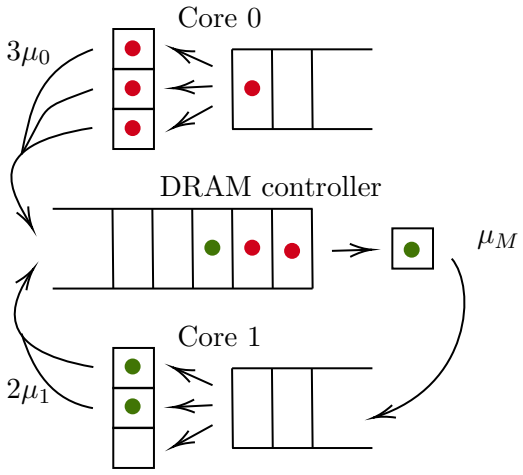
Processor Modelling Using Queueing Theory

Erik Leonards

June 23, 2023

Bachelor Thesis Mathematics and Computer Science

Supervisor: Prof. Dr. (Rude)sindo Núñez Queija, Dr. Anuj Pathania



Informatics Institute
Korteweg-de Vries Institute for Mathematics
Faculty of Sciences
University of Amsterdam

Abstract

This thesis gives a general introduction to closed queueing networks, describes mean value analysis, and formulates a closed queueing network model that predicts the parallel execution time of multiple benchmarks when the isolated execution time is known. The model aims at imitating the main source of interference between the cores: the DRAM. For an n-core processor, the model comprises $n+1$ queues: n cores and the DRAM. The accuracy of the model is determined for six benchmarks in the PARSEC benchmark suite. For 2 of the benchmarks, the model could very accurately predict the parallel execution time, while the model was less accurate for the other 4 benchmarks.

Title: Processor Modelling Using Queueing Theory
Authors: Erik Leonards, erik.leonards@student.uva.nl, 13170007
Supervisors: Prof. Dr. (Rude)sindo Núñez Queija, Dr. Anuj Pathania
Second grader: Prof. Dr. Ir. Cees de Laat, Dr. Sonja Cox
End date: June 23, 2023

Informatics Institute
University of Amsterdam
Science Park 904, 1098 XH Amsterdam
`http://www.ivi.uva.nl`

Korteweg-de Vries Institute for Mathematics
University of Amsterdam
Science Park 904, 1098 XH Amsterdam
`http://www.kdvi.uva.nl`

1 Introduction

Queueing networks are heavily studied and many practical applications of queueing networks are discussed in research papers. Due to the fact that collecting real-world data is costly, time-consuming, and practical applications are generally not interesting for mathematicians, there is often no comparison made to real-world data. This raises the following question: Can an abstract queueing network actually offer an accurate representation of a practical application? This thesis aims at answering this question for at least one practical application: the waiting time in a computer memory unit.

To answer this question, this thesis proposes a closed queueing network model that tries to answer the following research question: How can the isolated execution data of computer programs be used to predict the execution time of the programs running in parallel? In order to understand how this research question can answer the question from the first paragraph, we need to provide some more information about multiprogramming.

1.1 Multiprogramming

Multiprogramming is a form of parallel processing in which multiple computer programs run in parallel on a single processor with multiple processor cores. Each program uses one or more cores and each core can independently execute a list of instructions. There are many different types of programs, but this thesis only discusses user-independent programs (benchmarks) for testing and repeatability purposes.

Benchmarks running in parallel slow each other down, since they all use a shared memory unit: the Dynamic Random-Access Memory (DRAM). The benchmarks can use the DRAM by means of sending DRAM requests. Due to the fact that the DRAM can only handle a couple of requests at once, the average DRAM request waiting time increases as more benchmarks are run in parallel. This means that the parallel execution time of a benchmark is higher than the isolated execution of a benchmark. The proposed model will try to predict this increase in execution time by focusing on only the DRAM requests of the cores. If the increase in execution time can be correctly predicted, then we assume that closed queueing networks can offer an accurate representation of the waiting time in the DRAM.

It should be noted that heat transfer occurs between cores, but the analysis in this thesis is limited to a system running on a constant clock frequency, so the temperature of the core is irrelevant.

1.2 Motivation

The research question cannot only give insight into the usefulness of queueing networks for highly complicated systems, such as memory units, but an accurate model can also be useful for computer scientists. This is because benchmarks are often used to measure the performance of a computer system. The most accurate results are obtained when a benchmark is simulated using benchmark simulation software such as CoMeT [7]. While benchmark simulation is more accurate than benchmark execution, it should be noted that benchmark simulation takes a lot longer than benchmark execution, e.g. a benchmark with an execution time of a second can easily take an hour to simulate. Parallel execution of multiple benchmarks will take even longer to simulate. Thus, a model which can accurately predict the parallel execution time of a benchmark can be very useful.

1.3 Goal of thesis

Due to the fact that this is both a mathematics and computer science thesis, there are mathematics goals and computer science goals I would like to reach.

For the mathematics part, the main goal is understanding mean value analysis and creating and analyzing a specific closed queueing network model. After reaching this goal, I can discuss how other types of queueing networks or distributions can increase the accuracy of the model.

For the computer science part, the main goal is to get familiar with CoMeT, state an approach to determine the model parameters, and determine the accuracy of the model. After reaching this goal, I can discuss and implement possible extensions of the model.

1.4 Overview

Chapter 2 gives some more information about the two different types of DRAM requests, discusses the PARSEC benchmark suite, and specifies the CoMeT [7] software that will be used to generate accurate benchmark execution data.

Chapter 3 states some general information about Markov chains and queueing theory but also introduces a special type of queueing network: a closed queueing network. While finding the invariant distribution of such a network is easy, it is computationally infeasible to calculate. However, the performance measures of a closed queueing network, such as throughput and waiting time, can also be calculated without knowledge of the equilibrium distribution by using Mean Value Analysis (MVA) [6].

Chapter 4 simplifies the core-DRAM interaction and turns it into a closed queueing network. By mapping the performance measures of a closed queueing network to observable data, it is possible to state three criteria that must be satisfied in order to ensure that the model correctly represents the core-DRAM interaction. The chapter ends by specifying a method to calculate the model parameters from benchmark execution data.

Chapter 5 discusses the accuracy of the model and determines the best value for the two hyperparameters that are used in the parameter derivation method from Chapter 4.

Chapters 4 and 5 contain many diagrams and illustrations that are generated using code from `https://github.com/Erikwl/Processor-modelling`. The Github repository also contains a `c++` and Python implementation of mean value analysis.

1.5 Ethics

It will have an overall positive effect on the world if an accurate model can be found. This is because an accurate model can save computational resources and thus reduce carbon dioxide emissions.

2 Benchmark execution on a computer

A computer is an electronic device that can execute (computer) instructions. For this thesis, the two most important parts of the computer are the Dynamic Random-Access Memory (DRAM) and the Central Processor Unit (CPU).

2.1 The CPU, DRAM and databus

The CPU orchestrates the execution of computer programs and consists of independently operating CPU cores. Most CPUs have 4, 8, or 16 cores, but there are also CPUs known with hundreds of cores. Each core can independently execute a list of instructions. The instructions either consist of a calculation, like adding two numbers, or a memory request. A memory request wants to either read or write a value at a specific memory address. The memory request will travel through the computer system in order to find the memory unit which contains the memory address. The memory request first visits the memory within the core to see if the memory address is contained in one of those memory units. Most of the time, the memory address can be found here and the request is served quickly. However, if the memory address cannot be found in the core, then the request must leave the core and travel to the shared memory unit: the DRAM. In order to do this, the request must travel to one of the DRAM controllers. The DRAM controllers are located in the cores and only a subset of the cores have a DRAM controller. The DRAM controller can then serve the request because it can access the DRAM. A write request will only write a value into the DRAM and then disappear, while a read request will send the data at the memory address back to the core.

The DRAM controllers can only serve a finite number of requests at once, so it can occur that a request must wait before it is served. This means that more active cores cause an increase in the DRAM controller waiting time. The measure of the amount of data a component can handle is called bandwidth. The bandwidth is measured in bytes per second (B/s).

2.2 Benchmarks

We will make a distinction between tasks and programs. A list of instructions for one core will be called a task, while a set of tasks will be called a program. Thus, a task executes on one core, while a program requires multiple cores.

