Imperial College
London

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Analyzing Closed Multiclass Queueing Networks with Fork-Join Constructs

Author:
Joel Choo

Supervisor:
Dr Giuliano Casale

Submitted in partial fulfillment of the requirements for the MRes degree in Advanced Computing of Imperial College London

September 2016

Abstract

Multiclass closed queueing networks with fork-join constructs are a very useful way to model the performance of systems that use parallelization. This paper introduces a method to approximate the queue lengths and throughputs of such queueing networks. This method can be considered an extension of an existing approximation method (Decay Rate Approximation) for queueing networks without fork-join constructs. The main idea is to consider each queue in isolation by approximating the arrival process for each queue, and then calculate the decay rates for each queue and use that along with a product form solver to get a more accurate approximation. We also present a way to increase the computational efficiency of our approximation method and present results that suggest that this helps to make the runtime scale better.

Our results show that our approximations are equal or better than the current state of the art in a wide range of test cases varying the number of fork-join queues, the heterogeneity of the queues, and the service distributions of the queues. Our method has a mean queue length percentage error of less than 10% in all our tests.

Acknowledgments

I would like to thank my supervisor Dr Giuliano Casale for his guidance and support through this project.

I would also like to thank my loved ones for all the support and love they have showered on me.

I couldn't have done it without all of you.

Thanks from the bottom of my heart.

> If I have seen further it is by standing on the sholders of giants.
>
> *Isaac Newton*

Contents

1 Introduction **1**
- 1.1 Motivation . . . 2
- 1.2 Objectives . . . 3
- 1.3 Challenges . . . 3
- 1.4 Contributions . . . 4
- 1.5 Overview . . . 4

2 Background **7**
- 2.1 Organization . . . 7
- 2.2 Fork-Join Queues . . . 7
  - 2.2.1 Split-Merge Model . . . 8
  - 2.2.2 Solving and Approximating Fork-Join Queues . . . 8
- 2.3 Closed Queueing Networks . . . 9
- 2.4 Phase Type Distribution . . . 10
- 2.5 Markovian Arrival Process . . . 11
  - 2.5.1 Examples . . . 12
- 2.6 Matrix Geometric Method . . . 14
- 2.7 Modelling Departure Process of MAP/MAP/1 Queues . . . 15
  - 2.7.1 Level Probability Based Truncation Method . . . 17
- 2.8 Approximate Model for Single Class Fork-Join Queues . . . 17
  - 2.8.1 Size of D_0 and D_1 . . . 19
- 2.9 Marked Markovian Arrival Processes . . . 19
- 2.10 MMAP[K]/PH[K]/1 Queues . . . 20
- 2.11 Product Form Queueing Networks . . . 20

3 Decay Rate Approximation **23**
- 3.1 Single Class Approximation . . . 23
- 3.2 Notation . . . 24
- 3.3 Methodology . . . 24
  - 3.3.1 Initialization . . . 25
  - 3.3.2 Obtaining Individual Queues . . . 26
  - 3.3.3 Calculating Decay Rates . . . 26
  - 3.3.4 Calculating Queue Length . . . 26
  - 3.3.5 Optimization . . . 27
- 3.4 Performance . . . 27

4 Fork-Join Approximate Mean Value Analysis 29
4.1 Single Class . . . 29
4.2 Multiclass . . . 30
4.2.1 Complexity . . . 31
4.3 Performance . . . 31

5 Fork-Join DRA 33
5.1 Initialization . . . 33
5.1.1 AMVA-FCFS . . . 34
5.1.2 FJ-AMVA and Other Approximations . . . 34
5.1.3 Comparison . . . 35
5.2 Arrival Distribution . . . 36
5.3 Calculating Queue Length . . . 37

6 Efficiency 39
6.1 Objectives . . . 39
6.2 Profiling . . . 39
6.2.1 Increasing Parallel Queues . . . 40
6.2.2 Increasing Synchronization Queue Length . . . 40
6.2.3 Observations . . . 41
6.3 Threshold Based Resizing of the MMAP . . . 41
6.3.1 First Order MMAP . . . 41
6.3.2 Second Order MMAP . . . 43
6.3.3 Summary . . . 44

7 Results 45
7.1 Experimental Set Up . . . 45
7.2 Number of Fork-Join Queues . . . 46
7.3 Heterogeneity Of Fork-Join Queues . . . 47
7.4 Other Service Distributions . . . 48
7.4.1 Erlang-2 . . . 48
7.4.2 Markov Modulated Poisson Process (MMPP) . . . 49
7.5 Summary . . . 50

8 Conclusion 53
8.1 Future Work . . . 53

Chapter 1

Introduction

Fork-join constructs in queueing networks are those where one job is *forked* into multiple independent jobs which are processed in parallel, and when all those jobs have completed processing they are all *joined* and depart as a single job [55]. A simple example of a fork-join construct can be seen in Figure 1.1 and we explain the model and its assumptions in more detail in Section 2.2. Fork-join constructs arise in contexts which use parallel and/or distributed computing; for example query processing in parallel databases, parallel disk access in RAID, and the MapReduce framework all can be modelled using queueing networks that consist of fork-join constructs. A fork-join construct can be used to model the situation when a job is split into multiple parts and has to wait for all parts to finish processing before the job can proceed to the next part of the system. A simple example of that would be sorting in parallel, where the initial list is split into multiple parts which can be sorted in parallel and subsequently merged. However, the merging process cannot begin until all parts have been sorted.

Multiclass queueing networks are a widely used framework for modelling various types of stochastic systems; ranging from telecommunication systems to production lines to computer hardware and software systems [14]. The difference between a multiclass queueing network and a single class queueing network is that the jobs in the multiclass queueing network can have different labels/classes. The jobs can have different service time distributions at the queues and can have different populations (for closed queueing networks) or different arrival time distributions (for open queueing networks). What this means simply is that the queueing network can have different types of jobs that can arrive and be serviced differently. This is useful to model situations where jobs can be categorized into distinct classes that have different service requirements. A real life example would be trying to model a customer service call centre, where most customers have a simple

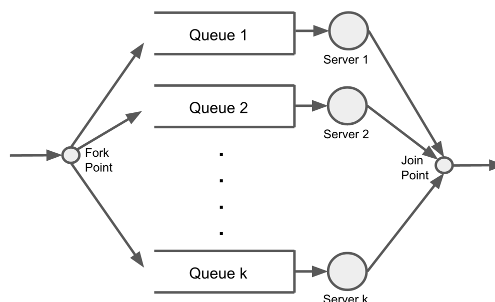

Figure 1.1: Simple Fork-Join Queue

question and can be serviced quickly, but a minority of customers have more complex problems or are difficult to deal with and take longer to service. In this example, we have two classes, with one that arrives very regularly and gets serviced quickly, and another class that arrives more irregularly and takes longer to service. Multiclass queueing networks are used to evaluate the performance of such systems and figure out how such systems can be optimized and scaled to larger networks and larger number of jobs.

Closed queueing networks are those where there are no external arrivals or departures of jobs, and instead jobs cycle around within the queueing network. There are a fixed number of jobs in the queueing network, and it is known that as the number of jobs grows, the closed queueing network can be used to accurately approximate a network with external arrivals and departures [50]. This means that the closed queueing network model can be used to model different types of networks as long as the runtime grows slowly as the population increases.

Despite their utility in modelling complex systems in the real world, closed multiclass queueing networks with fork-join constructs are difficult to perform exact analysis on. Presently, there only exists an exact solution to single class fork-join constructs with 2 queues with exponentially distributed service times [29]. For all other cases, there are only approximations. Multiclass queueing networks can be analyzed using mean-value analysis and product-form theory [51], but only for the situations where the model complies with the product-form assumptions. In the situation where the multiclass queueing network has queues that are first come first served, then the queueing network can only be analyzed when all the service times are exponentially distributed and where the mean service times for all classes is the same at each queue [17]. As such, most of the systems in the real world are optimized using these approximations instead of an exact analysis.

In this paper, we introduce a method to analyze closed multiclass queueing networks with fork-join constructs where the queues are first come first served and have phase type distributed service times and where service time moments can be chosen arbitrarily and differently from each class. This method is an extension to an existing method of analyzing multiclass queueing networks: Decay Rate Approximation (DRA) [22]. The DRA method is an approximation for closed multiclass queueing networks with the same constraints on the queues, but without the ability to model fork-join constructs.

1.1 Motivation

In the world we live in today, we rely on many important and highly complex systems. These systems range from manufacturing (e.g. production lines, supply flow) [19] to communication (e.g. internet, telecommunication networks) [31] to computer hardware and software systems. In particular, the use of parallel and distribued computing has become increasingly widespread in recent years and the trend looks to continue. For example, there are distributed databases/data storage like Google's Bigtable and Amazon's Dynamo, and also distributed computing frameworks for processing large amounts of data like MapReduce. These distributed systems involve large amounts of parallelization. In 2010, a single Google search query uses more than 1,000 computers to get the results for the user [26] and it is not unthinkable that the number of computers used per search query has increased since. In addition, consider the fact that Google processes at least 40,000 search queries a second (or 1.2 trillion search queries per year). The performance of such systems is very important to these technology companies. Amazon has calculated that a slow down in page

loads of one second could result in a reduction of US$1.6 billion in sales per year [25]. For Google, an extra half a second in responding to search queries resulted in a reduction in traffic by 20% [26], which can dramatically affect the revenue from ads. Models such as closed multiclass queueing networks with fork-join constructs are very useful to evaluate and optimize the performance of such systems.

In addition to the examples above, there are hardware systems that are designed for and make use of parallelization like graphics processing units (GPUs). Large technology companies like Google and Facebook have huge clusters of GPU-based systems forming complex systems with extremely large amounts of parallelization [27]. This is driven largely due to the recent successes and development of machine learning and deep learning, leading to the demand for machines that can scale to large data sets [12]. Designing these systems also requires a good knowledge and understanding of how jobs are processed in the system, and this can be gained by using models with fork-join constructs.

In general, processing jobs within these massively parallel and distributed systems can be modelled using a queueing network with fork-join constructs. In particular, the fork-join construct has been described as one of the key models for the performance analysis of parallel and distributed systems [18]. Therefore, being able to accurately and efficiently analyze queueing networks with fork-join constructs is getting increasingly important and allows us to be able to design, build and optimize such systems.

1.2 Objectives

The main aim of this project is to extend the decay rate approximation method [22] to queueing networks with fork-join constructs. We aim to use the research from [23] and improve on the approximation in [2] to get a more accurate approximation of the queueing network which is still efficient.

The eventual outcome of this project will be an extension of DRA including an implementation of that method in MATLAB. We aim to have an implementation that is:

- Accurate - Produces approximations that are more accurate than the state of the art method in [2]. We will test the approximation methods against values obtained from simulation and we aim to produce values that are closer to those obtained from the simulation.
- Universal - Works for a wide range of queueing networks. This includes different topologies and different service time distributions. Ideally, we want the method to produce accurate approximations for all queueing network topologies and for all types of service time distributions.
- Efficient - Ideally, the approximation should be efficient enough to work on large queueing networks with many parallel queues and large number of classes.

1.3 Challenges

The following challenges need to be addressed when developing the approximation method:

While our method can be seen as an extension of the Decay Rate Approximation (DRA) method, it is not as simple as writing a stand alone script that the existing DRA method can call when it needs to deal with fork-join queues. We need to thoroughly understand the DRA method and modify the parts where certain assumptions do not hold when fork-join queues are used.

Based on the research done on approximating single class fork-join queues, we know that as we increase the number of fork-join queues in parallel, there is an exponential growth in the number of states of the process that is used to model the fork-join queue [23]. As such, the run time and space required to run the approximation grows exponentially and does not scale to queueing networks with large fork-join constructs. We need to find a way to overcome this state space explosion in order to make the approximation more efficient and work for more topologies.

In addition, research suggests that the accuracy of the approximation increases as the length of the synchronization queue in the model increases [23]. However, this also causes an exponential increase in the number of states used to model the fork-join queue and therefore the time and space efficiency of the approximation is not ideal. We need to find a way to balance between increasing the length of the synchronization queue and increasing run time and space used.

1.4 Contributions

The contributions of this project are as follows:

- Developed an approximation method for multiclass queueing networks with fork-join constructs. This method is equal or better than the existing methods for a majority of our test cases, and has less than 10% mean queue length error for all our tests.
- Investigated methods to make the method more computationally efficient. By resizing the MMAPs to have fewer states, the runtime is reduced dramatically and scales better as we increase both the number of fork-join queues as well as the length of the synchronization queue.
- Implemented the approximation method in MATLAB.

1.5 Overview

There are two big building blocks to the project:

- Decay Rate Approximation (DRA)
- Fork-Join Approximate Mean Value Analysis (FJAMVA)

DRA is an approximation for multiclass closed queueing networks, but **without** fork-join constructs. We use the same high level ideas in DRA in our method, but modify parts of it to work when we consider fork-join constructs in the queueing network. We describe DRA in more detail in Chapter 3.

FJAMVA is an approximation for multiclass closed queueing networks **with** fork-join constructs. This is the exact same problem we are trying to solve. FJAMVA is the best known method for solving this, and we will be comparing our results against those obtained using FJAMVA. We describe FJAMVA in more detail in Chapter 4.

Chapter 2

Background

In this chapter, we introduce the background knowledge and research done on the various components of the paper.

2.1 Organization

This project is based on the following important concepts:

- Approximate model for single class fork-join queues in Section 2.8
- Decay Rate Approximation (DRA) in Chaper 3
- Approximate Mean Value Analysis (AMVA) of multiclass fork-join queueing networks in Chapter 4

This chapter is organized in a way that starts off with basic concepts, introducing the fork-join construct and the existing research on solving it. Next, we introduce the concepts required to understand the approximate model for single class fork-join queues in Section 2.8. Following that, we cover the concepts relating to multiclass queueing networks. The DRA and AMVA concepts are more complex and the project heavily relies on both, so they are in chapters 3 and 4 respectively.

