, $c_2 = 1$, $c_3 = 1500$, $\lambda = 1$. The service rates of the servers are taken to be:

- $\mu_1 = 2.2$ and $\mu_2 = 4$ in the first example,
- $\mu_1 = 4$ and $\mu_2 = 2.2$ in the second example.

The corresponding optimal decision policies are shown in Figures 2.3 and 2.4. Note that we present only the states for which $0 \leq x_1 \leq 400$ and $0 \leq x_2 \leq 100$ in order to exclude possible boundary effects.

Analyzing numerous cases for various parameter sets, we suspect that the optimal policy for such a queueing network is of a threshold type. We denote the different state-space regions, where a certain action is optimal, as follows:



(a) $s_2 = 0$, i.e., second server is off (b) $s_2 = 1$, i.e., second server is on

Figure 2.4: Optimal decision policy for $\mu_1 = 4$ and $\mu_2 = 2.2$.

- *region* $(0, 0)$ - the optimal action is 1, namely, both of the servers should be off;
- *region* $(1, 0)$ - the optimal action is 2, namely, the first server should be on, whereas the second one - off;
- *region* $(0, 1)$ - the optimal action is 3, namely, the first server should be off, whereas the second one - on;
- *region* $(1, 1)$ - the optimal action is 4, namely, both of the servers should be on.

Note that the numerical approach used in the current section can be used only for systems of a relatively small size. In many real-world instances the large buffer sizes, N and K, require a more computationally-efficient algorithm. As an example, to optimize the running costs associated with the database caching mechanism, one needs a method different than the value iteration technique, as the latter becomes computationally unfeasible. On the other hand, as discussed in Section 2.1, even the special case of no start-up cost associated with the second server (i.e., the special case of $c_3 = 0$) has withstood an analytic analysis so far. Therefore, the goal of our research is to develop a scalable algorithm with respect to N and K that approximates the optimal policy.

2.4 Approximation technique

In the following, we present our technique for approximating the optimal policy. Our approach to obtaining the control policy is to estimate the regions described in Section 2.3, rather than deriving an approximation of the value function. To do this, we decompose the original system into two sub-models. We refer to the parameters for these sub-models by appending a superscript $^{(1)}$ and $^{(2)}$, respectively.

Sub-model 1 consists of two single-server queues in a tandem setting. The input is modeled as a Poisson process and the service times of the jobs are assumed to be independent and exponentially distributed. There are holding costs $c_1^{(1)}$ and $c_2^{(1)}$ per time unit for each job waiting at the corresponding queue. In contrast to the main network analyzed in this chapter, there is no start-up cost for the second server. The optimal policy for such a system is proven to be defined by a switching curve, see [75]. In other words, for any given number of jobs $x_1^{(1)}$ at the first queue, there is a threshold value $S^{(1)}(x_1)$ for the number of jobs at the second node, $x_2^{(1)}$. If $x_2^{(1)}$ exceeds the corresponding threshold, then it becomes optimal to switch off the first server. We denote the arrival rate as $\lambda^{(1)}$ and the service rates as $\mu_1^{(1)}$ and $\mu_2^{(1)}$.

Sub-model 2 consists of an $M/M/1/N$ queue with holding costs $c_2^{(2)}$ per time unit for each waiting job and a start-up cost $c_3^{(2)}$. One can control the system by switching on/off the server. As in the main system analyzed in this chapter, the server cannot be idle, i.e., it switches off whenever there are no jobs. The optimal policy for such a system is proven to be a threshold policy, see [33]. More specifically, it is optimal to switch on the server when the number of jobs in the system, $x_2^{(2)}$, exceeds a given threshold value, $S^{(2)}$. We denote the arrival rate for this system as $\lambda^{(2)}$ and the service rate as $\mu_2^{(2)}$.

Recall from Section 2.1 that an analytic solution for sub-model 1 is unavailable. However, there are a number of studies that develop efficient numerical algorithms for calculating the optimal threshold levels. Therefore, we assume that the optimal policy for sub-model 1 is given.

On the other hand, there is a closed-form solution for the optimal threshold

value $S^{(2)}$ for sub-model 2, given by:

$$S^{(2)}\left(\lambda^{(2)}, \mu_2^{(2)}, c_2^{(2)}, c_3^{(2)}\right) = \sqrt{2\lambda^{(2)}\left(1 - \lambda^{(2)}/\mu_2^{(2)}\right) c_3^{(2)}/c_2^{(2)}}. \tag{2.1}$$

Therefore, one can derive the optimal policy also for this sub-model.

Intuitively, it might seem that simply combining the solutions of those two sub-models would give a good approximation of the optimal policy for the main model. However, this is not the case, mainly due to the complex dependency between the working regime of the first server and the arrival rate at the second server. Therefore, in our algorithm, we try to take into account this dependency.

2.4.1 Second server switched off

In this subsection, we show how to approximate each of the three switching curves (see Figures 2.3a and 2.4a) in the optimal decision policy in case the second server is switched off (i.e., for states $(x_1, x_2, 0)$, where $0 \leq x_1 \leq N$ and $0 \leq x_2 \leq K$). Considering the $(0, 0, 0)$ state (i.e., an empty system) as a reference point, we denote the *first* threshold levels to correspond to the decision when to switch on the first server, i.e., the curve separating *region* $(0, 0)$ from *region* $(1, 0)$. We characterize this curve by the function $p_1(x_1)$, where $0 \leq x_1 \leq N$. The value of the function gives the threshold value of x_2 for the corresponding $0 \leq x_1 \leq N$. Consequently, as the *second* curve, we consider the one describing when to switch on the second server, i.e., the transition between *region* $(1, 0)$ and *region* $(1, 1)$. Similarly, we introduce the function $p_2(x_1)$ that defines this curve. Finally, the *third* set of threshold levels, $p_3(x_1)$, gives the states for which it is optimal to switch off the first server and have only the second one working - *region* $(0, 1)$. In the remainder of this subsection we outline how to estimate those three switching curves.

Switching on the first server
Under certain system parameters, the optimal decision is to keep both servers off whenever there are not many jobs in the network. Namely, if $c_2 > c_1$ it would be optimal to keep jobs waiting at the first queue rather than at the second one. Next to that, intuitively, the lower the load of the network is, the bigger this region should be. As a first step of approximating this region, we determine the endpoints of the corresponding switching curve. Namely, we are

interested in the value of $p_1(0)$ and the specific i_1' for which $p_1(i_1') = 0$. To obtain these values, we use the following equations:

$$p_1(0) = S^{(2)}(\lambda, \mu_2, c_1 + c_2, c_3),$$
$$i_1' = \left\lfloor \frac{S^{(2)}(\lambda, \mu_2, c_1, c_3) + S^{(2)}(\mu_1, \mu_2, c_1 + c_2, c_3) + S^{(2)}(\lambda, \mu_2, c_2, c_3)}{3} \right\rfloor,$$

where the operator $S^{(2)}$ denotes the threshold value determined by solving sub-model 2 with the corresponding parameters (see Equation (2.1)) and $\lfloor x \rfloor$ denotes the floor function that outputs the largest integer less than or equal to x.

The idea to take the average value over three solutions of sub-model 2 for obtaining i_1' is to incorporate three different regimes of the system. The first one being the regime just before taking the decision to switch on the first server and having jobs only at the first server. The second regime is when server 1 is switched on and there is a number of jobs in queue 1. This implies that the arrival rate to the second node is μ_1. In the third regime, the queue at server 1 is empty, and hence, there are no holding costs acquired there and the arrival rate to the second queue is λ.

Note that if $\mu_1 > \mu_2$, then sub-model 2 with arrival rate μ_1 and service rate μ_2 becomes unstable, i.e., the inflow to the system exceeds the outflow. It is clear that in such case the optimal policy is to switch on the server whenever a job arrives, and therefore we take $S^{(2)}(\mu_1, \mu_2, c_1 + c_2, c_3) = 1$.

As a second step, we approximate the curve by a straight line that connects the two endpoints $p_1(0)$ and i_1'. This results in the following switching curve:

$$p_1(x_1) = p_1(0)\left(1 - \frac{x_1}{i_1'}\right)^+,$$

for $0 \leq x_1 \leq N$.

Switching on the second server
Once there is a certain number of jobs at the second queue, it becomes optimal to switch on the corresponding server. Following the same approach as the previous case, we first estimate the endpoints of the switching curve $p_2(0)$ and $p_2(N)$. We use sub-model 2 as follows:

$$p_2(0) = S^{(2)}(\lambda, \mu_2, c_2, c_3),$$
$$p_2(N) = S^{(2)}(\mu_1, \mu_2, c_1 + c_2, c_3),$$

where again $S^{(2)}(\mu_1, \mu_2, c_1 + c_2, c_3) = 1$ if $\mu_1 > \mu_2$. Our reason for choosing these parameters for sub-model 2 is that when there are no jobs at server 1, there are no holding costs c_1 acquired. Furthermore, due to the fact that in sub-model 2 the first queue is an $M/M/1/N$ queue, the inflow to server 2 equals the inflow to the tandem system. On the other hand, when the first queue is full, the arrival rate to the second server becomes μ_1, and furthermore, one should take into account also the holding costs c_1.

As a next step, one has to approximate the shape of the curve. In this case, a straight line proved to be a relatively inaccurate fit for the switching curve. Analysis of the optimal policies, which were derived by numerically solving systems with small buffer sizes N and K, lead us to the following fit:

$$p_2(x_1) = p_2(N) + \frac{p_2(0) - p_2(N)}{\sqrt{x_1}},$$

for $0 < x_1 < N$. We found this to be a good approximation while at the same time has a simple, tractable form.

Switching off the first server
For sufficiently low holding costs at the first queue there will be certain cases where it is optimal to switch off the first server. This will result in jobs waiting at the first queue rather than waiting at the more expensive second queue. We estimate the switching curve separating *region* $(1, 1)$ and *region* $(0, 1)$ by using the results obtained from our approximation algorithm so far, together with the solution for sub-model 1. More precisely, we obtain $p_3(x_1)$ for $0 \leq x_1 \leq N$, by the following equation:

$$p_3(x_1) = p_2(x_1) + S^{(1)}(x_1; \lambda, \mu_1, \mu_2, c_1, c_2),$$

where the operator $S^{(1)}$ denotes the threshold value determined by solving sub-model 1 with the corresponding parameters.

2.4.2 Second server switched on

It is clear that if the second server is working it is optimal to keep it on. Therefore, the optimal decision when $s_2 = 1$ can be either action 3 or action 4, corresponding to *region* $(0, 1)$ and *region* $(1, 1)$. Based on studying the conducted numerical examples and evaluating the performance of different approaches, we decided to approximate the switching curve between those two

regions in the same manner as in Section 2.4.1, i.e., when $s_2 = 0$. Namely, the switching curve is given by the points $p_3(x_1)$, where $0 \leq x_1 \leq N$. This way, one can directly use the results derived from the above-described procedure, which implies that this case does not increase the complexity of the algorithm.

2.5 Results

In this section, we evaluate the performance of the approximation algorithm. Recall that our method is based on estimating the switching curves of the optimal policy. Hence, the main idea is to derive a graph as similar as possible to the optimal decision policy graph (see Figures 2.3, and 2.4). However, in practice, the goal of 'optimizing' the system is often times to reduce the average costs. Therefore, although our algorithm is approximating the various switching curves, in this section, we will not examine how close are the approximated fitting functions to the optimal switching curves. Instead, we present the relative difference, denoted as E_r, between the acquired long-term average cost if one uses the decision policy obtained by our procedure, g^{est}, and the optimal one, g^{opt}, derived by numerically solving the MDP. More precisely, the relative difference is defined by:

$$E_r = \frac{|g^{est} - g^{opt}|}{g^{opt}} \times 100\%.$$

To cover the full spectrum of parameter values, we created multiple test suites with approximately 2000 parameter sets in total. We examine systems with loads in the range $[0.1, 0.9]$ for each of the queues, and ratios between the two holding costs: $c_1/c_2 \in [0.1, 0.9]$. More precisely, we varied the parameters as follows:

- systems where $\mu_1 < \mu_2$. We take μ_2 fixed at 10, while varying μ_1 from 1.1 to 9.9 with a step size of 1.1;
- systems where $\mu_1 > \mu_2$. We take μ_1 fixed at 10, while varying μ_2 from 1.1 to 9.9 with a step size of 1.1;
- systems where $\mu_1 = \mu_2$. Once again we take the same range of values for the service rates - from 1.1 to 9.9 with a step size of 1.1.

In all three test suites, we further varied c_1 between 0.1 and 0.9 with a step

of 0.1 and $c_3 = 500, 1000$ or 1500. Next to that, we fixed $\lambda = 1$. The reason is that only the ratio between the various service rates is important with respect to the approximation error. This comes from the fact that scaling the rates is equivalent to scaling the time, which does not influence the results. Due to the same reasoning, we also fixed one of the costs: $c_2 = 1$. We note that in all tests the start-up costs are considerably higher than the holding costs. We chose such values to ensure that the optimal decision policy is not a trivial one, i.e., switching the second server whenever there are jobs in the queue.

To evaluate the accuracy of the algorithm, we compared the average costs acquired by implementing the approximated optimal policy to those derived from numerically solving the MDP. Due to the computational complexity of the numerical approach, we created a benchmark only for relatively small buffer sizes. Therefore, we composed two test suits with the following system configurations: $N = 100, K = 50$, and $N = 1000, K = 100$.

The results of the conducted tests are shown in Tables 2.1 and 2.2. We present the approximation error of the algorithm in each of the three values for the start-up cost c_3. Furthermore, motivated by the differences of the optimal policies from Figures 2.3 and 2.4, we aggregate the results according to the following three cases for the service rates: $\mu_1 < \mu_2$, $\mu_1 > \mu_2$ and $\mu_1 = \mu_2$. Finally, next to the median of the relative error, we also list the 80^{th}, the 90^{th}, and the 95^{th} percentile for the corresponding nine test suites.

Based on the results from Tables 2.1 and 2.2, we conclude that our algorithm's performance is not significantly influenced by the value of the start-up cost, c_3. Next to that, the errors for the two system sizes and the three service rate configurations are also comparable. Nevertheless, the 80^{th} percentiles are two to three times larger than the median, which indicates that for specific systems the algorithm performs significantly worse than on average.

Therefore, we further studied the cases corresponding to the highest approximation errors. We found a common pattern among these systems, which shows that the algorithm performed worst in the following three service rate configurations:

- $\mu_1 = 1.1, \mu_2 = 9.9$;
- $\mu_1 = 9.9, \mu_2 = 1.1$;
- $\mu_1 = 1.1, \mu_2 = 1.1$.

Table 2.1: Approximation errors of the algorithm for test cases with $N = 100$ and $K = 50$.

Start-up cost	Service rates	Median E_r	E_r percentiles 80^{th}	90^{th}	95^{th}
$c_3 = 500$	$\mu_1 < \mu_2$	1.27%	3.55%	6.69%	8.44%
	$\mu_1 > \mu_2$	1.95%	5.29%	8.82%	11.38%
	$\mu_1 = \mu_2$	1.97%	5.22%	8.35%	9.28%
$c_3 = 1000$	$\mu_1 < \mu_2$	1.68%	4.27%	9.12%	10.31%
	$\mu_1 > \mu_2$	1.97%	4.34%	7.28%	11.14%
	$\mu_1 = \mu_2$	1.67%	4.35%	8.42%	10.20%
$c_3 = 1500$	$\mu_1 < \mu_2$	1.89%	4.68%	9.64%	11.25%
	$\mu_1 > \mu_2$	1.85%	4.43%	7.94%	10.29%
	$\mu_1 = \mu_2$	1.41%	4.26%	7.27%	9.85%

Note that these parameter sets correspond to systems where both of the servers operate under either very low or very high load. We believe that this feature differentiates the above-described cases from the rest and perhaps creates specific server dynamics that are not fully captured by our approximation algorithm.

Nevertheless, in all nine test suites, the median for the relative error is less than 2%. We believe that this approximation accuracy together with the intuitive and easy to implement nature of our algorithm makes it a suitable choice in practice.

2.6 Conclusion

In this chapter, we analyzed the control of a two-node tandem queuing network with holding costs at both queues. Next to the holding costs acquired at each time unit we introduced a start-up cost for the second server. This framework allows modeling of various real-world systems that exist in computer science, logistics, manufacturing, and many other fields. We presented

Table 2.2: Approximation errors of the algorithm for test cases with $N = 1000$ and $K = 100$.

Start-up cost	Service rates	Median E_r	E_r percentiles 80^{th}	90^{th}	95^{th}
$c_3 = 500$	$\mu_1 < \mu_2$	1.22%	3.50%	6.64%	8.34%
	$\mu_1 > \mu_2$	1.93%	5.27%	8.78%	11.28%
	$\mu_1 = \mu_2$	1.96%	5.20%	8.32%	9.24%
$c_3 = 1000$	$\mu_1 < \mu_2$	1.67%	4.23%	8.92%	10.01%
	$\mu_1 > \mu_2$	1.91%	4.28%	7.24%	11.02%
	$\mu_1 = \mu_2$	1.63%	4.31%	8.36%	10.09%
$c_3 = 1500$	$\mu_1 < \mu_2$	1.86%	4.58%	9.34%	10.75%
	$\mu_1 > \mu_2$	1.83%	4.40%	7.90%	10.18%
	$\mu_1 = \mu_2$	1.38%	4.25%	7.24%	9.80%

an efficient algorithm to approximate the optimal decision policy for such tandem models. The conducted numerical evaluation showed that our technique is highly accurate for any parameter combination tested.

Another advantage of our algorithm is that it is simple and rather intuitive, which facilitates its implementation in practice. Therefore, together with the achieved average costs within a few percentages of the optimal, we believe that the technique is a reasonable choice for managing such systems.

Finally, we address a few topics for further research. We find promising the idea of extending the algorithm in order to solve tandem queues composed of more than two nodes. Next to that, it might be of a practical interest to also study a model in which there is a start-up cost for the first server or a start-up time associated with the nodes.

Chapter 3

Analysis and Control of a Single Server Queue with Backlog Processing

In Chapter 2, we analyzed the generic cache mechanisms by modeling them as a two-node tandem queueing network. However, certain cache implementations introduce system dynamics that require different modeling. Motivated by this, in the current chapter we focus on a specific cache mechanism and model the corresponding system as a single server that switches between two processing stages. The first stage corresponds to pre-processing jobs one at a time. Next, the job is accumulated into a finite capacity buffer of jobs that require post-processing. In the second stage jobs in the buffer are processed, possibly multiple at once. Once processed, the jobs leave the system. Switching between modes involves initialization time. Therefore, accumulating jobs into a batch may greatly improve the overall performance. On the other hand, a larger batch size will increase the waiting time of new jobs.

In this study, we analyze such a system with a fixed batch size and derive the corresponding steady-state behavior. Next to that, we prove that the case of a batch of size two can be solved analytically by finding the roots of a polynomial of degree three. In addition, we obtain an approximation for a system with a large batch size. The main contribution is that we present a technique that combines the above-described results and obtains a function that captures the relationship between the size of the batch and the mean waiting time in the queue. Using our method, one can determine in a computationally efficient way the 'optimal' group size, i.e., the one that minimizes the mean waiting time. Extensive numerical experimentation shows that the approximation works extremely well for a wide range of parameter combinations.

The work in this chapter is based on A.V. Hristov, J.W. Bosman, R.D. van der Mei and S. Bhulai. Analysis and control of a single server queue with backlog processing (2018) [39]. *Submitted.*

3.1 Introduction

The majority of modern data centers are using shared storage solutions in order to improve performance (see, e.g., [70]). Caching mechanisms play an important role in the reduction of storage contention. Caching solutions operate as a data exchange buffer between application and storage. We are in particular interested in the behavior of write requests. The 'write behind' [93] caching mechanism is a means to improve write performance. In contrast to the more simple 'write through' caching, where a write to the cache results in an immediate write to the application, in a 'write behind' results are accumulated in the cache and processed at a specific point in time in a batch. In such a way, the 'write behind' mechanism reduces the number of writes to storage. Moreover, it improves performance as applications handle asynchronous requests faster than synchronous updates with a few records each [93]. A main challenge is to properly balance the *trade-off* between the *achievable throughput* on the one hand and *contention* at the application layer on the other hand. The contention can be highly reduced by first writing the data on the cache, and subsequently synchronizing the cached data with the storage layer. As a consequence of the 'write behind' mechanism, requests might have to wait for the cache in two cases: namely, the cache can be busy managing another write, or transferring the accumulated results to the storage. In general, this 'flush' of updates from the cache to storage is triggered by one of the following events: a specific time elapsed, or a given number of results that were written since the last flush [59].

The contribution in this study is fourfold. First, we propose a new performance model that captures the trade-off between throughput and contention on synchronizing data. Second, for this model we propose a simple analytic approximation of the expected contention level as a function of the cache-level threshold K. Third, we use this method to approximate the optimal cache-level threshold. This enables caching mechanisms to promptly adapt to changing circumstances. Fourth, we show by extensive numerical experimentation that the approximation works extremely well for a wide range of parameter values.

The model proposed in this research is related to the class of so-called polling systems, which have been extensively studied in the literature. We refer to [92] for an overview of the available results, and to [10] for a survey on the applicability of such systems. In polling models, the server works on multiple queues in some order.