There are different types of programs, but for the sake of repeatability, we will only discuss deterministic programs. Thus, we do not consider programs that require input from the user. Furthermore, in the field of parallel computing, resource-intensive programs

benchmark	execution time (ms)	number of cores	average DRAM utilization
`blackscholes`	178	2	1%
`bodytrack`	962	3	12%
`dedup`	2789	4	18%
`fluidanimate`	1444	2	1%
`streamcluster`	1015	2	28%
`swaptions`	433	2	1%

Table 2.1: Listed is information about the benchmarks in the PARSEC benchmark suite when executed on the least possible number of cores.

Parameter	Value
Number of DRAM controllers	1
DRAM controller bandwidth	7,110 GB/s
DRAM request size	64 B
DRAM controller capacity	1 request
DRAM controller service time	9 ns
DRAM service time	45 ns
Number of cores	16

Table 2.2: Configuration settings.

can be interesting to execute due to the fact that they can measure the performance of the memory and CPU. Those programs are called benchmarks and Princeton Application Repository for Shared-Memory Computers (PARSEC) [3] is a benchmark suite we will use to validate the model. The different benchmarks with their specifications are listed in Table 2.1, where the average DRAM utilization denotes the average percentage of the DRAM bandwidth that is used by the benchmark. The information in the table concerns the execution of each benchmark on the lowest number of possible cores, e.g. the `dedup` benchmark can be executed on $4, 7, 10, 13$ or 16 cores, but only the results are shown for the execution on 4 cores. Note that the execution time and average DRAM utilization depend on the configuration of the system. Some of the chosen configuration settings are listed in Table 2.2. The DRAM controller capacity denotes the number of requests a DRAM controller can handle simultaneously.

It is important to know that the value for the DRAM controller service time cannot be chosen but results from the values of the DRAM controller bandwidth, DRAM request size, and the number of requests a DRAM controller can serve simultaneously. The DRAM controller service time is then given by

$$\text{DRAM controller service time } = \frac{\text{capacity } \times \text{ request size}}{\text{bandwidth}} \approx 9 \text{ ns}.$$

The data in Table 2.1 is generated using the CoMeT [7] simulation software. This software is designed for accurate thermal simulations but also keeps track of much more simulation data, such as core utilization and the total number of DRAM requests.

3 Queueing theory and Markov chains

This chapter discusses relevant prior knowledge concerning Markov chains and queueing theory. Markov chains are simple mathematical models that are able to describe random phenomena evolving in time. In the field of Markov chains, the number of possible states of the system can be infinite, but we only care about finite state spaces because the proposed queueing network model that models a processor is finite.

There are many different types of Markov chains, but we will only discuss continuous-time Markov chains because those Markov chains are the most relevant when discussing queueing theory. We will start by discussing some basic properties and definitions of continuous-time Markov chains before we apply the theory to the relevant networks within queueing theory: the closed queueing networks.

3.1 Continuous-time Markov chains

This section introduces continuous-time Markov chains and discusses some basic properties. We will also state some theorems, but before we do that, we must first give the definition of a state space and a Q-matrix.

Definition 3.1.1 (State space)**.** *A state space I is a countable set. The elements of I are called states.*

Definition 3.1.2 (Q-matrix)**.** *Let I be a state space. A Q-matrix on I is a matrix $Q = (q_{ij} : i, j \in I)$ for which the following conditions hold:*

- $q_{ij} \geq 0$ *for all* $i \neq j$*;*
- $0 \leq -q_{ii} < \infty$ *for all* $i \in I$*;*
- $\sum_{j \in I} q_{ij} = 0$ *for all* $i \in I$*.*

A Q-matrix, state space, and initial distribution $\lambda = (\lambda_i : i \in I)$ induce a continuous-time Markov chain.

Definition 3.1.3. *(Continuous-time Markov chain) A process $(X_t)_{t \geq 0}$ is called a continuous-time Markov chain (CTMC) and denoted by $X_t \sim \text{Markov}(\lambda, Q)$ if for time t and state i it holds that*

$$\mathbb{P}[X_t = i] = \left(\lambda e^{tQ}\right)_i.$$

Defining a CTMC in such a way may seem arbitrary at first, but the following properties can be shown for this definition:

- The time it takes for the system to leave state i is $E(-q_{ii})$ (exponentially with parameter $-q_{ii}$) distributed.
- The probability of transitioning from state i to state $j \neq i$ is 0 if $q_{ii} = 0$ and $\frac{q_{ij}}{-q_{ii}}$ if $q_{ii} \neq 0$.
- The process is memoryless: the future probability distribution over the state space only depends on the current probability distribution over the state space. This is caused by the memoryless property of the exponential distribution.

Those properties give rise to the following mapping of a CTMC to a closed queueing network with one customer traveling between the different service centers:

- A state maps to a service center.
- The system being in state i with probability p maps to the customer being in center i with probability p.

Let $P = (p_{ij} : i, j \in I)$ be the transition matrix between the states of the CTMC, such that p_{ij} corresponds to the probability of transitioning from state i to j. It thus for all $i, j \in I$ that

$$p_{ij} = \begin{cases} \frac{q_{ij}}{-q_{ii}} & : j \neq i, q_{ii} \neq 0 \\ 0 & : j \neq i, q_{ii} = 0 \end{cases}, \quad p_{ii} = \begin{cases} 0 & : q_{ii} \neq 0 \\ 1 & : q_{ii} = 0 \end{cases}.$$

The earlier stated properties of a CTMC induce the following properties for a closed queueing network with one customer:

- Service center i has service rate $-q_{ii}$ (the service time is $E(-q_{ii})$ distributed). The service of the customer will never be completed if $q_{ii} = 0$.
- When the customer has been serviced by center i, it travels with probability $\frac{q_{ij}}{-q_{ii}}$ to center $j \neq i$.

Figure 3.1 illustrates the equivalent definition. Observe that the service rates and transition probabilities induce the Q-matrix and transition matrix that are given by

$$Q = \begin{pmatrix} -4 & 0 & 4 \\ 3 & -3 & 0 \\ 2 & 6 & -8 \end{pmatrix}, \quad P = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0.25 & 0.75 & 0 \end{pmatrix}.$$

The advantage of thinking about a CTMC as a network of service centers is that the network can be extended by adding more customers and by adding 'interference' between the customers. Some of those extensions are discussed in Section 3.2.

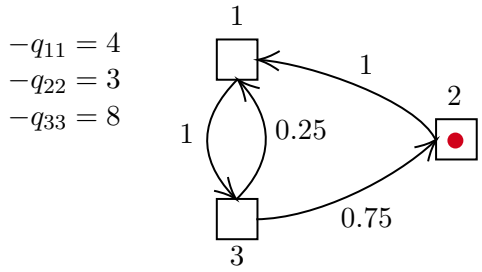


Figure 3.1: An example of a CTMC illustrated as a network of service centers and a customer (red dot) travelling through the network. The service rates of the service centers are given by $-q_{ii}$ for service center i and the numbers on the arrows indicate transition probabilities for the customer after the service has been completed at a service center. The current distribution of the system is given by $\lambda = (0, 1, 0)$.

3.1.1 The invariant distribution

For a process evolving in time, such as a CTMC, the equilibrium distribution is often a topic of interest. Definition 3.1.4 formalizes this concept.

Definition 3.1.4 (Invariant distribution)**.** *A distribution $\pi = (\pi_i : i \in I)$ is an invariant distribution over a Q-matrix Q if*

$$\pi Q = 0.$$

Defining an invariant distribution like this makes sense because $\pi Q = 0$ implies that

$$\begin{aligned}
\mathbb{P}[X_t = i] &= \left(\pi e^{tQ}\right)_i \\
&= \left(\pi \left(Q^0 + tQ + \frac{t^2 Q^2}{2!} + \dots\right)\right)_i \\
&= \left(\pi + t\pi Q + \frac{t^2 \pi Q^2}{2!} + \dots\right)_i \\
&= \pi_i,
\end{aligned}$$

which means that the distribution over the state space is invariant with evolving time. Note that the condition $\pi Q = 0$ is equivalent to

$$\pi_i \sum_{j \in I \setminus \{i\}} q_{ij} = \sum_{j \in I \setminus \{i\}} \pi_j q_{ji}, \quad i \in I, \tag{3.1}$$

due to the fact that $-q_{ii} = \sum_{j \in I \setminus \{i\}} q_{ij}$. Equation 3.1 will be referred to as the balance equation.

3.1.2 Convergence to equilibrium

A question one may ask is which initial distributions converge to the invariant distribution when time goes to infinity. There exists a convergence theorem, but we need the definition of irreducibility in order to formulate the theorem.