2.2 Fork-Join Queues

At a high level, fork-join constructs are simply just a part of a queueing network where an incoming job arrives at a point where it gets split into multiple jobs which get processed in parallel. Then when all those jobs have completed processing, they are resynchronized and the job leaves the fork-join construct as a single job [55].

Figure 2.1 shows a fork-join construct in more detail. We define the terms fork point and join point as the points in the network where the job is forked into multiple jobs and joined back into a single job respectively. These points are labelled in Figure 2.1.

Note that the incoming job gets split into n independent jobs when it arrives at the fork point. These n independent jobs immediately join n separate and parallel first come first served queues where it waits to get serviced by n different servers. When one of the n independent jobs has been serviced,

it joins a synchronization queue and waits until all of the other jobs that split off at the fork point have completed servicing. Once the last job finishes servicing and joins the synchronization queue, the n jobs then resynchronize at the join point and get combined back into one job again.

Note that the n servers can have different service time distributions and do not have to be related to each other. Note that the n jobs do not necessarily start getting processed by the servers at the same time and can spend different amounts of time in the queues before getting serviced. In addition, the jobs are assumed to be processed in first come first served order.

In some cases, the fork-join model has interfering jobs that can join the n queues but in this paper we consider the model without the interfering jobs.

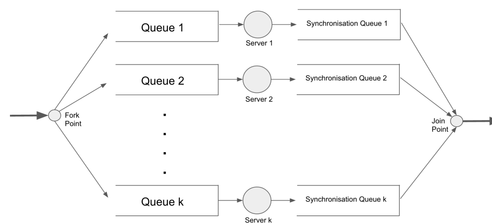

Figure 2.1: Detailed Fork-Join Queue

2.2.1 Split-Merge Model

A similar model to the fork-join construct is the split-merge model. The split-merge model is similar to the fork-join in that the incoming job is split into n sub-jobs and processed in parallel. However, the difference is that the next incoming job can only split and start getting processed when all the sub-jobs have finished servicing and are rejoined [28].

The split-merge model has been solved for the following homogeneous queues: M/G/1, M/G/1/N, M/G/C, and G/G/1/N [28]. For heterogeneous split-merge queues, the case where $n = 2$ and the servers have exponentially distributed service times has been solved [28].

The split-merge model can be seen to be an upper bound on the fork-join model, as the networks topologies and the way that jobs are split and synchronized are exactly the same, with the difference being that even though a server has completed servicing a sub-job, it cannot begin servicing the next sub-job (from the next job) until all of the other $n - 1$ sub-jobs have finished servicing and synchronized.

2.2.2 Solving and Approximating Fork-Join Queues

In this section, we explore the current research in solving fork-join constructs and focus in particular on how matrix geometric methods have been used to analyse them. We explore:

- n way fork-join queueing systems with Poisson arrivals and exponential service times
- n way fork-join queueing systems with Poisson arrivals and general service times

Poisson Arrivals and Exponential Service Times

In this situation, the arrivals are modelled using a Poisson process with rate λ and the service time distributions of all the queues are exponential with the same rate μ.

Let us define ρ as the utilization of the queues, $R(\rho) = \frac{1}{\mu-\lambda}$ as the mean response time for a single server queue, and $R_n^{FJ}(\rho)$ as the response time of a fork-join system with utilization ρ and n queues. Let $R_n^{max}(\rho)$ be the maximum of n response times for queues with utilization ρ.

An exact solution has been found for $n = 2$, and it is as follows[43]:

$$R_2^{FJ}(\rho) = \frac{12-\rho}{8} R(\rho) \tag{2.1}$$

For all $n > 2$, there is no exact solution and there are only approximations based on known upper and lower bounds. One key result is that $R_n^{\max}(\rho)$ is an upper bound for $R_n^{FJ}(\rho)$[43]. Using that result, an approximation was obtained for $R_n^{FJ}(\rho)$ for $2 < n \leq 32$ which is based on the bounds[43]:

$$\frac{H_n}{\mu} \leq R_n^{FJ}(\rho) \leq H_n \cdot R(\rho) \tag{2.2}$$

where H_n is the nth harmonic number, i.e. $H_n = \sum_{i=1}^{n} \frac{1}{i}$.

In addition to that, there are two other approximations obtained by Varki et al[57] and Varma and Makowski[58]. These approximations were compared in a simulation study[39] against an approximation using the maximum order statistic, which is just taking the maximum of all the n response times. The performance of the various approximations vary with different parameters such as having homogeneous/heterogeneous servers, n, and service distributions.

Maximum of Response Times

As mentioned earlier, an approximation exists using the expected value of the maximum of multiple response times. This is the same as $R_n^{max}(\rho)$ defined earlier. This corresponds exactly to the solution of the split-merge model.

Poisson Arrivals and General Service Times

There is an approximation for fork-join queues with general service times[56] but it requires some precomputation of a value by running some simulations. The value obtained by running the simulations depends on n and ρ. As such, the performance and usefulness of this approach depends greatly on being able to know and fix n and ρ beforehand.

2.3 Closed Queueing Networks

There are two types of queueing networks: open and closed. In this project, we are interested in studying closed queueing networks. In an open queueing network, jobs arrive from an external source and depart to an external destination. In contrast, in the closed queueing network, jobs do not arrive from an external source and do not depart to an external destination. Instead, jobs

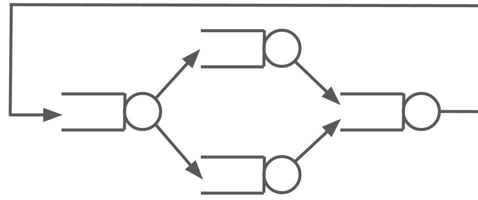

Figure 2.2: Simple Example of a Closed Queueing Network with a Fork-Join Construct

move around the queues in the system but never actually leave the system. The outgoing jobs are connected and redirected back to the input and become an arrival to the network [60]. This is illustrated in Figure 2.2. It is also common for closed queueing networks to have a think time, which is some time between cycling back from the end of the network to the front. This can be a constant amount of time, or can be modelled using any distribution like arrival or service times.

One key difference between open and closed queueing networks is that closed queueing networks have a fixed population of jobs. If the population is large enough, a closed network can accurately approximate an open network [50]. This means that the closed queueing network model can be used to model different types of networks as long as the runtime grows slowly as the population increases. In the multiclass case where jobs have different classes, there is a fixed initial population of jobs for each class. In some cases, the populations are fixed but others allow class switching, where a job can change classes. However, the total population still remains the same.

There are many computer systems that can be modelled with a closed queueing network. For example: users interacting with a system, threads acquiring a lock, processes blocking for I/O [24].

2.4 Phase Type Distribution

The phase type distribution is a probability distribution that generalizes the exponential distribution [44], both in series and in parallel [32]. Consider a continuous time Markov chain with $m+1$ states, where state 0 is absorbing and all the other m states are transient. A phase type distribution is a probability distribution of the time until absorption into the absorbing state (state 0) of such a Markov chain [45].

The parameters of a phase type distribution are:

- α, a probability row vector of length m
- S, an $m \times m$ matrix which is the transition rate matrix for the Markov chain excluding state 0

The overall transition rate matrix can then be written as:

$$Q = \begin{bmatrix} 0 & \mathbf{0} \\ -S \cdot \mathbf{1} & S \end{bmatrix}$$

where **0** is a $1 \times m$ vector of zeroes and **1** is a $m \times 1$ vector of ones.

For a random variable X that has a phase type distribution, the moments of X can be computed as follows [44]:

$$E[X^n] = (-1)^n \cdot n! \cdot \alpha \cdot S^{-n} \cdot \mathbf{1}$$

Here are some special examples of the phase type distribution:

An **exponential** distribution with rate μ has:

$$\alpha = [1],\ S = [-\mu]$$

An **Erlang-2** distribution with rate μ has:

$$\alpha = \begin{bmatrix} 1 & 0 \end{bmatrix},\ S = \begin{bmatrix} -\mu & \mu \\ 0 & -\mu \end{bmatrix}$$

A **hypo-exponential** distribution with two phases with transition rates μ_1 and μ_2 going from phase 1 to 2 and 2 to 0 respectively has:

$$\alpha = \begin{bmatrix} 1 & 0 \end{bmatrix},\ S = \begin{bmatrix} -\mu_1 & \mu_1 \\ 0 & -\mu_2 \end{bmatrix}$$

A **hyper-exponential** distribution with two phases where the probability of selecting phase 1 is p and where the rate for phases 1 and 2 are μ_1 and μ_2 respectively has:

$$\alpha = \begin{bmatrix} p & 1-p \end{bmatrix},\ S = \begin{bmatrix} -\mu_1 & 0 \\ 0 & -\mu_2 \end{bmatrix}$$

It is known that any distribution can be arbitrarily well approximated by a phase type distribution [3], and there are well studied methods of fitting a phase type distribution to data such as the expectation maximization algorithm [6] and the method of moments [3].

2.5 Markovian Arrival Process

The Markovian arrival process (MAP) is a model for events occuring in a system. It can be seen as an extension of the Poisson process, where the time between events is exponentially distributed. Consider the continuous time Markov chain representing a Poisson process. The Markov chain has only one state and one transition from the state to itself, where the transition corresponds an observable event lke an arrival (explained in more detail in Section 2.5.1).

The MAP extends this such that instead of just having one state, any number of states is allowed. The transitions between those states are Poisson processes and each transition can have a different rate from the others. Also, instead of having every transition correspond to an observable event occuring, the MAP transitions in the Markov chain can be hidden. This means that the transitions do not need to result in any observable event occuring and only mean that internally, the state of the Markov chain has changed.

The infinitesimal generator for the Markov chain for MAPs, D, can then be represented as the sum, $D = D_0 + D_1$, where D_1 is for the transitions corresponding to an event and D_0 is for the hidden

transitions.

One property of MAPs is that in theory any stationary point process can be approximated arbitrarily closely to a MAP [5].

The average rate of events in a MAP, λ, is called the fundamental rate of the MAP and is equals to $\lambda = \theta D_1 \mathbf{1}$, where θ is the stationary probability vector of the Markov chain with infinitesimal generator D [46].

2.5.1 Examples

Other than the Poisson process, other processes that can be represented as a MAP include Markov-modulated Poisson processes and phase-type renewal processes which include the Erlang distribution, hyperexponential distribution, hypoexponential distribution, and Coxian distribution. Here are some examples of D_0 and D_1 matrices for some of those distributions:

Poisson

The Markov chain for the Poisson process has only one state with one observable transition and no hidden transitions. This is shown in Figure 2.3.

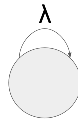

Figure 2.3: Markov chain for Poisson process in Section 2.5.1

In this case, the D_0 and D_1 matrices will be:

$$D_0 = \begin{bmatrix} -\lambda \end{bmatrix}$$
$$D_1 = \begin{bmatrix} \lambda \end{bmatrix}$$

Erlang-2

The Markov chain for the Erlang-2 process has two states with one observable transition and one hidden transition. This is shown in Figure 2.4.

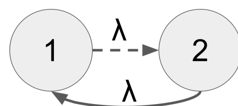

Figure 2.4: Markov chain for Erlang-2 process in Section 2.5.1

In this case, the D_0 and D_1 matrices will be:

$$D_0 = \begin{bmatrix} -\lambda & \lambda \\ 0 & -\lambda \end{bmatrix}$$
$$D_1 = \begin{bmatrix} 0 & 0 \\ \lambda & 0 \end{bmatrix}$$

Markov Modulated Poisson Process

This process is the case where we switch between two Poisson processes and where the process of switching between the Poisson processes is also a Poisson process. So this is modelled using a Markov chain with two states, two observable transitions and two hidden transitions. This is shown in Figure 2.5.

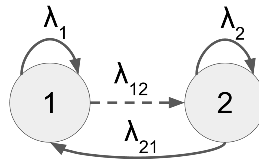

Figure 2.5: Markov chain for MMPP in Section 2.5.1

In this case, the D_0 and D_1 matrices will be:

$$D_0 = \begin{bmatrix} -(\lambda_1 + \lambda_{12}) & \lambda_{12} \\ \lambda_{21} & -(\lambda_2 + \lambda_{21}) \end{bmatrix}$$
$$D_1 = \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix}$$

Generic Example

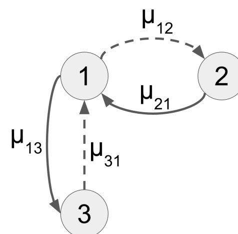

Figure 2.6: Markov chain for example MAP in Section 2.5.1

Consider the example MAP in Figure 2.6. Let the transitions rate from state x to state y be μ_{xy}. The dotted lines in the figure represent hidden transitions and the solid lines represent observable transitions. Then the MAP can be represented by the following matrices:

$$D_0 = \begin{bmatrix} -(\mu_{12}+\mu_{13}) & \mu_{12} & 0 \\ 0 & -\mu_{21} & 0 \\ \mu_{31} & 0 & -\mu_{31} \end{bmatrix}$$
$$D_1 = \begin{bmatrix} 0 & 0 & \mu_{13} \\ \mu_{21} & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

2.6 Matrix Geometric Method

The matrix geometric method is a technique used to solve queueing system problems that have a special structure. The transition rate matrix for the queueing system needs to have a tridiagonal block structure like this:

$$Q = \begin{pmatrix} B_{00} & B_{01} & & & & & \\ B_{10} & A_1 & A_2 & & & & \\ & A_0 & A_1 & A_2 & & & \\ & & A_0 & A_1 & A_2 & & \\ & & & A_0 & A_1 & A_2 & \\ & & & & A_0 & A_1 & A_2 \\ & & & & & \ddots & \ddots & \ddots \end{pmatrix} \tag{2.3}$$

where $B_{00}, B_{01}, B_{10}, A_0, A_1, A_2$ are all matrices.

Processes that can be described with such a transition rate matrix are called quasi-birth-death (QBD) processes. Let the states of the process be grouped together by blocks, where the ith block consists of all the states that correspond to the ith row of blocks in the matrix above.