One switching strategy is the k-limited serving discipline. In k-limited polling models the switch between queues is triggered by one of the following two events: a predefined number of k customers get served, or the queue becomes empty. This discipline does not satisfy the branching property [72], which complicates the analysis and makes exact results very hard to obtain. Most of the papers [52,94] that derive exact solutions consider models with two queues. Such systems can be related to the one described in the present chapter by modeling the customers from one of the queues as the requests to be written to the cache, whereas the jobs in the other queue to stand for the accumulated requests, waiting to be written to the storage. Nevertheless, there are two major differences between the k-limited service discipline and the system policy considered in the present study. First, in the former models, the server switches queues whenever it becomes idle, and second, the arrival processes of customers in the two queues are usually assumed to be independent.

Service of customers in groups is also considered in networks with "bulk service" [1, 21]. Of particular interest for the current research are systems that have batch-size-dependent service [30] and that incorporate a two-phase policy [13, 22]. For a survey on this type of queues, the reader is referred to [68]. Furthermore, [83] reviews a broad range of papers on the subject of optimal control of queues.

The remainder of this chapter is organized as follows. In Section 3.2, we introduce the model. In Section 3.3, we illustrate how the system performance varies depending on the batch size, and we analyze two specific cases of K, namely, $K = 2$ and $K \to \infty$. Next, in Section 3.4, we present how one can use the obtained results to find the optimal value for the batch size in a scalable and efficient way. We conclude with numerical results and a summary together with ideas for possible further research.

3.2 Model

We model the 'write behind' mechanism by considering a single server queueing network. For an illustration of the model, the reader is referred to Figure 3.1.

Jobs are assumed to arrive according to a Poisson process with rate λ. Initially, each job has to be *pre-processed* by the server. The server will send a response after a job has been pre-processed. Each pre-processed job will be

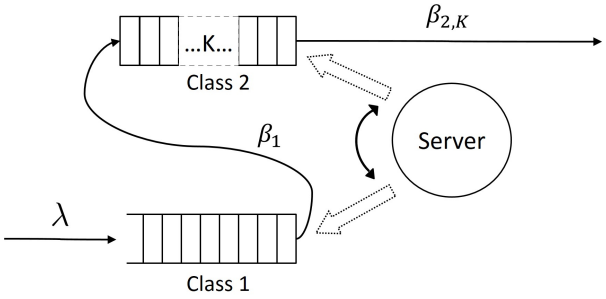


Figure 3.1: The 'write behind' mechanism as a queueing model.

accumulated in a second queue, i.e., the buffer, until a threshold of $K \geq 1$ jobs has been reached. At this threshold the server switches to the *post-processing* mode where all jobs from the second queue are processed in one batch of K jobs. Note that this is actually the case in practice when the system is heavily loaded, which is indeed the conditions under which it is crucial to analyze and optimize the system from a practical point of view. After the batch has completed, the corresponding jobs leave the system. If a job arrives and finds the server busy, it will be queued until the server takes this job from the first, pre-processing queue.

In order to model this serving policy, we distinguish two classes of jobs. The first class represents the jobs that are sent to the second queue, whereas the second class job triggers the server to switch to the batch mode. We strictly impose that every K-th job is class 2, and all others are class 1. In such a way, we assure that the server starts the service of the batch exactly when it contains K jobs. The service time of the first class is taken to be exponentially distributed, with mean β_1 and corresponding service rate $\mu_1 = \frac{1}{\beta_1}$. This corresponds to the required time to pre-process a request. For the sake of simplicity, the service time of the second class is modeled as one exponentially distributed phase, with mean $\beta_{2,K}$ with corresponding service rate $\mu_{2,K} = \frac{1}{\beta_{2,K}}$. This service time includes the time required to pre-process the K-th job and post-process the corresponding batch of size K. We assume $\beta_{2,K}$ to increase proportionally to the batch size and therefore we take $\beta_{2,K} = a + bK$, where a and b are parameters. One can interpret $a \geq \beta_1$ as the time required to pre-process the K-th job and subsequently the required initialization time

for post-processing the K jobs in the buffer, i.e., the batch service. On the other hand, $b > 0$ represents the expected time to post-process one single job. In the sequel, we omit the second subindex $_K$ and replace $\beta_{2,K}$ by β_2 when it is clear from the context.

To analyze the system, we consider a continuous-time Markov chain with two-dimensional states (i, j) and state space S such that

$$S := \mathcal{N}_0 \times \{0, 1, \ldots, K-1\},$$

where i corresponds to the number of jobs waiting for completion of the pre-process step, while j represents the number of jobs in the post-process buffer. One can easily verify that this Markov chain has a generator matrix Q with the following non-zero, non-diagonal entries:

$$\begin{aligned} q_{(i,l),(i+1,l)} &= \lambda && \text{for } i \geq 0 \text{ and } 0 \leq l \leq K, \\ q_{(i,l),(i-1,l+1)} &= \mu_1 && \text{for } i > 0 \text{ and } 0 \leq l < K, \\ q_{(i,K),(i-1,0)} &= \mu_2 && \text{for } i > 0, \end{aligned} \tag{3.1}$$

where q_{s_1,s_2} is the transition rate from state s_1 to state s_2. Furthermore, the load of the system for a given K is :

$$\rho_K = \lambda \left(\frac{(K-1)\,\beta_1}{K} + \frac{\beta_2, K}{K} \right). \tag{3.2}$$

For stability, we assume that $\rho_K < 1$. Also, note that standard balancing arguments show that the system is idle for a fraction $1 - \rho_K$ of the time. In other words, if $\pi = \{\pi_{i,j}, (i,j) \in S\}$ stands for the stationary distribution, it follows that

$$\pi_{0,0} + \pi_{0,1} + \cdots + \pi_{0,K-1} = 1 - \rho_K. \tag{3.3}$$

3.3 Analysis

In this section, we analyze the impact of the batch size on the system behavior. Recall that the rationale behind 'write behind' mechanism is to reduce the delay experienced by write requests on the application level, i.e., at the server's queue. In our model, we incorporate the required time to empty the internal buffer in the service time of the K-th job. Please observe that this modeling choice does not influence the waiting time at the buffer for any of the writing

requests. I.e., jobs that find the server in post-processing mode have to wait anyway until the post-processing has completed. Therefore, in this chapter, we consider the mean waiting time in the pre-processing queue as our main performance metric. The *waiting time* W is defined as the time between a job arrival until that job has started pre-processing, i.e., the waiting time in the first queue. This leads to the following two main goals of the current research: (1) to determine the expected waiting time until a job will be post-processed for a given batch size K, and (2) to find the optimal value of K for which this metric is minimized. The mean waiting time $E[W]$ can be expressed in terms of stationary distribution π as follows:

$$E[W] := \frac{1}{\lambda} \sum_{i=1}^{\infty} \sum_{j=0}^{K-1} (i-1)\pi_{i,j}, \qquad \text{for } K > 0. \tag{3.4}$$

In the remainder of this section we analyze the mean waiting time for given values of K. First, we note that the case of $K = 1$ corresponds to an $M/M/1$ queue, and therefore, the system performance can be obtained easily. Furthermore, in the Appendix we describe how the matrix-geometric method (MGM) [60] can be used to calculate the mean waiting time for any given K. Although in theory, this method applies to any value of K, due to its computational complexity, the MGM approach becomes inefficient or even unfeasible for large buffer sizes. Therefore, in Sections 3.3.1 and 3.3.2, we present techniques that result in analytic expressions. In contrast to MGM, these two approaches are related to specific cases of K. Namely, we derive an exact solution for $K = 2$ and a fluid approximation for $K \to \infty$.

3.3.1 Analytic solution for $K = 2$

In this subsection, we analyze the system in the special case $K = 2$. This means that every second job that arrives is of class 2; in other words, the two different classes of jobs arrive in strictly alternating fashion. Since $\beta_1 \leq \beta_2$ and the PASTA property, one can conclude that an arriving job will find the server working on a class 1 job with less or equal probability than working on a class 2 job. Therefore, conditioned on a given number of jobs in the system, one knows that there is a higher or equal probability that the server is working on (or waiting) a client of class 2 rather than class 1. In particular, for any given $i \geq 0$ it holds that $\pi_{i,0} \leq \pi_{i,1}$. Note that in the special case of $\beta_1 = \beta_2$, the two classes become identical and indeed trivially $\pi_{i,0} = \pi_{i,1}$ for any given $i \geq 0$.

We denote by $c_i = \lambda/\mu_i$ for $i = 1, 2$ the *potential load* of the system in case of only class i arrivals. By the transition rates in (3.1) for states $(0, 0)$ and $(0, 1)$, it follows:

$$\begin{aligned} \pi_{1,1} &= (\lambda\pi_{0,0})/\mu_2 = c_2\pi_{0,0}, \\ \pi_{1,0} &= (\lambda\pi_{0,1})/\mu_1 = c_1(1-\rho-\pi_{0,0}), \end{aligned} \tag{3.5}$$

where we also used Equation (3.3) to express $\pi_{0,1}$ in terms of $\pi_{0,0}$. Moreover, from Equation (3.2) it follows that $\rho = \frac{1}{2}(c_1 + c_2)$. Now, using Equation (3.5) in the balance equations for states $\pi_{1,0}$ and $\pi_{1,1}$, gives us:

$$\begin{aligned} \pi_{2,1} &= \frac{-\lambda\pi_{0,0} + (\lambda+\mu_1)\pi_{1,0}}{\mu_2} \\ &= \frac{1}{2}c_2\left((-c_1-1)(c_1+c_2-2) - 2(c_1+2)\pi_{0,0}\right), \\ \pi_{2,0} &= \frac{-\lambda\pi_{0,1} + (\lambda+\mu_2)\pi_{1,1}}{\mu_1} \\ &= \frac{1}{2}c_1\left(c_1+c_2+2(c_2+2)\pi_{0,0} - 2\right). \end{aligned} \tag{3.6}$$

As shown in the Appendix, there exists a unique 2-by-2 matrix R that satisfies the following equation:

$$\begin{pmatrix} \pi_{0,0} & \pi_{0,1} \\ \pi_{1,0} & \pi_{1,1} \end{pmatrix} R - \begin{pmatrix} \pi_{1,0} & \pi_{1,1} \\ \pi_{2,0} & \pi_{2,1} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}.$$

Now, using equations (3.5) and (3.6) together and denoting by r_{ij} the element in the i-th row and j-th column in the matrix R, we can derive the following expressions:

$$\begin{aligned} r_{11} &= \tfrac{c_1(1-\rho-\pi_{0,0})(1-\rho-2\pi_{0,0})}{\pi_{0,0}^2 c_2 - c_1(1-\rho-\pi_{0,0})^2}, \\ r_{12} &= \tfrac{c_2\left((1-\rho)(\pi_{0,0}(2c_1+3)-(c_1+1)(1-\rho)) - \pi_{0,0}^2(c_1-c_2+2)\right)}{\pi_{0,0}^2 c_2 - c_1(1-\rho-\pi_{0,0})^2}, \\ r_{21} &= \tfrac{c_1\left((1-\rho)(\pi_{0,0}(2c_1-1)-c_1(1-\rho)) - \pi_{0,0}^2(c_1-c_2-2)\right)}{\pi_{0,0}^2 c_2 - c_1(1-\rho-\pi_{0,0})^2}, \\ r_{22} &= \tfrac{\pi_{0,0}c_2(1-\rho-2\pi_{0,0})}{\pi_{0,0}^2 c_2 - c_1(1-\rho-\pi_{0,0})^2}. \end{aligned} \tag{3.7}$$

Next to that, normalizing the stationary distribution gives us one more con-

straint with regard to the matrix R:

$$\sum_{i=0}^{\infty} \begin{pmatrix} \pi_{i,0} \\ \pi_{i,1} \end{pmatrix}^T R^i \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} \pi_{0,0} \\ 1-\rho-\pi_{0,0} \end{pmatrix}^T [I-R]^{-1} \begin{pmatrix} 1 \\ 1 \end{pmatrix} = 1. \tag{3.8}$$

Finally, using the derived substitutions from Equations (3.3) and (3.7) into (3.8) leads us to a real-valued third-degree polynomial with a root $\pi_{0,0}$:

$$f(x) = a + bx + cx^2 + dx^3, \tag{3.9}$$

where the coefficients are:

$$\begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix} = \begin{pmatrix} c_1^2 \left(c_1 + c_2 - 2\right)^2 \\ 2\left(c_1 + c_2 - 2\right)\left(c_2 + c_1\left(3c_1 + c_2 - 1\right)\right) \\ 12\left(c_1 - 1\right)c_1 + 4\left(c_1 + 3\right)c_2 \\ 8\left(c_1 - c_2\right) \end{pmatrix}.$$

Note that in the special case of $c_1 = c_2$ the leading coefficient of the polynomial defined in Equation (3.9) becomes 0, hence it simplifies to a quadratic equation. The solutions to this quadratic equation are:

$$\begin{aligned} x_{1,2} &= \frac{-16c_1^2(c_1-1) \pm \sqrt{\left(16c_1^2(c_1-1)\right)^2 - 4(4c_1^2(c_1-1)^2)(16c_1^2)}}{32c_1^2} \\ &= \frac{1-c_1}{2}. \end{aligned}$$

On the other hand, $c_1 = c_2$ corresponds to $\mu_1 = \mu_2$, i.e., the two classes are identical. Using the well-known result for an $M/M/1$ queueing system, one can derive

$$\pi_{0,0} = \pi_{0,1} = \frac{1-\rho}{2} = \frac{1-c_1}{2},$$

which verifies that the two results are the same. Therefore, in the following we take $c_2 > c_1 > 0$. Furthermore, we consider the system to be stable, hence $c_1 + c_2 < 2$.

Lemma 1 *The third-degree polynomial defined in Equation* (3.9) *has a negative value for a given lower bound of* $\pi_{0,0}$.

Proof: First, we show that

$$\pi_{0,0} \geq \frac{c_1(1-(c_1+c_2)/2)}{c_1+c_2}.$$

Indeed, as discussed in this section where $\pi_{1,0} \leq \pi_{1,1}$, and therefore, using Equation (3.5) we derive that

$$c_1(1-\rho-\pi_{0,0}) \leq c_2\pi_{0,0},$$

which after trivial manipulation leads us to the desired lower bound for $\pi_{0,0}$. Now, evaluating the polynomial defined in Equation (3.9) for this lower bound and simplifying the resulted expression gives us:

$$f\left(\frac{c_1(1-(c_1+c_2)/2)}{c_1+c_2}\right) = -\frac{c_1(c_1-c_2)^2c_2(-2+c_1+c_2)^2}{(c_1+c_2)^3}.$$

Therefore, we conclude that under the conditions that $c_2 > c_1 > 0$ and $c_1+c_2 \neq 2$, the third-degree polynomial has a strictly lower value than 0 when evaluated for a given lower bound of $\pi_{0,0}$.

Lemma 2 *The third-degree polynomial defined in Equation* (3.9) *has a strictly positive value for a given upper bound of* $\pi_{0,0}$.

Proof: To find an upper bound for $\pi_{0,0}$ we use the following two observations from the beginning of the section:

$$\pi_{0,0} \leq \pi_{0,1} \qquad \text{and} \qquad \pi_{0,0}+\pi_{0,1} = 1-\rho.$$

Substituting and working out the expression gives:

$$\pi_{0,0} \leq \frac{1-\rho}{2} = \frac{1-(c_1+c_2)/2}{2}.$$

Evaluating the polynomial defined in Equation (3.9) for this upper bound and simplifying the resulted expression leads to the following value:

$$f\left(\frac{1-(c_1+c_2)/2}{2}\right) = \frac{1}{8}(c_1-c_2)^2(-2+c_1+c_2)^2.$$

From this, we conclude that under the conditions that $c_1+c_2 \neq 2$ and $c_1 \neq c_2$, the third-degree polynomial has a strictly larger value than 0 when evaluated for a given upper bound of $\pi_{0,0}$.

Theorem 3 *The third-degree polynomial defined in Equation* (3.9) *has three real roots and uniquely identifies* $\pi_{0,0}$.

Proof Due to the condition that $c_2 > c_1 > 0$, the leading coefficient of the third-degree polynomial defined in Equation (3.9) is $8(c_1 - c_2) < 0$. Together with the results from Lemmas 1 and 2 we derive the following inequalities:

$$\begin{aligned} &f(-\infty) > 0; \\ &f\left(\frac{c_1(1-(c_1+c_2)/2)}{c_1+c_2}\right) < 0; \\ &f\left(\frac{1-(c_1+c_2)/2}{2}\right) > 0; \\ &f(\infty) < 0. \end{aligned} \tag{3.10}$$

Moreover, under the considered conditions with regards to c_1 and c_2 we have:

$$\frac{c_1}{c_1+c_2}\left(1-(c_1+c_2)/2\right) < \frac{1}{2}\left(1-(c_1+c_2)/2\right),$$

and therefore the upper bound found in Lemma 2 is indeed strictly larger than the lower bound from Lemma 1. Together with the derived inequalities in (3.10) we conclude that the polynomial is changing signs three times which proves that there are three real roots. In addition, there is exactly one root smaller than the lower bound for $\pi_{0,0}$; one larger than the upper bound and one in between.

In conclusion, from Theorem 3 it directly follows that $\pi_{0,0}$ is uniquely identified by solving the third-degree polynomial (3.9). Consequently, one can derive the steady-state distribution π for the analyzed system by $\pi_i = \pi_1 R^{i-1}$ for $i > 0$, where and R is obtained from (3.7).

3.3.2 Fluid approximation for large K

In this section we study the asymptotic growth in waiting time when $K \to \infty$ by means of a fluid-limit approximation. We outline the technique and present the results of the approximation.

The idea behind the approach is to let $K \to \infty$ while speeding up time by the same factor K. Observe that when there are no class 2 jobs active, the

service time seen by individual class 1 jobs vanishes:

$$\beta_1/K \stackrel{K\to\infty}{=} 0.$$

In this case, no class 1 jobs have been accumulated due to class 2 job processing. When the system contains class 2 jobs, fresh arriving jobs will be preceded by the service time of class 1 jobs that wait together with class 2 jobs. I.e., each class 2 job is preceded by $(K-1)$ class 1 jobs. The additional service time brought by the other $K-1$ class 1 jobs is a convolution of $K-1$ exponential service times. Due to the law of large numbers this additional service time will converge to a deterministic value β_1:

$$(K-1)\beta_1/K \stackrel{K\to\infty}{=} \beta_1.$$

Also the inter-arrival time between class 2 jobs becomes deterministic with mean $\frac{1}{\lambda}$. However, the service time of each K-th job remains exponentially distributed with mean:

$$(a+bK)/K \stackrel{K\to\infty}{=} b.$$

In the limit, the system becomes a single server queueing system with deterministic inter-arrival times and service times that are the convolution of two distributions: an exponential distribution with mean b and a deterministic one with mean β_1.

Because we are interested in the mean waiting time, one can analyze an equivalent queueing system where the service time is exponentially distributed with mean b and the inter-arrival time is $1/\lambda-\beta_1$. This implies that the scaled process can be analyzed by a $D/M/1$ queueing system with corresponding load:

$$\tilde{\rho} = \frac{b}{\frac{1}{\lambda}-\beta_1}. \tag{3.11}$$

Now, using Theorem X.5.1b from [2], we obtain the average workload for this queue as $\tilde{\rho}\frac{1}{\eta}$, where η is the unique solution >0 of:

$$1 = \frac{1/b}{1/b-\eta}e^{-\eta(1/\lambda-\beta_1)}.$$

From this expression we derive the following fixed-point equation for obtaining η:

$$\eta = \frac{1}{b}(1-e^{-\eta(1/\lambda-\beta_1)}). \tag{3.12}$$

Using the expected workload of the fluid scaling we can characterize the asymptotic growth of the mean waiting time, $E[W(K)]$, as a function of batch size K:

$$E[W(K)] = f(K) \to \xi_1 + K\xi_2, \tag{3.13}$$

where the constant ξ_2 is given by

$$\xi_2 = \frac{\tilde{\rho}}{\eta}. \tag{3.14}$$

3.4 Optimization of the batch size K

In this section, we describe an efficient and scalable method to find the optimal batch size for a given set of parameters $(N, \lambda, \beta_1, a, b)$. As we are interested in the mean waiting time in the queue, $E[W(K)]$, we want to capture how K relates to the value of this specific performance metric. One way to find the optimal batch size is to use the matrix-geometric method described in the Appendix and calculate the corresponding $E[W(K)]$ for all $1 \leq K \leq N$. However, this approach is not efficient and might even be infeasible for large N. In many real-world applications, the maximum buffer size, N, prohibits such a straightforward approach. In such cases, one needs a scalable algorithm for estimating the optimal K.

Another approach is to simplify the model and assume that on *average* (instead of *strictly*) every K-th job has a service time β_2. This results in an $M/G/1$ queue where the service time is a random variable S, whose distribution is a mixture of two random variables. Namely, with probability $(K-1)/K$ it is exponentially distributed with mean β_1, and otherwise – with mean β_2. This implies:

$$E[S] = \frac{(K-1)\beta_1 + \beta_2}{K},$$
$$E[S^2] = \frac{2\left((K-1)\,\beta_1^2 + \beta_2^2\right)}{K}.$$

Consequently, the mean waiting time in the queue, $E[W]_{M/G/1}$, for this $M/G/1$ queue can be derived by the Pollaczek-Khinchin formula:

$$E[W]_{M/G/1} = \frac{\lambda E[S^2]}{2(1-\rho)} = \frac{\lambda\left(a^2 - \beta_1^2\right) + \lambda\left(2ab + \beta_1^2\right)K + \lambda b^2 K^2}{\lambda\left(\beta_1 - a\right) + \left(1 - \lambda\left(\beta_1 + b\right)\right)K}. \tag{3.15}$$

Although this model simplification results in an analytic solution, the obtained estimation for the mean waiting time is rather inaccurate (see also Section 3.5 below). Therefore, in the following, we develop an approximation approach which is both scalable and accurate.