Definition 3.1.5 (Irreducible)**.** *A CTMC is irreducible if for all states $i, j \in I$ there exist states $i_1, \ldots, i_n$ such that $p_{i,i_1} p_{i_1,i_2} \cdots p_{i_n,j} > 0$.*

Irreducibility thus implies that you can travel from any state to any other state. Using this definition, we can formulate Theorem 3.1.6.

Theorem 3.1.6 (Convergence to equilibrium)**.** *Let $X_t \sim \text{Markov}(\lambda, Q)$ be irreducible, Q be finite and having invariant distribution π. Then*

$$\mathbb{P}[X_t = i] \to \pi_i \quad as \quad t \to \infty.$$

Corollary 3.1.7. *Let $X_t \sim \text{Markov}(\lambda, Q)$ be irreducible, Q be finite and having invariant distribution π. Then there is a unique invariant distribution.*

3.2 Closed Queueing networks

This section discusses closed queueing networks and their relations with CTMCs. Remember from Section 3.1 that a CTMC can be thought of as a network of service centers with one customer travelling through the network. If you let multiple customers of different classes travel through the network, you obtain a closed queueing network. To be more precise, a closed queueing network has a finite set of service centers, a finite set of customer classes, and a finite number of customers of each class. This section discusses the different components of a closed queueing network and states the invariant distribution. However, we will first introduce some notations before we formally define a closed queueing network.

Let I and R be respectively the number of service centers and customer classes. For convenience, we will index the service centers from 1 to M and we let $1 \leq i, j \leq M$ denote service centers while letting $1 \leq r, s \leq R$ denote customer classes. Let $\mathbf{N} = (N_1, \ldots, N_R)$ denote the population vector where N_r corresponds to the total number of customers of class r.

3.2.1 The customers

The customers travel through the network according to transition probabilities. Those transition probabilities are the same for all customers within the same class. Let $p_{i,j}^{(r)}$ denote the transition probability of a customer of class r to travel from service center i to j. For simplicity, we assume $p_{i,i}^{(r)} = 0$.

3.2.2 The service centers

All service centers are exhibiting the First In First Out (FIFO) queueing discipline, with each center having multiple servers. This means that a customer arriving at a service station is sent to the back of the queue and each customer moves up a spot when a customer leaves the service center. Furthermore, the first k customers in the queue are served by the service center if the service center has k servers. The queue contains both

the non-served and served customers and the queue of service center i will be called queue i. The number of customers center i can serve simultaneously is denoted by C_i and will be called the capacity of center i. The service rate of a customer does not depend on its class. Here, a service rate of μ means that the service time is $E(\mu)$ distributed.

For n customers present at center i, let $\phi_i(n)$ and $\phi_{i,k}(n)$ be respectively the total service rate of center i and the service rate that the customer at position k in queue i receives. The total service rate is equally divided between the first C_i customers in the queue, so we have

$$\phi_{i,k}(n) = \begin{cases} \frac{\phi_i(n)}{\min\{C_i,n\}} & : k \leq \min\{C_i, n\} \\ 0 & : \text{else} \end{cases}.$$

3.2.3 The state of the network

The state of the network is denoted by

$$\boldsymbol{x} = (\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_M),$$

where

$$\mathbf{x}_i = \left(x_{i,1}, x_{i,2}, \ldots, x_{i,n_i(\boldsymbol{x})}\right).$$

denotes the state of service center i. Here, $n_i(\boldsymbol{x}) \coloneqq |\mathbf{x}_i|$ equals the number of customers in queue i and $x_{i,k}$ is the class of the customer at the k-th position in the queue. Formally, the state space is given by

$$\chi \coloneqq \left\{ \boldsymbol{x} : \boldsymbol{x} = (\mathbf{x}_1, \ldots, \mathbf{x}_M), 1 \leq x_{i,k} \leq R, 1 \leq i \leq M, 1 \leq k \leq n_i(\boldsymbol{x}), \sum_{i=1}^{M} \sum_{k=1}^{n_i(x)} \delta_{r,x_{i,k}} = N_r \right\},$$

where the last condition ensures that exactly N_r class r customers are in the network and

$$\delta_{i,j} = \begin{cases} 1 & : i = j \\ 0 & : i \neq j \end{cases},$$

denotes the Dirac delta function.

Notice how the transition probabilities $p_{i,j}^{(r)}$ and service rate functions $\phi_{i,k}(n)$ induce transition rates between the different states. This means that for each closed queueing network there is an equivalent CTMC. The only non-zero transition rates are between the states where exactly one customer from queue i travels to the back of queue $j \neq i$. This means that the Q-matrix is given by $Q = \left(q_{\boldsymbol{x},\boldsymbol{y}} : \boldsymbol{x}, \boldsymbol{y} \in \chi\right)$, where

$$q_{\boldsymbol{x}\boldsymbol{y}} = p_{i,j}^{(x_{i,k})} \phi_{i,k}(n_i(\boldsymbol{x})), \quad j \neq i,$$

and $\boldsymbol{y} \in \chi$ is the state that is created from $\boldsymbol{x} \in \chi$ when the customer at position k in queue i is served and moves to spot $n_j(\boldsymbol{x}) + 1$ in queue $j \neq i$.

3.2.4 Aggregate states

The exact order of the customers in the queue is often not important to know, but only how many customers of each class are in each queue. This is what aggregate states describe. An aggregate state is denoted by $\boldsymbol{n} = (\mathbf{n}_1, \ldots, \mathbf{n}_M)$ with $\mathbf{n}_i = (n_{i,1}, \ldots, n_{i,R})$ where $n_{i,r}$ denotes the number of class r customers at service center i. The aggregate state space for a population vector $\mathbf{N}$ is defined by

$$\mathcal{N}(\mathbf{N}) \coloneqq \left\{ \boldsymbol{n} : \boldsymbol{n} = (\mathbf{n}_1, \ldots, \mathbf{n}_M), n_{i,r} \geq 0, 1 \leq i \leq M, 1 \leq r \leq R, \sum_{j=1}^{M} n_{j,r} = N_r \right\}.$$

Just like for states $\boldsymbol{x} \in \chi$, we will use $n_i(\boldsymbol{n})$ to denote the total number of customers at service center i. Thus we have

$$n_i(\boldsymbol{n}) = \sum_{r=1}^{M} n_{i,r}.$$

3.2.5 The invariant distribution

Remember that the invariant distribution $\pi = (\pi_x : \boldsymbol{x} \in \chi)$ for a CTMC can be found by solving the balance equations. Before presenting the solution to the balance equations we must first introduce the **one customer invariant measure terms**.

Definition 3.2.1. *The **one customer invariant measure terms** are given by*

$$e = (e_{i,r} : 1 \leq i \leq M, 1 \leq r \leq R),$$

such that the equations

$$\sum_{j=1}^{M} e_{j,r} p_{j,i}^{(r)} = e_{i,r}, \quad 1 \leq i \leq M, \quad 1 \leq r \leq R, \tag{3.2}$$

are satisfied.

In order to conclude using Theorem 3.1.6 that there exists at most one invariant distribution, we make the following assumptions about closed queueing networks:

- When in equilibrium, all class r customers visit the same set of centers and we will denote this set by $M(r)$, and this set is formally defined by

  $$M(r) \coloneqq \{i : e_{i,r} \neq 0\}.$$

  Similarly, we can define $R(i)$ to be the set of customer classes that visit center i, and this is formally defined by

  $$R(i) \coloneqq \{r : e_{i,r} \neq 0\}.$$

- The transition matrix $P^{(r)} = \left(p_{i,j}^{(r)} : 1 \leq i, j \leq M\right)$ is irreducible over $M(r)$.

Theorem 3.2.2 gives the general solution to the balance equations for a closed queueing network. The theorem originates from [2].