Note that for these QBD processes, transitions can occur:

- from states in the ith block to states in the ith block
- from states in the ith block to states in the $i-1^{\text{th}}$ block
- from states in the ith block to states in the $i+1^{\text{th}}$ block

And that those transitions from the ith block to the:

- i^{th} block are represented by the matrices B_{00} and A_1
- $i-1^{\text{th}}$ block are represented by the matrices B_{10} and A_0
- $i+1^{\text{th}}$ block are represented by the matrices B_{01} and A_2

In order to calculate the stationary distribution π, we need to define π_i as the subvector corresponding to the states for block i. This means that the size of the vector π_i equals to the number of states in block i.

Then note that in a similar way that the M/M/1 queue can be solved, we first obtain the balance equations for the subvectors which are as follows:

$$\pi_0 B_{00} + \pi_1 B_{10} = 0 \tag{2.4}$$
$$\pi_0 B_{01} + \pi_1 A_1 + \pi_2 A_0 = 0 \tag{2.5}$$
$$\pi_1 A_2 + \pi_2 A_1 + \pi_3 A_0 = 0 \tag{2.6}$$
$$\vdots$$
$$\pi_{i-1} A_2 + \pi_i A_1 + \pi_{i+1} A_0 = 0 \tag{2.7}$$
$$\vdots$$

Then similar to how the M/M/1 queue can be solved, there exists a matrix R such that for $i \geq 1$:

$$\pi_i = \pi_{i-1} R \tag{2.8}$$

Then, clearly it follows that:

$$\pi_i = \pi_0 R^i \tag{2.9}$$

Then substituting that into Equation 2.7 we get that:

$$\pi_0 R^{i-1} A_2 + \pi_0 R^i A_1 + \pi_0 R^{i+1} A_0 = 0 \tag{2.10}$$
$$\pi_0 R^{i-1}(A_2 + RA_1 + R^2 A_0) = 0 \tag{2.11}$$
$$A_2 + RA_1 + R^2 A_0 = 0 \tag{2.12}$$

Then from Equation 2.12, we can solve for R. There are a few efficient numerical algorithms that can solve for R, such as cyclic reduction [15] and logarithmic reduction [37] [49].

Once we have obtained R, we can use R and substitute Equation 2.9 into Equation 2.4 to solve for π_0. Once we have solved for π_0, obtaining any other π_i can be done by using Equation 2.9.

2.7 Modelling Departure Process of MAP/MAP/1 Queues

In this section, we consider how we can model the departure process of a MAP/MAP/1 queue as a MAP. A MAP/MAP/1 queue is a queue that has both arrival and service times modelled using MAPs. Then we know that the generator of the continuous time Markov chain (CTMC) that models the queue has the following structure [36]:

$$Q = \begin{pmatrix} \bar{A}_0 & A_1 & & & \\ A_{-1} & A_0 & A_1 & & \\ & A_{-1} & A_0 & A_1 & \\ & & \ddots & \ddots & \ddots \end{pmatrix} \tag{2.13}$$

where the arrival MAP is characterized by B_0, B_1 and the service MAP is characterized by S_0, S_1 and:

$$A_1 = B_1 \otimes I \tag{2.14}$$
$$A_0 = B_0 \oplus S_0 \tag{2.15}$$
$$A_{-1} = I \otimes S_1 \tag{2.16}$$
$$\bar{A}_0 = B_0 \otimes I \tag{2.17}$$

Note that Q has the structure of a QBD as described in Section 2.6 and therefore we can use matrix geometric methods to solve for the stationary distribution of Q.

The departure process from the MAP/MAP/1 queue can be represented by a MAP with infinitely many phases [36]. Note that Q is the background Markov chain for the departure process MAP. Then note that the transitions corresponding to the departure of a job are non-hidden and are part of the D_1 matrix, while all other transitions are hidden transitions and are part of the D_0 matrix. Therefore, that the departure MAP can then be represented using the two matrices:

$$D_0 = \begin{pmatrix} \bar{A}_0 & A_1 & & & \\ & A_0 & A_1 & & \\ & & A_0 & A_1 & \\ & & & \ddots & \ddots \end{pmatrix} \tag{2.18}$$

$$D_1 = \begin{pmatrix} A_{-1} & & \\ & A_{-1} & \\ & & \ddots \end{pmatrix} \tag{2.19}$$

One way that can be used to approximate the infinite MAP is to truncate the infinite matrix after a certain point. A truncation at level n means that the matrices are truncated after n repeats of the diagonal part containing A_{-1}, A_0, A_1. There are multiple truncation methods such as ETAQA[34] and level probability based truncation method[52] but in both of those methods, the result after truncation is that we obtain matrices $D_0^{(n)}$ and $D_1^{(n)}$ with the following structure [36]:

$$D_0^{(n)} = \begin{pmatrix} \bar{A}_0 & A_1 & & & & & \\ & A_0 & A_1 & & & & \\ & & A_0 & A_1 & & & \\ & & & \ddots & \ddots & & \\ & & & & \ddots & \ddots & \\ & & & & & A_0 & A_1 \\ & & & & & & \hat{A}_0 \end{pmatrix} \tag{2.20}$$

$$D_1^{(n)} = \begin{pmatrix} A_{-1} & & & & & \\ & A_{-1} & & & & \\ & & \ddots & & & \\ & & & \ddots & & \\ & & & & \hat{A}_{-1} & \check{A}_{-1} \end{pmatrix} \tag{2.21}$$

with the only difference between the truncation methods being the definition of the block matrices $\hat{A}_{-1}, \check{A}_{-1}$. The basic idea behind the truncation methods is that all the levels greater than n will be merged into the last level.

2.7.1 Level Probability Based Truncation Method

In the code written for this paper, the level probability based truncation method was used. Recall the definition for π_i from Section 2.6. The level probability based method approximates the probability that the actual model is at level n when the truncated model is at level n using the vector π_n. In addition, the probability that the actual model is at level greater than n when the truncated model is at level n is approximated using the sum of the π_i of all $i > n$, i.e.

$$\pi_n^+ = \sum_{i=n+1}^{\infty} \pi_i \tag{2.22}$$

Given the geometric relation of all the π_i as seen in Equation 2.9, that sum can easily be computed as follows:

$$\begin{aligned} \pi_n^+ &= \sum_{i=n+1}^{\infty} \pi_i \\ &= \pi_0 R^{n+1} (I - R)^{-1} \end{aligned} \tag{2.23}$$

And so using this notation, the matrix blocks $\hat{A}_0, \hat{A}_{-1}, \check{A}_{-1}$ can be defined as follows:

$$\hat{A}_0 = A_0 + A_1 \tag{2.24}$$

$$\hat{A}_{-1} = \text{diag}(\pi_n) \cdot \text{diag}(\pi_n + \pi_n^+) \cdot A_{-1} \tag{2.25}$$

$$\check{A}_{-1} = \text{diag}(\pi_n^+) \cdot \text{diag}(\pi_n + \pi_n^+) \cdot A_{-1} \tag{2.26}$$

where diag(v) is defined as the diagonal matrix consisting of the elements in vector v.

2.8 Approximate Model for Single Class Fork-Join Queues

This section is based on the research done in [23]. This is another approximation method for single class fork-join queueing networks with MAP arrival distribution and MAP service times.

The approximate model makes two modifications to the fork-join model:

- Arrival process - Instead of a single job splitting into n sub-jobs at a fork point, each of the n queues in the fork-join construct has its own arrival process independent of the others.
- Synchronization queues - The synchronization queues are finite length and when they are full any additional jobs are dropped.

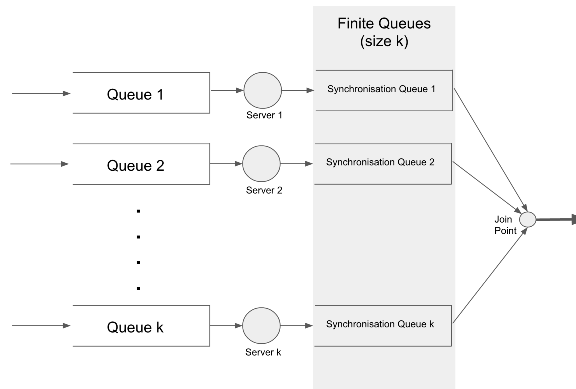

Figure 2.7: Approximate Fork-Join Model

This can be visualized in Figure 2.7.

So given the arrival and service MAPs of a fork-join construct, we approximate it by producing a MAP describing the departure process of the approximate model. This is done by considering each queue in isolation. Each of the n queues can be considered invidually as a MAP/MAP/1 queue, where the arrival process is just the arrival process into the fork-join construct and the service process is the same. In this way, we have n MAP/MAP/1 queues. Then using the method in Section 2.7 we can approximate the departure process of those as n MAPs.

Then the next step is to synchronize those n MAPs. Consider the example where there are two queues in the fork-join consruct and the length of the synchronization queue is set as one. Then the state transition diagram for the fork-join construct is like that in Figure 2.8 where the state labels (x, y) represent the number of jobs that have been serviced and are waiting to be synchronized (x for one queue, y for the other queue). The dotted lines represent hidden transitions corresponding to one queue finishing servicing, and the continuous lines represent the visible transitions corresponding to when the synchronization is complete and the job departs the join point.

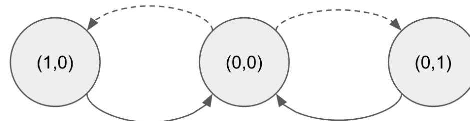

Figure 2.8: Synchronizing 2 MAPs with synchronization queue length = 2

Recall that a MAP is described using the D_0 and D_1 matrices. Let A_0, A_1 represent those matrices for one queue and B_0, B_1 represent those matrices for the other queue. Then the MAP that describes the synchronization process can be written as:

$$D_0 = \begin{pmatrix} A_0 \oplus B_0 & A_1 \otimes I & I \otimes B_1 \\ 0 & I \otimes B_0 & 0 \\ 0 & 0 & A_0 \otimes I \end{pmatrix} \tag{2.27}$$

$$D_1 = \begin{pmatrix} 0 & 0 & 0 \\ I \otimes B_1 & 0 & 0 \\ A_1 \otimes I & 0 & 0 \end{pmatrix} \tag{2.28}$$

where I is the identity matrix and $\oplus$ and $\otimes$ are the Kronecker sum and product respectively.

In general, as we extend the length of the synchronization queue, the MAP for the synchronization process can be represented as:

$$D_0 = \begin{pmatrix} A_0 \oplus B_0 & A_1 \otimes I & I \otimes B_1 & 0 & 0 & 0 & 0 & 0 \\ 0 & A_0 \oplus B_0 & 0 & A_1 \otimes I & 0 & 0 & 0 & 0 \\ 0 & 0 & A_0 \oplus B_0 & 0 & I \otimes B_1 & 0 & 0 & 0 \\ 0 & 0 & 0 & \ddots & \ddots & \ddots & 0 & 0 \\ 0 & 0 & 0 & 0 & A_0 \oplus B_0 & 0 & A_1 \otimes I & 0 \\ 0 & 0 & 0 & 0 & 0 & A_0 \oplus B_0 & 0 & I \otimes B_1 \\ 0 & 0 & 0 & 0 & 0 & 0 & I \otimes B_0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & A_0 \otimes I \end{pmatrix} \tag{2.29}$$

$$D_1 = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ I \otimes B_1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ A_1 \otimes I & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & I \otimes B_1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & A_1 \otimes I & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \ddots & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & I \otimes B_1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & A_1 \otimes I & 0 & 0 \end{pmatrix} \tag{2.30}$$

2.8.1 Size of D_0 and D_1

One important thing to note is the size of the resultant D_0 and D_1 matrices. The Kronecker product and sum of two matrices A and B with dimensions $m \times n$ and $p \times q$ respectively result in matrices with dimensions $mp \times nq$. This means that the resultant D_0 and D_1 matrices have dimensions $(2S+1)mp \times (2S+1)nq$ where S is the length of the synchronization queue.

This means that when synchronizing more than two queues, the size of the resultant D_0 and D_1 grow extremely quickly. For example, with a third parallel queue with matrices of dimensions $r \times s$, the resultant D_0 and D_1 matrices have dimensions $(2S+1)^2 mpr \times (2S+1)^2 nqs$. In general, the resultant D_0 and D_1 matrices will have dimensions $(2S+1)^{K-1} \cdot (\prod_{i=1}^{K} m_i) \times (2S+1)^{K-1} \cdot (\prod_{j=1}^{K} n_j)$, where the dimensions of the matrices for the ith MAP are $m_i \times n_i$.

2.9 Marked Markovian Arrival Processes

Marked MAPs are a generalization of the MAP where the MAP has multiple job classes and arrivals are marked with a class label. Recall that the MAP can be described using the D_0 and D_1 matrices

which describe the hidden and observable transitions respectively. A MMAP with K classes can be described with a D_0 matrix representing the hidden transitions, and K non-negative matrices $D_{11}, D_{12}, D_{13}, \ldots, D_{1k}$, representing the observable transitions (i.e. the arrivals) of each of the K classes.

To convert a MMAP into a MAP by ignoring the class labels, the D_{1x} matrices can all be summed up to obtain the D_1 matrix, i.e.

$$D_1 = \sum_{i=1}^{K} D_{1i}$$

The MMAP has been used to model multi-class arrivals queueing systems [35]. There are methods to fit MMAPs to data with an arbitrary number of classes [53].

Consider the example MAP in Figure 2.6 and the case where there are two types of jobs. Let the rate of non-hidden transitions (i.e. arrivals) going from state x to state y of job label j be defined as $\mu_{j,xy}$. Then we have:

$$D_0 = \begin{bmatrix} -(\mu_{12} + \mu_{1,13} + \mu_{2,13}) & \mu_{12} & 0 \\ 0 & -(\mu_{1,21} + \mu_{2,21}) & 0 \\ \mu_{31} & 0 & -\mu_{31} \end{bmatrix}$$
$$D_{11} = \begin{bmatrix} 0 & 0 & \mu_{1,13} \\ \mu_{1,21} & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$
$$D_{12} = \begin{bmatrix} 0 & 0 & \mu_{2,13} \\ \mu_{2,21} & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

So notice how the D_{11} and D_{12} matrices can be summed to obtain the D_1 matrix in the previous section, where $\mu_{13} = \mu_{1,13} + \mu_{2,13}$ and $\mu_{21} = \mu_{1,21} + \mu_{2,21}$.