In Section 3.3.2, we outlined a method to derive the mean waiting time for large K. As discussed, the fluid approximation implies a linear relationship between the batch size K and the mean waiting time $E[W(K)]$ for large values of K. On the other hand, in Section 3.3 we showed that there is rarely a linear dependence between K and $E[W(K)]$ for the whole range $1 \leq K \leq N$ and in most of the cases there is a non-trivial optimal batch size K. These two findings together with the form of the above-derived Equation (3.15) for the $M/G/1$ simplification, led us to choose the following function as an approximation of $E[W(K)]$:

$$f(K) := \frac{l + mK + nK^2}{s + K}, \tag{3.16}$$

where l, m, n, and s are parameters.

In the following, we describe how one can estimate l, m, n, and s. First, we note that n corresponds to the asymptotic slope for $K \to \infty$. Using, the results from the fluid approximation, we derive $n = \xi_2$, where ξ_2 is given by Equation (3.14). Now, to estimate the remaining three parameters, l, m, and s, we obtain the value of f for three distinct values of K. As discussed, the case of $K = 1$ stands for an $M/M/1$ queue with an arrival rate λ and service rate $1/(a+b)$, and therefore $f(1) = E[W(1)] = \lambda(a+b)^2/(1-\lambda(a+b))$. Next to that, as depicted in Theorem 3, the stationary distribution for $K = 2$ can be analytically derived, from which consequently one can easily compute $f(2) = E[W(2)]$. Finally, we numerically compute $E[W(K)]$ for a third value of K using the MGM from the Appendix. As we have already obtained $E[W(K)]$ at the beginning and at the end of the range of possible batch sizes, we use the MGM to calculate $E[W(K)]$ for $K = \lfloor N/2 \rfloor$.

Now, using $f(K)$ in Equation (3.16), and the estimated parameter values, one can approximate the optimal batch size by finding $1 \leq K \leq N$ that minimizes $f(K)$. Namely, if we denote with $x := \sqrt{s^2 - (ms - l)/n} - s$ the larger root of $f'(K)$:

$$K_{est} = \begin{cases} 1, & \text{for } x \leq 1, \\ \lfloor x \rfloor \text{ or } \lceil x \rceil, & \text{for } 1 < x \leq N, \\ N, & \text{for } N \leq x, \end{cases}$$

where for the case $1 < x \leq N$, one can simply compute $f(\lfloor x \rfloor)$ and $f(\lceil x \rceil)$ and take the one for which the function has the lower value.

Our technique approximates the optimal value of the batch size using the MGM only once. Moreover, although ideally one would apply the MGM to a system with $K = \lfloor N/2 \rfloor$, we note that our approximation algorithm works with any value of $K > 2$. Therefore, one might use a constant value of K, which allows for a computationally feasible numerical solution, in case this is not possible for $K = \lfloor N/2 \rfloor$. In such a way, the computational complexity of our algorithm becomes a constant, $\mathcal{O}(1)$. We emphasize that this is a significant improvement in comparison to the alternative approach of running MGM for all possible values of the batch size, which would result in a complexity of $\mathcal{O}\left(1^3 + 2^3 + \cdots + N^3\right) = \mathcal{O}\left(N^4\right)$ [42]. Furthermore, the fact that for many practical applications our technique would be the only feasible approach is of even greater importance.

3.5 Results

In the following, we evaluate the proposed method for estimating the optimal batch size. To create a benchmark for a system with a specific set of parameters we apply the matrix-geometric approach for each possible K. Therefore, to obtain this benchmark for a broad range of combinations for the parameters (β_1, a, b), our choice for N is limited to relatively small values. More precisely, we examine the accuracy for $N = 20$, 50 and 100, β_1 varying from 0.01 to 0.96 with a step of 0.05, and a and b from 0.01 to 1 with a step size of 0.01. In all test cases, we fix $\lambda = 1$ as scaling λ, β_1, a and b with the same factor is equivalent to scaling time which does not influence the steady-state behavior of the system. Furthermore, we decided to exclude trivial cases in which the optimal batch size is 1, as those are straightforward to identify by simply checking whether $E[W(1)] \leq E[W(2)]$, and thus not representative for the algorithm's accuracy in general. In addition, we also omitted cases in which the system was unstable. This procedure resulted in nearly 50,000 test cases.

As discussed, we consider the mean waiting time in the queue as the key performance metric of the system. Moreover, we are interested in the value of the batch size that minimizes this metric. Therefore, we evaluate the accuracy of the algorithm by comparing the mean waiting times in the system with the estimated K_{est} and the one with the real optimal batch size. More precisely,

as an error metric we define:

$$E_r = \frac{|E[W]_{est} - E[W]_{opt}|}{E[W]_{opt}} \times 100\%,$$

where $E[W]_{est}$ and $E[W]_{opt}$ are the mean waiting times given the estimated batch size and the optimal one, respectively.

In Figure 3.2, we examine the test case for which the accuracy of our algorithm was the lowest. Note that even in this worst case, the fitting curve is a good approximation of the mean waiting time. However, the optimal value of K is not the estimated one. The relative difference in the performance metric in case the approximated optimal value of K is applied instead of the real one is $|E[W(3)] - E[W(2)]| / E[W(2)] = 0.69\%$. On the other hand, this error becomes more than 9% if the batch size is chosen to be 1 instead of the optimal value of 2. Therefore, we believe that the relative difference in the mean waiting time is more relevant than the one in the corresponding values of K.

The aggregated results of the tests are shown in Table 3.1. We calculate in what percentage of the cases our algorithm derived the optimal value of K and output the result in the column "Exact" of Table 3.1. Next to that, we present the highest error from the tests where the estimated K was not the optimal one. As discussed, we measure the error as the relative difference between the mean waiting times in a system with the optimal K and with the estimated one. As a comparison, we include the results of the $M/G/1$ model simplification discussed in Section 3.4.

Table 3.1: Accuracy of the approximation of the optimal batch size.

Method	Exact	Max E_r
Our approximation	99.8%	0.69%
$M/G/1$	77.0%	10.11%

After analyzing the 0.2% of test cases in which our algorithm made an error, we found out that for all of them the optimal batch size is 2. We believe that system parameters that lead to such a solution with optimal $K = 2$ are rarely observed in practice. In contrast, the parameters that result in wrong approximation by the $M/G/1$ technique do not follow any particular pattern, and moreover, are significantly more in comparison to our algorithm.

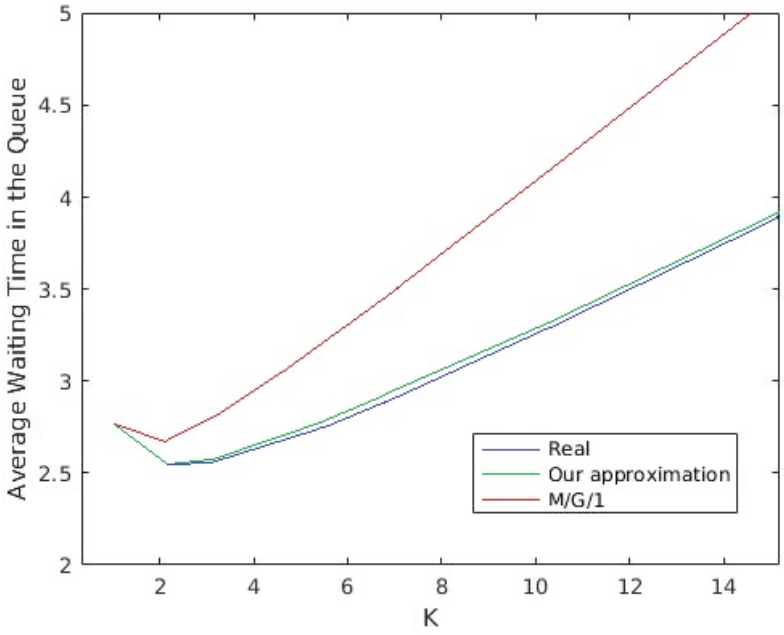


Figure 3.2: Mean waiting time as a function of the batch size K for the system with parameters $N = 50, \lambda = 1, \beta_1 = 0.51, a = 0.55$ and $b = 0.23$.

In addition, the relative errors of the two approximation approaches differ by one order of magnitude. Tables 3.2 and 3.3 provide the result details for ten parameter configurations corresponding to some of the worst cases for each of the two approximation algorithms (i.e., our technique and the $M/G/1$ approach, respectively).

Based on the results from these test cases we conclude that our algorithm identifies the optimal batch size with remarkably high precision. Next to that, the systems for which the approximated K is not the optimal one are rather unrealistic from a practical point of view. Moreover, even in these cases the relative error in the performance metric of interest is negligibly small.

Table 3.2: Result details for system configurations corresponding to some of the worst cases for our approximation technique.

System parameters				K values			Errors	
N	β_1	a	b	K_{opt}	K_{est}	$K_{M/G/1}$	$E_{r,est}$	$E_{r,M/G/1}$
50	0.51	0.55	0.23	2	3	2	0.69	0.00
20	0.41	0.46	0.18	2	3	2	0.66	0.00
50	0.41	0.48	0.25	2	3	2	0.64	0.00
50	0.46	0.51	0.22	2	3	2	0.64	0.00
50	0.36	0.46	0.34	2	3	2	0.59	0.00
20	0.61	0.63	0.20	2	3	2	0.59	0.00
50	0.36	0.41	0.15	2	3	2	0.58	0.00
20	0.56	0.59	0.22	2	3	2	0.56	0.00
50	0.31	0.41	0.23	2	3	2	0.54	0.00
100	0.51	0.54	0.17	2	3	2	0.52	0.00

Table 3.3: Result details for system configurations corresponding to some of the worst cases for the $M/G/1$ approach.

System parameters				K values			Errors	
N	β_1	a	b	K_{opt}	K_{est}	$K_{M/G/1}$	$E_{r,est}$	$E_{r,M/G/1}$
20	0.31	0.39	0.30	2	2	1	0.00	10.11
100	0.46	0.54	0.41	3	3	2	0.00	6.83
50	0.41	0.61	0.37	4	4	3	0.00	4.75
20	0.56	0.66	0.30	5	5	3	0.00	3.77
50	0.61	0.73	0.23	6	6	4	0.00	2.57
50	0.71	0.79	0.19	7	7	5	0.00	2.02
50	0.86	0.89	0.09	12	12	9	0.00	0.58
20	0.81	0.89	0.08	15	15	11	0.00	0.55
100	0.76	0.89	0.09	14	14	11	0.00	0.54
50	0.86	0.89	0.08	13	13	9	0.00	0.51

3.6 Conclusion

Inspired by the 'write behind' caching mechanism that is used in data center storage solutions we formulated and analyzed a queueing model. Each job in this queueing model represents a write operation request from application to storage. Each job is processed by a single server in two stages. In the first stage service jobs are pre-processed. After the pre-processing stage a response will be sent. However, the job will remain in the system until the post-processing stage has completed. The server alternates between pre- and post-processing stages. Jobs that require post-processing will be accumulated in a cache until a threshold of K jobs has been reached. When this threshold has been reached, all K-jobs in cache are served at once in a batch. The goal is to minimize expected waiting time of jobs in the pre-processing stage by choosing the proper batch size K. This model is representative for various real-world systems that exist in computer science, manufacturing, logistics, and many others. We showed how one can analyze and derive the stationary distribution of jobs in such systems with a fixed batch size. In addition, we considered a special case where an analytic exact solution is possible by finding the roots of a third-degree polynomial. Next to that, we outlined a fluid approach which results in approximation of the expected waiting time for large batch sizes. Finally, we presented how one can combine the insights from these techniques in order to find the optimal static control policy in an efficient and scalable way. Extensive numerical tests showed that this method works extremely well.

By using explicit results for $K = 1$ and $K = 2$ in combination with the asymptotic factor from the fluid analysis of Subsection 3.3.2 we only require the application of the MGM for a moderate size of K. In this way we are able to significantly reduce required memory and computation time.

Finally, we address a few topics for further research. A practical relevant challenge would be to analyze a system with time-varying arrival and service rates. We believe that the low computational requirements of our algorithm allow an extension which can tackle such models. Nevertheless, in order to verify that, one should thoroughly test the possible generalization of our technique. Another open problem is the dynamic control of the 'write behind' cache. Motivated by this, in the next chapter we analyze the corresponding system within the Markov Decision Processes framework.

3.7 Appendix

In the following, we show how the matrix-geometric method [60] can be applied to find the stationary distribution of the system for a given K. Using the transition rates in (3.1), one derives the following block structure of the generator matrix Q:

$$Q = \begin{pmatrix} B_0 & A_0 & 0 & 0 & 0 & \cdots \\ A_2 & A_1 & A_0 & 0 & 0 & \cdots \\ 0 & A_2 & A_1 & A_0 & 0 & \cdots \\ 0 & 0 & A_2 & A_1 & A_0 & \cdots \\ \vdots & \vdots & \vdots & \ddots & \ddots & \ddots \end{pmatrix},$$

where

$$-B_0 = A_0 = \begin{pmatrix} \lambda & 0 & \cdots & 0 \\ 0 & \lambda & \cdots & 0 \\ \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda \end{pmatrix}, \quad A_2 = \begin{pmatrix} 0 & \mu_1 & 0 & \cdots & 0 \\ 0 & 0 & \mu_1 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 & \mu_1 \\ \mu_2 & 0 & \cdots & 0 & 0 \end{pmatrix},$$

and $A_1 = diag(-(A_0+A_2)e)$ are K-by-K square matrices. By definition this is a Quasi Birth Death (QBD) process and the following normalization equation holds:

$$1 = \sum_{i=0}^{\infty} \pi_i e = \pi_0 e + \pi_1 (I + R + R^2 + \cdots)e = \pi_o e + \pi_1 (I - R)^{-1} e, \quad (3.17)$$

where e is the unit vector, $\pi_i = (\pi_{i,0}, \pi_{i,1}, \ldots, \pi_{i,K})$ are the equilibrium probability vectors and R is such that $\pi_i = \pi_1 R^{i-1}$. It follows that R is the minimal non-negative solution of the matrix-quadratic equation

$$A_0 + RA_1 + R^2 A_2 = 0. \quad (3.18)$$

One simple scheme to compute R is to iteratively solve (3.18) by the following successive substitutions:

$$R_{k+1} = -(A_0 + R_k^2 A_2) A_1^{-1}, \qquad k = 0, 1, 2, \ldots, \quad (3.19)$$

starting with $R_0 = 0$. More complicated and efficient techniques to find R have been developed in the literature (see, for example, [51]). Next, to find

the stationary distribution one can compute π_0 and π_1 using the boundary condition:

$$\pi_0 B_0 + \pi_1 A_2 = 0, \tag{3.20}$$

together with (3.17), and subsequently π_i for any $i > 1$ by the following equation:

$$\pi_i = \pi_1 R^{i-1}. \tag{3.21}$$

Once the stationary distribution of the system is obtained, one can derive the mean waiting time in the queue by Equation (3.4).

Chapter 4

Closed-Form Control Policies of Markov Decision Processes Using Symbolic Regression

In this chapter, we introduce a new approach to optimize the control of systems that can be modeled as Markov Decision Processes (MDPs) with a threshold-based optimal policy. Our method is based on a specific type of evolutionary algorithm known as Symbolic Regression (SR). We present how both the execution time and the accuracy of this algorithm can be greatly improved by taking into account the corresponding MDP framework in which we apply it.

The proposed method has two main advantages: (1) it results in near-optimal decision policies, and (2) in contrast to other algorithms, it generates closed-form approximations. Obtaining an explicit expression for the decision policy gives the opportunity to conduct sensitivity analysis, and allows instant calculation of a new threshold function for any change in the parameters. We emphasize that the introduced technique is highly generic and applicable to MDPs that have a threshold-based policy. Extensive numerical experimentation demonstrates the usefulness of the method.

The work in this chapter is based on A. Hristov, S. Bhulai, R.D. van der Mei and J.W. Bosman. Deriving explicit control policies for Markov Decision Processes using Symbolic Regression (2018) [36]. *Submitted.*

4.1 Introduction

In practice, many stochastic control problems exhibit an optimal policy that is of a threshold type. In most cases, this structure can be proven by mathematical induction within the MDP framework [46]. In real applications, however, one does not only need to know the structure of the optimal policy, but also

the threshold value for implementation purposes. Unfortunately, deriving this value explicitly remains a hard problem and usually has to be solved by numerical computation.

There are several advantages of having the threshold value in an explicit form. First, it allows one to easily implement a threshold for different system parameters without having to solve the corresponding MDP. Second, the robustness of the threshold function can be assessed through sensitivity analysis, which is important when system parameters are estimated from real data.

In this chapter, we propose a new approach to obtain an analytic expression for the decision policy of a given MDP. The main idea of our method is to combine the field of Markov decision theory with a specific evolutionary algorithm: the SR algorithm. In the current study, we focus on threshold-based decision policies. We examine this sub-class of policies as they can be expressed in a natural and easy to interpret closed-form. Namely, one can examine the threshold value for one of the dimensions of the system's state space as a function of the other dimensions and the system parameters. We outline guidelines and useful practices when tailoring the algorithm.

An introduction to MDPs and the classical numerical techniques to solve such problems (e.g., value iteration and function iteration) is described in [62, 69,84]. Next to the numerical approach, one might tackle the challenge of optimal control also by using algebraic techniques. However, due to the complexity of most of the MDP problems, obtaining an algebraic solution is usually not feasible. Therefore, the vast body of literature deals with proving structural properties of various Markov decision problems rather than finding the explicit structure of the decision policy (see [7, 46, 77]).

On the other hand, mostly due to practical reasons, there is a need for an efficient procedure that yields an implementable decision policy. There are several papers that show how one can make use of machine learning techniques to obtain such a solution. Most of the research is focusing on one of the following two types of algorithms: reinforcement learning [81, 82], or genetic programs [4, 15, 57, 96].

We believe that the method proposed in this chapter can serve as a link between the above described two major approaches. Our technique exploits certain structural properties of the given MDP to produce a closed-form solution. To obtain the function characterizing the decision policy, we use SR

(see [43, 48, 80] for an introduction to SR). A related research that applies SR within an MDP framework is the one conducted in [63]. However, in contrast to [63], our research aims at finding the control policy rather than the value function. Therefore, with our approach one can incorporate in the SR algorithm possible domain knowledge and insights for the optimal policy.

In the following section, we outline our implementation of the SR method for approximating the decision policy of a given MDP. To present our guidelines more comprehensibly, we apply our technique to two running examples. In Section 4.3, we analyze the 'write-behind' cache model and present how the approximated decision policy performs compared to the optimal one. Next, in Section 4.4 we discuss how the generated expressions from our method resemble the real closed-form solution for an $M/M/1$ queueing model with customer rejection. We conclude with a summary in Section 4.5.

4.2 Method

In this section, we introduce our method of finding a closed-form solution for the control policy of a given MDP by using SR. Below, we briefly outline the main concepts of this regression algorithm. The reader is referred to [43,48,80] for more details on SR.

SR is a type of regression analysis that searches the space of algebraic expressions to find the one that best fits a given dataset, both in terms of accuracy and simplicity. Within the SR framework, an *individual* represents a specific formula, which is expressed as a *tree* (for an example we refer to Figure 4.1). Note that each leaf contains a parameter, whereas each node gives the mathematical operator.

Like any other evolutionary algorithm, SR forms an initial population of individuals. Next, it iteratively generates a new *offspring* of individuals (e.g., a new generation) by *crossing* and/or *mutating* already existing individuals. Figure 4.2 illustrates a simple crossing scheme and Figure 4.3 shows a possible mutation. The underlying idea is that over time the population's accuracy increases due to evolving the good performing individuals (i.e., survival of the fittest).

Figure 4.4 presents the four main steps of our technique, outlined as follows:

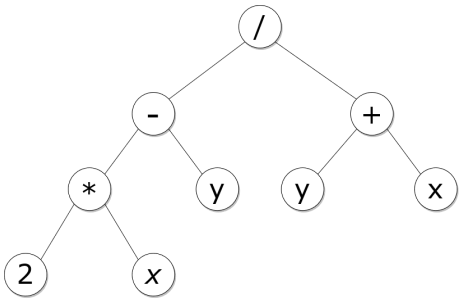


Figure 4.1: An example of individual: the formula $\frac{2x-y}{y+x}$ expressed as a tree.

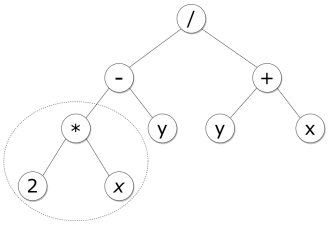
Step 1: Model
The analyzed system is modeled as an MDP by defining the system states and the corresponding transition probabilities, associated costs and action space.