Theorem 3.2.2. *For a closed queueing network it holds that the unique invariant distribution $\pi = (\pi_{\boldsymbol{x}} : \boldsymbol{x} \in \chi)$ is given by*

$$\pi_{\boldsymbol{x}} = \frac{1}{G(\mathbf{N})} \prod_{i=1}^{M} \prod_{k=1}^{n_i(\boldsymbol{x})} \frac{e_{i,x_{i,k}}}{\phi_i(k)}, \tag{3.3}$$

where $G(\mathbf{N})$ is a normalising constant such that $\sum_{\boldsymbol{x}\in\chi} \pi_{\boldsymbol{x}} = 1$, i.e.

$$G(\mathbf{N}) = \sum_{\boldsymbol{x}\in\chi} \prod_{i=1}^{M} \prod_{k=1}^{n_i(\boldsymbol{x})} \frac{e_{i,x_{i,k}}}{\phi_i(k)}.$$

Proof. Uniqueness follows using Corollary 3.1.7 from the fact that the transition matrix $P^{(r)}$ is irreducible over $M(r)$. The theorem can now be proven by showing that π satisfies the balance equations

$$\pi_{\boldsymbol{x}} \sum_{\boldsymbol{y}\in\chi\setminus\{\boldsymbol{x}\}} q_{\boldsymbol{x}\boldsymbol{y}} = \sum_{\boldsymbol{y}\in\chi\setminus\{\boldsymbol{x}\}} \pi_{\boldsymbol{y}} q_{\boldsymbol{y}\boldsymbol{x}}, \quad \boldsymbol{x} \in \chi.$$

For a state $\boldsymbol{x} \in \chi$ it holds that the only non-zero transition rates $q_{\boldsymbol{x}\boldsymbol{y}}$ correspond to the transition of exactly one customer moving to another service center. This means that the left-hand side of the balance equations can be calculated by summing over all tuples (i, j, k) corresponding to the transition of the customer at position k of queue i to queue j, and thus we have

$$\begin{aligned}
\pi_{\boldsymbol{x}} \sum_{\boldsymbol{y}\in\chi} q_{\boldsymbol{x}\boldsymbol{y}} &= \pi_{\boldsymbol{x}} \sum_{i=1}^{M} \sum_{k=1}^{n_i(\boldsymbol{x})} \sum_{j=1}^{M} \phi_{i,k}(n_i(\boldsymbol{x})) p_{i,j}^{(x_{i,k})} \\
&= \pi_{\boldsymbol{x}} \sum_{i=1}^{M} \sum_{k=1}^{n_i(\boldsymbol{x})} \phi_{i,k}(n_i(\boldsymbol{x})) \sum_{j=1}^{M} p_{i,j}^{(x_{i,k})} \\
&= \pi_{\boldsymbol{x}} \sum_{i=1}^{M} \sum_{k=1}^{n_i(\boldsymbol{x})} \phi_{i,k}(n_i(\boldsymbol{x})) \\
&= \pi_{\boldsymbol{x}} \sum_{i=1}^{M} \phi_i(n_i(\boldsymbol{x})).
\end{aligned}$$

In order to calculate the right-hand side of the balance equations, we define

$$T_{\boldsymbol{x}} \coloneqq \left\{(k, j, i) \in \mathbb{N}^3 : 1 \le i, j \le M, 1 \le k \le n_j(\boldsymbol{x}) + 1, n_i(\boldsymbol{x}) \ge 1\right\},$$

as the set of all possible transitions that 'result in state $\boldsymbol{x}$'. Here, (k, j, i) corresponds to the k-th customer of service center j moving to service center i, such that the resulting

state is state $\boldsymbol{x}$. Notice how $n_i(\boldsymbol{x}) \geq 1$ is a necessary condition because no transition to center i can occur if there is no customer at center i in state $\boldsymbol{x}$.

Now for $\boldsymbol{x} \in \chi$ and $(k, j, i) \in T_x$ let $\boldsymbol{y} \in \chi$ correspond to the state such that the k-th customer moving center j to center i results in state $\boldsymbol{x}$. Then in state $\boldsymbol{y}$ there is one more customer in center j and one less in center i, thus

$$\pi_{\boldsymbol{y}} = \pi_{\boldsymbol{x}} \frac{e_{j,x_{i,n_i(\boldsymbol{x})}}}{\phi_j(n_j(\boldsymbol{x})+1)} \frac{1}{\frac{e_i,x_{i,n_i(\boldsymbol{x})}}{\phi_i(n_i(\boldsymbol{x}))}} = \pi_{\boldsymbol{x}} \frac{e_{j,x_{i,n_i(\boldsymbol{x})}}}{\phi_j(n_j(\boldsymbol{x})+1)} \frac{\phi_i(n_i(\boldsymbol{x}))}{e_i, x_{i,n_i(\boldsymbol{x})}}.$$

The right-hand side of the balance equations can now be calculated by summing over all elements of T_x:

$$\begin{aligned}
\sum_{\boldsymbol{y}\in\chi} \pi_{\boldsymbol{y}} q_{\boldsymbol{y}\boldsymbol{x}} &= \pi_{\boldsymbol{x}} \sum_{i=1,n_i(\boldsymbol{x})\geq 1}^{M} \sum_{j=1}^{M} \sum_{k=1}^{n_j(\boldsymbol{x})+1} \frac{e_{j,x_{i,n_i(\boldsymbol{x})}}}{\phi_j(n_j(\boldsymbol{x})+1)} \frac{\phi_i(n_i(\boldsymbol{x}))}{e_i, x_{i,n_i(\boldsymbol{x})}} p_{j,i}^{(x_{i,n_i(\boldsymbol{x})})} \phi_{j,k}(n_j(\boldsymbol{x})+1) \\
&= \pi_{\boldsymbol{x}} \sum_{i=1,n_i(\boldsymbol{x})\geq 1}^{M} \frac{\phi_i(n_i(\boldsymbol{x}))}{e_i, x_{i,n_i(\boldsymbol{x})}} \sum_{j=1}^{M} \frac{e_{j,x_{i,n_i(\boldsymbol{x})}}}{\phi_j(n_j(\boldsymbol{x})+1)} p_{j,i}^{(x_{i,n_i(\boldsymbol{x})})} \sum_{k=1}^{n_j(\boldsymbol{x})+1} \phi_{j,k}(n_j(\boldsymbol{x})+1) \\
&= \pi_{\boldsymbol{x}} \sum_{i=1,n_i(\boldsymbol{x})\geq 1}^{M} \frac{\phi_i(n_i(\boldsymbol{x}))}{e_i, x_{i,n_i(\boldsymbol{x})}} \sum_{j=1}^{M} \frac{e_{j,x_{i,n_i(\boldsymbol{x})}}}{\phi_j(n_j(\boldsymbol{x})+1)} p_{j,i}^{(x_{i,n_i(\boldsymbol{x})})} \phi_j(n_j(\boldsymbol{x})+1) \\
&= \pi_{\boldsymbol{x}} \sum_{i=1,n_i(\boldsymbol{x})\geq 1}^{M} \frac{\phi_i(n_i(\boldsymbol{x}))}{e_i, x_{i,n_i(\boldsymbol{x})}} \sum_{j=1}^{M} e_{j,x_{i,n_i(\boldsymbol{x})}} p_{j,i}^{(x_{i,n_i(\boldsymbol{x})})} \\
&= \pi_{\boldsymbol{x}} \sum_{i=1}^{M} \frac{\phi_i(n_i(\boldsymbol{x}))}{e_i, x_{i,n_i(\boldsymbol{x})}} e_{i,x_{i,n_i(\boldsymbol{x})}} \qquad \text{(by 3.2)} \\
&= \pi_{\boldsymbol{x}} \sum_{i=1}^{M} \phi_i(n_i(\boldsymbol{x})),
\end{aligned}$$

which is equal to the left-hand side of the balance equations. This implies that $\pi = (\pi_{\boldsymbol{x}} : \boldsymbol{x} \in \chi)$ is the unique invariant distribution. □

The invariant distribution of Equation 3.3 enables us to calculate the equilibrium state probabilities of the aggregate states in $\mathcal{N}(\mathbf{N})$. Theorem 3.2.3 states the invariant distribution of the aggregate states in $\mathcal{N}(\mathbf{N})$. The theorem originates from [2].

Theorem 3.2.3. *The equilibrium state probabilities for $\boldsymbol{n} \in \mathcal{N}(\mathbf{N})$ are given by*

$$\pi_{\boldsymbol{n}} = \left[\prod_{i=1}^{N} \binom{n_i}{n_{i,1}, \ldots, n_{i,R}} \right] \pi_{\boldsymbol{x}},$$

where $\boldsymbol{x} \in \chi$ is any state that induces the aggregate state $\boldsymbol{n}$.