One important thing to note is that superposing phase type distributed processes results in a MMAP.

2.10 MMAP[K]/PH[K]/1 Queues

In the decay rate approximation method described in the following chapter, we refer to the concept of a MMAP[K]/PH[K]/1 queue. This refers to the queue where arrivals are described by a MMAP with K job classes, and where the service times are PH distributed. The service time distributions for each job class do not need to be the same, and in fact are independent from each other.

2.11 Product Form Queueing Networks

Product form queueing networks are the type of queueing networks that have a product form solution. A product form solution for a queueing network is one where the join probability of the queue

sizes in the network is the product of all the probabilities of queue sizes at the individual queues [7]. More formally, a product form solution for queueing networks is when:

$$P(n_1, n_2, \ldots, n_M) = C \prod_{i=1}^{M} P(n_i) \tag{2.31}$$

where M is the number of queues and C is a normalizing constant, and where n_i is the number of jobs in queue i.

The stationary state distribution of product form queueing networks have a simple closed form expression and therefore such queueing networks can be solved efficiently even for large values of M. There are algorithms with runtime that is polynomial in the number of components in the network.

Only under certain conditions will a queueing network have a product form solution. These conditions can be found in [11], but the one that is most pertinent to this project is that for first come first served queues, a product form solution is only possible if the service time distribution is exponential and all job classes have the same service time distribution.

Chapter 3

Decay Rate Approximation

The decay rate approximation (DRA) method is a matrix analytic approximation for closed queueing networks with general first come first served (FCFS) queues [22]. It aims to improve on the performance of existing approximations of multiclass closed queueing networks that cannot be handled by product form theory. This chapter is based on the research done in [22] describing the DRA method.

3.1 Single Class Approximation

The DRA method can be considered to be an extension of a prior approximation method for single class closed queueing networks [21]. That method involves modelling the arrivals to the queues as a MAP, and then iteratively looking at each MAP/PH/1 queue in isolation.

The method is based on the observation that the stationary queue-length distribution of a MAP/PH/1 queue may be accurately approximated by:

$$p_i(n) = \begin{cases} (1-\rho_i) & n = 0 \\ \rho_i(1-\eta_i)\eta_i^n & n \geq 1 \end{cases} \tag{3.1}$$

where i refers to the ith queue, ρ_i is the utilization, and η_i is the caudal characteristic of the queue. The caudal characteristic of the queue is also the dominant eigenvalue of the rate matrix of the quasi-birth-death process used to solve the MAP/PH/1 queue using matrix analytic methods. Then using that, they formulated a heuristic product form probability expression as follows:

$$p(n_1, \ldots, n_M) = \frac{1}{C} \prod_{i=1}^{M} p_i(n_i) \tag{3.2}$$

where C is a normalizing constant. The values of ρ_i and η_i are then computed by analyzing each queue in the system in isolation. When considering an individual queue, the arrival process is computed by scaling the service process of all input queues by their utilization, and then obtaining the MAP which is the superposition of all of the scaled phase type processes. In this way, each individual queue can be analyzed as a MAP/PH/1 queue. We can then use matrix geometric methods to solve for the decay rate for the queue, η_i.

In order to do that, the utilization of the queue can be computed to be $\rho_i = X\theta_i$ where X is the current estimate of the network throughput and θ_i is the mean service demand at queue i.

A new estimate of the network throughput is computed as the average of the throughputs at each queue obtained by the product form equation 3.2 using the values of ρ_i and η_i.

At each iteration, the whole process is done and the new estimate of the network throughput is compared against the previous value. If the difference is less than some threshold value, then the loop is terminated and the last computed estimate of throughput is used.

To summarize, the algorithm comprises of:

Step 1 Initialize throughput, X, with any choice of approximation. The suggested methods are to use exponential service times (instead of phase type) with the same mean, and then use the convolution algorithm or mean value analysis algorithm [20, 38].

Step 2 Compute utilization for each queue by doing $\rho_i = X\theta_i$.

Step 3 Compute caudal characteristics for each queue by using matrix geometric methods to analyze the MAP/PH/1 queue (where the input MAP is obtained by superposition of the scaled service processes of input queues).

Step 4 Solve for a new throughput value, X', using the computed values of ρ_i and η_i in the product form equation 3.2.

Step 5 If $|X' - X|$ is less than some threshold value, we terminate and return X'. Otherwise, set $X = X'$ and go back to **Step 2**

3.2 Notation

Before starting to describe the DRA method, it is helpful to define the notation we will be using.

Let M be the number of queues in the network, and K be the number of job classes.

Let X_k be the estimate of mean throughput for class k. Then let $\mathbf{X} = (X_1, X_2, \ldots, X_K)$ be a vector of estimates of mean throughputs for all K classes.

Also, let ρ_{ik} be the mean per-class utilization at each queue when the throughputs are given by $\mathbf{X}$, and computed by $\rho_{ik} = X_k\theta_{ik}$.

In addition, let $\tilde{\rho}_{ik}$ be the mean per-class utilisation given by AMVA-PF when the model is evaluated with decay rate η_{ik} and X_k, where η_{ik} is the decay rate for class k at queue i.

3.3 Methodology

Similar to the method in Section 3.1 and in [21], one of the key ideas of DRA is that each queue in the network is considered in isolation. In the single class case, each queue was a MAP/PH/1 queue. However, in DRA we are dealing with the multiclass case, so the queues will be MMAP[K]/PH[K]/1

queues instead.

At a high level, the DRA method has the same steps as the method in Section 3.1. The basic idea is similar:

- Iteratively solve for the throughput and queue lengths.
- At each iteration, compute utilization values based on the current estimate thoughput.
- Consider each queue individually as a MMAP[K]/PH[K]/1 queue and compute the decay rate.
- Use a product form solver to compute a new estimate for the queue utilizations, system throughput and queue length distributions.
- Minimize the difference between the estimates for the utilizations.

The main differences are in performing those steps, as the multiclass case is more complicated. We discuss those differences below.

3.3.1 Initialization

DRA obtains an initial estimate of the system throughput as the one returned by the approximate mean-value analysis method for FCFS queues (AMVA-FCFS) [17]. As with the single class version, the intial estimate can be done any way desired, but the authors of [21] found experimentally that the DRA results are accurate if the throughput is initialized using AMVA-FCFS.

The AMVA-FCFS method is a simple iterative algorithm that is based on the arrival theorem. The arrival theorem is that the waiting time of a class k job at queue i can be expressed as:

$$W_{ik} = \theta_{ik} + \theta_{ik} \sum_{s=1}^{K} A_{is}^{(k)} \tag{3.3}$$

where θ_{ik} is the mean service demand of class k jobs at queue i and $A_{is}^{(k)}$ is the mean class s queue length at queue i observed by an arriving job of class k.

While the equation is exact for some queueing systems like processor-sharing queues, it is not exact for first come first served (FCFS) queues. AMVA-FCFS modifies the equation to deal with FCFS queues by approximating the waiting time as:

$$W_{ik} \approx \theta_{ik} + \sum_{s=1}^{K} \theta_{is} A_{is}^{(k)} \tag{3.4}$$

Given the stationary mean queue lengths Q_{ik}, the term $A_{is}^{(k)}$ can be computed as:

$$A_{is}^{(k)} = \begin{cases} Q_{ik} \cdot \frac{N_r - 1}{N_r} & s = k \\ Q_{is} & s \neq k \end{cases} \tag{3.5}$$

where N_k is the population of class k.

With that, we can compute W_{ik} using Equation 3.4 and with that the throughputs can be computed as:

$$X_k = \frac{K_k}{\sum_{i=1}^{M} W_{ik}} \tag{3.6}$$

Then with the throughputs, the queue lengths can be computed as:

$$Q_{ik} = W_{ik} \cdot X_k \tag{3.7}$$

AMVA-FCFS begins with an initial estimate of mean queue lengths. This can be done simply by assuming that all queues have an equal number of class k jobs and just dividing the total population of jobs by the number of queues.

At each iteration, AMVA-FCFS calculates new waiting times based on Equation 3.4 (using Equation 3.5 to calculate the $A_{is}^{(k)}$ terms). Then, new throughputs and queue lengths are computed by using Equations 4.5 and 3.7. The new queue length is then used in the next iteration to calculate the waiting times. This is performed for a set number of iterations or until the difference between two successively computed queue lengths is less than some epsilon.

Therefore, the computational complexity of the AMVA-FCFS method is $O(IMK)$ where I is the number of iterations, M is the number of queues, and K is the number of classes.

3.3.2 Obtaining Individual Queues

As mentioned earlier, to compute the caudal characteristics we consider each queue individually as a MMAP[K]/PH[K]/1 queue. The service distribution of the queue is kept as the same PH distribution. The arrival MMAP of a queue i is computed by using the superposition of the phase type service processes of the queues with output flowing to queue i. This is done on a class by class basis, considering one class at a time and getting the superposition of the processes for that class only. As with the single class version, the processes are scaled by the utilization at the queues, i.e. if class k jobs flow from queue j to queue i then the output process is multiplied by ρ_{jk}).

3.3.3 Calculating Decay Rates

This is one of the main challenges of translating the method to the multiclass version. Unlike the single class version, there is no known straightforward matrix geometric method analysis of the MMAP[K]/PH[K]/1 queue. The authors were not able to analytically determine the decay rates of the exact distribution but provided a numerical method to determine the decay rates.

3.3.4 Calculating Queue Length

From numerical experiments, the authors found that when the total number of jobs in the system is fixed, the distribution of the number of jobs belonging to different job classes is reasonably close to a multinomial distribution. The multinomial approximation of the queue length is:

$$p(\boldsymbol{n}) \approx p^*(\boldsymbol{n}) = \left(1 - \sum_{i=1}^{K} \eta_i\right) \frac{(n_1 + n_2 + \ldots + n_K)!}{n_1! \cdot n_2! \ldots n_K!} \cdot \eta_1^{n_1} \cdot \eta_2^{n_2} \ldots \eta_K^{n_K} \tag{3.8}$$

where η_k is the decay rate for class k for the specific queue that we are looking at. Note that this requires us to have already computed the decay rates using the numerical method mentioned in Section 3.3.3.

3.3.5 Optimization

DRA begins by initializing the system throughput, **X**. This throughput estimation is done using a AMVA-FCFS solver, which gives us a good starting point to begin the optimization. DRA then optimizes locally around that initial **X** using the following objective function:

$$\min f(\mathbf{X}) = \sum_{i=1}^{M} \sum_{k=1}^{K} |\tilde{\rho}_{ik} - \rho_{ik}| \text{ subject to } \mathbf{X} \geq \mathbf{X}^{-} \tag{3.9}$$

where $\mathbf{X}^{-}$ is a lower bound on the throughput which can be set to a small positive value (like $\epsilon = 10^{-3}$) just to exclude the trivial solution with zero throughput. Also, $\rho_{ik} = X_r \theta_{ik}$ is the utilization for class k jobs at queue i when the system throughput is estimated as **X**. Additionally, $\tilde{\rho}_{ik}$ is the utilization for class k jobs at queue i that is obtained from using an AMVA product form solver given the decay rates calculated when each queue is considered individually.

This optimization is performed by a non-linear optimization program such as a solver using the interior point method and DRA outputs the results obtained from the AMVA product form solver in the last iteration.

3.4 Performance

Based on the tests performed by the authors, DRA results are quite accurate. The test cases considered were those where there were 2 job classes with 2, 3, 4, 8 first come first served queues. The authors also varied the total number of jobs in the system and the relative demand across job classes and queues.

	Method		
Error (%)	DRA	AMVA-FCFS	AMVA
0 - 5	42.5%	33.75%	20%
5 - 10	45%	30%	38.75%
10 - 15	12.5%	27.5%	26.25%
15 - 20	-	7.5%	11.25%
20 - 25	-	1.25%	3.75%

Figure 3.1: Error rates for different methods

In all their tests, DRA had less than 15% error on mean queue-lengths. In fact, DRA obtains errors that are mostly below 10% and in many cases are between 2% and 5%. One thing to note is that the error for the test cases with 8 queues is larger than the ones with fewer queues. The authors did not observe any patterns in the error in relation to the number of jobs and different demands across clases and queues.

DRA can therefore be considered to be more robust than AMVA-FCFS, as it occasionally incurs errors larger than 20%-25% while DRA never has any errors above 15%. This can be visualized better in Figure 3.1.

Chapter 4

Fork-Join Approximate Mean Value Analysis

This section is based on this paper [2] that introduces new approximations for multiclass fork-join queues in both open and closed networks. In this section we describe the approximations for the heterogeneous fork-join construct in closed queueing network in both single class as well as multi-class cases.

The paper uses a generic fork-join model where the job does not necessarily have to fork into all the queues in the fork-join construct, and instead has a routing probability ϵ_i of going to the ith queue in the fork-join construct. To simplify notation, let $\epsilon_i = 1$ for all queues that are not in the fork-join construct.

4.1 Single Class

For simplicity, we deal first with the case where there is only one fork-join construct in the queueing network. Let F be the number of queues in the fork-join construct.

Let G be the number of non-FJ queues (numbered $1, \dots, G$) in the queueing network and let the queues in the FJ construct be numbered $G+1, \dots, G+F$. Let s_i be the service demand of queue i.

First, compute the residence time $R'_i(n)$ for all queues in the queueing network, including the K queues in the fork-join construct using the standard MVA equation:

$$R'_i(n) = \epsilon_i \times s_i[1 + \bar{n}_i(n-1)] \tag{4.1}$$

where $\bar{n}_i(n-1)$ is the average queue length at queue i when there is one less job in the queueing network.