Step 2: Data generation
The regression program requires a dataset on which the individuals will be tested. Therefore, one needs to numerically solve a number of system instances, e.g., to obtain the optimal decision policy with regards to specific parameter values.

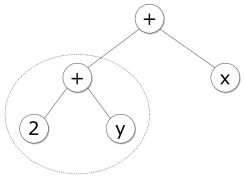
Step 3: Algorithm settings
The algorithm settings are configured to control the way the SR program evolves the generations. One can specify the duration of the evolution, the initial population, the set of mathematical operators, etc.

Step 4: Results
Different setting configurations lead to different results. Next to following the guidelines that we present, one should iteratively analyze the obtained expressions and further adjust additional settings that might lead to a better result.

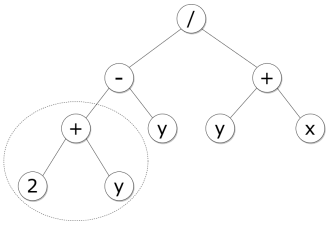
To evaluate the performance of the algorithm, we examine two specific MDP models as running examples. The first one defines a system which analysis is highly challenging and, to the best of our knowledge, there is still no efficient technique for obtaining the optimal policy. Furthermore, there is no analytic solution available even for specific cases of this system. Therefore, we take the corresponding MDP model as our first running example and as a



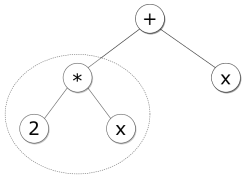
(a) Tree 1 before crossing



(b) Tree 2 before crossing

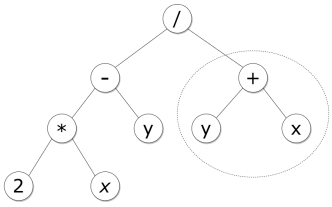


(c) Tree 1 after crossing

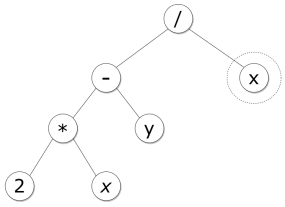


(d) Tree 2 after crossing

Figure 4.2: The crossing operator illustrated on the two trees in Figures 4.2(a) and 4.2(b). The encircled subtrees are exchanged, resulting in the trees in Figures 4.2(c) and 4.2(d).



(a) Before mutation



(b) After mutation

Figure 4.3: Mutation removing the subtree of the encircled node in Figure 4.3(a) (representing the term $y + x$) and replaces it by a randomly generated subtree. The new subtree contains, in this case, only the element x and is encircled in Figure 4.3(b).

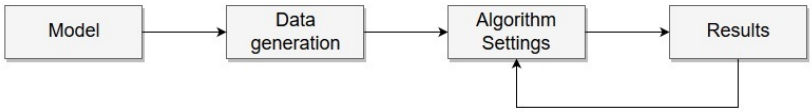


Figure 4.4: The four steps of our technique.

benchmark for evaluating the algorithm's accuracy in terms of achieved system performance. In contrast, the optimal policy for our second MDP example, namely, an $M/M/1$ queue with customer rejection, can be derived in a closed-form. Hence, by comparing the expressions generated by our technique to the optimal formula, we can study how well the algorithm approximates the algebraic form of the decision policy.

4.3 Running example 1: The 'write behind' cache

In this section, we present our first running example. Recall that we are interested in studying the dynamic control of the 'write behind' caching mechanism. Therefore, in the following, we briefly outline the model (see Section 3.2 for more details).

4.3.1 Model

We consider a model with one server which has to complete a two-step process for each job that arrives in the system (see Figure 3.1 for an illustration of the model). Jobs receive an initial service in a FIFO fashion and are subsequently accumulated in a buffer. The second phase consists of serving those jobs that are in the buffer as a batch, i.e., perform a flush of the cache. This way, the server can accumulate jobs in the buffer to some level before serving them all together as a group.

In the following, we denote the maximum size of the server queue and the

buffer as M and N, respectively. Next to that, jobs are assumed to arrive according to a Poisson process with rate λ and join the queue if they find the server busy at this moment. Furthermore, the time to store a job in the buffer is taken to be exponentially distributed, with mean β_1. The service time required for serving K jobs from the buffer is also assumed to be exponentially distributed, with mean $\beta_{2,K}$. We model $\beta_{2,K}$ to increase proportionally to the batch size and therefore we take $\beta_{2,K} = a + bK$, where a and b are known parameters. In addition, we note that the system load, ρ, is dependent on the batch size and hence the decision policy. Therefore, in the following as an approximation of ρ we use the case of a batch size two, which corresponds to the decision policy with highest load (see Section 3.2). Hence, we define $\rho = 0.5\lambda(\beta_1 + a + 2b)$.

Minimizing the mean waiting time in the queue for the above-described system has proven to be challenging. Due to the complexity of the problem [55], there are studies on the simpler case of a static system control when the jobs in the buffer are served whenever they reach a predefined number K, regardless of the number of jobs at the server queue. However, even in this case, there is still no analytic solution available. Therefore, we believe that finding an analytic expression for a dynamic decision policy would greatly facilitate managing such systems. At the same time, it promises a far better performance than the static one, as it takes into account also the number of jobs at the server queue.

4.3.2 Data generation

As discussed, we use SR to derive an expression for the threshold policy function for a given MDP. To produce an estimate, the regression needs a training data set as an input. Once trained on the corresponding samples, the approximation's accuracy can be obtained by comparing the predictions on a given set of test samples with the real values.

There are a few ways to solve an instance of a given MDP problem, e.g., by running the value iteration technique [69, 84], the policy iteration method [69, 84] or by Temporal Difference (TD) learning [81]. Once the optimal threshold policy is obtained, one can transform it into a function $f(P_s, x) = y$, where P_s denotes the specific system parameter vector and y gives the corresponding threshold level for x. Note that for an MDP with an n-dimensional state

space the vector x would be $(n-1)$-dimensional and y an integer. In our example, x is an integer and stands for the number of jobs at the server queue, whereas y is the number of jobs that are already in the buffer. Next to that, we believe that in addition to the initial core parameters for the system (e.g., the local variables), one might also consider including self-composed ones (e.g., structured features). This way, one can facilitate the algorithm in discovering important dependencies between the parameters. For example, in most of the cases, the system load, ρ, greatly influences the behavior of the MDP, and hence, the optimal control. Therefore, in some of the generated SR instances we include ρ as a structured feature.

In the following, we outline the procedure we have performed to generate the data set for our running example. Our objective is to obtain an approximation that can be used for a system with any parameters set, $P_s = (\lambda, \beta_1, a, b)$. Therefore, we design model instances with the idea of generating samples for systems as diverse as possible: $\beta_1 \in \{1.2, 2.4, 3.6, ..., 12\}$, $a/\beta_1 \in \{1, 1.4, 1.8, ..., 5\}$, and $b/\beta_1 \in \{0.01, 0.014, 0.18, ..., 0.5\}$. In all of these cases we take $\lambda = 1$ to reduce the number of parameters without loss of generality. This way, we produce examples of systems with a load ranging from 0.1 to 0.9. Note that for a given set of system parameters, P_s, we have $M+1$ samples in the data. Namely, one for each $0 \leq x \leq M$ (the number of jobs at the queue cannot exceed the queue length), implemented with the corresponding threshold value y indicating the optimal batch size.

Next to that, in the analyzed MDP system, scaling the arrival rate and the service rates does not influence the optimal decision policy. Therefore, we incorporate this insight by generating additional samples for systems with the following parameter set transformation:

$$(\lambda/\tau, \beta_1\tau, a\tau, b\tau),$$

where τ takes the values 100 and $1/100$. Note that we append the already derived x and y to these copies and use the result as additional data samples. In this way, we assist SR in finding an expression that is scale-free with regards to the parameter set, and therefore less probable to be over-fitting the specific range of training values.

For the majority of cases, we took $M = 100$ and $N = 50$, so that we could solve the above-described systems with the value iteration technique within seconds. Nevertheless, we did also generate samples with $M = 1000$ and $N = 500$ and included them in the test set. As a final remark, note that

we have split the obtained dataset in a training and test set by a 70/30 ratio, including MDP instances with various loads in both subsets.

4.3.3 Algorithm settings

Once the MDP problem is modeled, and a dataset is generated by solving instances of this problem, one needs to specify the desired algorithm settings. We note that for an SR implementation we use the `gplearn` [80] Python package. Its efficiency together with the `scikit-learn` [64] inspired and compatible API, made this package our choice for representative SR solution. An extensive list of the possible settings for the `gplearn`'s SR implementation is described in the corresponding package documentation (see [80]). Therefore, in this chapter, we focus only on the features that we find particularly interesting with regards to our technique, i.e., when one uses SR within an MDP framework. Namely, settings that one should consider adjusting in a way different than the default one are listed in the following paragraphs.

Set of operators
The set of operators contains a list of the mathematical operators that are allowed in building and evolving the trees. Due to a trade-off between the complexity of the formulas and their accuracy, our approach is to start with the four basic binary operations: *addition*, *subtraction*, *multiplication*, and *division*. Next to that, we believe that often the threshold function might contain *square root* operation and/or *logarithm*. Therefore, we suggest running a few SR instances where trees can use various subsets of those mathematical operators. Comparing the outcomes of the configurations one can decide which results suit better one's goal. Note that there is no benefit in implicitly allowing exponentiation if one does not expect an exponent higher than two, as the trees $exponent(a, 2)$ and $multiply(a, a)$ have the same depth and length.

Initial depth and parsimony coefficient
One can control the length and the depth of the trees, i.e., the complexity of the expressions, by adjusting the *initial depth* (*init_d*) and the *parsimony coefficient* (*pc*). More precisely, the *init_d* is given by a tuple that defines the minimum and the maximum size allowed for the first generation of trees, whereas the *pc* is influencing how the further generations evolve by penalizing longer expressions. In this way, one can control the *"bloating" effect*, which is characterized as an increase in the trees' size that corresponds to an insignif-

icant improvement in their approximation accuracy. Larger values of the *pc* penalize larger trees more and make them less favorable for selection. In our MDP example, we generated instances with values for the *pc* between 0.001 and 0.1 and *init_d* in ranges varying from as low as $(2, 3)$ to the higher values of $(7, 8)$. Based on the conducted tests, we believe that setting the *pc* to 0.01 in combination with *init_d* range containing the number of system parameters, $|P_s|$, would be a reasonable default choice for our technique. In particular, we conclude that for our MDP problem *init_d* $= (3, 6)$ tends to produce the best results as the first generation consists of relatively simple expressions that nevertheless are complex enough to capture all system parameters.

Fitness function
The fitness of each *individual* is evaluated according to the fitness function, which is based on the approximation accuracy of the training samples. This implies that there are two important components of the *fitness function* – the way the error on a single training sample is defined and the way the overall fitness is obtained from those accuracy scores. Note that in our running example each system instance is represented by $M + 1$ data samples, one for each value of $0 \leq x_1 \leq M$. Therefore, it is crucial to assign appropriate weights to the various training samples associated with a common system instance. The reason is that the more probable a system state is, the greater impact an error in the corresponding decision has. Hence, one might consider using the steady state probability of each state as its weight. However, in some models, it is difficult to obtain the steady-state distribution. As a consequence, in our first running example, we tested the two simpler weight functions, namely, ρ^x and $(M - x)^2$, where ρ is the load of the system from the corresponding sample (P_s, x, y). Note that the second function does not require any additional computations as the maximum queue length M is given as a system parameter. Next to that, we believe that in the context of estimating a control policy of a threshold-type one should consider a fitness function that computes the relative error on a given sample instead of the absolute one. In conclusion, we recommend to use the weighted mean absolute percentage error (wMAPE) or weighted root mean squared error (wRMSE) [41].

4.3.4 Results

In this section, we evaluate the results from the various SR instances. The output of the algorithm depends on the specific settings. Therefore, one has

to decide whether it is possible to further adjust one or more of these settings to produce an expression that fits better one's goal: a simpler threshold function, or a more accurate one. Nevertheless, we believe that certain settings result in both better performing and less complex threshold decision policies. For that reason, next to the provided setting recommendations, we advice that one initially explores a larger number of various SR configurations and only afterward further evolve a few of the best ones. In such a case, it might become important to optimize the running time of each algorithm's instance. One way to achieve this is to train and/or test the first couple of configurations only on a subset of the corresponding data instead of the full one. Furthermore, one can keep track of the best fitness score for each generation and terminate the instance earlier if there is not much of an improvement in the score for a few generations in a row.

Next, we discuss the accuracy of our technique for our running example. We configure the SR algorithm in accordance with the guidelines described in Subsections 4.3.2 and 4.3.3. To study the possible generalization of our technique, we test settings that can be derived for any system within the studied MDP framework, e.g., weights and structured features that depend only on the system parameters. To evaluate the added value of our approach we compare it to SR instances with default settings. Next to that, we also include cases where only part of our recommendations were implemented. In Table 4.1, we list some of these instances, numbered with Roman numerals (i.e., I, II,..., VII) and the corresponding parameters. The first column, *Instance*, is used for reference purposes, whereas the other columns are self-explanatory.

Table 4.1: Setting configurations for running example 1.

Instance	Operators	ρ as a feature	Fitness function	Weights
I	$+, -, *, /$	No	RMSE	
II	$+, -, *, /$	No	wMAPE	ρ^x
III	$+, -, *, /$	Yes	wMAPE	ρ^x
IV	$+, -, *, /$	Yes	wRMSE	ρ^x
V	$+, -, *, /$	Yes	wRMSE	$(M-x)^2$
VI	$+, -, *, /, \sqrt{}$	Yes	wRMSE	$(M-x)^2$
VII	$+, -, *, /, \sqrt{}$	Yes	wRMSE	$(M-x)^2$

Our goal is to find the decision policy which minimizes the mean waiting

time of the system. Therefore, although the algorithm is approximating the threshold function, we will not examine how good of a fit the estimation is. Instead, we present the relative difference, E_r, between the mean waiting time if one uses the threshold policy from the generated expression, g^{est}, and the optimal one, g^{opt}. More precisely, we define

$$E_r := \frac{|g^{est} - g^{opt}|}{g^{opt}} \times 100\%.$$

The generated expressions and their accuracies are shown in Table 4.2. Next to the median, we state the 95^{th}, and the 99^{th} percentile of E_r. Note that although the formula from instance I approximates the optimal decision policy relatively well on average, the 99^{th} percentile error is extremely large. We believe that this is due to not using weighted score function for the corresponding SR program. This way, the SR algorithm assigns equal importance for each state whereas the mean waiting time depend mainly on the decision policy for states associated with a high steady-state probability. Based on the results for instances I to V, we conclude that incorporating the guidelines in accordance with the MDP framework greatly improves the accuracy of the expressions without adding complex terms.

Table 4.2: Numerical results for running example 1.

Instance	Threshold function	E_r percentiles		
		50^{th}	95^{th}	99^{th}
I	$\frac{3a\lambda-0.37}{\lambda\beta_1} - \frac{\lambda}{\lambda-1/\beta_1} + 0.27x + 0.96$	1.31	5.05	1280.08
II	$\lambda(\beta_1 + a) + \frac{a-bx}{\beta_1} + x$	0.32	1.65	2.60
III	$\lambda(\beta_1 + a + b) + \frac{a-2b}{\beta_1} + 0.75x + 0.4$	0.30	1.10	1.45
IV	$2\lambda(\beta_1 + a + b) + \frac{0.75(a-2b)}{\beta_1} + 0.5x$	0.26	1.28	3.48
V	$1.5\lambda(\beta_1 + a + b) + a(2\lambda + 1/\beta_1) + 0.33x$	0.27	0.93	1.04
VI	$\sqrt{0.4(a\lambda + x)}(1.6\sqrt{a/\beta_1} + 0.6\lambda)$	0.03	0.34	0.60
VII	$a\lambda + \sqrt{x}(a\lambda + 1.5) + \sqrt{\sqrt{x}(a\lambda + 1.5)}$	0.01	0.18	0.39

Next to that, we observe that including the square root operator can decrease the error even further. The significant accuracy improvement in both VI and VII instances indicates that the function characterizing the optimal decision policy might involve the *square root* operator. Indeed, in both cases

the threshold value contains the term $\sqrt{x}$. However, the remarkable accuracy of the threshold functions of VI and VII comes at the expense of an increased complexity of the expressions.

4.4 Running example 2: The M/M/1 queue with customer rejection

Next to numerically evaluating the performance of the generated functions, we are also interested in how well these symbolic expressions resemble the optimal one. Therefore, in this section, we apply our technique to an MDP system that has a closed-form solution. This enables us to compare the derived expressions to the real one. Moreover, we use this example to once again go through the steps of our approach. In the following, we implement the algorithm according to the guidelines described in Subsections 4.3.2 and 4.3.3. Furthermore, we apply the default settings in order to examine our technique in its general form, without any adjusting and tailoring to a specific model.

4.4.1 Model

We study a single server queue with Poisson arrivals with rate λ and exponentially distributed service times with rate μ. There are holding costs, c_h, associated with each customer in the queue. Furthermore, one can decide to reject a customer upon arrival. In such a case, rejection cost c_r is acquired. The control policy for this MDP problem is given by a threshold value τ. More precisely, a customer is admitted if and only if there are less than τ waiting customers at the queue. Even for such a simple system, obtaining a closed-form expression for the optimal decision policy is very challenging. However, it was shown in [6] that the long run average cost, g, for the special case $c_h = c_r = 1$ is given by:

$$g = \frac{\rho - \frac{(\tau+1)(1-\rho)\rho}{(1/\rho)^{\tau} - \rho}}{1-\rho} + \frac{(1-\rho)\lambda}{\left(\frac{1}{\rho}\right)^{\tau} - \rho},$$

where $\rho = \lambda/\mu$ is the system load and τ is the rejection threshold value. Now, minimizing g with respect to τ gives the optimal threshold value, denoted τ_{opt}:

$$\begin{aligned} \tau_{opt} = \mu - \lambda - 1 \\ - \frac{1}{\log(\rho)} \\ - \frac{L\left(-\rho \exp\left(\log\left(\rho\right)\left(\mu - 1 - \lambda\right) - 1\right)\right)}{\log(\rho)}, \end{aligned} \tag{4.1}$$

where $L(.)$ is the *Lambert-W* function [24, 49], also called the *omega function*, and given as the inverse function of

$$f(W) = We^{W}.$$

4.4.2 Data generation

Note that the state space of this MDP is one-dimensional, namely, the number of customers in the queue, and therefore each system configuration results in exactly one data sample (λ, μ, ρ, y) where y denotes the threshold value. We generated samples for systems with 100 equally spread values of μ in the range [1 ... 1000] and 100 values of λ for each μ resulting in ρ from 0.05 to 0.95.

4.4.3 Algorithm settings

Three different setting configurations were tested. In the first one we used only the four basic mathematical operators (*addition*, *subtraction*, *multiplication* and *division*), whereas in the second configuration we add one more that is present in the closed-form expression: *natural logarithm*. Finally, we generated a third program instance including also the *Lambert-W* function [24, 49] as a possible mathematical operator. In all configurations, the *init_d* and the *pc* were assigned to the suggested default values, namely $(1, 3)$ and 0.01, respectively. Furthermore, since each system instance is associated with exactly one sample, there is no need of using weights in the *fitness function*.

4.4.4 Results

The above-described three configurations and the corresponding approximations τ_1, τ_2, and τ_3 for the optimal threshold policies are listed in Table 4.3.

Table 4.3: Setting configurations for running example 2.

Instance	Operators	Threshold value τ
I	$+, -, *, /$	$\tau_1 = \mu - \lambda - 0.657 + \frac{\lambda}{\mu - \lambda}$
II	$+, -, *, /, log(.)$	$\tau_2 = \mu - \lambda - 1 - \frac{1}{\log(\rho)}$
III	$+, -, *, /, log(.), L(.)$	$\tau_3 = \mu - \lambda - 0.597 + \frac{\lambda}{\mu - \lambda}$

Interestingly, the results show that the technique is able to find the most influential terms (namely, $\mu - \lambda$) in all three configurations. Furthermore, in the first case (i.e., instance I) it approximated $(-1 - 1/\log(\rho))$ using the term $\lambda/(\lambda - \mu)$ and a constant. It is interesting to note that this is indeed a very good estimation as it is exactly the first (and most important) term, $\rho - 1$, from the Taylor expansion of $\log(\rho)$ at $\rho \uparrow 1$, namely,

$$\frac{1}{\log(\rho)} \approx \frac{1}{\rho - 1} = \frac{\mu}{\lambda - \mu} = \frac{\lambda}{\mu - \lambda} - 1, \text{ for } \rho \uparrow 1.$$

In addition, we believe that the high accuracy of this approximation led the algorithm to use it also in τ_3, although the log operand was allowed by the third settings configuration. The fact that τ_2 contains the exact term $-1 - 1/\log(\rho)$ shows that given more evolutionary time (e.g., more generations, and/or different random seeds) would have helped the third configuration to discover this term.

Finally, note that the last part of the exact threshold expression that involves the Lambert-W function was not included by the algorithm. After further analysis, we found that the mean and the variance of this term across the sampled systems were -0.005 and 0.003, respectively. Therefore, given that the mean threshold was 1252, we believe that this additional term is indeed negligible. Based on these findings, we conclude that one might use our technique to both derive a very well-performing decision policy and study the performance sensitivity in the system parameters.