Proof. For $\boldsymbol{n} \in \mathcal{N}(\mathbf{N})$ define $\chi_{\boldsymbol{n}}$ as the set of states that induce the aggregate state $\boldsymbol{n}$. Formally, this is defined as

$$\chi_{\boldsymbol{n}} \coloneqq \left\{ \boldsymbol{x} \in \chi \ : \ \sum_{k=1}^{n_i(\boldsymbol{n})} \delta_{r,x_{i,k}} = n_{i,r}, 1 \leq i \leq M, 1 \leq r \leq R \right\}.$$

Notice how

$$|\chi_{\boldsymbol{n}}| = \prod_{i=1}^{M} \binom{n_i(\boldsymbol{n})}{n_{i,1}, \ldots, n_{i,R}},$$

because there are

$$\binom{n_i(\boldsymbol{n})}{n_{i,1}, \ldots, n_{i,R}}$$

different queues possible at center i if the number of customers of each class at center i is given. For $\boldsymbol{x} \in \chi_{\boldsymbol{n}}$ and service center i we have that the expression

$$\prod_{k=1}^{n_i(\boldsymbol{n})} e_{i,x_{i,k}} = \prod_{r=1}^{R} e_{i,n_{i,r}},$$

only depends on $\boldsymbol{n}$ and not on the exact element within $\chi_{\boldsymbol{n}}$. This implies that $\pi_{\boldsymbol{x}}$ is the same for all $\boldsymbol{x} \in \chi_{\boldsymbol{n}}$. We can use this to obtain

$$\pi_{\boldsymbol{n}} = \sum_{\boldsymbol{x} \in \chi_{\boldsymbol{n}}} \pi_{\boldsymbol{x}} = |\chi_{\boldsymbol{n}}| \, \pi_{\boldsymbol{x}} = \left[\prod_{i=1}^{M} \binom{n_i(\boldsymbol{n})}{n_{i,1}, \ldots, n_{i,R}} \right] \pi_{\boldsymbol{x}}.$$

□

3.3 Mean value analysis

The calculation of the normalization constant $G(\mathbf{N})$ in Equation 3.3 requires a summation over all possible states. Due to the extremely large number of possible states, this is computationally infeasible. However, most of the time the exact invariant distribution is not of interest and only the performance measures (e.g. utilization, throughput) of a closed queueing network are of interest. As it turns out, the calculation of those performance measures does not require the computation of the normalization constant. An efficient algorithm to calculate the performance measures was found in 1980 by [6]. The algorithm is called Mean Value Analysis and this section is devoted to explaining and proving the correctness of the algorithm. The algorithm is only applicable to closed queueing networks, but it was later extended by [4] to also allow open customer classes. We will only consider closed queueing networks in this thesis, so the theorems and results in this section originate from [6].

Section 3.3.1 gives an overview of the algorithm and the performance measures, while Section 3.3.2 introduces some notations. Section 3.3.3 gives proofs for the relations between the performance measures. Section 3.3.4 states how the relations between the performance measures can be combined to formulate the MVA algorithm for Constant Rate (CR) closed queueing networks.

3.3.1 An overview

This section states the performance measures and the relations between them. The relations are proven in Section 3.3.3. We will use $\mathbf{N}'$ and $\mathbf{N}''$ to denote arbitrary population vectors and we use $\mathbf{N}'' \leq \mathbf{N}'$ to denote that $N_r'' \leq N_r'$ for all $1 \leq r \leq R$. MVA is a recursive algorithm and starts by calculating the performance measures for the population vector $\mathbf{0}$. The algorithm inductively adds customers until the desired population vector of $\mathbf{N}$ is obtained. For all $\mathbf{N}' \leq \mathbf{N}$ the MVA algorithm will calculate the following performance measures:

- The mean throughput $\overline{x}_{i,r}(\mathbf{N}')$ is the mean number of class r customers that leave center i in 1 unit of time.
- The utilization $u_i(\mathbf{N}')$ is the mean number of busy servers at service center i.
- The mean queue length $\overline{n}_i(\mathbf{N}')$ of center i.
- The mean waiting time $\overline{w}_{i,r}(\mathbf{N}')$ of class r customers at service center i.
- For $n \in \mathbb{N}$, the marginal queue length $p_i(n, \mathbf{N}')$ is the probability that there are exactly n customers at center i, while $p_i(\mathbf{N}'', \mathbf{N}')$ is the probability that the population vector at center i is given by $\mathbf{N}''$.

The recursion of the MVA algorithm is based on the following relations between the performance measures:

$$
\begin{aligned}
&\overline{n}_{i,r}(\mathbf{N}') = \overline{w}_{i,r}(\mathbf{N}')\overline{x}_{i,r}(\mathbf{N}'),\\
&p_i(n, \mathbf{N}') = \frac{1}{\phi_i(n)} \sum_{r=1}^{R} \overline{x}_{i,r}(\mathbf{N}') p_i(n-1, \mathbf{N}' - \mathbf{1}_r),\\
&\overline{w}_{i,r}(\mathbf{N}') = \frac{1}{C_i \mu_i} \left[1 + \overline{n}_i(\mathbf{N}' - \mathbf{1}_r) + \sum_{n=0}^{C_i - 2} (C_i - n - 1) p_i(n, \mathbf{N}' - \mathbf{1}_r) \right],\\
&u_i(\mathbf{N}') = \frac{1}{\mu_i} \sum_{r=1}^{R} \overline{x}_{i,r}(\mathbf{N}'),\\
&\overline{x}_{l^*(r),r}(\mathbf{N}') = \frac{N_r}{\sum_{i=1}^{M} e_{i,r} \overline{w}_{i,r}(\mathbf{N}')}.
\end{aligned}
$$

All equations, except for the first one, are proven in Section 3.3.3.

3.3.2 The normalization constant

Remember that the normalization constant for a population vector $\mathbf{N}'$ is given by

$$
G(\mathbf{N}') = \sum_{n \in \mathcal{N}(\mathbf{N}')} \prod_{i=1}^{M} f_i(\mathbf{n}_i),
$$

where

$$f_i(\mathbf{n}) \coloneqq C(\mathbf{n})E_i(\mathbf{n})\Phi_i(|\mathbf{n}|),$$

with

$$\begin{aligned}
|\mathbf{n}_i| &\coloneqq n_{i,1} + \ldots + n_{i,R},\\
C(\mathbf{n}_i) &\coloneqq \binom{|\mathbf{n}_i|}{n_{i,1}, \ldots, n_{i,R}},\\
E_i(\mathbf{n}_i) &\coloneqq \prod_{r=1}^{R} (e_{i,r})^{n_{i,r}},\\
\Phi_i(n) &\coloneqq \prod_{k=1}^{n} \frac{1}{\phi_i(k)}, \quad n \in \mathbb{N}.
\end{aligned}$$

For the sake of completeness, we define the normalization constant $G_i(\mathbf{N}'', \mathbf{N}')$ by

$$G_i(\mathbf{N}'', \mathbf{N}') \coloneqq \sum_{n \in \mathcal{N}_i(\mathbf{N}'', \mathbf{N}')} \prod_{j=1, j\neq i}^{M} f_j(\mathbf{n}_j),$$

where

$$\mathcal{N}_i(\mathbf{N}'', \mathbf{N}') \coloneqq \left\{ n \in \mathcal{N}(\mathbf{N}') : \mathbf{n}_i = \mathbf{N}'' \right\}.$$

The normalisation constant $G_i(\mathbf{N}'', \mathbf{N}')$ can be interpreted as fixing $\mathbf{N}''$ out of $\mathbf{N}'$ number of customers at service station i.

3.3.3 The relations between the performance measures

Little's law is an important law that we will use in a number of proofs and it relates the different performance measures.

Theorem 3.3.1 (Little's law)**.** *For a queueing network in equilibrium distribution, the mean number of customers in a component of the queueing network equals the product of the waiting time and throughput in that component of the network.*

Corollary 3.3.2. *We can either view the entire service center as a component or only the C_i servers in the center. For the first case, we have that*

$$\overline{n}_{i,r}(\mathbf{N}') = \overline{w}_{i,r}(\mathbf{N}')\overline{x}_{i,r}(\mathbf{N}'),$$

while for the second case, we obtain

$$u_i(\mathbf{N}') = \frac{1}{\mu_i} \sum_{r=1}^{R} \overline{x}_{i,r}(\mathbf{N}').$$

We will now continue by stating the relations between the different performance measures. Lemmas 3.3.3 and 3.3.4 state respectively an equation for the throughput and the marginal probability, while Theorem 3.3.5 states a recurrence for the mean waiting time. The notation $\mathbf{1}_r$ will be used to denote an R component vector whose components are zero except for the r-th component, which is 1.