Next, reorder the residence times of the fork-join devices in descending order. Compute the overall fork-join device residence time as follows:

$$R'^{*}(n) = \sum_{i=G+1}^{G+F} \frac{1}{i-M} R'_i(n) \tag{4.2}$$

where $R'_{G+1}(n) \geq R'_{G+2}(n) \geq \cdots \geq R'_{G+F}(n)$.

Then, compute the system throughput by applying Little's Law and considering that the response time is equal to the residence time at the fork-join construct plus the sum of the residence times at all non-FJ queues:

$$X(n) = \frac{n}{R'^{*}(n) + \sum_{i=1}^{G} R'_i(n)} \tag{4.3}$$

Compute the average queue length for each queue in the queueing network, including those in the fork-join construct using the standard MVA equation based on the residence times of each queue and the overall throughput:

$$\bar{n}_i(n) = X(n) \times R'_i(n) \tag{4.4}$$

4.2 Multiclass

G and F are defined in the same way as the single class case. Let $\boldsymbol{N}$ be the vector of length equal to the number of classes that describes the population of all the classes in the network. Let $R'^{*}_k(\boldsymbol{N})$ be the average residence time of class k jobs at the fork-join construct when the job populations are given by $\boldsymbol{N}$. Similar to the single class case, the model for the multiclass case also has routing probabilities. Let $\epsilon_{i,k}$ be defined as the probability that class k jobs visit queue i, and define $\epsilon_{i,k} = 1$ for queues that are not part of the fork-join construct.

The throughput for class k can be computed by:

$$X_k(\boldsymbol{N}) = \frac{N_k}{Z_k + R'^{*}_k(\boldsymbol{N}) + \sum_{i=1}^{G} R'_{i,k}(\boldsymbol{N})} \tag{4.5}$$

where Z_k is class k jobs' think time and $R'^{*}_k(\boldsymbol{N}) + \sum_{i=1}^{G} R'_{i,k}(\boldsymbol{N})$ is the average response time for class k jobs.

Then the average number of class k jobs at queue i is:

$$\bar{n}_{i,k}(\boldsymbol{N}) = X_k(\boldsymbol{N}) \times R'_{i,k}(\boldsymbol{N}) \tag{4.6}$$

And then the residence time equation is:

$$R'_{i,k}(\boldsymbol{N}) = \epsilon_{i,k} \cdot s_{i,k} \left[1 + \sum_{t=1}^{K} \bar{n}_{i,t}(\bar{N} - \mathbf{1}_k)\right] \tag{4.7}$$

where $\mathbf{1}_k$ is a vector of zeroes with a 1 in the kth position.

Then, similar to the single class approach, the residence times for the FJ queues are computed individually. The residence times of the queues in the fork-join construct are sorted in descending order and the overall fork-join residence time for each class can be computed by doing:

$$R'^{*}_{k}(\boldsymbol{N}) = \sum_{i=G+1}^{G+F} \frac{1}{i-G} R'_{i,k}(\boldsymbol{N}) \tag{4.8}$$

where $R'_{G+1,k}(\boldsymbol{N}) \geq R'_{G+2,k}(\boldsymbol{N}) \geq \cdots \geq R'_{G+F,k}(\boldsymbol{N})$.

The throughput can be computed as shown above in the Equation 4.5. The queue lengths can be computed using the individual fork-join queues residence time and the system throughput.

4.2.1 Complexity

At each iteration, the complexity is dominated by determining the total queue length with Bard-Schweitzer which is $O(MK)$ (where M is the total number of queues and K is the number of classes) as we have to loop over all combinations of queues with classes. Therefore, the overall complexity of the FJ-AMVA method is $O(IMK)$ where I is the number of iterations.

4.3 Performance

The relative error of the approximation is between 5-15 percent. However, the authors of the paper only tested exponential service time distributions. A figure with the results is shown in Figure 4.1, where the bottom axis is the utilization of the queue.

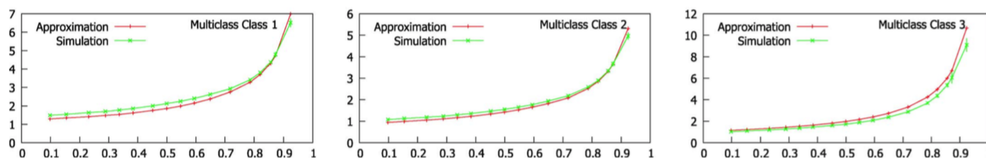

Figure 4.1: Results for the approximation for closed queueing networks

Chapter 5

Fork-Join DRA

In this chapter, we introduce our extension to the DRA method such that it now works for multiclass closed queueing networks with fork-join constructs.

Recall that in Chapter 3 we introduced the DRA method for approximating the performance of multiclass closed queueing networks with general first come first served queues with phase type service distributions. In this project, we extend that method to work on queueing networks with fork-join constructs. We do this by modifying and extending the parts of the DRA method that deal with the *initialization* of the approximation of system throughput and queue lengths as well as the *computation of the arrival distributions* for the individual MMAP[K]/PH[K]/1 queues.

As such, the general outline of the algorithm for our extension to the DRA generally remains the same:

- Iteratively solve for the throughput and queue lengths.
- At each iteration, compute utilization values based on the current estimate thoughput.
- Consider each queue individually as a MMAP[K]/PH[K]/1 queue and compute the decay rate.
- Use a product form solver to compute a new estimate for the queue utilizations, system throughput and queue length distributions.
- If the difference between the estimates for the utilizations is less than some threshold value, terminate the loop and return the last computed system throughput and queue length distributions.

5.1 Initialization

Recall that in the DRA method, we initialize the approximation of the estimate of the system throughput using the result of running the approximate mean-value analysis method for FCFS queues (AMVA-FCFS) [17] on the queueing network (in Section 3.3.1). In this section, we investigate two different methods to initialize the queue lengths and throughputs: the AMVA-FCFS method and the fork-join AMVA method described in Chapter 4 (FJ-AMVA).

5.1.1 AMVA-FCFS

The AMVA-FCFS method is a simple iterative solver for first come first served queues and is described in Section 3.3.1. The AMVA-FCFS method can still be used on queueing networks with fork-join constructs, but the way that the input is prepared needs to be slightly modified.

In order to use the AMVA-FCFS method, we need to be be able to compute the mean service demand for all classes at all queues. The mean service demand for class k at queue i can be computed by $\theta_{ik} = v_{ik} \cdot s_{ik}$ where v_{ik} is the mean number of visits of class k jobs to queue i and s_{ik} is the mean service time per visit of a class k job to queue i. The mean service time per visit can be computed in the same way as done in the DRA method, $\frac{1}{\mu_{ik}}$, where μ_{ik} is the the mean of the service distribution for class k jobs at queue i.

In the DRA method, the mean number of visits is computed using the routing matrix, P, as input (where P_{ij} is the routing probability from queue i to queue j). Note that the number of visits to queue i equals to the number of visits to all queues feeding queue i multiplied by their respective routing probabilities, i.e. $v_{ik} = \sum_{j=1}^{M} P_{ji} \cdot v_{jk}$. We can use this to set up a system of equations for all i, where one of the queues is set as the reference queue with the mean number of visits set to 1, and can solve that system of equations for the mean visits for each queue.

With the addition of the fork-join construct, the same principle no longer holds. We consider the two cases that are different. The first case is for the queues that are in parallel in the fork-join construct. For those, the mean number of visits would be equal to the mean number of visits to the queue feeding into the fork point multiplied by the routing probability of a sub-job being routed to the queue (in the simple case this equals to 1). This gives us the exact same equation as before.

The second case is for the queue that is after the join point. In this case, the mean number of visits cannot be the sum of the feeding queues as the feeding queues will be synchronized and taking the sum will be double counting. The easiest way to deal with this is that the mean number of visits to that queue should equal the mean number of visits to the queue(s) feeding into the fork point.

With this modification, we are able to use the AMVA-FCFS method to compute an initial estimate of the mean queue lengths and throughputs of a queueing network with fork-join constructs.

5.1.2 FJ-AMVA and Other Approximations

Another option for initializing the mean queue lengths and throughputs is to use the FJ-AMVA method described in Chapter 4. Unlike the AMVA-FCFS method, the FJ-AMVA method is designed to work on queueing networks with fork-join constructs. This means that we can use the FJ-AMVA directly and that there is no modification needed.

Essentially, any approximation method that can be used to approximate queueing networks with fork-join constructs can be used. As it can be expected, there is a trade off between effficiency and accuracy of the various approximation methods, and in this project we choose to investigate the AMVA-FCFS and FJ-AMVA methods because of their ability to provide relatively accurate estimates despite their simplicity and efficiency.

$\lambda_{1,1}$	$\lambda_{1,2}$	$\lambda_{2,1}$	$\lambda_{2,2}$	$\lambda_{3,1}$	$\lambda_{3,2}$	N_1	N_2	Error (AMVA-FCFS)	Error (FJ-AMVA)
2	4	2	4	2	4	1	1	0.1082	0.1082
2	4	2	4	2	4	2	2	0.1196	0.1196
2	4	2	4	2	4	3	3	0.1248	0.1247
2	4	2	4	2	4	4	4	0.1289	0.1289
2	4	2	2	4	8	2	3	0.0476	0.0476
2	4	2	4	3	6	2	3	0.0862	0.0862
2	4	2	4	4	8	2	3	0.0607	0.0607
2	4	2	4	6	12	2	3	0.0467	0.0467

Table 5.1: Error for Different Initialization Methods ($M = 3$)

5.1.3 Comparison

In this section, we compare the performance of our extension when initialized with the two different methods. We ran our tests on a queueing network like in Figure 5.1 with 2 and 4 queues in parallel and with varying amount of complexity and heterogeneity in the service distributions, and with varying populations of jobs (which results in varying amounts of utilization).

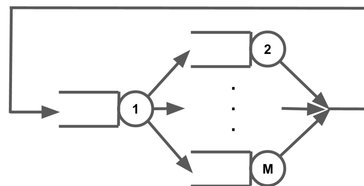

Figure 5.1: Initialization Test - Queueing Network

In this comparison and in all the results presented, we refer to the mean absolute error in the mean queue length across all classes and stations. This is obtained from [22] and is as follows:

$$\text{error} = \frac{1}{2N} \sum_{i=1}^{M} \sum_{k=1}^{K} |Q_{i,k} - \hat{Q}_{i,k}| \tag{5.1}$$

where $Q_{i,k}$ is the mean queue length for class k jobs at queue i obtained using our method, and $\hat{Q}_{i,k}$ is the same but obtained using a simulation [13]. N is the total population of jobs.

In the range of tests conducted, the difference in the error obtained when initializing using the two methods is negligible. In fact, the maximum difference of the errors in all our tests is only 0.01%. This is despite the fact that the initial estimates obtained from the two methods are usually different. In all of our test cases, the final queue lengths obtained after convergence is in between the initial estimates obtained from the two methods, with AMVA-FCFS tending to under estimate and FJ-AMVA tending to over estimate.

The errors obtained and the test parameters can be seen in Figures 5.1 and 5.2.

From this, it seems that there is not too much difference between the two methods of initializing the mean queue lengths and throughputs. However, the FJ-AMVA method is designed specifically for queueing networks with fork-join constructs, and therefore the initial estimate obtained using

$\lambda_{1,1}$	$\lambda_{1,2}$	$\lambda_{2,1}$	$\lambda_{2,2}$	$\lambda_{3,1}$	$\lambda_{3,2}$	$\lambda_{4,1}$	$\lambda_{4,2}$	$\lambda_{5,1}$	$\lambda_{5,2}$	N_1	N_2	Error (AMVA-FCFS)	Error (FJ-AMVA)
2	4	2	4	2	4	2	4	2	4	2	2	0.3636	0.3636
2	4	3	5	4	6	5	7	6	8	3	4	0.1578	0.1578
3	5	3	5	5	7	7	9	9	11	3	4	0.1586	0.1586

Table 5.2: Error for Different Initialization Methods ($M = 5$)

FJ-AMVA is more accurate and closer to the simulated value. In addition, it is easier to prepare the input for FJ-AMVA than for AMVA-FCFS. In terms of computational efficiency, both methods are polynomial in the number of queues and number of classes ($O(IMK)$, explanations found in Sections 4.2.1 and 3.3.1), and their actual practical runtime is orders of magnitude smaller than the overall iterative solver, especially once we consider longer synchronization queue lengths. Therefore, given that there is no discernible difference in accuracy between the two, we suggest using the FJ-AMVA method to initialize the mean queue lengths and throughputs and for the remainder of the paper we will use the FJ-AMVA method to initialize the mean queue lengths and throughputs.

5.2 Arrival Distribution

Recall that in the DRA method we consider each queue individually and find the decay rates for the individual queues. To do this, we need to estimate the arrival process to each individual queue. In the DRA method, we did this by approximating the departure process from each queue to be their service process scaled by the utilization of the queue. Then the arrival process for each queue would be the superposition of the departure processes of the queues that feed into it.

With the addition of fork-join constructs, we need to modify the above process. First, consider the queues in parallel in the fork-join construct. The arrival distribution to those queues is exactly the arrival distribution to the fork point, which can be approximated by the superposition of the departure processes of queues feeding into the fork point. This is essentially the same process as before and we just need to consider all queues that feed into the fork point as feeding directly into each of the parallel queues.

Next, consider the queues which have the departures from the join point feeding into them. In this case, we need to be able to estimate the departure process from the join point. Given the departure process of the join point, we can compute the superposition of that and all other feeding queues and use that as the arrival process into the queue.

To estimate the departure process from the join point, we use the results of the research from [23] that is described in detail in Section 2.8 (referred to from here on as the FJ-Sync method). That research provides a model to approximate the departure process of a fork-join construct for single class queueing networks. To approximate the synchronization of the parallel queues, the model uses the assumption of having finite length synchronization queues and creates a Markov chain with states based on the number of jobs in the synchronization queues. The departure MAP is then created based on that Markov chain.