4.5 Conclusion

This study is a pioneering contribution, in which we present a new and promising technique to obtain an analytic solution to MDPs that have a threshold-based optimal policy. The outlined method makes use of a specific machine learning algorithm - the Symbolic Regression. We showed how one could apply and tailor this evolutionary algorithm to the MDP framework. Our approach obtains a near-optimal decision policy that is given in a closed-form expression.

Moreover, the technique introduced in this chapter was tested on two MDP models, resulting in highly accurate approximations both in terms of the achieved system performance and the form of the expression. To further study the possible generalization of our method, we would have to apply it to other MDP problems. In addition, we find promising the idea of incorporating even more insights for the analyzed system to facilitate the SR algorithm.

Chapter 5

Closed-Form Performance Evaluation of Markov Decision Processes Using Symbolic Regression

In the previous chapters, we studied networks in which jobs are incoming from a single channel. However, in practice, there are also systems that manage the demand from multiple sources using a common pool of resources modeled as servers. The traditional approach would be to assign a number of these servers for each of the incoming streams of jobs. In case the demand rates vary over time, though, this technique would lead to congestion during peak times and idle resources during off-peak periods. A common solution is to *dynamically* allocate the resources to the different streams, hence the term *blending* systems.

As a continuation of Chapter 4, we further explore the potential of combining queueing theory together with tools from machine learning, in particular Symbolic Regression (SR). We present an *approximation algorithm* to obtain *closed-form expressions* for the performance metrics of a given system. In this chapter, we consider a two-stream *blending* model as a running example to demonstrate how one can incorporate valuable insights of the analysis of the system to greatly improve the algorithm's performance. The proposed method derives remarkably accurate algebraic expressions for the performance metrics of interest. The obtained formulas allow for sensitivity analysis, and furthermore, instant system resolving in case of a change in the parameters.

The work in this chapter is based on A. Hristov, S. Bhulai, R.D. van der Mei and J.W. Bosman. Performance evaluation through symbolic regression with an application to a two-stream blending system (2018) [37]. *Submitted.*

5.1 Introduction

In this chapter, we present a new method to approximate the performance metrics of a given system. Our algorithm exploits known results for special cases of the analyzed system to tailor and facilitate SR in obtaining a closed-form approximation (see Chapter 4 and [43, 48] for an introduction to SR). The closed-form expressions greatly facilitate studying the performance sensitivity in the system parameters and assessing the robustness in the parameter estimation. In addition, exact expressions allow one to directly recalculate the metrics for time-varying parameters. These advantages make our solution appealing for an application in real-time systems where parameters are changing continuously. The main contributions of this chapter are the high accuracy of the approximations and their explicit algebraic form. Next to that, we believe that our technique is promising and can be further studied and used for a broader range of queueing problems.

As a running example we analyze a specific two-stream blending system [8, 91]. One of the streams represents urgent jobs that are received according to a given stochastic process. The other stream consists of non-urgent (e.g., backlog) jobs available in an infinite quantity. This naturally implies the following two objectives for such a system: (1) minimize the mean waiting time of the former stream, and (2) maximize the throughput of the backlog work. However, there is a trade-off between these two goals as the two demand types share the same resource pool. This poses the question of how to control the number of servers allocated for each of the demands. Due to the abundance of such systems in practice and the challenging nature of the problem, there are a number of papers on this topic. The research conducted in [8,91] models the two-stream blending system as an MDP and derives the optimal threshold value for the control policy given a specific performance constraint. In our study, we consider the related subject of determining the expected performance for a given threshold value. Despite the considerable amount of research done on such systems, deriving analytic results remains a hard problem and the solutions so far involve computationally intensive numerical procedures.

In the following section, we list the main steps of our technique. Next, in Section 5.3, we demonstrate our approach by applying it to the two-stream blending model. After discussing the obtained results, we conclude the chapter with a summary in Section 5.4.

5.2 Method

In this section, we adapt our method for finding closed-form expressions for the performance metrics of a given MDP model (see Figure 4.4). Namely, the main four steps of our technique are:

Step 1: Model
Model the analyzed system as a queueing network. We define the system states, the corresponding transition rates and the performance metrics we are interested in. Next to that, we present a way to numerically obtain these metrics for a given system.

Step 2: Data generation
Generate data that is required by the SR algorithm. This data is a collection of samples, where each sample contains the parameter values and the associated performance metrics for a given system instance.

Step 3: Algorithm settings
Configure the algorithm parameters. There are many settings that determine the speed and the accuracy of SR. By adjusting these parameters, one can control the duration of the evolution, the initial population, and perhaps most importantly, the way the generations evolve.

Step 4: Results
Evaluate the results. As mentioned, different setting configurations lead to different results. There are certain choices that we believe are optimal, whereas for others, we iteratively analyze the obtained expressions and further adjust settings that might lead to a better result.

The contribution of our technique is twofold: (1) it derives very accurate approximations of the key performance metrics, and (2) it generates a closed-form expression, which allows both sensitivity analysis and instantaneous solution for any instance of the system. Therefore, our goal is to obtain formulas that are good approximations and at the same time relatively simple and easy to interpret. We believe that our solutions can be used for deriving both optimal system performance and fundamental insight into the dependency of the specific performance metric with regards to various system parameters.

5.3 Running example: Two-stream blending system

In this section, we analyze the two-stream blending system and apply our technique to obtain closed-form approximation of the mean waiting time and the throughput of the network. We first define the queueing model which we use to capture the characteristics of the system. Next, we present our approach in the following two crucial procedures: preparing the data and adjusting the settings of the algorithm. Furthermore, we evaluate the obtained approximations.

5.3.1 Model

We consider a system of s servers with two queues, i.e., two classes of jobs. For an illustration of the model, see Figure 5.1. Class 1 jobs arrive at the corresponding queue according to a Poisson process with rate λ, whereas the other queue is saturated and contains an infinite number of jobs. We will refer to the former queue as the first one and the corresponding jobs as the first class, and the latter as the second one with jobs of the second class. Furthermore, we assume the required service times of jobs of class $i = 1, 2$ to be independent and exponentially distributed with mean $\beta_i = 1/\mu_i$. Therefore, the load of the system, ρ, is given by $\rho = \frac{\lambda}{s\mu_1}$.

The first class of jobs represents the urgent jobs in the system. We are interested in their mean waiting time and give them priority over the second class of jobs, which stand for the backlog work. The case of preemptive service leads to trivial optimal control where class 2 jobs are being processed whenever there is an idle server. Hence, we assume that all s servers operate in a non-preemptive manner. Next to the implementation of priority policy, a second mechanism that is used in such systems is designating a number of servers, $0 \leq c \leq s$, that can serve both classes of jobs and reserve the rest $s - c$ only for the urgent demand. This leads to the following threshold-type policy:

- The first class of jobs is taken into service as soon as a server becomes available.
- The second class of jobs is taken into service only if the first queue is

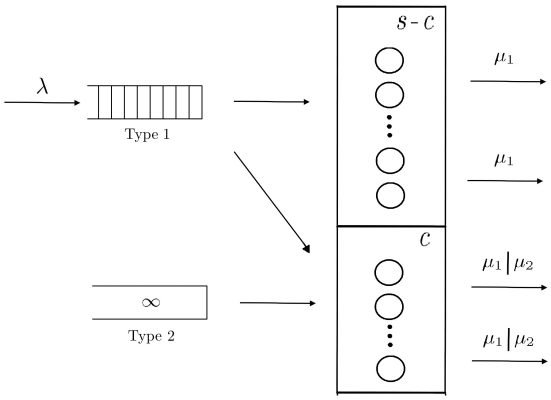


Figure 5.1: The two-stream blending model.

empty and there are fewer than c class 1 jobs in the system. This implies that there cannot be more than c class 2 jobs being served simultaneously as the rest $s - c$ are kept solely for the first queue.

We capture the system state by a couple (x_1, x_2), where $x_1 \geq 0$ is the number of class 1 jobs that are receiving service or are waiting in the first queue, and $0 \leq x_2 \leq c$ stands for the number of class 2 jobs in service. Consequently, we can determine the transition rates and the generator matrix Q of the corresponding continuous-time Markov chain. It is easy to verify that the non-zero non-diagonal entries of Q are:

$$\begin{aligned}
&q_{(i,j),(i+1,j)} = \lambda && \text{for } i \geq 0, \\
&q_{(i,j),(i-1,j)} = \min\{i, s-j\}\mu_1 && \text{for } i \geq 1,\ \max\{c-i, 0\} < j \leq c, \\
&q_{(i,j),(i-1,j+1)} = i\mu_1 && \text{for } 1 \leq i \leq c,\ j = c - i, \\
&q_{(i,j),(i,j-1)} = j\mu_2 && \text{for } i > c - j \text{ and } 1 \leq j \leq c.
\end{aligned} \tag{5.1}$$

Note that in our model c is not restricted to be an integer. Instead, we handle the case $x < c < x + 1$ where $0 \leq x < s$ is an integer, by randomizing between the following two threshold values: $c = x$ and $c = x + 1$. The probability of applying the lower value, x, is given by $0 < c - x < 1$.

5.3.2 Data generation

We use SR to derive an expression for the mean waiting time at the first queue and the throughput of the second one. To produce an estimate, the regression technique requires a set of training samples as input. Once trained on the corresponding data, the performance of the approximation can be evaluated on a separate set of test samples by comparing the predictions with the actual, real values. This procedure ensures that the approximation works well also on previously unseen examples, i.e., it does not overfit the specific training set.

In the following, we describe how we generate the data set for our approximations. Our objective is to obtain an estimate that can be used for a system with any parameters set $P_s = (\lambda, \mu_1, \mu_2, c, s)$. Therefore, we design model instances with the idea of generating samples for systems as diverse as possible. More precisely, we vary the parameters as follows:

- both μ_1 and μ_2 take value within $\{0.1, 0.2, 0.5, 1, 2, 5, 10, 20, 50, 100\}$,
- $s \in \{10, 20, 30, \ldots, 150\}$, and $c \in \{0, s/15, 2s/15, \ldots, s\}$,
- $\lambda \in \{0.1\rho, 0.3\rho, 0.5\rho, 0.7\rho, 0.9\rho\}$.

The above-listed choice of parameter values results in slightly more than 100000 samples. Next, we numerically solve these systems with the matrix-geometric method as outlined in the Appendix.

In addition to these base parameters for the model (i.e., the local variables), we also include self-composed ones (i.e., structured features). For example, one such variable that we include in the data set is the system load ρ as this metric is crucial for any queueing system. As discussed in Section 5.1, there is already a number of studies on two-stream blending systems. We believe that problem-specific insights should be used in order to further tailor the algorithm and as a consequence obtain better predictions in a more efficient manner. Although there is no closed-form solution for the general model that we consider, there are a few results in specific cases, e.g., when the service demands of the two classes of jobs are equal. Next to that, it is easy to verify that when the threshold value is zero (i.e., $c = 0$) one might use the Erlang C formula [27] to obtain the mean waiting time of the first stream. Based on these results we also include the following two parameters: ρ^{s-c} and $c!/s!$,

where $x!$ denotes the factorial of x. Finally, we note that in case $c = s$, the throughput of the second stream can be derived as $(1 - \rho)c\mu_2$, and therefore we add this self-composed variable as well.

According to the common terminology for prediction problems, we refer to the two metrics of interest, the mean waiting time and the throughput, as y_1 and y_2, respectively. We consider cases where one of the response variables is less than 0.05 or larger than 500 as unrealistic from a practical point of view, and therefore we exclude such cases from the data set. Next to that, we make sure that the generated samples are indeed representing diverse systems and there are instances with mean waiting time and throughput performance across the desired range of values. Therefore, we define the following four ranges for both y_1 and y_2: $[0.05, 0.5], [0.5, 1], [1, 10]$ and $[10, 500]$, which might be interpreted as *very low*, *low*, *high*, and *very high*, respectively. To ensure that the response variables are evenly spread among these categories, we randomly sample an equal number of instances from each of the four ranges. More precisely, for each performance metric and each category, we randomly select 1000 system instances for the training data set and another 1000 for the test data set. Note that this procedure is required to avoid overfitting specific type of systems (e.g., such that have *very low* throughput).

An analysis of the distribution of y_1 within the above-described data set shows that nearly 99% of the values are less than 1 (i.e., the mean waiting time is *very low* or *low*). Therefore, to provide a reasonable number of instances with $y_1 \in (1, 10)$ and $y_1 \in (10, 500)$, we introduce more samples with relatively high load ρ. Namely:

- Both μ_1 and μ_2 take values within $\{0.1, 1, 10, 50, 100\}$,
- $s \in \{1, 11, 21, \ldots, 91\}$, and $c \in \{0, s/10, 2s/10, \ldots, s\}$,
- $\lambda \in \{0.91\rho, 0.92\rho, \ldots, 0.99\rho\}$.

In such a way, we generate more than 6000 additional samples with target variable $y_1 > 1$. We note that the distribution of y_2 is also heavily right-skewed, but nevertheless, there are a considerable number of cases to sample from for each of the four categories.

5.3.3 Algorithm settings

After we generated the data set by solving the defined system instances and adding the self-composed variables, we specify the desired algorithm settings. An extensive list of the various settings is described in the corresponding `gplearn`'s package documentation (see [80]). In this chapter, we focus only on the settings that we find particularly relevant with regards to our technique, i.e., using SR to analyze a given queueing model. Namely, we adjust in a way different than the default the following parameters.

Set of operators
The set of operators contains a list of the mathematical operators that are allowed in building and evolving the trees. Based on the closed-form expressions for the special cases $c = 0$ and $c = s$, we decided to include *addition*, *subtraction*, *multiplication*, *division*, and *exponentiation*. For our running example, adding the *square root* and the *logarithm* operator resulted in an increase in the algebraic complexity of the expressions without corresponding improvement in the approximation accuracy.

Initial depth and parsimony coefficient
By adjusting the *initial depth* (*init_d*) and the *parsimony coefficient* (*pc*), we control the depth and the length of the trees, i.e., the complexity of the expressions. More precisely, the *init_d* is given by a tuple that defines the minimum and the maximum size allowed for the first generation of trees, whereas the *pc* is influencing how the further generations evolve by penalizing longer expressions. In such a way, one can manage the "bloating" effect – an increase in the trees' size that corresponds to a not significant increase in their fitness. Greater values of the *pc* penalize larger trees more and make them less favorable for selection. Based on the conducted tests, we believe that the optimal values for these two parameters are: $init\_d = (2, 6)$, and $pc \in (0.001, 0.2)$. Approximating the throughput performance metric in a closed-form expression proves to be more challenging, and therefore we use lower values of *pc* for these SR programs. Next to that, we further decrease the *pc* in case we let the specific SR instance evolve for more generations.

Fitness function
The fit of each individual (i.e., the tree representing a mathematical formula) is evaluated based on its approximation accuracy on the training samples. As discussed, our goal is to obtain approximations that are applicable for a

broad range of system parameters. Hence we want to avoid overfitting specific type of systems, while inaccurately predicting others. In other words, the fitness function has to be sensitive to large errors. Next to that, we construct our training set by including the same number of instances from each of the considered performance ranges. Therefore, we should apply the same weight for each estimation error, i.e., take their average. Due to these two observations and its common usage when determining the quality of an estimator, we choose the mean squared error (MSE) measurement as our fitness function:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2,$$

where n is the number of system instances in the data, $y^{(i)}$ the value to be predicted for the i-th system instance, and $\hat{y}^{(i)}$ the estimated value for the i-th system instance.

5.3.4 Results

As described, we produce a few SR algorithm instances with different settings for both the mean waiting time at the first queue, y_1, and the throughput of the second queue, y_2. We note that in most cases the approximation's accuracy might be further increased, e.g., by simply evolving the SR instance for more generations, including additional mathematical operators, decreasing *pc*, etc. However, there is a potential trade-off between the achieved accuracy and the algebraic complexity of the closed-form expressions. Therefore, we discuss only those results that we believe are relatively simple and at the same time are highly accurate.

Approximating the mean waiting time
First, we analyze the mean waiting time, y_1. Using only the four basic binary operations together with the logarithm and evolving for 30 generations, we could generate an extremely accurate closed-form prediction, $y_{1,pred}$, resulting in an MSE of 0.17 on both the train and the test data set for $\rho < 1$:

$$y_{1,pred} = \frac{\rho}{s\mu_1 - \lambda} + \frac{1}{s}\left(\frac{c}{s\mu_2} - \frac{\log(s)}{\mu_1}\right). \tag{5.2}$$

To establish a benchmark, we analyze the special case when the load of the first queue is nearly 1 (i.e., $\rho \uparrow 1$). In this setting, together with the

fact that class 1 jobs have priority, one can argue that the system behaves like an $M/M/1$ queue with a service rate $s\mu_1$. This implies the following approximation, $y_{1,\rho\uparrow 1}$, for $\rho \uparrow 1$:

$$y_{1,\rho\uparrow 1} = \frac{\rho}{s\mu_1 - \lambda}. \tag{5.3}$$

As a fair comparison, evaluating the two methods only on systems with $\rho = 0.99$, results in an MSE value of 0.57 for the heavy-traffic estimation and 0.1 for our approximation, $y_{1,pred}$. Therefore, we conclude that our closed-form prediction is almost exact and at the same time allows sensitivity analysis. It is interesting to note that the algorithm has derived the formula for the case $\rho \uparrow 1$ as it is part of the closed-form expression (5.2). Furthermore, it has included additional terms that not only adjust the heavy-traffic approximation to general cases, but even significantly increases the accuracy of the special case $\rho \uparrow 1$ itself.

As a final remark, we discuss the contribution of adding the self-composed features, i.e., $s\mu_1$, $s\mu_2$, ρ^{s-c}, $c!s!$, c/s, and $(1-\rho)c\mu_2$. We have run multiple SR instances without including these features and the most accurate formula amongst these approximations achieved an MSE of 29.19 on the train and 30.11 on the test data. Note that this error is more than 100 times higher than the one of the above-discussed SR. In addition, omitting the self-composed features resulted in significant increase in the algebraic complexity of the closed-form expressions. Therefore, we conclude that by incorporating these insights we could greatly facilitate the SR algorithm in finding an accurate and algebraically simple closed-form approximation of the mean waiting time.

Approximating the throughput

Approximating the throughput of the second queue, y_2, proves to be more involved. As discussed, we can explicitly derive $y_2 = (1-\rho)c\mu_2$ for the special case of $c = s$, and $y = 0$ for $c = 0$. Next to that, in [8] it was observed that the throughput seems to increase nearly linearly with the threshold value c. This leads to the approximation, $(1-\rho)c\mu_2$, which we use as a benchmark. Using our technique and the composed features we generated two SR instances. We evolved the first one for twenty generations with $pc = 0.005$, whereas the second one for thirty generations with $pc = 0.001$. As shown in Table 5.1, instance I results in a relatively compact formula that is 40 times more accurate than the benchmark. Furthermore, instance II achieves even lower MSE score. However, this improvement is accompanied by rapid growth in the algebraic

complexity and includes terms that are hard to interpret. We note that the SR configurations that we have run without using the self-composed features resulted in nearly five times higher error than instances I and II(i.e., MSE of 848.20). Moreover, the algebraic complexity of the SR instances without self-composed features prohibits listing them in Table 5.1.

Table 5.1: Approximation results for the expected throughput.

Instance	Expression	MSE	
		Train	Test
Benchmark	$(1-\rho)c\mu_2$	6890.43	7898.91
I	$\frac{(1-\rho)c^2\mu_2\rho^{s-c}}{s}$	170.79	195.58
II	$\frac{(1-\rho)c\mu_2\rho^{s-c}}{s} + \mu_2 c(1-\rho)(\rho-0.867)$	111.85	126.96

We conclude that our technique generates highly accurate approximations for both the mean waiting time and the throughput of the analyzed queueing system. Moreover, this accuracy is achieved without including too many terms in the algebraic expressions, and as a result, the formulas are relatively compact.

5.4 Conclusion

In this chapter, we further studied our method to learn the performance of a queueing system by using the SR algorithm. We showed that this algorithm is able to benefit from insights of the performance by analysis of the system in specific cases. By expanding on these results, our algorithm generates an expression for a broader range of parameters, and therefore, has the potential to push the field of queueing theory further.

We applied our method to approximate two main performance metrics of a given two-stream blending system. We showed how one could apply and tailor the SR algorithm to this queueing framework by incorporating insights of already known results for this system. The obtained closed-form expressions result in remarkably accurate estimations for both performance metrics.

The results raise a number of questions for further research. For example:

'How do the generated approximations relate to the design of the training data set?' Next to that, an extensive analysis of the SR settings that might be optimal in a typical queueing system would significantly facilitate the application of the presented technique to other problems within this field of research.