Lemma 3.3.3. *For the throughput, it holds that*

$$\overline{x}_{i,r}(\mathbf{N}') = e_{i,r}\frac{G(\mathbf{N}'-\mathbf{1}_r)}{G(\mathbf{N}')}. \tag{3.4}$$

Lemma 3.3.4. *For the marginal queue length distribution, it holds that*

$$p_i(n,\mathbf{N}') = \frac{1}{\phi_i(n)}\sum_{r=1}^{R}\overline{x}_{i,r}(\mathbf{N}')p_i(n-1,\mathbf{N}'-\mathbf{1}_r), \quad (n\geq 0). \tag{3.5}$$

Proof. We have that

$$\begin{aligned}
p_i(\mathbf{N}'',\mathbf{N}') &= E_i(\mathbf{N}'')\Phi_i(|\mathbf{N}''|)C(\mathbf{N}'')\frac{G_i(\mathbf{N}'',\mathbf{N}')}{G(\mathbf{N}')}\\
&= \sum_{r=1}^{R} e_{i,r}E_i(\mathbf{N}''-\mathbf{1}_r)\frac{\Phi_i(|\mathbf{N}''|-1)}{\phi_i(|\mathbf{N}''|)}C(\mathbf{N}''-\mathbf{1}_r)\frac{G_i(\mathbf{N}''-\mathbf{1}_r,\mathbf{N}'-\mathbf{1}_r)}{G(\mathbf{N}'-\mathbf{1}_r)}\frac{G(\mathbf{N}'-\mathbf{1}_r)}{G(\mathbf{N}')}\\
&= \frac{1}{\phi_i(|\mathbf{N}''|)}\sum_{r=1}^{R} e_{i,r}p_i(\mathbf{N}''-\mathbf{1}_r,\mathbf{N}'-\mathbf{1}_r)\frac{G(\mathbf{N}'-\mathbf{1}_r)}{G(\mathbf{N}')}\\
&= \frac{1}{\phi_i(|\mathbf{N}''|)}\sum_{r=1}^{R}\overline{x}_{i,r}(\mathbf{N}')p_i(\mathbf{N}''-\mathbf{1}_r,\mathbf{N}'-\mathbf{1}_r).
\end{aligned}$$

Equation 3.5 is now obtained by taking a summation over all $\mathbf{N}''$ with $|\mathbf{N}''| = n$. □

Theorem 3.3.5. *A recurrence for the mean waiting time is given by*

$$\overline{w}_{i,r}(\mathbf{N}') = \sum_{n=1}^{|\mathbf{N}'|}\frac{n}{\phi_i(n)}p_i(n-1,\mathbf{N}'-\mathbf{1}_r)$$

Proof. The mean number of class r customers at center i is given by

$$\begin{aligned}
\overline{n}_{i,r}(\mathbf{N}') &= \sum_{\mathbf{N}''=\mathbf{0}}^{\mathbf{N}'} N_r''p_i(\mathbf{N}'',\mathbf{N}')\\
&= \sum_{\mathbf{N}''=\mathbf{1}_r}^{\mathbf{N}'} N_r''C(\mathbf{N}'')\Phi_i(|\mathbf{N}''|)E_i(\mathbf{N}'')\frac{G_i(\mathbf{N}'',\mathbf{N}')}{G(\mathbf{N}')}\\
&= \sum_{\mathbf{N}''=\mathbf{1}_r}^{\mathbf{N}'} |N''|C(\mathbf{N}''-\mathbf{1}_r)\frac{\Phi_i(|\mathbf{N}''|-1)}{\phi_i(|\mathbf{N}''|)}e_{i,r}E_i(\mathbf{N}''-\mathbf{1}_r)\frac{G_i(\mathbf{N}'-\mathbf{1}_r,\mathbf{N}''-\mathbf{1}_r)}{G(\mathbf{N}'-\mathbf{1}_r)}\frac{G(\mathbf{N}'-\mathbf{1}_r)}{G(\mathbf{N}')}\\
&= \sum_{\mathbf{N}''=\mathbf{1}_r}^{\mathbf{N}'}\frac{|N''|}{\phi_i(|\mathbf{N}''|)}\overline{x}_{i,r}(\mathbf{N}')p_i(\mathbf{N}''-\mathbf{1}_r,\mathbf{N}'-\mathbf{1}_r)\\
&= \overline{x}_{i,r}(\mathbf{N}')\sum_{n=1}^{|\mathbf{N}'|}\frac{n}{\phi_i(n)}p_i(n-1,\mathbf{N}'-\mathbf{1}_r).
\end{aligned}$$

By using Little's law we obtain

$$\overline{w}_{i,r}(\mathbf{N}') = \sum_{n=1}^{|\mathbf{N}'|} \frac{n}{\phi_i(n)} p_i(n-1, \mathbf{N}' - \mathbf{1}_r).$$

□

The waiting time equation in Theorem 3.3.5 can be simplified for a Constant Rate closed queueing network. In a CR network, service center i has C_i server with a service rate of μ_i. This is formally defined in Definition 3.3.6 and the equation from theorem 3.3.5 is simplified for CR closed queueing networks in Corollary 3.3.7.

Definition 3.3.6 (CR queueing network)**.** *A Constant Rate (CR) queueing network is a network for which holds that*

$$\phi_i(n) = \min\{n, C_i\}\,\mu_i, \quad i = 1, \ldots, M.$$

Corollary 3.3.7. *For a CR queueing network, we have*

$$\overline{w}_{i,r}(\mathbf{N}') = \frac{1}{C_i\mu_i}\left[1 + \overline{n}_i(\mathbf{N}' - \mathbf{1}_r) + \sum_{n=0}^{C_i-2}(C_i - n - 1)p_i(n, \mathbf{N}' - \mathbf{1}_r)\right].$$

Proof. For convenience we will use p_i to denote $p_i(n, \mathbf{N}' - \mathbf{1}_r)$. We then have that

$$\begin{aligned}
\overline{w}_{i,r}(\mathbf{N}') &= \sum_{n=1}^{\mathbf{N}'} \frac{n}{\phi_i(n)} p_i(n-1, \mathbf{N}' - \mathbf{1}_r) \\
&= \frac{1}{\mu_i}\left[\sum_{n=1}^{C_i} p_i(n-1, \mathbf{N}' - \mathbf{1}_r) + \frac{1}{C_i}\sum_{n=C_i+1}^{|\mathbf{N}'|}(n-1+1)p_i(n-1, \mathbf{N}' - \mathbf{1}_r)\right] \\
&= \frac{1}{\mu_i}\left[\sum_{n=0}^{C_i-1} p_i + \frac{1}{C_i}\sum_{n=C_i}^{|\mathbf{N}'-\mathbf{1}_r|} np_i + \frac{1}{C_i}\sum_{n=C_i}^{|\mathbf{N}'-\mathbf{1}_r|} p_i - \frac{1}{C_i}\sum_{n=0}^{C_i-1} np_i + \frac{1}{C_i}\sum_{n=0}^{C_i-1} np_i\right] \\
&= \frac{1}{\mu_i}\left[\frac{1}{C_i}\sum_{n=0}^{C_i-1} C_i p_i + \frac{1}{C_i}\sum_{n=0}^{|\mathbf{N}'-\mathbf{1}_r|} np_i + \frac{1}{C_i}\left(1 - \sum_{n=0}^{C_i-1} p_i\right) - \frac{1}{C_i}\sum_{n=0}^{C_i-1} np_i\right] \\
&= \frac{1}{C_i\mu_i}\left[1 + \overline{n}_i(\mathbf{N}' - \mathbf{1}_r) + \sum_{n=0}^{C_i-1}(C_i - n - 1)p_i\right].
\end{aligned}$$

□

Notice that from the throughput equation 3.7 and the fact that $e_{l^*(r),r} = 1$ we can deduce that

$$\overline{x}_{i,r}(\mathbf{N}') = e_{i,r}\overline{x}_{l^*(r),r}(\mathbf{N}').$$

We can apply this equation with Little's law to obtain

$$\overline{n}_{i,r}(\mathbf{N}') = \overline{x}_{i,r}(\mathbf{N}')\overline{w}_{i,r}(\mathbf{N}') = e_{i,r}\overline{x}_{l^*(r),r}(\mathbf{N}')\overline{w}_{i,r}(\mathbf{N}'), \tag{3.6}$$

Summing Little's law over all service centers gives us

$$N_r = \sum_{i=1}^{M} \overline{n}_{i,r}(\mathbf{N}') = \sum_{i=1}^{M} \overline{x}_{i,r}(\mathbf{N}')\overline{w}_{i,r}(\mathbf{N}'),$$

which implies that

$$\overline{x}_{l^*(r),r}(\mathbf{N}') = \frac{N_r}{\sum_{i=1}^{M} e_{i,r}\overline{w}_{i,r}(\mathbf{N}')}. \tag{3.7}$$

3.3.4 The algorithm

This section states the MVA algorithm for CR queueing networks. The algorithm contains two parts: the base case and the recursion. The recursion results from the relations between the performance measures, while the base case still needs to be stated. That is, we need to find the values of $\overline{x}_{i,r}(\mathbf{0})$, $\overline{n}_i(\mathbf{0})$, $p_i(0,\mathbf{0})$ and $p_i(0,\mathbf{N}')$. To decrease the computational complexity of the algorithm, we will choose a service center $l^*(r) \in M(r)$ such that and normalize $e_r = (e_{i,r} : i = 1,\ldots,M)$ such that $e_{l^*(r),r} = 1$.