In this project, we extend that method to work for multiclass queueing networks. First, we scale

the service processes by multiplying them by the utilization at the queues. Note that this is done for each class at each queue and therefore at one queue, the service process for one class can be scaled differently from another class. Recall that the queues are MMAP[K]/PH[K]/1 queues and that the service distributions both before and after scaling are PH processes.

Next, we approximate the departure process from the join point by considering each class individually. The idea is that when considering only one specific class, we essentially end up with just the single class problem, and then we can apply the FJ-Sync method to get an approximate departure process for that class from the join point.

To do this, consider the scaled PH[K] service processes for the parallel queues feeding into the join point. As all PH processes are also MAPs, we will consider and refer to a scaled PH[K] service process as a MMAP[K] instead.

So we consider all the $D_0, D_1, D_{1,1}, D_{1,2}, \ldots, D_{1,K}$ matrices representing the MMAP[K] individually. Recall that the D_0 matrix corresponds to the hidden transitions, the D_1 matrix corresponds to all visible transitions, and the K other matrices correspond to the visible transitions for each of the K classes.

When we only consider one class, we have a MAP where the hidden transitions are described by the D_0 matrix and the visible transitions are described by the $D_{1,k}$ matrix. We can then apply the FJ-Sync method directly to obtain a new MAP of the synchronized departure process, described by D'_0 and $D'_{1,k}$.

We then repeat this method for all classes and end up with D'_0 and $D'_{1,1}, D'_{1,2}, \ldots, D'_{1,K}$. In order to ensure that this is still a valid MMAP, we normalize it by ensuring that the diagonal values of D'_0 are set to a value such that the sum of the rows of $D'_0 + \sum_{k=1}^{K} D'_{1,k}$ is 0.

And so the synchronized departure MAP from the join point can be described with the $D'_0, D'_{1,1}, \ldots, D'_{1,K}$ transition rate matrices.

5.3 Calculating Queue Length

In the DRA method, after looking at each queue individually and calculating the decay rates, we used a product form solver to calculate the new queue lengths (described in Section 3.3.4). When dealing with fork-join constructs, the jobs split at the fork point and so the queues are going to experience duplicate jobs. This causes the mean queue lengths for the queues in the fork-join queue to be longer.

For example, consider Q_1, a closed queueing network with two homogeneous queues in series, and Q_2, a closed queueing network with two homogeneous queues in a fork-join construct. If both queueing networks have a job population of size 1, you would expect that the mean queue length at each queue in Q_1 would be approximately 0.5. However, for Q_2, the mean queue length would be larger than that. Both queues in Q_2 start servicing the job at the same time, and any waiting time (where queue length is 0) is between the time a queue finishes servicing and the time the other queue finishes servicing. In contrast, a queue in Q_1 only starts servicing the job when the other queue finishes servicing, so the waiting time would be the full service time of the other queue.

Therefore, we need some way to ensure that this is accounted for. We managed this by using the queue lengths obtained from the approximation in the initialization stage as a baseline. We set the job populations to the sum of the mean queue lengths obtained from the initial FJ-AMVA approximation, i.e.

$$N'_k = \sum_{i=1}^{M} Q_{i,k}^{\text{init}} \tag{5.2}$$

where N'_k is the scaled population for class k that will be used as input to the product form solver, and $Q_{i,k}^{\text{init}}$ is the mean queue length for queue i and class k obtained by the initial approximation (in our case, FJ-AMVA).

Chapter 6

Efficiency

In this chapter, we discuss the various efforts that we made to make the method more efficient especially in terms of runtime. We start by discussing our rationale and objectives for increasing the efficiency of the method. Then we move on to profiling the code and figuring out which parts of the code have the most effect on the runtime. After that, we go into the methods we used to make the code more efficient.

6.1 Objectives

One of the objectives of this project is to come up with a *practical* method of analyzing queueing networks with fork-join constructs. This means that not only does it have to produce accurate results, but we want it to be able to run efficiently enough to be used in practical situations. One of the drawbacks of our previous research [23] is that the method does not scale to larger fork-join constructs. In this project, we aim to come up with ways to make our method more efficient and therefore more useful in a practical setting. More concretely, we hope to be able to scale relatively well as we increase the number of parallel queues in the fork-join construct.

Another drawback of our previous research is that while the results of the approximation seemed promising, the accuracy is only better than other methods when we use larger lengths of the synchronization queue. In doing so, the runtime increases exponentially and again becomes impractical for real world use. Therefore, we aim to again be able to scale relatively well with increasing synchronization queue length at least to a point where results that are better than other state of the art approximations can be obtained within a reasonable amount of time.

6.2 Profiling

We profile the code by using MATLAB's built in profiler. The profiler runs the code and times it, and can help us to identify which functions are taking the most time to run as well as track the number of calls made to all functions. The full documentation and details about the profiler can be found on the MATLAB website [41]. In all our tests, we run the profiler on a Macbook Pro with a 2.4 GHz Intel Core i5 processor, 8GB of RAM, and an Intel Iris 1536MB graphics processor.

Parallel Queues	Total Time (s)	MMAPPHK1FCFSQueue4 Time (s)	% of Total Time
2	2.412	1.697	70.35655058
3	5.536	4.48	80.92485549
4	35.643	33.659	94.43368964
5	814.599	805.878	98.92941189

Table 6.1: Increasing Number of Parallel Queues and Runtime

As mentioned in Section 6.1, we are interested in how the runtime scales with increasing number of parallel queues and increasing length of the synchronization queue used. We discuss this in the following two sub-sections.

6.2.1 Increasing Parallel Queues

The queueing network used in our profiling tests can be seen in Figure 5.1 and is where all the service distributions are Poisson processes with mean 1. We fixed the length of the synchronization queue to be 1. In all the tests we fix the number of classes to be 2.

From our tests, the runtime is dominated by the time taken by the `MMAPPHK1FCFSQueue4` function calls. The `MMAPPHK1FCFSQueue4` function is used to compute performance measures of MMAP[K]/-PH[K]/1 queues, and in our method it is specifically used to compute the decay rates of all the queues in the network when they are looked at individually as a MMAP[K]/PH[K]/1 queue.

The `MMAPPHK1FCFSQueue4` function takes in as input the list of $D_0, \ldots, D_K$ matrices representing the arrival processes, as well as the list matrices describing the PH service distributions. The main reason that the runtime is dominated by the function is that the matrices for the arrival processes grow quickly (Section 2.8.1), and the `MMAPPHK1FCFSQueue4` function involves performing many operations on those matrices.

Table 6.1 shows how the runtime increases as the number of parallel queues increases, and the percentage of runtime spent in the `MMAPPHK1FCFSQueue4` function.

As seen in the table, the runtime increases exponentially as the number of parallel queues increases. The runtime for even 4 or 5 queues in parallel is already too large to be useful practically.

As discussed in Section 2.8.1, the size of the arrival matrices increases exponentially as the number of parallel queues increases, and unsurprisingly the time taken to process these matrices increases exponentially as well.

6.2.2 Increasing Synchronization Queue Length

We use the same set up for the queueing network as before, but fix the number of parallel queues at 2 and vary the synchronization queue length instead.

As seen in Table 6.2, the runtime increases exponentially as the length of the synchronization queue increases, approximately doubling each time the length of the synchronization queue is increased

Sync Queue Length	Total Time (s)	MMAPPHK1FCFSQueue4 Time (s)	% of Total Time
1	4.501	3.568	79.27127305
2	8.092	7.08	87.49382106
3	14.925	13.701	91.79899497
4	33.174	31.762	95.74365467
5	88.88	86.643	97.48312331

Table 6.2: Increasing Synchronization Queue Length and Runtime

by 1.

And similar to previously, the `MMAPPHK1FCFSQueue4` function dominates the runtime, again due to the fact that the size of the arrival matrices increases exponentially as the synchronization queue length increases.

6.2.3 Observations

It is clear that the growth in the size of the matrices representing the arrival processes is causing the runtime of the method to grow exponentially. Therefore, to prevent this exponential increase in the runtime, we need to come up with some methods to reduce the size of those matrices. In the following sections, we introduce a few of those methods.

6.3 Threshold Based Resizing of the MMAP

In this section, we describe our method of threshold based resizing of the MMAP. The idea is simple: once the state size of the MMAP increases past a certain point, we replace the MMAP with a simpler one with much fewer states. However, in doing so, the approximation could get less accurate and therefore we need to investigate this trade off. In this project, we tried resizing the MMAP to have one state and two states.

6.3.1 First Order MMAP

To resize the MMAP to have only one state is simple: we calculate the mean for the MAP representing the visible transitions for each class and replace the MAP with a Poisson process with the same mean. Since there is only one state, there are no hidden transitions and the D_0 matrix is just $-D_1$ (where the D_1 matrix is the sum of all the new $D_{1,k}$ matrices).

Computing the rate for the Poisson process is done by finding the equilibrium distribution of the underlying continuous time Markov chain, then multiplying the equilibrium probabilities with the transitions rates out of those states, and summing them all up, i.e.

$$\lambda_k = \pi \cdot D_{1,k} \cdot e(n, 1) \tag{6.1}$$

where:

- λ_k is the rate for class k

- n is the number of states of the MAP
- π is the equilibrium distribution (with dimensions $1 \times n$) of the MAP
- $D_{1,k}$ is the D_1 matrix for class k (with dimensions $n \times n$)
- $e(n, 1)$ is the column vector of all 1's (with dimensions $n \times 1$)

The results for increasing synchronization queue length can be seen in Figure 6.1, and the results for increasing the number of parallel queues can be seen in Figure 6.2. For these tests, we used the same set up as in Section 6.2, and set the threshold size to 1, i.e. when computing the arrival processes, we will resize the MMAP down to a single state MMAP at any point where it has more than one state.

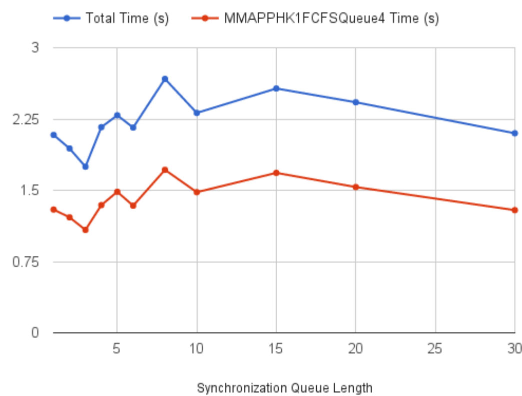

Figure 6.1: Resizing to First Order MAP: Synchronization Queue Length

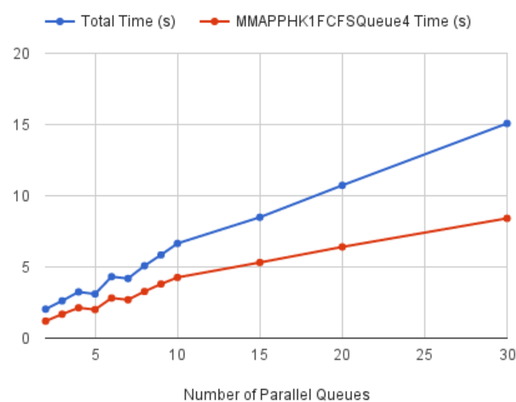

Figure 6.2: Resizing to First Order MAP: Number of Parallel Queues

The results show that when we do the resizing to a first order MMAP, increasing the synchronization queue length does not seem to affect the runtime. This is probably because the only extra work needed to be done when increasing the synchronization queue length is when finding the mean of the MAPs that get larger as synchronization queue length increases. Finding the mean of the MAP

is done by finding the equilibrium distribution of the underlying continuous time Markov chain, multiplying the equilibrium probabilities with the transitions rates out of those states, and summing them all up. Multiplying the probabilities with the transitions rates and summing them can be done with basic matrix multiplications which are performed very efficiently in MATLAB and can exploit parallelization which explains why the runtime does not seem to be increasing. The most computationally expensive part of finding the equilibrium distribution is to solve a set of linear equations. The efficiency of that is difficult to analyze as MATLAB has various optimizations and uses different solvers for matrices with different characteristics. However, from our results it seems like the increasing size of the matrices does not affect the runtime of the solver in this situation. This is great as it allows us to increase the length of the synchronization queue arbitraily and obtain more accurate results.

When increasing the number of parallel queues, the runtime seems to scale with that linearly. This is probably because when we synchronize the parallel queues, we do that iteratively two at a time, and ensure that we resize the matrices at each iteration. Therefore the size of the matrices does not grow with each iteration as it used to do before. And therefore increasing the number of parallel queues by n just requires us to repeat the same process n more times but with approximately inputs of the same size.

6.3.2 Second Order MMAP

We also tried to resize the MMAP to have two states. To do this, we used the `mamap2m_fit_gamma_fb_mmap` function that already existed as part of the DRA method. The `mamap2m_fit_gamma_fb_mmap` function performs approximate fitting of a MMAP, yielding a second-order acyclic MMAP that matches the class probabilities, the forward and backward moments.

The results for increasing synchronization queue length can be seen in Figure 6.3, and the results for increasing the number of parallel queues can be seen in Figure 6.4. For these tests, we used the same set up as in Section 6.2, and set the threshold size to 2, i.e. when computing the arrival processes, we will resize the MMAP down to a two state MMAP at any point where it has more than two states.

The results show that when resizing to a second order MMAP, increasing the length of the synchronization queue helps the runtime scale much better. Previously, setting synchronization length of 5 took 88.88 seconds to run compared to 39.298 seconds with the resizing. Also, previously the runtime approximately doubled as the synchronization length increased, and is definitely increasing at a much slower rate when we use resizing.

Regarding increasing the number of parallel queues, the runtime also definitely scales better with resizing. Without resizing, the runtime increases extremely quickly, with just 5 parallel queues taking 814.599 seconds. With resizing, the same set up takes only 61.405 seconds. The growth of the runtime is much slower, with a seemingly linear increase.