5.5 Appendix

In the following, we show how the matrix-geometric method [60] can be used to find the stationary distribution of the system for given system parameters λ, μ_1, μ_2, s and c. As discussed, the transition rates are given by (5.1). This implies that the generator matrix, Q, has a block structure of the following form:

$$Q = \begin{pmatrix} B_{00} & B_{01} & 0 & 0 & 0 & \cdots \\ B_{10} & A_1 & A_0 & 0 & 0 & \cdots \\ 0 & A_2 & A_1 & A_0 & 0 & \cdots \\ 0 & 0 & A_2 & A_1 & A_0 & \cdots \\ \vdots & \vdots & \vdots & \ddots & \ddots & \ddots \end{pmatrix},$$

with corresponding matrices

$$A_0 = \begin{pmatrix} s\mu_1 & 0 & \cdots & 0 \\ 0 & (s-1)\mu_1 & \cdots & 0 \\ \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & (s-\lceil c \rceil)\mu_1 \end{pmatrix}, \tag{5.4}$$

$$A_1 = \begin{pmatrix} q_{1,1} & 0 & \cdots & \cdots & 0 \\ \mu_2 & q_{2,2} & 0 & \cdots & 0 \\ 0 & 2\mu_2 & q_{3,3} & \cdots & 0 \\ \vdots & \ddots & \ddots & \vdots & \\ 0 & 0 & \cdots & \lceil c \rceil \mu_2 & q_{\lceil c \rceil+1, \lceil c \rceil+1} \end{pmatrix}, \tag{5.5}$$

$$A_2 = \begin{pmatrix} \lambda & 0 & \cdots & 0 \\ 0 & \lambda & \cdots & 0 \\ \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda \end{pmatrix}, \tag{5.6}$$

where $\lceil c \rceil$ and $\lfloor c \rfloor$ stand for the least integer greater or equal to c and the greatest integer less or equal to c correspondingly, and $q_{i,i} = -(A_0 + A_2)e_i$

for $1 \leq i \leq \lceil c \rceil + 1$ and e_i being the unit vector. Furthermore, A_0, A_1, and A_2 are $(\lceil c \rceil + 1)$-by-$(\lceil c \rceil + 1)$ square matrices, whereas B_{00}, B_{10}, and B_{10} are of size $s(\lceil c \rceil + 1)$-by-$s(\lceil c \rceil + 1)$. Finally, we note that we do not explicitly present the non-zero entities of B_{00}, B_{10}, and B_{10} due to readability and size constraints. Nevertheless, the exact values can be easily derived from the transition rates (5.1).

By definition this is a QBD process and the following normalization equation holds:

$$1 = \sum_{i=0}^{\infty} \pi_i e = \pi_0 e + \pi_1 (I + R + R^2 + \cdots) e = \pi_o e + \pi_1 (I - R)^{-1} e, \qquad (5.7)$$

where e is the unit vector, $\pi_i = (\pi_{i,0}, \pi_{i,1}, \ldots, \pi_{i,K})$ are the equilibrium probability vectors and R is such that $\pi_i = \pi_1 R^{i-1}$.

Now, one can apply the technique outlined in Section 3.7 to derive R, and consequently, the equilibrium probability vectors. Once the stationary distribution of the system is obtained, one can derive the mean waiting time of the priority jobs and the throughput of the rest.

Chapter 6

Performance Evaluation of Nested Systems

Various types of systems across a broad range of disciplines may be modeled by tandem queues with nested sessions. Strong dependence between the servers has proven to make such networks complicated and difficult to study. An exact analysis is in most of the cases intractable. Moreover, even when performance metrics such as the saturation throughput and the utilization rates of the servers are known, determining the limiting factor of such a network can be far from trivial. In this chapter, we present a simple, tractable and nevertheless relatively accurate method for approximating the above-mentioned performance measurements for any server in a given network. In addition, we propose an extension to the intuitive *'slowest server rule'* (SSR) for identification of the bottleneck, and show through extensive numerical experiments that this method works very well.

The work in this chapter is based on A. Hristov, J.W. Bosman, R.D. van der Mei and S. Bhulai. Throughput and bottleneck analysis of tandem queues with nested sessions. To appear in *Probability in the Engineering and Informational Sciences* [38].

6.1 Introduction

Networks where the service time at a given server is dependent on the queueing behavior of other queues are called Layered Queueing Networks (LQNs). Such systems find application in various fields like computer science [25, 74], health care [97], telecommunication software systems [79], and assembly lines [12]. In this chapter, we analyze a specific type of LQNs, which are characterized by having exactly one serving node per layer. We refer to such systems as tandem queues with nested sessions. As an example, consider a queueing network that models the customer dynamics at a gas station. Drivers have to first fuel their

cars at the fuel pump and then pay to a cashier inside the station. In this case, there are two nodes, i.e., servers: the fuel dispensers and the cashiers. Each one of these servers is characterized by a service rate and a number of available slots, i.e., sessions. Note that the customers occupy a place at the first node (the fuel dispensers) throughout the whole procedure – clients leave and free the space at the fuel pump only after filling gas and paying the cashier. Therefore, the service speed of the second server (the cashiers) influences the sojourn time at the first node as well.

The dependency between the various layers in an LQN may be significant and therefore must be taken into account. This makes the mathematical analysis of such networks challenging. Solutions based on approximate versions [5, 14, 78] of the mean value analysis algorithm [71] are researched in [25, 74, 95]. Other algorithms for obtaining estimates of the performance metrics of LQNs include those developed in [32,86,87], where stochastic process algebras are used for the analysis. Next to the above-mentioned approaches for generalized LQNs, there is a number of studies on special cases of such networks. A specific polling queueing system analyzed in [19] can be considered as an instance of an LQN. Other examples are systems with exactly two layers. Such networks are studied in [20] by means of the power-series algorithm [9] and in [65–67] with the help of matrix-analytic methods [60]. The algorithm presented in our study is approximating the performance metrics of an LQN without a restriction on the number of layers. However, each layer should contain exactly one server.

The goal of any bottleneck identification technique is to find the limiting factor for the performance of a given system. In queueing networks without a layered structure this task is rather trivial – the most loaded server is the one that is slowing down the system [76, 85]. However, in an LQN it might be the case that more than one server is influencing the throughput. In such cases, we need a new definition of a bottleneck and a new technique to identify it. In this chapter, by a bottleneck we refer to the server whose service rate modification results in the largest change in throughput for the system as a whole. This definition agrees with the one for traditional queueing models in case the network is not layered: the most loaded server is the one whose service rate modification is most influential for the throughput of the system. However, the dependency of the service times among servers with nested sessions turns the bottleneck identification in such networks into a far more complicated task. This is due to the fact that the most utilized server is not always the one that plays the biggest role in slowing down the system [26].

The first contribution of this research is the presented algorithm for approximating the performance of tandem queues with nested sessions. Given that the system is overloaded we estimate the throughput, i.e., the saturation throughput, and the utilization rate of each server. As a second contribution, we formulate an extended definition for an LQN bottleneck. Finally, we introduce a novel layer-specific measure that could be used as a bottleneck identifier.

The remainder of the chapter is organized as follows. Section 6.2 introduces the model that we consider throughout the chapter and the approximation algorithm that can be applied to such models in order to estimate the performance metrics. Furthermore, Section 6.3 extends the analysis of those systems by describing a method for bottleneck identification. The accuracy of the presented algorithms is shown in Section 6.4 and Section 6.5, respectively. Finally, in Section 6.6 we discuss further research and state our conclusions.

6.2 Model

In this section, we present the specific class of LQNs that we consider in the chapter. The model assumptions and the notation that is used throughout the chapter are introduced in this section. Finally, we further elaborate on our definition of a bottleneck.

6.2.1 Saturated tandem queues with nested sessions

Figure 6.1 will serve as an example in this section. It models a system consisting of three tandem queues with nested sessions. In such a network, customers acquire service from all the nodes before leaving the system. They visit the nodes according to a predefined path, which is the same for all of them. Customers start at the first node and only after receiving full service there, they go to the next node (the path is from left to right in Figure 6.1). Customers free the resource they occupy at a given node only after they receive their service from all the nodes in the network. In comparison, in traditional tandem queueing networks without a nested structure, the customers free the resource at a node as early as they finish with their service at the node itself. Furthermore, the terminology used in systems with a nested structure differs from

the one describing traditional queueing networks in a few important aspects. First, the nodes are referred to as servers. Second, the possible number of places that can be occupied by customers at those nodes are called sessions.

As we are interested in the maximum possible throughput that can be achieved for a given network and pinpoint the bottleneck in case a queue starts to build up, we analyze the system under the assumption that it is saturated. This implies that all sessions at the first server are constantly occupied. Therefore, we can model the system as a closed network with the number of customers equal to the number of sessions at the first node. Next to that, we assume the service time of server i to be exponentially distributed.

As input parameters for such models we use:

$N :=$ the number of layers in the network;

$c_i :=$ the number of sessions at the i-th server, where $1 \leq i \leq N$;

$\mu_i :=$ the service rate at the i-th server, where $1 \leq i \leq N$ and $\mu_i = 1/\beta_i$.

Furthermore, we denote the layer containing the first server as the top layer or equivalently the first layer. On one layer lower, i.e., on the second layer, is situated the second server and the queue behind it, and so on. The network consisting of the i-th layer, together with all the layers below it, is referred to as the i-th subsystem.

To illustrate, we refer to Figure 6.1. In the presented network, there are $N = 3$ layers with a number of sessions per server: $c_1 = 12, c_2 = 8$, and $c_3 = 5$. The occupied sessions are marked with black or striped depending whether the user is requiring service from the specific server or waiting for a service from another node. For example, although all twelve sessions are occupied at the first server, only three of them are being served there, whereas the other nine are waiting for/receiving service from either the second or the third node. Therefore, in total there are nine customers at the second subsystem.

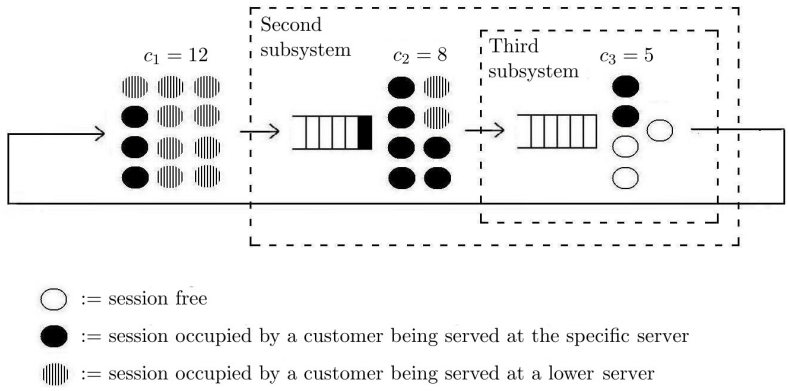


Figure 6.1: A three-layered model (with $c_1 = 12, c_2 = 8, c_3 = 5$).

6.2.2 Performance metrics

The goal of our approximation algorithm is to derive the following performance metrics:

$$T^{(i)} := \text{the saturation throughput of the } i\text{-th subsystem;}$$
$$T^{(i)}_{\max} := \text{the throughput of the } i\text{-th subsystem assuming no waiting times;}$$
$$U^{(i)}_{\text{norm}} := \text{the } \textit{normal} \text{ utilization rate of the } i\text{-th server;}$$
$$U^{(i)}_{\text{eff}} := \text{the } \textit{effective} \text{ utilization rate of the } i\text{-th server,}$$

where

- *normal* utilization is the average portion of sessions that are occupied at the corresponding server. The sessions can be occupied due to a request that currently receives service from the server or such that holds the session because it still needs to receive service at the sub-layers.
- *effective* utilization is the average portion of sessions that are occupied by requests that receive service from the corresponding server.

To show the difference between the *normal* and the *effective* utilization rate, we again refer to the system in Figure 6.1. Assume that the specific state

depicted there represents the average number of sessions occupied in the long run across the three servers. To obtain the *normal* utilization of a given server (i.e., $U^{(2)}_{\text{norm}}$), one has to calculate the fraction of the occupied sessions. This gives $U^{(2)}_{\text{norm}} = 1$. However, only six out of the eight customers at the second node are acquiring service there, whereas the other two are at the third server. Therefore, $U^{(2)}_{\text{eff}} = 0.75$.

Furthermore, we argue that one may assume $c_i \geq c_j$ for $i < j$ as, due to the layered structure of the network, the maximum number of sessions that can be occupied on a server j is $\min\{c_i \,|\, i < j\}$.

The models we consider have strictly one server per layer. Therefore, the sojourn time of a client at a given server i is exactly the sum of the waiting time at server i, w_i, the service time acquired at this node β_i, and the sojourn time in the $(i+1)^{th}$ subsystem.

6.2.3 Bottleneck analysis

Having obtained the maximum possible throughput of a given system, we determine what would be the most efficient way to increase this performance. We identify the bottleneck as the server that has the highest impact on the throughput of the network.

As an example, we take the model in Figure 6.1 with $\mu_1 = \mu_2 = \mu_3 = 5$ as a baseline. We plot the throughput in the three cases of μ_1, μ_2, or μ_3 varying while having the other two parameters fixed at their baseline values. As it can be seen in Figure 6.2, changes in the service rate of the third node correspond to largest gains/losses of the overall throughput. Hence, we conclude that in this case the bottleneck is the third server.

6.3 Performance analysis

In this section, we describe our algorithm for approximating the performance metrics and our bottleneck identification technique for tandem queues with nested sessions. We present the method used to estimate the throughput and the utilization rates by first applying it to a two-layered system. Second, we

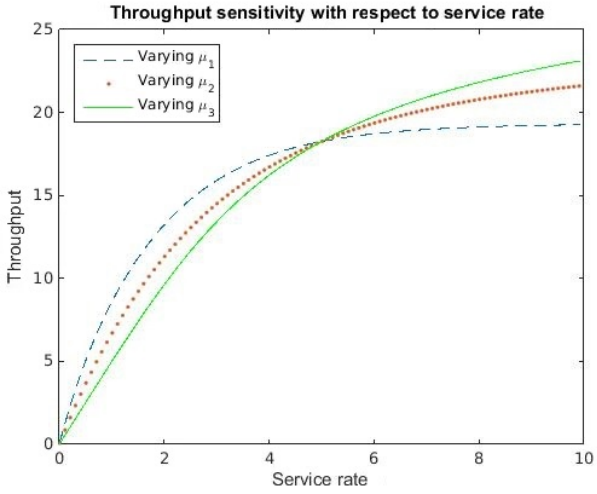


Figure 6.2: Throughput as a function of the service rates.

show how one can use a recursive scheme for approximating the performance metrics of a network with any given number of layers $N > 2$. Finally, we describe our bottleneck identification technique, which relies on the saturation throughput of the various subsystems.

6.3.1 Exact solution for $N = 2$

We analyze the special case of a two-layered system. We model the network as a continuous time Markov chain with corresponding state space $\mathcal{S} := \{0, 1, 2, \ldots, c_1\}$, where state i denotes the total number of occupied sessions at the second server. One can easily verify that this Markov chain has a generator matrix Q with the following non-zero, non-diagonal entries:

$$\begin{aligned} q_{k,k+1} &= \mu_1(c_1 - k), & k &= 0, 1, \ldots, c_1 - 1, \\ q_{k,k-1} &= \mu_2 \min\{c_2, k\}, & k &= 1, 2, \ldots, c_1, \end{aligned}$$

where $q_{i,k}$ is the transition rate from state i to state k. In addition, we define

$$\rho_k := \frac{q_{k-1,k}}{q_{k,k-1}}, \qquad k = 1, 2, \ldots, c_1. \tag{6.1}$$

As the analyzed system is a birth-death process, one can obtain the stationary distribution $\pi = (\pi_0, \pi_1, \ldots, \pi_{c_1})$ as a product form:

$$\pi_0 = \left(1 + \sum_{i=1}^{c_1} \prod_{j=1}^{i} \rho_j \right)^{-1},$$

$$\pi_k = \pi_0 \prod_{i=1}^{k} \rho_i, \qquad k = 1, 2, \ldots, c_1.$$

Furthermore, using the stationary distribution π, one can obtain the desired performance metrics in the following manner:

$$T^{(1)} = \sum_{i=0}^{c_1} \pi_i (c_1 - i)\mu_1, \tag{6.2}$$

$$U_{\text{eff}}^{(1)} = \sum_{i=0}^{c_1} \pi_i \frac{(c_1 - i)}{c_1}, \tag{6.3}$$

$$U_{\text{eff}}^{(2)} = U_{\text{norm}}^{(2)} = \sum_{i=0}^{c_1} \pi_i \frac{\min\{i, c_2\}}{c_2}. \tag{6.4}$$

6.3.2 Recursive scheme for $N > 2$

We introduce a few notations that are used in this section:

- $\pi^{(i,j)}$:= the stationary distribution of the i-th subsystem conditioned that it contains exactly j customers;
- $T^{(i,j)}$:= the saturation throughput of the i-th subsystem conditioned that it contains exactly j customers;
- $U_{eff}^{(\ell,i,j)}$:= the *effective* utilization rate of the ℓ-th server in case the i-th subsystem is studied in isolation and contains exactly j customers;
- $U_{norm}^{(\ell,i,j)}$:= the *normal* utilization rate of the ℓ-th server in case the i-th subsystem is studied in isolation and contains exactly j customers.

In the following, we describe how one can estimate $\pi^{(i,j)}$, $T^{(i,j)}$, $U_{\text{eff}}^{(\ell,i,j)}$, and $U_{\text{norm}}^{(\ell,i,j)}$, for any given $1 \leq i \leq N-1$; $0 \leq j \leq c_{i+1}$, and $1 \leq \ell \leq i$. To analyze

the i-th subsystem in isolation we assume it to be saturated. Therefore, all the sessions on the i-th server should be occupied. Hence, we take $\min\{j, c_i\}$ as the number of sessions at the i-th server. Moreover, due to the assumption that there is an infinite number of customers in the i-th queue, there is no difference in the networks with $j \geq c_i$. Therefore, in the following analysis we let $j \leq c_i$.

The approach is similar to the one described in the case of a two-layered system. Again, we model the i-layered network as a continuous time Markov chain with a state space $\mathcal{S} := \{0, 1, 2, \ldots, j\}$, where the state denotes the total number of customers in the $(i-1)$-layered subsystem. In this case, however, we use the following approximation to the transition rates:

$$\begin{aligned} q_{k,k+1} &= \mu_i(j-k), & k &= 0, 1, \ldots, j-1, \\ q_{k,k-1} &= T^{(i+1,k)}, & k &= 1, 2, \ldots, j. \end{aligned}$$

Having the approximated transition rates, one can use those values in Equation (6.1) to obtain ρ_k for $1 \leq k \leq j$. Furthermore, by the same method described in Section 6.3.1, one can derive the stationary state probabilities $\pi^{(i,j)}$. Once the values of $\pi^{(i,j)}$ have been obtained, one can calculate the performance metrics for the corresponding (i, j) tuple as follows:

$$T^{(i,j)} = \sum_{k=0}^{j} \pi_k^{(i,j)} (j-k)\mu_i,$$

$$U_{\text{eff}}^{(\ell,i,j)} = \begin{cases} \sum\limits_{k=0}^{j} \pi_k^{(i,j)} \frac{j-k}{c_i}, & \ell = i, \\ \sum\limits_{k=0}^{j} \pi_k^{(i,j)} U_{\text{eff}}^{(\ell,i+1,j)}, & i < \ell \leq N, \end{cases}$$

$$U_{\text{norm}}^{(\ell,i,j)} = \begin{cases} \frac{j}{c_i}, & \ell = i, \\ \sum\limits_{k=0}^{j} \pi_k^{(i,j)} U_{\text{norm}}^{(\ell,i+1,j)}, & i < \ell \leq N. \end{cases}$$

As can be seen, the results for the i-th subsystem depend on the values for the $(i+1)$-th one. Therefore, starting from the last, i.e., the $(N-1)$-th, subsystem one can obtain the desired metrics by the method for the two-layered network. Using the technique described above, one can further recursively derive the values for higher subsystems and eventually the ones for the initial system: $T^{(1,c_1)}$, $U_{\text{eff}}^{(\ell,1,c_1)}$, and $U_{\text{norm}}^{(\ell,1,c_1)}$ for $1 \leq \ell \leq N$.

6.3.3 Bottleneck identification technique

In this section, we explain our technique for bottleneck identification. We want to determine the server whose service rate is the most influential one to the overall throughput. Therefore, similar to the *'slowest server rule'* (SSR), we first calculate each server's speed in isolation $c_i\mu_i$, where $1 \leq i \leq N$. Next to that, due to the nested structure of the system, one should take into account the dependence between the various servers. Thus, we introduce a server-specific metric, which we refer to as the *effectiveness* rate of the server. One can obtain this metric as follows:

$$T^{(i)}/T^{(i)}_{\max} = \frac{c_i}{\sum_{i\leq k\leq N}(\beta_k + w_k)} \Big/ \frac{c_i}{\sum_{i\leq k\leq N}\beta_k} = \frac{\sum_{i\leq k\leq N}\beta_k}{\sum_{i\leq k\leq N}(\beta_k + w_k)},$$

for $1 \leq i < N$. Note that this is exactly the ratio between the total service time and the actual expected sojourn time. Therefore, the higher this ratio is (i.e., the closer to 1 it is), the less waiting occurs in the nodes below it, and hence the less influential this nested structure is for the specific server. On the other hand, a small ratio would stand for a large waiting time in lower nodes, which means that the server is not working at its full potential.