The base case

We have that

$$p_i(0,\mathbf{0}) = 1, \quad \overline{n}_i(\mathbf{0}) = 0.$$

The value of $p_i(0,\mathbf{N}')$ is a bit more difficult to obtain, but it can be calculated using the fact that the mean number of idle servers is given by

$$\sum_{n=0}^{C_i-1} (C_i - n)p_i(n,\mathbf{N}') = C_i - u_i(\mathbf{N}'),$$

and the fact that the utilization can be expressed in terms of the throughput:

$$u_i(\mathbf{N}') = \frac{1}{\mu_i}\sum_{r=1}^{R} \overline{x}_{i,r}(\mathbf{N}'). \tag{3.8}$$

Combining these equations gives

$$p_i(0,\mathbf{N}') = 1 - \frac{1}{C_i}\left[u_i(\mathbf{N}') + \sum_{n=1}^{C_i-1}(C_i - n)p_i(n,\mathbf{N}').\right] \tag{3.9}$$

The pseudo code

The obtained results can then be combined to obtain algorithm 1.

Algorithm 1 Calculate the performance measures of a CR queueing network

Require: $e_{l^*(r),r} = 1, r = 1, 2, \ldots, R$

{initialization}
for $i = 1, 2, \ldots, M$ **do**
$\overline{n}_i(\mathbf{0}) \leftarrow 0$
$p_i(0, \mathbf{0}) \leftarrow 1$
end for

{iteration over all possible population vectors (the main loop)}
for $\mathbf{N}' \leq \mathbf{N}$ **do**

{mean waiting time and throughput}
for $r = 1, 2 \ldots, R$ **do**

{mean waiting time calculation using Corollary 3.3.7}
for $i \in M(r)$ **do**
$\overline{w}_{i,r}(\mathbf{N}') \leftarrow \frac{1}{C_i \mu_i} \left[1 + \overline{n}_i(\mathbf{N}' - \mathbf{1}_r) + \sum_{n=0}^{\min\{C_i - 2, |\mathbf{N}'| - 1\}} (C_i - n - 1) p_i(n, \mathbf{N}' - \mathbf{1}_r)\right]$
end for

{throughput calculation using Equation 3.7}
$\overline{x}_{l^*(r),r}(\mathbf{N}') \leftarrow \frac{N_r}{\sum_{i \in M(r)} e_{i,r} \overline{w}_{i,r}(\mathbf{N}')}$
end for

{mean queue length, utilization, and queue length distribution}
for $i = 1, 2, \ldots, M$ **do**

{mean queue length calculation using Equation 3.6}
$\overline{n}_i(\mathbf{N}') \leftarrow \sum_{r \in R(i)} e_{i,r} \overline{x}_{l^*(r),r}(\mathbf{N}') \overline{w}_{i,r}(\mathbf{N}')$

{utilization calculation using Equation 3.8}
$u_i(\mathbf{N}') \leftarrow \frac{1}{\mu_i} \sum_{r \in R(i)} \overline{x}_{l^*(r),r}(\mathbf{N}') e_{i,r}$

{queue length distribution calculation using Lemma 3.3.4 and Equation 3.9}
for $n = 1, 2, \ldots, \min\{C_i - 1, |\mathbf{N}'|\}$ **do**
$p_i(n, \mathbf{N}') \leftarrow \frac{1}{\phi_i(n)} \sum_{r \in R(i)} \overline{x}_{i,r}(\mathbf{N}') p_i(n - 1, \mathbf{N}' - \mathbf{1}_r)$
end for
$p_i(0, \mathbf{N}') \leftarrow 1 - \frac{1}{C_i} \left[u_i(\mathbf{N}') + \sum_{n=1}^{\min\{C_i - 1, |\mathbf{N}|\}} (C_i - n) p_i(n, \mathbf{N}')\right]$
end for
end for

Performance measure	Time complexity for a $\mathbf{N}' \leq \mathbf{N}$	Upper bound
Mean waiting time	$\mathcal{O}\left(\sum_{r=1}^{R} \sum_{i \in M(r)} C_i\right)$	$\mathcal{O}\left(RM'C'\right)$
Throughput	$\mathcal{O}\left(\sum_{r=1}^{R} \lvert M(r) \rvert\right)$	$\mathcal{O}\left(RM'\right)$
Mean queue length	$\mathcal{O}\left(\sum_{i=1}^{M} \lvert R(i) \rvert\right)$	$\mathcal{O}\left(R'M\right)$
Utilization	$\mathcal{O}\left(\sum_{i=1}^{M} \lvert R(i) \rvert\right)$	$\mathcal{O}\left(R'M\right)$
Queue length distribution	$\mathcal{O}\left(\sum_{i=1}^{M} \lvert C_i \rvert \lvert R(i) \rvert\right)$	$\mathcal{O}\left(R'MC'\right)$

Table 3.1: The time complexities of calculating the performance measures for a specific population vector $\mathbf{N}' \leq \mathbf{N}$. Concise upper bounds for the time complexities are also given.

The time complexity

In order to give a concise formula for the total time complexity, we will introduce the constants

$$\begin{aligned} M' &\coloneqq \max\left\{\lvert M(r) \rvert : r = 1, 2, \ldots, R\right\} \\ C' &\coloneqq \max\left\{C_i : i = 1, 2, \ldots, M\right\} \\ R' &\coloneqq \max\left\{\lvert R(i) \rvert : i = 1, 2, \ldots, M\right\}. \end{aligned}$$

We have that the main for-loop of the algorithm iterates over $\prod_{r=1}^{R} N_r$ possible population vectors $\mathbf{N}'$. Then for each population vector, the time complexities of calculating the performance measures are visible in Table 3.1.

This means that the total time complexity of the algorithm can be bounded above by

$$\mathcal{O}\left(N_1 N_2 \cdots N_R \max\left\{RM'C', R'MC'\right\}\right) \leq \mathcal{O}(N_1 N_2 \cdots N_R RMC').$$

This formula suggests that the execution time of the algorithm is largely determined by the number of customers of each class and not so much by the number of service centers or other parameters of the system. It can also be observed that even for networks with relatively small numbers of customer classes it can be computationally infeasible to calculate the performance measures.

4 Model description and analysis

The queueing theory from Chapter 3 can now be used to create a closed queueing network model which aims at imitating the core-DRAM interaction.

Section 4.1 describes the model and introduces three criteria that should be satisfied in order to ensure that the model correctly reproduces the behavior of the cores and DRAM, while Section 4.3 uses this analysis to state an algorithm for determining the model parameters of a benchmark. The presented algorithm is not able to satisfy all the criteria and this is partly due to two limitations of a closed queueing network: fixed model parameters and no priority system. However, there are other types of queueing networks that don't have those limitations and we will briefly discuss these networks in Section 4.4.

4.1 Model description and explanation

This section will begin by stating a general overview of the most important interactions in the cores and DRAM. A model is derived from those interactions. After this, the values of the one invariant measure terms are calculated and the performance measures from Section 3.3.1 are mapped to benchmark execution data. This mapping induces the throughput, waiting time, and average number criteria.

4.1.1 The model dynamics overview

As discussed in Section 2.1, most of the memory requests being sent out by the cores can be served by the memory unit in the core, but some memory requests cannot be served by the memory unit in the core, so they go to a DRAM controller. Since our field of interest concerns the DRAM, we do not incorporate the memory requests that can be served by the memory unit into our model. The dynamics induced by the service of DRAM read and write requests are visualized in Figure 4.1. The aim is to turn this interaction into a closed queueing network so we can apply the theorems and results from Section 3.2. The different components of the computer turn into service centers and the different DRAM requests turn into customers. We will discuss the service centers and customers of the model in Section 4.1.1.