While the runtime seems to scale better, it is still undoubtedly true that the actual runtimes are still relatively large. The minimum runtimes when varying the synchronization queue length and number of parallel queues are 23.742 seconds and 24.865 seconds respectively. These runtimes will definitely be improved with better hardware, but are definitely much slower and less practical than the runtimes obtained resizing to MMAPs with just one state.

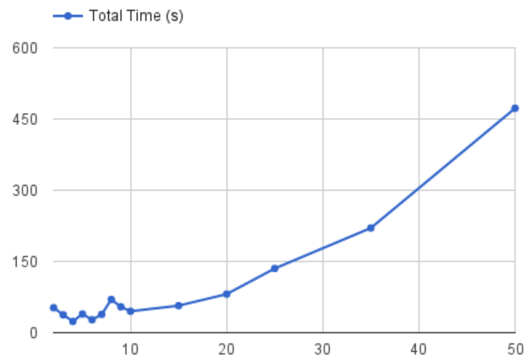

Figure 6.3: Resizing to Second Order MAP: Synchronization Queue Length

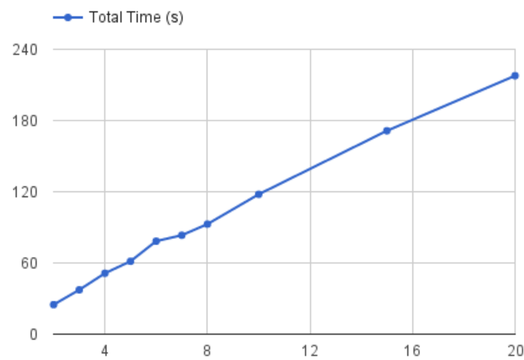

Figure 6.4: Resizing to Second Order MAP: Number of Parallel Queues

6.3.3 Summary

The threshold based resizing works very well to help the runtimes scale better with increasing the synchronization queue lengths and number of parallel queues. In particular, resizing to a first order MMAP seem to allow us to increase the synchronization queue length without any or much additional runtime. This is crucial because increasing the synchronization queue length to 10 helps to increase the accuracy up to 90% in the single class case [23].

Chapter 7

Results

In this chapter, we present the results of our extension to the DRA method. We compare the accuracy of our method against the FJ-AMVA approximation method described in Chapter 4. One of our objectives of this project is to develop an approximation method that is accurate on a wide range of queueing networks. Therefore, in our experiments below, we investigate how accurate the FJ-DRA approximation is when we vary the following factors:

- Number of fork-join queues in the fork-join construct
- Heterogeneity of the fork-join queues
- Complexity of service distributions

7.1 Experimental Set Up

For all our experiments, we present three sets of results: simulated, FJ-AMVA, and FJ-DRA. The simulated values are those that are obtained by running the Java Modelling Tools Simulation [13] with the confidence interval set to 0.99 and the maximum relative error set to 0.03. The FJ-AMVA results are those obtained by using the FJ-AMVA method in Chapter 4 with the tolerance set to 0.00001 and max iterations set to 300. The FJ-DRA results are obtained by using the method in Chapter 5.

We used the threshold based resizing described in Section 6.3 to resize the MMAP to a first order MMAP with a threshold size of 1, i.e. the MMAP is resized at any point where it is larger than 1. As presented in Chapter 6, this allows us to use longer synchronization queues and so in all the tests we use a synchronization length of 50.

In all the results presented below, we refer to the mean absolute error in the mean queue length across all classes and stations. This is obtained from [22] and is as follows:

$$\text{error} = \frac{1}{2N} \sum_{i=1}^{M} \sum_{k=1}^{K} |Q_{i,k} - \hat{Q}_{i,k}| \tag{7.1}$$

where $Q_{i,k}$ is the mean queue length for class k jobs at queue i obtained using our method, and $\hat{Q}_{i,k}$ is the same but obtained using a simulation [13]. This is the same equation as used in Chapter 5.

Parameter	Value
N_1	3
N_2	4
$\lambda_{1,1}$	1
$\lambda_{1,2}$	2
$\lambda_{i,1}$ $(\forall i \neq 1)$	2
$\lambda_{i,2}$ $(\forall i \neq 1)$	4

Table 7.1: Parameters for Experiments Testing Increasing Number of Parallel Queues

Except for the tests with more complex service distributions in Section 7.4, we fix the service distributions to be exponential distributions. We use the notation $\lambda_{i,k}$ to be the rate of class k jobs at queue i.

In all our tests, we use the notation N_k to refer to the population of class k.

7.2 Number of Fork-Join Queues

In this section, we investigate how the accuracy of the FJ-DRA method is affected as we change the number of fork-join queues. In the experiments below we use the queueing network as shown in Figure 7.1, and vary the number of queues in the fork-join construct. The parameters for this experiment can be found in Table 7.1.

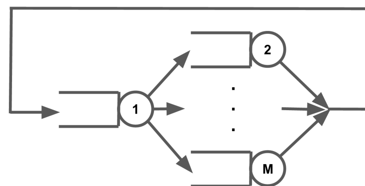

Figure 7.1: Testing Increasing Number of Fork-Join Queues

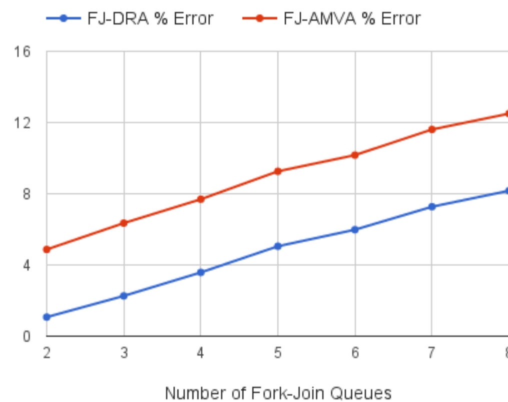

Figure 7.2: Results - Increasing Number of Fork-Join Queues

Parameter	Value
N_1	2
N_2	3
$\lambda_{1,1}$	1
$\lambda_{1,2}$	2
$\lambda_{i,1}$ $(\forall i \neq 1)$	1+0.1(i-1)h
$\lambda_{i,2}$ $(\forall i \neq 1)$	2+0.2(i-1)h

Table 7.2: Parameters for Experiments Testing Varying Heterogeneity

The results are presented in Figure 7.2. We observe that the mean queue length error of our FJ-DRA method is less than the error when the FJ-AMVA method is used. The error is very low when the number of fork-join queues is small, at 1.07% for two fork-join queues. However, the error increases as the number of fork-join queues increases, reaching 8.17% for 8 fork-join queues. Note that the error for the FJ-AMVA method also increases as the number of fork-join queues increases, and the percentage error for both methods increase at approximately the same rate.

Therefore, our results suggest that the approximations obtained from our FJ-DRA method are more accurate than those obtained from the FJ-AMVA method for queueing networks with varying numbers of fork-join queues.

7.3 Heterogeneity Of Fork-Join Queues

In this section, we investigate the effect of varying the heterogeneity of the fork-join queues on the accuracy of the FJ-DRA method. We use the queueing network shown in Figure 7.1 and fix the number of fork-join queues to be 4. The parameters for the experiment can be seen in Table 7.2 where h is a variable we vary in our experiments to change the amount of heterogeneity.

The results can be seen in Figure 7.3. The results show that the performance of the FJ-DRA method is slightly better than the FJ-AMVA when there is less heterogeneity as the FJ-DRA method was better up to $h = 7$ and worse from $h = 8$ onwards. For ease of reading, the exact parameters for those values are presented in Table 7.3.

From $h = 8$ onwards, the performance difference between the two methods seems to increase as h increases. This suggests that for queueing networks with extremely heterogeneous fork-join constructs, the FJ-AMVA method has more accurate approximations. In general, the amount of error is relatively low, with the maximum error of the FJ-DRA method being 9.58%. Also, there seems that when the heterogeneity of the fork-join queues increases, the accuracy of both approximations increases as well, with the error decreasing from $h = 1$ onwards, and going below 5% for $h \geq 7$.

One possible reason that the performance of the FJ-DRA method is not as good when the amount of heterogeneity increases is that we need to use a longer synchronization queue. In all the tests, the length of the synchronization queue is fixed. However, when the queues get more heterogeneous, it gets more likely that we need a longer synchronization queue. One possible way to mitigate this is to scale the length of the synchronization queue as the heterogeneity increases.

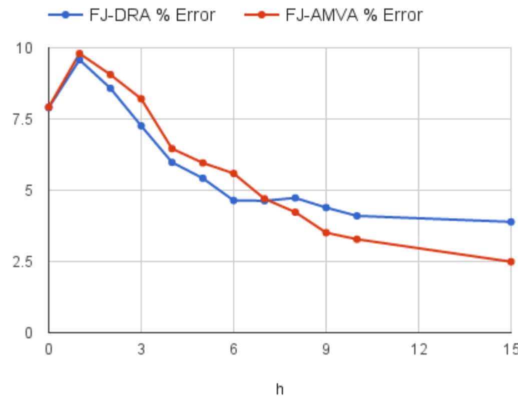

Figure 7.3: Results - Heterogeneity

Parameter	h = 7	h = 8
$\lambda_{1,1}$	1	1
$\lambda_{1,2}$	2	2
$\lambda_{2,1}$	1.7	1.8
$\lambda_{2,2}$	3.4	3.6
$\lambda_{3,1}$	2.4	2.6
$\lambda_{3,2}$	4.8	5.2
$\lambda_{4,1}$	3.1	3.4
$\lambda_{4,2}$	6.2	6.8
$\lambda_{5,1}$	3.8	4.2
$\lambda_{5.2}$	7.6	8.4

Table 7.3: Parameters for h=7 and h=8

7.4 Other Service Distributions

In this section, we investigate how the accuracy of the FJ-DRA method is affected by the complexity of the service distributions. For the experiments below, we use the same queueing network topology as before and as shown in Figure 7.1 and fix the number of fork-join queues to be 2.

In our experiments, we set the service distributions to the Erlang-2 distribution.

- Erlang-2
- Markov Modulated Poisson Process (MMPP)

7.4.1 Erlang-2

The Erlang-2 distribution is described in Section 2.5.1 and is parameterized by the rate λ, and we use the notation $\lambda_{i,k}$ to describe the rate of class k jobs at queue i.

For the experiments below, we fix some of the parameters as in Table 7.4 and make all the fork-join queues homogeneous, i.e. fix all $\lambda_{i,1}$ to be equal and all $\lambda_{i,2}$ to be equal (for $i \neq 1$).

Parameter	Value
N_1	1
N_2	2
$\lambda_{1,1}$	2
$\lambda_{1,2}$	4

Table 7.4: Parameters for Erlang-2 Experiments

$\lambda_{i,1}$	$\lambda_{i,2}$	FJ-DRA % Error	FJ-AMVA % Error
2	4	4	4
3	4	3.12	3.16
4	4	2.63	4.36
3	5	2.76	3
3	6	2.56	3.11
4	8	3.85	3.39
5	10	4.27	3.68
6	12	4.23	3.72
10	20	3.57	3.18

Table 7.5: Results - Erlang-2

The results can be seen in Table 7.5. For most of the cases, FJ-DRA is slightly more accurate than FJ-AMVA. The performance for both methods is generally very good, with the maximum error for all the test cases being less than 5%. The FJ-DRA approximation seems to perform worse than the FJ-AMVA in situations where the rates of the Erlang-2 process for the fork-join queues are much greater than the other queue. The error seems to be somewhat constant and there is no clear increasing or decreasing trend as the λ's change.

7.4.2 Markov Modulated Poisson Process (MMPP)

The MMPP is described in detail in Section 2.5.1. In our experiments, we use a two state MMPP (as seen in Figure 2.5) to represent the service process at all the queues. We use $\lambda_{i,k}^{(s)}$ to represent the rate of servicing class k jobs at queue i in state s, and use $\sigma_{i,k}^{(s_1,s_2)}$ to represent the rate of transitioning from state s_1 to state s_2 for class k jobs at queue i. Like before, we fix some parameters as seen in Table 7.6, and have all the fork-join queues be homogeneous.

One important difference in this case is that the MMPP is **not** a PH process, but instead is a MAP. However, the product form solver we use is for MMAP[K]/PH[K]/1 queues and not MMAP[K]/-MMAP[K]/1 queues, so we need to obtain a PH that best represents the MMPP. Recall that a PH process is characterized by α, a probability vector of the probabilities of starting in each state, and S, the transition rate matrix for the process. We set the D_0 matrix of the MMPP as the S matrix, and set the equilibrium distribution of the MMPP as α. With this transformation, we can use our method to approximate the case where the service processes are MMPPs.

The results can be seen in Figure 7.4. The FJ-DRA method is equal or better than the FJ-AMVA method in all the test cases, with the error for FJ-DRA less than 2% for most test cases, and under

Parameter	Value
N_1	2
N_2	3
$\lambda_{1,1}^{(1)}$	1
$\lambda_{1,1}^{(2)}$	2
$\lambda_{1,2}^{(1)}$	2
$\lambda_{1,2}^{(2)}$	4
$\lambda_{i,1}^{(1)}$ $(\forall i \neq 1)$	1
$\lambda_{i,2}^{(1)}$ $(\forall i \neq 1)$	2
$\lambda_{i,2}^{(2)}$ $(\forall i \neq 1)$	$2 \cdot \lambda_{i,1}^{(2)}$
$\sigma_{i,k}^{(s_1,s_2)}$ $(\forall i, k, s_1, s_2)$	2

Table 7.6: Parameters for MMPP Experiments

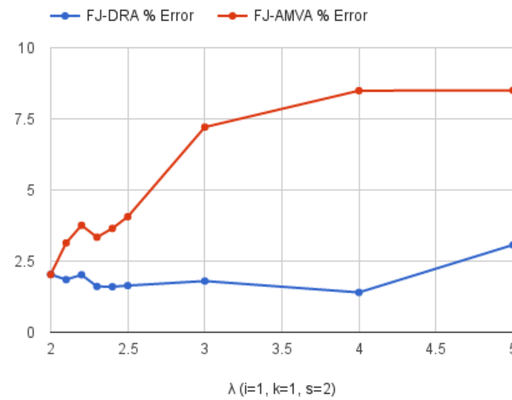

Figure 7.4: Results - MMPP

3% for all but one test case. In contrast, the FJ-AMVA method has errors of less than 5% for the test cases where $\lambda_{i,1}^{(2)}$ is smaller, but rises to errors of more than 8% when that rate is increased. This might be because the FJ-AMVA method only takes into account the mean of the service distributions, and not any other higher order moments. In contrast, the FJ-DRA method uses the original MMAP when computing the synchronized departure process, and only loses information about the higher order moments when it needs to resize the MMAP.