In the following, we summarize our bottleneck identification procedure:

Step 1: Obtain $T^{(i)}$ and $T^{(i)}_{\max}$ for all $1 \leq i \leq N$. The value of $T^{(i)}$ can be estimated by applying our approximation algorithm on the i-th subsystem, whereas $T^{(i)}_{\max}$ can be directly derived from the model parameters:

$$T^{(i)}_{\max} = \frac{c_i}{\sum_{i\leq k\leq N}\beta_k};$$

Step 2: Assign the following "Eff" score to each server:

$$\text{Eff}^{(i)} = \frac{T^{(i)}}{T^{(i)}_{\max}} c_i\mu_i,$$

where $1 \leq i \leq N$;

Step 3: Determine the server with the smallest "Eff" score as the bottleneck.

6.4 Results

In this section, we present the results of the approximation algorithm described in Section 6.3.2. In addition, we also show how our technique relates to methods introduced in similar studies.

The system parameters that we vary in the numerical tests are the number of layers N, the number of sessions per server, $c = (c_1, c_2, \ldots, c_N)$, and the service rate at each node, $\mu = (\mu_1, \mu_2, \ldots, \mu_N)$. To evaluate our method we want to identify a possible relationship between the model parameters and the algorithm's accuracy. Since each layer is associated with two variables, extensive test suites for systems with many layers are unfeasible. Therefore, we first analyzed the performance of the algorithm for $N = 3$, in which case we could examine more than $2,000,000$ different model instances with a broad range of values for the vectors c and μ. We used the results of these initial tests to find out how the number of sessions and the service rates influence the estimation accuracy. Next to that, we identified the worst-case scenarios for three-layered systems. Based on the findings, we could tailor the test settings for networks with more layers in such a way that we get the average and the worst case performance indicators for the algorithm. Moreover, we verified that the observed relationship between the system parameters and the performance of our method remain valid also for networks with more layers, e.g., ten layers. In these cases, we compared the calculated estimates to values obtained by a simulation as the exact solutions are computationally intractable.

6.4.1 Results for three-layered networks

First, we designed three test suites with a different number of sessions per server $c = (c_1, c_2, c_3)$. Namely, one that represents a small system with $c = (3, 2, 1)$, one for $c = (12, 8, 5)$, and the third having $c = (100, 90, 70)$. In all test suites we vary the values of the service rates μ_1 and μ_2 from 0.1 to 10 with a step size of 0.1. Next to that, without loss of generality, we have fixed $\mu_3 = 1$. The reason is that only the ratio between the various service rates is important with respect to the approximation error. This comes from the fact that scaling the rates is equivalent to scaling the time, which does not influence the performance metrics estimated by the algorithm.

At this point, we examine only three-layered systems as they can also be

solved exactly, which makes the analysis faster and more accurate in comparison to simulation. This allows us to perform a vast number of test cases and draw conclusions based on the results.

As it can be seen from the plots in Figure 6.3 for the three-layered systems, the throughput approximation error in each of the test suites is largest in a specific region and goes to 0 outside of it. Therefore, we will not examine the average approximation error as it would be highly dependent on the chosen range of parameters. Instead, we focus on the worst-case scenarios.

Another observation is that the lower the number of sessions per server there are, the greater the error is. For example, the maximum error in the first test suite is 0.7%, which is more than twenty times higher than the one in the case of a relatively large number of sessions.

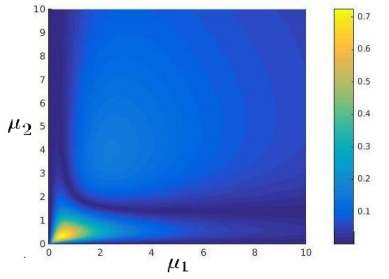
In order to test these two conclusions, we design 220 additional test suites of three-layered models. The service rates are again set to vary in the same manner: μ_1 and μ_2 range from 0.1 to 10 with a step size of 0.1 and $\mu_3 = 1$. However, this time we examine all possible tuples $c = (c_1, c_2, c_3)$ up to ten sessions per server, i.e., all combinations of c_1, c_2, and c_3 where $c_3 \leq c_2 \leq c_1 \leq 10$.

As expected, all 220 plots of the approximation error look almost the same as the one from Figure 6.3(a) – the relatively high error rates form a specific region. In those test suites, the region is around the point for which $c_1\mu_1 = c_2\mu_2 = c_3\mu_3$. The results of those 2.2 million additional test cases also agree with our second observation – the fewer the number of sessions, the worse the algorithm performs. Based on these, one can correctly identify the worst cases from all the tests conducted so far - the highest error of 2.4% is observed at the systems with the following parameters: $c = (2, 1, 1)$ and $\mu = (0.6, 1, 1)$; $c = (3, 1, 1)$ and $\mu = (0.3, 1, 1)$.

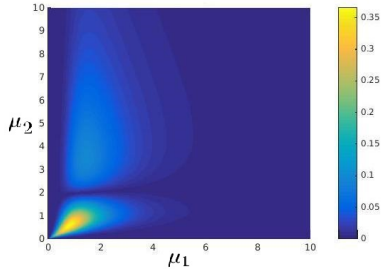
6.4.2 Results for ten-layered networks

With the test suites consisting of three-layered systems we examined the dependency between the accuracy of the algorithm and the number of sessions and the service rate of the various servers. Next to that, we want to analyze the influence of the number of layers on the approximation error of the algorithm. Therefore, we look at systems with more than three layers. However,

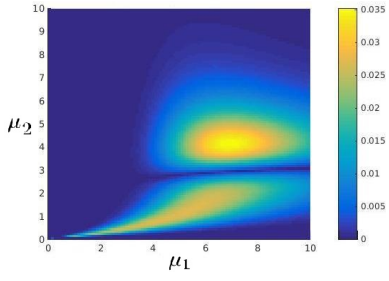
in those cases, we use simulation to obtain benchmark results as the exact solution is computationally intractable.



(a) $c = (3, 2, 1)$



(b) $c = (12, 8, 5)$



(c) $c = (100, 90, 70)$

Figure 6.3: Throughput approximation errors for the various test suites for three-layered systems.

In the following, we investigate systems with ten layers. As there are twenty different parameters for those systems, extensive test suites where all those parameters are varied are not possible. First, we performed more than $10,000$

test cases with random parameter sets. Complying with our observations so far, the relative error of those tests was in most of the cases insignificant and rarely larger than 0.1%. Therefore, we tried to tailor the parameters in order to find the worst-case scenarios, which we can compare to those found in the three-layered systems. The highest error that we found is 7.7% for the ten-layered network with the following parameters: $c = (4, 3, 3, 2, 2, 2, 1, 1, 1, 1)$, and $\mu = (0.5, 0.75, 0.75, 0.9, 0.9, 0.9, 1, 1, 1, 1)$. Next to systems with the above stated number of sessions per server, we analyzed two more test suites with parameters $c = (10, 9, 8, 7, 6, 5, 4, 3, 2, 1)$ and $c = (15, 14, 13, 12, 11, 10, 9, 8, 7, 6)$. We varied the service rates in the range $20\% - 500\%$ of the following values: $\mu = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)$ and $\mu = (1, 1, 1, 2, 2, 2, 2, 3, 3, 3)$ correspondingly. These two additional test suites were designed to examine the algorithm's accuracy error in unfavorable parameter sets - a few number of sessions per node and similar servers' speed. Nevertheless, the approximation error was rarely significant with a 95^{th} percentile value of 3.76%.

Due to its recursive nature, the accuracy of our algorithm indeed gets worse with an increase in the number of layers. Nevertheless, even the largest errors found for ten-layered systems are comparable with the average ones stated for other algorithms [25, 87]. At the same time, one has to consider that our method is designed specifically for tandem queues with nested sessions, whereas the above-cited algorithms can be used to solve more general LQNs.

6.5 Bottleneck identification results

In the following, we present the results with regards to our bottleneck identification method. To the best of our knowledge, the prominent techniques researched so far are the ones shown in [26] and the intuitive *'slowest server rule'* (SSR). Therefore, we compare those three bottleneck identification metrics.

Recall that the bottleneck identified by our method is the server with the lowest $\text{Eff}^{(i)}$ score. Similarly, the server considered as the bottleneck according to the SSR is the one with the smallest $c_i\mu_i$ product. Note that the SSR is used for traditional tandem queues without nested structure, where the throughput of the whole network is determined by the speed of the slowest node in the chain. Therefore, considering that the throughput of a given server i is the number of sessions times the service rate of each one of them, $c_i\mu_i$, it follows

directly that the node with the lowest $c_i\mu_i$ is the limiting factor.

Following the technique in [26], we assign one more score $\text{BStrength}^{(i)}$ to each server except for the last one. The main idea is to take the ratio of the *normal* utilization rate $U^{(i)}_{\text{norm}}$ of server i over the largest *normal* utilization rate of a server below it, $\max_{\{j>i\}} U^{(j)}_{\text{norm}}$. The server with the largest score is the one identified as the bottleneck, i.e.,

$$\max_{1\leq i<N} \text{BStrength}^{(i)} = \max_{1\leq i<N} \frac{U^{(i)}_{\text{norm}}}{\max_{j>i} U^{(j)}_{\text{norm}}}.$$

Finally, we describe the method which we use to determine the real bottleneck. According to our definition, this is the server whose speed modification results in the largest change in the saturation throughput of the system. Therefore, given a system with N servers, we examine N modified networks. In each one of those N systems, the service rate of one of the nodes is increased by 0.01%, whereas all other parameters are kept the same. Using exact analysis we derive the saturation throughput of those modified networks and identify the largest one. In this way, we find the server that is the most influential to the performance of the initial system.

Based on the above-described test framework, we evaluate the *Eff*, *SSR*, and *BStrength* techniques on test suites, similar to those described in Section 6.4 above. Once again, for the three-layered systems we take: $c = (3, 2, 1)$; $c = (12, 8, 5)$, and $c = (100, 90, 70)$ with μ_1 and μ_2 vary from 0.1 to 10 with a step size of 0.1. However, this time, next to the test cases where μ_3 is fixed at 1, we include such with μ_3 fixed at 10 and at 100. The test suites are extended in such a way in order to achieve a fair number of cases in which a specific server is the bottleneck. In addition, we evaluated the bottleneck identification techniques for all the configurations of the ten-layer system that we described previously.

The results from the test suites with three-layer systems are listed in Table 6.1. The table shows the percentage of cases in which the corresponding technique identified the bottleneck correctly. Due to the fact that no BStrength score can be assigned to the last node, we exclude the cases that have the third server as a bottleneck when evaluating this specific technique. From the results, one can conclude that our technique is performing better than the other two. Note that the higher in the network the bottleneck is, the harder it is for the three techniques to identify it correctly. This can be explained by the

nested structure being more influential for those servers than for the lower ones.

Table 6.1: Cases of correct bottleneck identification.

Technique	Server 1	Server 2	Server 3	Total
Eff	85.07%	95.41%	99.98%	93.74%
SSR	83.42%	94.16%	100.00%	92.88%
BStrength	36.49%	100.00%	–	72.24%

Figure 6.4(a) shows that the cases, in which our technique does not give the correct result, lie on the borderline of the parameters ranges for which the bottleneck shifts from one server to another. In those cases, two or more servers are almost equally influential to the overall throughput, and hence even though the server identified by our method is not the real bottleneck, it still does limit the performance of the system. Therefore, we believe that the following relative error gives more insight into the performance of the algorithms:

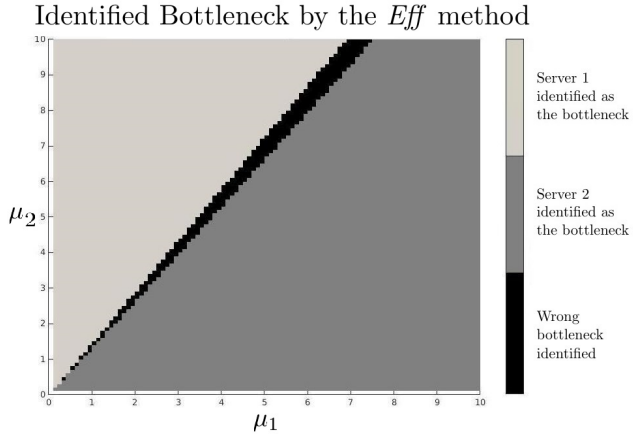
$$E_r = \frac{T_{new} - T_{alg}}{T_{new} - T_{old}} \times 100\%,$$

where T_{old} denotes the saturated throughput of the tested system, T_{alg}, the one where the rate of the server identified as the bottleneck by the corresponding algorithm is increased with 0.01%, and T_{new}, the one where the rate of the server that is the real bottleneck is modified with 0.01%. The average relative error E_r from all test cases (both the three-layered and the ten-layered configurations) is 0.92% for our technique, 0.95% for the SSR and 38.49% for the BStrength.

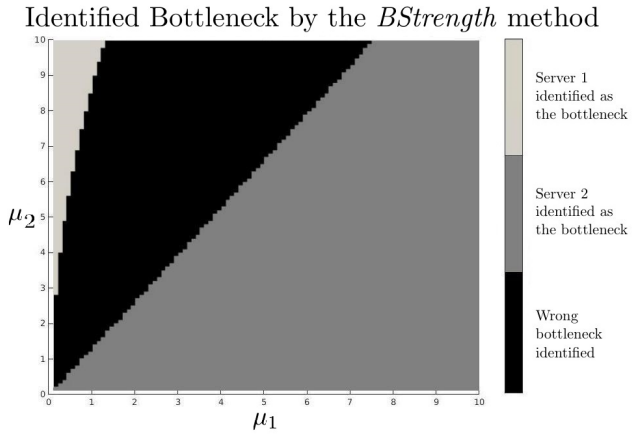
As a final remark, we would like to point out that our method for bottleneck identification coincides with the SSR in case of a tandem queueing network or in the similar case of an equal number of sessions per server. Therefore, we believe that our method can be considered as a generalization of the SSR.

6.6 Conclusion

This chapter has studied the performance evaluation of tandem queues with nested sessions. An approximation algorithm for obtaining the servers' utiliza-



(a) Eff method



(b) BStrength method

Figure 6.4: Bottleneck identification results for $c = (3, 2, 1)$ and $\mu_3 = 100$.

tion rates and throughput was presented. We believe that the high estimation accuracy of the algorithm is an indication of low approximation errors also in

case the model is further extended to fit more general LQNs. To study the generalization of our technique, we would have to apply it to LQNs that have a non-linear structure. Therefore, a promising further research might allow incorporating our method in more complicated layered networks.

In addition, we introduced a server-wise metric that extends the analysis by identifying the bottleneck in a given queueing network. Even though it did not pinpoint the most throughput-limiting server in a small percentage of the cases, it still identified a server that is severely impeding the performance of the system. Moreover, being an extension of the SSR, it is applicable to queueing networks without a layered structure as well. Therefore, we believe that this bottleneck identification method is more general and at the same time more accurate than the existing techniques so far.

Bibliography

[1] S. Aalto. Optimal control of batch service queues with finite service capacity and linear holding costs. *Mathematical Methods of Operations Research*, 51(2):263–285, 2000.

[2] S. Asmussen. *Applied Probability and Queues*. Springer-Verlag New York, 2003.

[3] S. Balsamo. Closed queueing networks with finite capacity queues: Approximate analysis. In *Proceedings of the 14th European Simulation Multiconference on Simulation and Modelling*, pages 593–600. SCS Europe, 2000.

[4] D. Barash. A genetic search in policy space for solving Markov decision processes. In *Proceedings of AAAI Spring Symposium on Search Techniques for Problem Solving under Uncertainty and Incomplete Information*. AAAI Press, 1999.

[5] Y. Bard. Some extensions to multiclass queueing network analysis. In *Proceedings of the 3rd International Symposium on Modelling and Performance Evaluation of Computer Systems*, pages 51–62, 1979.

[6] S. Bhulai. *Markov Decision Processes: the Control of High-Dimensional Systems*. PhD thesis, Vrije Universiteit Amsterdam, 2002.

[7] S. Bhulai and G.M. Koole. On the structure of value functions for threshold policies in queueing models. *Journal of Applied Probability*, 40(3):613–622, September 2003.

[8] S. Bhulai and G.M. Koole. A queueing model for call blending in call centers. *IEEE Transactions on Automatic Control*, 48(8):1434–1438, August 2003.

[9] J.P.C. Blanc. *Performance Evaluation of Computer and Communication Systems, Lecture Notes in Computer Science*, pages 53–80. Springer, Berlin, Heidelberg, 1993.

[10] M. Boon, R.D. van der Mei, and E.M.M. Winands. Applications of polling systems. *Surveys in Operations Research and Management Science*, 16:67–82, March 2011.

[11] A. Brandwajn and Y.-L.L. Jow. An approximation method for tandem queues with blocking. *Operations Research*, 36(1):73–83, 1988.

[12] R. Buitenhek, G.-J. van Houtum, and H. Zijm. AMVA-based solution procedures for open queueing networks with population constraints. *Annals of Operations Research*, 93:15–40, 2000.

[13] K. Chandan and K.S. Kumar. Analysis of two-phase N-policy $M/M^{(B)}/1$ queueing system with server startup. *Asian Journal of Computer Science and Information Technology*, 5(8), 2015.

[14] K.M. Chandy and D. Neuse. Linearizer: a heuristic algorithm for queueing network models of computing systems. *Communications of the ACM*, 25:126–134, 1982.

[15] H.S. Chang, H.-G. Lee, M.C. Fu, and S.I. Marcus. Evolutionary policy iteration for solving Markov decision processes. *IEEE Transactions on Automatic Control*, 50(11):1804–1808, November 2005.

[16] K.-H. Chang and W.-F. Chen. Admission control policies for two-stage tandem queues with no waiting spaces. *Computers & Operations Research*, 30(4):589–601, April 2003.

[17] J.W. Cohen. *The Single Server Queue*. Elsevier Science Publishers B.V., Amsterdam: North-Holland, 1983.

[18] A.E. Conway and J. Keilson. Analysis of a two-stage finite buffer flow controlled queueing model by a compensation method. *Operations Research Letters*, 18(2):65–74, September 1995.

[19] J.L. Dorsman, N. Perel, and M. Vlasiou. Server waiting times in infinite supply polling systems with preparation times. *Probability in the Engineering and Informational Sciences*, 30(2):153–184, 2016.

[20] J.L. Dorsman, R.D. van der Mei, and M. Vlasiou. Analysis of a two-layered network by means of the power-series algorithm. *Performance Evaluation*, 70:1072–1089, 2013.

[21] J.L. Dorsman, R.D. van der Mei, and E.M.M. Winands. Polling systems with batch service. *OR Spectrum*, 34(3):743–761, 2011.

[22] B.T. Doshi. Analysis of a two phase queueing system with general service times. *Operations Research Letters*, 10(5):265–272, 1991.

[23] A.E. Eiben and J.E. Smith. *Introduction to Evolutionary Computing*. Springer-Verlag Berlin Heidelberg, 2008.

[24] L. Euler. Observationes variae in mathesin puram. *Acta Academiae Scientiarum Petropolitanae*, 2:29–51, 1783.

[25] G. Franks, T. Al-Omari, M. Woodside, O. Das, and S. Derisavi. Enhanced modeling and solution of layered queueing networks. *IEEE Transactions on Software Engineering*, 35(2):148–161, 2009.

[26] G. Franks, D. Petriu, M. Woodside, J. Xu, and P. Tregunno. Layered bottlenecks and their mitigation. In *Proceedings of the 3rd international conference on the Quantitative Evaluation of Systems (QEST '06)*, pages 103–114. IEEE Computer Society, 2006.

[27] N. Gans, G.M. Koole, and A. Mandelbaum. Telephone call centers: Tutorial, review, and research prospects. *Manufacturing & Service Operations Management*, 5(2):79–141, April 2003.

[28] W.K. Grassmann and S. Drekic. An analytical solution for a tandem queue with blocking. *Queueing Systems*, 36(1):221–235, November 2000.

[29] N. Guerouahane, D. Aissani, N. Farhi, and L. Bouallouche-Medjkoune. M/G/c/c state dependent queuing model for a road traffic system of two sections in tandem. *Computers & Operations Research*, 87:98–106, 2017.

[30] U.C. Gupta and S. Pradhan. Queue length and server content distribution in an infinite-buffer batch-service queue with batch-size-dependent service. *Advances in Operations Research*, Article ID 102824, 2015.

[31] M. Harchol-Balter. *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*. Cambridge University Press, New York, NY, USA, 2013.

[32] U. Herzog and J.A. Rolia. Performance validation tools for software / hardware systems. *Performance Evaluation*, 45:125–146, 2001.

[33] D.P. Heyman. Optimal operating policies for $M/G/1$ queuing systems. *Operations Research*, 16(2):362–382, 1968.

[34] A. Hordijk and G.M. Koole. On the shortest queue policy for the tandem parallel queue. *Probability in the Engineering and Informational Sciences*, 6(1):6379, 1992.

[35] A.V. Hristov, S. Bhulai, J.W. Bosman, and R.D. van der Mei. Control of a tandem queue with a start-up cost for the second server. To appear in *Stochastic Models*, 2018.

[36] A.V. Hristov, S. Bhulai, R.D. van der Mei, and J.W. Bosman. Deriving explicit control policies for Markov decision processes using symbolic regression. *Submitted*.

[37] A.V. Hristov, S. Bhulai, R.D. van der Mei, and J.W. Bosman. Performance evaluation through symbolic regression with an application to a two-stream blending system. *Submitted*.

[38] A.V. Hristov, S. Bhulai, R.D. van der Mei, and J.W. Bosman. Throughput and bottleneck analysis of tandem queues with nested sessions. To appear in *Probability in the Engineering and Informational Sciences*, 2018.