The results suggest that the FJ-DRA method is much more accurate than the FJ-AMVA in approximating a fork-join construct with MMPP service processes.

7.5 Summary

In this chapter, we tested the accuracy of the FJ-DRA method against the FJ-AMVA method varying three factors: number of fork-join queues, heterogeneity of the fork-join queues, and complexity of the service distributions.

In most of our test cases, the FJ-DRA method is at least as accurate as the FJ-AMVA method. There are some test cases where FJ-DRA method is much better than the FJ-AMVA method, such as in our tests for increasing the number of parallel queues and when we set the service distributions to be MMPPs. The only test cases where the FJ-AMVA method performs better is when the fork-join queues are very heterogeneous.

Overall, the FJ-DRA method is relatively accurate, with less than 10% error in all our tests.

Chapter 8

Conclusion

Recall that our objective for the project is to create an approximation method for multiclass fork-join queues that is efficient, accurate, and works on a wide range of queueing networks. We presented the FJ-DRA approximation method and an implementation of it in MATLAB. We tested it while varying three factors: the number of fork-join queues, the heterogeneity of the fork-join queues, and the service distributions. From our results, the FJ-DRA method is more accurate than the FJ-AMVA approximation for a majority of our tests and has less than 10% error for the mean queue lengths in all our tests. In particular, the results are significantly better for the tests with increasing number of queues and when we set the service processes to MMPPs.

We also investigated some ways to ensure that the runtime scales well as we increase the number of fork-join queues. This ensures that the FJ-DRA method is reasonably efficient and can be used in practical real world situations. The runtime also scales relatively well as we increase the length of the synchronization queues in our model, allowing us to obtain more accurate approximations.

In this way, we feel that we have achieved the objectives that we set out at the start. However, there are still parts of the project that can be investigated and extended, and in the section below we detail a few of our ideas.

8.1 Future Work

One way to increase both the accuracy and efficiency of the method is when resizing the MMAP. Currently, we use a fixed synchronization queue length and generate the MMAP based on that. However, we notice that in that process we are actually generating a truncated quasi-birth-death (QBD) process, and if we use an infinite length synchronization queue we will obtain a non-truncated QBD. We can use matrix geometric methods to solve for the stationary distribution of the QBD. Then we can resize the MMAP to have one state by using the stationary distribution to approximate the rate of the process.

Bibliography

[1] Arnold O Allen. *Probability, statistics, and queueing theory*. Academic Press, 2014. pages

[2] Firas Alomari and Daniel A Menasce. Efficient response time approximations for multiclass fork and join queues in open and closed queuing networks. *Parallel and Distributed Systems, IEEE Transactions on*, 25(6):1437–1446, 2014. pages 3, 29

[3] Tayfur Altiok. On the phase-type approximations of general distributions. *IIE Transactions*, 17(2):110–116, 1985. pages 11

[4] Danilo Ardagna, Michele Ciavotta, GP Gibilisco, G Casale, and J Pérez. Prediction and cost assessment tool-proof of concept. *Project deliverable*, 2013. pages

[5] Søren Asmussen and Ger Koole. Marked point processes as limits of markovian arrival streams. *Journal of Applied Probability*, pages 365–372, 1993. pages 12

[6] Søren Asmussen, Olle Nerman, and Marita Olsson. Fitting phase-type distributions via the em algorithm. *Scandinavian Journal of Statistics*, pages 419–441, 1996. pages 11

[7] Simonetta Balsamo. Product form queueing networks. In *Performance Evaluation: Origins and Directions*, pages 377–401. Springer, 2000. pages 21

[8] Simonetta Balsamo, Lorenzo Donatiello, and Nico M Van Dijk. Bound performance models of heterogeneous parallel processing systems. *Parallel and Distributed Systems, IEEE Transactions on*, 9(10):1041–1056, 1998. pages

[9] Simonetta Balsamo and Ivan Mura. Approximate response time distribution in fork and join systems. *ACM SIGMETRICS Performance Evaluation Review*, 23(1):305–306, 1995. pages

[10] Simonetta Balsamo and Ivan Mura. On queue length moments in fork and join queuing networks with general service times. In *Computer Performance Evaluation Modelling Techniques and Tools*, pages 218–231. Springer, 1997. pages

[11] Forest Baskett, K Mani Chandy, Richard R Muntz, and Fernando G Palacios. Open, closed, and mixed networks of queues with different classes of customers. *Journal of the ACM (JACM)*, 22(2):248–260, 1975. pages 21

[12] Ron Bekkerman, Mikhail Bilenko, and John Langford. *Scaling up machine learning: Parallel and distributed approaches*. Cambridge University Press, 2011. pages 3

[13] Marco Bertoli, Giuliano Casale, and Giuseppe Serazzi. Jmt: performance engineering tools for system modeling. *SIGMETRICS Perform. Eval. Rev.*, 36(4):10–15, 2009. pages 35, 45

[14] Dimitris Bertsimas and Jose Nino-Mora. Optimization of multiclass queueing networks with changeover times via the achievable region approach: Part ii, the multi-station case. *Mathematics of Operations Research*, 24(2):331–361, 1999. pages 1

[15] Dario Bini and Beatrice Meini. On the solution of a nonlinear matrix equation arising in queueing problems. *SIAM Journal on Matrix Analysis and Applications*, 17(4):906–926, 1996. pages 15

[16] Henrik Bohnenkamp and Boudewijn Haverkort. The mean value of the maximum. In *Process Algebra and Probabilistic Methods: Performance Modeling and Verification*, pages 37–56. Springer, 2002. pages

[17] Gunter Bolch, Stefan Greiner, Hermann de Meer, and Kishor S Trivedi. *Queueing networks and Markov chains: modeling and performance evaluation with computer science applications*. John Wiley & Sons, 2006. pages 2, 25, 33

[18] Onno Johan Boxma, Ger Koole, and Zhen Liu. *Queueing-theoretic solution methods for models of parallel and distributed systems*. Centrum voor Wiskunde en Informatica, Department of Operations Research, Statistics, and System Theory, 1994. pages 3

[19] John A Buzacott and J George Shanthikumar. *Stochastic models of manufacturing systems*, volume 4. Prentice Hall Englewood Cliffs, NJ, 1993. pages 2

[20] Jeffrey P Buzen. Computational algorithms for closed queueing networks with exponential servers. *Communications of the ACM*, 16(9):527–531, 1973. pages 24

[21] Giuliano Casale and Peter Harrison. A class of tractable models for run-time performance evaluation. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, pages 63–74. ACM, 2012. pages 23, 24, 25

[22] Giuliano Casale, Gabor Horvath, and Juan Perez. A matrix-analytic approximation for closed queueing networks with general fcfs nodes. pages 2, 3, 23, 35, 45

[23] Joel Choo. Analysing queueing networks with fork-join constructs. pages 3, 4, 17, 36, 39, 44

[24] ChengFu Chou. Open queueing network and mva. http://www.cmlab.csie.ntu.edu.tw/ nypgand1/Perf2013/files/slides/lec7.pdf. pages 10

[25] Fast Company. How one second could cost amazon $1.6 billion in sales. https://www.fastcompany.com/1825005/how-one-second-could-cost-amazon-16-billion-sales, 2012. pages 3

[26] Jeff Dean. Building software systems at google and lessons learned. http://www.youtube.com/watch?v=modXC5IWTJI, 2010. pages 2, 3

[27] Facebook. Facebook to open-source ai hardware design. https://code.facebook.com/posts/1687861518126048/facebook-to-open-source-ai-hardware-design/, 2015. pages 3

[28] Pierre M. Fiorini and Lester Lipsky. Exact analysis of some split-merge queues. *SIGMETRICS Perform. Eval. Rev.*, 43(2):51–53, September 2015. pages 8

[29] Leopold Flatto and S Hahn. Two parallel queues created by arrivals with two demands i. *SIAM Journal on Applied Mathematics*, 44(5):1041–1053, 1984. pages 2

[30] Bennett Fox. Semi-markov processes: a primer. Technical report, DTIC Document, 1968. pages

[31] Erol Gelenbe and Isi Mitrani. *Analysis and synthesis of computer systems*, volume 4. World Scientific, 2010. pages 2

[32] Mor Harchol-Balter. *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*. Cambridge University Press, 2013. pages 10

[33] Qi-Ming He and Marcel F Neuts. Markov chains with marked transitions. *Stochastic processes and their applications*, 74(1):37–52, 1998. pages

[34] Armin Heindl, Qi Zhang, and Evgenia Smirni. Etaqa truncation models for the map/map/1 departure process. In *Quantitative Evaluation of Systems, 2004. QEST 2004. Proceedings. First International Conference on the*, pages 100–109. IEEE, 2004. pages 16

[35] András Horváth, Gábor Horváth, and Miklós Telek. A traffic based decomposition of two-class queueing networks with priority service. *Computer Networks*, 53(8):1235–1248, 2009. pages 20

[36] András Horváth, Gábor Horváth, and Miklós Telek. A joint moments based analysis of networks of map/map/1 queues. *Performance Evaluation*, 67(9):759–778, 2010. pages 15, 16

[37] Guy Latouche and V Ramaswami. A logarithmic reduction algorithm for quasi-birth-death processes. *Journal of Applied Probability*, pages 650–674, 1993. pages 15

[38] Edward D Lazowska, John Zahorjan, G Scott Graham, and Kenneth C Sevcik. *Quantitative system performance: computer system analysis using queueing network models*. Prentice-Hall, Inc., 1984. pages 24

[39] Abigail S Lebrecht and William J Knottenbelt. Response time approximations in fork-join queues. In *23rd UK Performance Engineering Workshop (UKPEW)*, 2007. pages 9

[40] David M Lucantoni, Kathleen S Meier-Hellstern, and Marcel F Neuts. A single-server queue with server vacations and a class of non-renewal arrival processes. *Advances in Applied Probability*, pages 676–705, 1990. pages

[41] MathWorks. Matlab profiler. https://uk.mathworks.com/help/matlab/matlab_prog/profiling-for-improving-performance.html. pages 39

[42] Takahashi Misa and Takahashi Yukio. Synchronization queue with two map inputs and finite buffers. *Advances in Algorithmic Methods for Stochastic Models, Notable Publications Inc.*, pages 375–390, 2000. pages

[43] Randolph Nelson and Asser N Tantawi. Approximate analysis of fork/join synchronization in parallel queues. *Computers, IEEE Transactions on*, 37(6):739–743, 1988. pages 9

[44] Marcel F Neuts. Structured stochastic matrices of m/g/1 type and their applications. 1989. *Marcel Decker Inc., New York*. pages 10

[45] Takayuki Osogami. Definition of ph distribution. pages 10

[46] Takayuki Osogami. Properties of markovian arrival processes. pages 12

[47] J. F. Pérez, J. Van Velthoven, and B. Van Houdt. Q-mam: A tool for solving infinite queues using matrix-analytic methods. In *Proceedings of the 3rd International Conference on Performance Evaluation Methodologies and Tools*, ValueTools '08, pages 16:1–16:9, ICST, Brussels, Belgium, Belgium, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering). pages

[48] Jhon Fernando Perez and Giuliano Casale. Assessing sla compliance from palladio component models. In *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2013 15th International Symposium on*, pages 409–416. IEEE, 2013. pages

[49] Juan F Pérez and Benny Van Houdt. Quasi-birth-and-death processes with restricted transitions and its applications. *Performance Evaluation*, 68(2):126–141, 2011. pages 15

[50] B Pittel. Closed exponential networks of queues with saturation: the jackson-type stationary distribution and its asymptotic analysis. *Mathematics of Operations Research*, 4(4):357–378, 1979. pages 2, 10

[51] Martin Reiser and Stephen S Lavenberg. Mean-value analysis of closed multichain queuing networks. *Journal of the ACM (JACM)*, 27(2):313–322, 1980. pages 2

[52] Ramin Sadre and Boudewijn Haverkort. Characterising traffic streams in networks of map— map— 1 queues. 2001. pages 16

[53] Andrea Sansottera, Giuliano Casale, and Paolo Cremonesi. Fitting second-order acyclic marked markovian arrival processes. In *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on*, pages 1–12. IEEE, 2013. pages 20

[54] Misa Takahashi, Hideo Ōsawa, and Takehisa Fujisawa. On a synchronization queue with two finite buffers. *Queueing Systems*, 36(1-3):107–123, 2000. pages

[55] Alexander Thomasian. Analysis of fork/join and related queueing systems. *ACM Computing Surveys (CSUR)*, 47(2):17, 2015. pages 1, 7

[56] Alexander Thomasian and Asser N Tantawi. Approximate solutions for m/g/1 fork/join synchronization. In *Proceedings of the 26th conference on Winter simulation*, pages 361–368. Society for Computer Simulation International, 1994. pages 9

[57] Elizabeth Varki, Arif Merchant, and Hui Chen. The m/m/1 fork-join queue with variable sub-tasks. *Unpublished–http://www. cs. unh. edu/varki/publication/open. pdf*, 2008. pages 9

[58] Subir Varma and Armand M Makowski. Interpolation approximations for symmetric fork-join queues. *Performance Evaluation*, 20(1):245–265, 1994. pages 9

[59] Sanjeev Verma. Matrix geometric technique: Theory and application to queueing problems. 1993. pages

[60] Ward Whitt. Open and closed models for networks of queues. *AT&T Bell Laboratories Technical Journal*, 63(9):1911–1979, 1984. pages 10