[39] A.V. Hristov, J.W. Bosman, R.D. van der Mei, and S. Bhulai. Analysis and control of a single server queue with post-process batching. *Submitted*.

[40] T. Jansen. *Analyzing Evolutionary Algorithms: The Computer Science Perspective*. Springer-Verlag Berlin Heidelberg, 2012.

[41] M.K. Jassal. *The Effect of Optimization of Error Metrics*. Master's thesis, School of Business and Informatics, University of Boras, 2010.

[42] E.L. Kaltofen and A. Storjohann. *Complexity of Computational Problems in Exact Linear Algebra*. Springer, Berlin, Heidelberg, pages 227-233, 2015.

[43] M.W. Khan and M. Alam. A survey of application: Genomics and genetic programming, a new frontier. *Genomics*, 100(2):65–71, 2012.

[44] L. Kleinrock. *Queueing Systems*. Wiley Interscience, 1975.

[45] A.G. Konheim and M. Reiser. Finite capacity queuing systems with applications in computer modeling. *SIAM Journal on Computing*, 7(2):210–229, 1978.

[46] G.M. Koole. A simple proof of the optimality of a threshold policy in a two-server queueing system. *System & Control Letters*, 26(5):301–303, December 1995.

[47] G.M. Koole. On the power series algorithm. *Performance Evaluation of Parallel and Distributed Systems - Solution Methods*, CWI Tract 105 & 106, CWI, Amsterdam, 1994.

[48] J.R. Koza, D. Andre, F.H. Bennett, and M.A. Keane. *Genetic Programming III: Darwinian Invention & Problem Solving*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1999.

[49] J.H. Lambert. Observationes variae in mathesin puram. *Acta Helveticae physico-mathematico-anatomico-botanico-medica*, pages 128–168, 1758.

[50] G. Latouche and M.F. Neuts. Efficient algorithmic solutions to exponential tandem queues with blocking. *SIAM Journal on Algebraic Discrete Methods*, 1(1):93–106, 1980.

[51] G. Latouche and V. Ramaswami. A logarithmic reduction algorithm for quasi-birth-death processes. *Journal of Applied Probability*, 30(3):650–674, 1993.

[52] D.-S. Lee. A two-queue model with exhaustive and limited service disciplines. *Stochastic Models*, 12(2):285–305, 1996.

[53] L. Leskelä. Stabilization of an overloaded queueing network using measurement-based admission control. *Journal of Applied Probability*, 43(1):231–244, 2006.

[54] L. Leskelä and J. Resing. *A tandem queueing network with feedback admission control*. Springer, Berlin, Heidelberg, pages 129-137, 2007.

[55] Y. Levy. A class of scheduling policies for real-time processors with switching system applications. In *Proceedings of the Eleventh International Teletraffic Conference, Kyoto, Japan*, 1985.

[56] Y. Li, X. Cai, F. Tu, and X. Shao. Optimization of tandem queue systems with finite buffers. *Computers & Operations Research*, 31(6):963–984, 2004.

[57] Z.-Z. Lin, J.C. Bean, and C.C. White. A hybrid genetic/optimization algorithm for finite-horizon, partially observed Markov decision processes. *INFORMS Journal on Computing*, 16(1):27–38, 2004.

[58] C.-M. Liu and C.-L. Lin. An efficient two-phase approximation method for exponential tandem queueing systems with blocking. *Computers & Operations Research*, 22(7):745–762, 1995.

[59] R. Love. *Linux Kernel Development*, chapter The Page Cache and Page Writeback, pages 323–337. Developer's Library. Pearson Education, 2010.

[60] M.F. Neuts. *Matrix-Geometric Solutions in Stochastic Models: an Algorithmic Approach*. Johns Hopkins series in the mathematical sciences. Johns Hopkins University Press, Baltimore, 1981.

[61] V.F. Nicola and T.S. Zaburnenko. Efficient importance sampling heuristics for the simulation of population overflow in Jackson networks. *ACM Transactions on Modeling and Computer Simulation*, 17(2), April 2007.

[62] J.M. Norman. *Heuristic Procedures in Dynamic Programming*. Manchester University Press Manchester, 1972.

[63] M. Onderwater, S. Bhulai, and R.D. van der Mei. Value function discovery in Markov decision processes with evolutionary algorithms. *IEEE Transactions on Systems, Man, and Cybernetics*, 46(9):1190–1201, September 2016.

[64] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[65] E. Perel and U. Yechiali. Queues where customers of one queue act as servers of the other queue. *Queueing Systems*, 60:271–288, 2008.

[66] E. Perel and U. Yechiali. On customers acting as servers. *Asia-Pacific Journal of Operational Research*, 30:1–23, 2013.

[67] E. Perel and U. Yechiali. Finite two layered queueing systems. *Probability in the Engineering and Informational Sciences*, 30(3):492–513, 2016.

[68] W.B. Powell and P.A. Humblet. *The Bulk Service Queue with a General Control Strategy: Theoretical Analysis and a New Computational Procedure*. LIDS-P. Laboratory for Information and Decision Systems, Massachusetts Institute of Technology, 1985.

[69] M.L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. New York, NY, USA, John Wiley & Sons, 1994.

[70] D. Qin, A.D. Brown, and A. Goel. Reliable writeback for client-side flash caches. In *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA*, pages 451–462, 2014.

[71] M. Reiser and S.S. Lavenberg. Mean-value analysis of closed multichain queuing networks. *Journal of the ACM*, 27:313–322, 1980.

[72] J.A.C. Resing. Polling systems and multitype branching processes. *Queueing Systems*, 4:13:409–426, 1993.

[73] T.G. Robertazzi. *Computer Networks and Systems: Queueing Theory and Performance Evaluation*. Springer-Verlag New York, 1990.

[74] J.A. Rolia and K.C. Sevcik. The method of layers. *IEEE Transactions on Software Engineering*, 21(8):689–700, 1995.

[75] Z. Rosberg, P. Varaiya, and J. Walrand. Optimal control of service in tandem queues. *IEEE Transactions on Automatic Control*, 27(3):600–610, June 1982.

[76] C. Roser, M. Nakano, and M. Tanaka. A practical bottleneck detection method. In *Proceedings of the 33rd Winter Simulation Conference (WSC '01)*, pages 949–953. IEEE Computer Society, 2001.

[77] D. Roubos and S. Bhulai. Approximate dynamic programming techniques for the control of time-varying queuing systems applied to call centers with abandonments and retrials. *Probability in Engineering and Information Sciences*, 24(1):27–45, January 2010.

[78] P.J. Schweitzer. Approximate analysis of multiclass closed networks of queues. In *Proceedings of the International Conference on Stochastic Control and Optimization*, pages 25–29, 1979.

[79] C. Shousha, D.C. Petriu, A. Jalnapurkar, and K. Ngo. Applying performance modelling to a telecommunication system. In *Proceedings of the 1st International Workshop of Software and Performance*, pages 1–6, 1998.

[80] T. Stephens. Gplearn version 0.2.0. `https://gplearn.readthedocs.io/en/stable/`, 2016.

[81] R.S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3(1):9–44, August 1988.

[82] R.S. Sutton and A.G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998.

[83] L. Tadj and G. Choudhury. Optimal design and control of queues. *Top*, 13(2):359–412, 2005.

[84] H. Tijms. *A First Course in Stochastic Models*. Wiley, 2003.

[85] P. Tregunno. *Practical Analysis of Software Bottlenecks*. Master's thesis, Department of Systems and Computer Engineering, Carleton University, 2003.

[86] M. Tribastone. Relating layered queueing networks and process algebra models. In *Proceedings of the 1st Joint WOSP/SIPEW International Conference on Performance Engineering*, pages 183–194, 2010.

[87] M. Tribastone. A fluid model for layered queueing networks. *IEEE Transactions on Software Engineering*, 39(6):744–756, 2013.

[88] G.D. Tsiotras and H. Badr. A recursive methodology for the derivation of the blocking probabilities of tandem queues with finite capacity. *Computers & Operations Research*, 17(5):475–479, 1990.

[89] R.D. van der Mei and E.M.M. Winands. Polling models with renewal arrivals: a new method to derive heavy-traffic asymptotics. *Performance Evaluation*, 64(9-12):1029–1040, October 2007.

[90] N.D. van Foreest, J.C.W. van Ommeren, M.R.H. Mandjes, and W.R.W. Scheinhardt. A tandem queue with server slow-down and blocking. *Stochastic Models*, 21(2-3):695–724, 2005.

[91] P. Vis. *Performance Analysis of Multi-Class Queueing Models*. PhD thesis, Vrije Universiteit Amsterdam, The Netherlands, 2017.

[92] V. Vishnevsky and O. Semenova. Mathematical methods to study the polling systems. *Automation and Remote Control*, 67:173–220, February 2006.

[93] L. Vuong. Extreme transaction processing patterns: Write-behind caching. `http://www.infoq.com/articles/write-behind-caching`, 2009.

[94] E.M.M. Winands, I.J.B.F. Adan, G.J. van Houtum, and D.G. Down. A state-dependent polling model with k-limited service. *Probability in the Engineering and Informational Sciences*, 2:23:385–408, 2009.

[95] C.M. Woodside, J.E. Neilson, D.C. Petriu, and S. Majumdar. The stochastic rendezvous network model for performance of synchronous client-server-like distributed software. *IEEE Transactions on Computers*, 44:20–34, 1995.

[96] A. Yener and C. Rose. Genetic algorithms applied to cellular call admission: local policies. *IEEE Transactions on Vehicular Technology*, 46(1):72–79, February 1997.

[97] G. Yom-Tov and A. Mandelbaum. Queues in hospitals: Semi-open queueing networks in the QED regime. Technical report, Technion, Israeli Institute of Technology, 2008.

[98] B. Zhang and H. Ayhan. Optimal admission control for tandem queues with loss. *IEEE Transactions on Automatic Control*, 58(1):163–167, January 2013.

Summary

The increasing complexity of IT infrastructures poses significant challenges in managing computer systems. Ensuring efficient usage of the available resources while preserving the desired Quality of Service (QoS) demands one to go beyond ad-hoc solutions. Motivated by this, we develop tools and methods to evaluate the performance of ICT service chains, and furthermore, to manage their control in an optimal manner. One of the main goals of our research is to bridge the gap between theory and practice. Therefore, we present possible IT applications as a context for each of the studied techniques. We explore a new solution concept by combining the fields of queueing theory and machine learning. We believe that the introduced techniques have a great potential offering remarkably accurate, and at the same time easily scalable, generic solutions to real-world problems.

Motivated by caching in database application, in Chapter 2 we analyze a tandem queueing model consisting of two servers where the output of one of the servers becomes the input of the second node. More specifically, we model the application level sending write requests to the cache as the first server and the cache as the second one. Furthermore, we introduce the possibility to switch on/off any of the two servers at any given moment. Next to that, we assume holding costs at each of the two queues and a start-up cost for the second server, i.e., the cache. This way, we formulate a Markov Decision Process (MDP) with an optimal control policy defined as the one minimizing the long-term average costs of the system. However, the existing numerical techniques to compute this optimal policy are inefficient or even unfeasible for some real-world applications. Therefore, we present a scalable approximation algorithm to obtain a threshold-type decision policy. The researched technique is rather intuitive and the approximated policy results in long-term average costs within a few percentages of the optimal. Therefore, we believe that our method contributes to expanding both the theoretical and the practical knowledge on the matter.

Chapter 3 further explores the performance evaluation and optimal control

of database caching mechanisms. In contrast to the preceding chapter, this chapter focuses on the so-called 'write-behind' cache mechanism. We model the system as a single server that processes jobs in two stages. The first stage corresponds to pre-processing jobs one at a time, i.e., requests being written in the cache. After pre-processing the jobs are accumulated in a batch of given size K and served at once, i.e., the requests are being transferred from the cache to the database. The goal of the study is to identify the batch size K that minimizes the expected waiting time of arriving jobs. To approximate the optimal value of K, we first derive an analytic solution for the stationary distribution of jobs in the system for two special cases: $K = 1$ and $K = 2$. Next, we outline a fluid approach which results in an approximation of the expected waiting time for large batch sizes. Finally, we show how to combine the insights from these techniques to find the optimal batch size in an efficient and scalable way. Extensive numerical experimentation shows that the approximation works extremely well for a wide range of parameter combinations.

In Chapter 4 we introduce a new technique to obtain closed-form approximations of the optimal threshold-based policy for MDPs. Our method uses the Symbolic Regression (SR) algorithm. We present how to significantly improve both the accuracy and the execution time of this evolutionary algorithm by tailoring it to the corresponding MDP framework. Applying our approach on two running examples results in analytic expressions that approximate the optimal control policy with great accuracy. Next to that, we show that the obtained mathematical formulas allow sensitivity analysis of the system parameters. In addition, the opportunity to instantly calculate a new threshold function for any change in the parameters makes our solution particularly appealing for an application in real-time systems. Furthermore, we believe that the introduced technique is highly generic and is applicable to a broad range of other MDPs.

In Chapter 5 we further research the potential of combining queueing theory together with tools from machine learning. We introduce a way to incorporate insights and results derived from queueing theory techniques into SR. The presented method leads to closed-form approximations for the relevant performance metrics of a given system. To illustrate the technique, we apply it to the so-called two-stream blending system. We use our method to obtain closed-form expressions for the mean waiting time and the throughput. The generated analytic formulas are remarkably accurate and at the same time algebraically simple.

Finally, we abstract over the models considered to this point and adopt a high-level view of the ICT service chains. In Chapter 6, we analyze a Layered Queueing Network (LQN) in which a given number of servers are organized in a nested fashion, and therefore influence each others service rate. We present a simple, computationally tractable and nevertheless highly accurate approximation algorithm for obtaining the servers' utilization rates and throughput of the given nested system. We believe that the high accuracy of the approximation opens up opportunities to extend the model to more general LQNs. Furthermore, we show that even when performance metrics such as the saturation throughput and the utilization rates of the servers are known, determining the limiting factor in the network is far from trivial. Therefore, we introduced a server-wise metric for identification of the bottleneck that extends the intuitive 'slowest server rule'. The conducted numerical tests show that the proposed bottleneck identification technique is more accurate than the existing algorithms so far. Next to that, being an extension of the 'slowest server rule', our method is applicable to queueing networks without a layered structure as well.

Samenvatting

De toenemende complexiteit van IT infrastructuren vormt een grote uitdaging voor de beheersbaarheid van computersystemen. Hierbij is het van groot belang de beschikbare bronnen efficiënt in te zetten onder voorwaarde dat de gewenste Quality of Service (QoS) wordt behaald. Om deze complexiteit en schaal het hoofd te kunnen bieden moet er verder worden gedacht dan voor de hand liggende ad-hoc oplossingen. Dit heeft ons geïnspireerd om prestatiemodellen en algoritmen te ontwikkelen die het gedrag van ICT-serviceketens beschrijven. De modellen en algoritmen kunnen bovendien gebruikt worden voor het bepalen van besturingsregels voor deze ICT-systemen. Een van de hoofddoelen van ons onderzoek is het overbruggen van de kloof tussen theorie en praktijk. Om dit te bewerkstelligen plaatsen we elk van de bestudeerde modellen en algoritmen in de context van een beoogde ICT-toepassing. We introduceren een nieuwe oplossingsmethode waarin technieken uit de wachtrijanalyse en machine learning vakgebieden worden gecombineerd. In deze methode passen we machine learning toe op concepten uit de wachtrijanalyse. De resulterende modellen zijn in staat om variëteit van relevante externe invloeden mee te nemen. Dit maakt de ontwikkeling mogelijk van accurate en schaalbare modellen en algoritmen met een bredere toepasbaarheid.

Het model in hoofdstuk 1 is geïnspireerd door de interactie tussen database toepassingen en de onderliggende opslag. Deze interactie vind plaats doormiddel van een cache die schrijfverzoeken tijdelijk opslaat in werkgeheugen. Ons model kan worden gezien als een tandemmodel waarbij de uitvoer van de eerste server genereert de invoer van de tweede server. De eerste server representeert het verzenden van schrijfverzoeken vanuit applicatieniveau naar de cache. De tweede server modelleert het schrijven van de cache naar de onderliggende opslag. Elk van de twee servers kan op elk gewenst moment worden in- of uitgeschakeld. Daarnaast veronderstellen we wachtkosten bij elk van de twee wachtrijen en opstartkosten voor de tweede server, d.w.z. de cache. We construeren een Markov Decision Process (MDP) om de optimale strategie te bepalen die de gemiddelde langetermijnkosten van het systeem minimaliseert. De bestaande numerieke technieken om de bedieningsstrategie te optimalis-

eren zijn inefficiënt en zelfs onhaalbaar voor bepaalde systemen in de realiteit. Daarom presenteren we een schaalbaare benadering om de dempelwaardestrategie te bepalen voor het in en uitschakelen van de servers. De onderzochte techniek is intuïtief en resulterende strategie zorgt dat de gemiddelde kosten over de lange termijn binnen enkele procenten van optimale strategie liggen.

Hoofdstuk 3 beschouwt de prestatiemodellen en bijhorende optimalisering van database caching-mechanismen vanuit een ander perspectief. In tegenstelling tot het vorige hoofdstuk, richt dit hoofdstuk zich op het zogenaamde 'write-behind' cachemechanisme. We modelleren het systeem doormiddel van een Markovmodel waarin een enkele server de betreffende taken in twee stappen verwerkt. De eerste stap correspondeert met plaatsen en voorbereiden van de binnenkomende taken in de cache. Na de voorverwerking worden de taken verzameld in een batch ter grootte K die in ëën keer wordt verwerkt als een grote schrijfopdracht. Het doel is om de grootte van de batch te bepalen die de verwachte wachttijd voor aankomende opdrachten minimaliseert. Om de optimale waarde van K te schatten, leiden we eerst een analytische oplossing af voor de stationaire verdeling van taken in het systeem voor twee speciale gevallen: $K = 1$ en $K = 2$. Vervolgens leiden we een vloeistofmodel af om de verwachte wachttijd van grote batches te benaderen. Ten slotte combineren we deze technieken tot een uitdrukking waarmee de optimale batchgrootte op een efficiënte en schaalbare manier kan worden bepaald. Uitgebreide numerieke experimenten tonen aan dat de benadering heel goed werkt voor een veelvoud van parametercombinaties.

In hoofdstuk 4 introduceren we een nieuwe techniek om expliciete benaderingen te bepalen voor drempelwaardestrategiën in MDP's. Onze methode maakt gebruik van het Symbolic Regression (SR) -algoritme. We zijn in staat zowel de nauwkeurigheid als de rekentijd van dit algoritme aanzienlijk verbeteren door gebruik te maken van domeinspecifieke kennis uit het MDP-raamwerk. We passen onze aanpak toe op twee voorbeelden. Voor deze voorbeelden hebben we analytische uitdrukkingen afgeleid die de optimale strategie met grote nauwkeurigheid benaderen. De gesloten vorm van deze uitdrukkingen stelt ons bovendien in staat een gevoeligheidsanalyse uit te voeren op de systeemparameters. De gevonden uitdrukkingen maken het mogelijk om drempelwaardestrategiën direct aan te passen aan veranderingen in de systeemparameters. Dit maakt onze oplossing bijzonder aantrekkelijk voor een toepassing in real-time systemen. We zijn er van overtuigd dat onze aanpak zeer generiek is en van toepassing is op een breed spectrum van MDP formuleringen.

In hoofdstuk 5 breiden we ons onderzoek uit naar de combinatie van wachtrijtheorie met technieken uit de machine learning. We introduceren een manier om de technieken en modelcomponenten uit wachtrijtheorie te vertalen naar SR. De gepresenteerde methode kan gesloten vorm benaderingen afleiden voor de belangrijkste prestatiematen van wachtrijmodellen. Ter demonstratie passen we onze aanpak toe op het zogenaamde 'two-stream blending' systeem. Ook voor dit model hebben we uitdrukkingen in gesloten vorm afgeleid voor de gemiddelde wachttijd en de maximale doorvoersnelheid. De gegenereerde analytische uitdrukkingen zijn nauwkeurig en tegelijkertijd algebraïsch eenvoudig.

Tenslotte beschouwen we een model dat op een hoger abstractieniveau ligt dan de modellen in voorgaande hoofdstukken. In hoofdstuk 6 analyseren we een Layered Queuing Network (LQN) waarin servers een gelaagde keten vormen en daarom elkaars doorvoersnelheid beïnvloeden. We presenteren een eenvoudig benaderingsalgoritme voor het bepalen van de bezettingsgraad en doorvoersnelheid van de gelaagde servers. Wij geloven dat de hoge nauwkeurigheid van de benadering potentie biedt voor uitbreiding naar generiekere gelaagde wachtrijmodellen. Verder laten we zien dat zelfs wanneer prestatiematen zoals de maximale doorvoersnelheid en de bezettingsgraad van de servers bekend zijn, het bepalen van de beperkende factor in het netwerk niet triviaal is. Daarom introduceren we een prestatieindex die een uitbreiding is van de 'slowest server rule'. Uit de uitgevoerde numerieke tests blijkt dat de voorgestelde bottleneckidentificatie techniek nauwkeuriger is dan de bestaande algoritmen. Daarnaast is onze methode ook toepasbaar op klassieke wachtrijnetwerken zonder gelaagde structuur.