The service centers and customers of the model

In order to decide which components we should incorporate into our model, we need to make a few observations and assumptions regarding Figure 4.1:

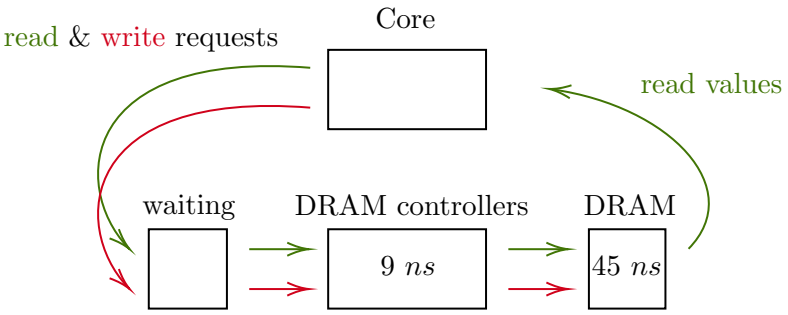


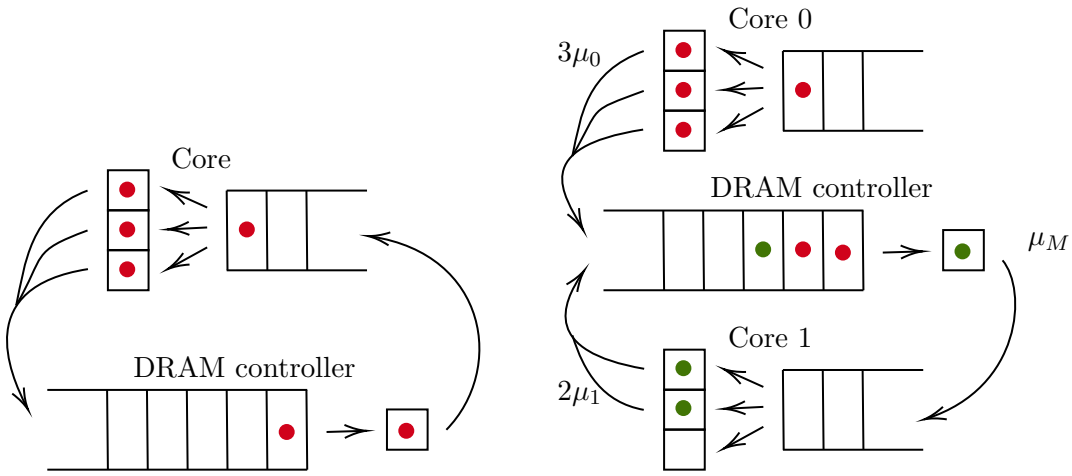
Figure 4.1: The path of DRAM read and write requests.

- The DRAM time is fixed and the waiting time only applies to the DRAM controllers. Since we wanted the model to be as simple as possible, we will leave out the DRAM.
- The read requests return to the core, while the write requests disappear. This is problematic because in a closed queueing network the customers don't leave the system. Therefore, in the model, the write requests also return to the core. Due to the fact that in practice a relatively small number of requests is a write request (6.7% for `bodytrack` and 13.3% for `blackscholes`), the model might still be accurate or useful. By letting the write requests also return to the core, there is no difference between read and write requests. This means it makes sense to talk about a request, without specifying the type of the request.
- For the chosen configuration (Section 2.2), the DRAM controller can only handle 1 request at once. This means that the DRAM controllers component from Figure 4.1 can be modeled as a queue with a capacity of 1 and a service time of 9 ns.
- We assume that the core functions as a queue. After a request has been served by the DRAM, it returns a value to the core. The core can then use that value to execute more instructions and thus sent out more requests.

These observations and assumptions give a simplified core-DRAM interaction (Figure 4.2a). In the figure, the red dots are customers and correspond to the DRAM requests. Note that this simplified model only contains the most crucial components and other components can easily be added to the model (such as the DRAM or the local memory inside the core) in order to make the model more realistic.

Note that the requests are recycled in the network. This means that the number of requests in the model is not necessarily equal to the number of DRAM requests that the core sends in a certain time period. For example, a core sending 10 DRAM requests in 200 ms can be modeled by letting one request travel through the network and letting this request visit the DRAM every 20 ms.

Multiple cores can easily be added to the network. The case of two cores is given in Figure 4.2b. The formal definition of the model is given in Definition 4.1.1. Service center M corresponds to the memory and centers $0 \leq i \leq M-1$ correspond to the cores.



(a) One core.

(b) Two cores and service rates.

Figure 4.2: The DRAM controller is modeled as a queue and the model does not distinguish between DRAM read and write requests. The red dots correspond to the DRAM requests. The system on the left contains one core, while the system on the right contains two cores and includes service rates. The service rate for the chosen configuration equals $\frac{1}{9}$, because the DRAM controller service time is 9 ns.

Definition 4.1.1. *The M-core single memory model is defined as the triple $(\mathbf{N}, \mu, C)$, where*

$$\mathbf{N} = (N_i \in \mathbb{N} : 0 \leq i \leq M-1)$$

is the population vector,

$$\mu = (\mu_i : 0 \leq i \leq M),$$

is the service rate vector and

$$C = (C_i : 0 \leq i \leq M)$$

is the capacity vector. There are $R = M$ classes of customers and

$$p_{i,M}^{(i)} = p_{M,i}^{(i)} = 1, \quad 0 \leq i \leq M-1.$$

4.1.2 The one customer invariant measure terms

For the terms $e_{i,r}$ it must hold that

$$e_{i,r} = \sum_{j=0}^{M} e_{j,r} p_{j,i}^{(r)} = \begin{cases} \sum_{j=0}^{M-1} e_{j,r} & : i = M \\ e_{M,r} & : i \neq M \end{cases},$$

which means that we can simply let $e_{M,i} = 1$, $e_{i,i} = 1$ $(0 \leq i \leq M-1)$, and all other terms are equal to 0. This means that

$$S(r) = \{M, r\}, \quad 0 \leq r \leq M-1,$$

and notice that the transition matrix $P^{(r)}$ is irreducible over this set. This directly implies that any $\boldsymbol{x} \in \chi$ with $x_{i,k} \neq i$ for $0 \leq i \leq M-1$ has 0 probability of occurring in equilibrium, because then $e_{i,x_{i,k}} = 0$. In other words, in equilibrium, there are only class i customers in core i, which is as it should be.

4.1.3 The performance criteria

Remember that the Mean Value Analysis (MVA) algorithm of Section 3.3 gives us a computationally feasible approach to measuring the performance of a closed queueing network. The performance measures can be mapped easily to some of the available execution data produced by the simulation software CoMeT. The values of $\overline{x}_{M,i}$ (throughput), $\overline{w}_{M,i}$ (waiting time) and $\overline{n}_{M,i}$ (mean number of customers) can be mapped to respectively the average number of served core-i-requests in 1 ns, the average access latency and the average number of core-i-requests in the DRAM. The mapping of these three performance measures induces Criteria 4.1.2, 4.1.3 and 4.1.4.

Criterion 4.1.2 (Throughput)**.** *The throughput criterion is given by*

$$\overline{x}_{M,i}(\mathbf{N}) = \frac{\textit{number of served core-i-requests by the DRAM}}{\textit{time (ns)}}$$

Criterion 4.1.3 (Waiting time)**.** *The waiting time criterion is given by*

$$\overline{w}_{M,i}(\mathbf{N}) = \textit{average DRAM access latency for core-i-requests in the DRAM.}$$

Criterion 4.1.4 (Average number)**.** *The average number criterion is given by*

$$\overline{n}_{M,i}(\mathbf{N}) = \textit{average number of core-i-requests in DRAM}$$

It should be noted that due to Little's law, $\overline{n}_{M,i} = \overline{w}_{M,i}\overline{x}_{M,i}$, we can only satisfy a maximum of two criteria. However, the third criterion should automatically be satisfied if it turns out that the DRAM is in equilibrium during the execution of a benchmark because then Little's law also applies to the DRAM during benchmark execution. Section 4.1.3 discusses whether Little's law holds in the DRAM.

It is worth emphasizing that prioritizing the fulfillment of the throughput criterion takes precedence over satisfying the waiting time or average number criteria. This is due to the fact that the throughput can be used to determine the total number of sent DRAM requests, which can then be used to track the progress of a benchmark.

In the remainder of this chapter, we will discuss how both the throughput and waiting time criteria can be fulfilled, but the same theory applies to fulfilling the throughput and the average number criteria.

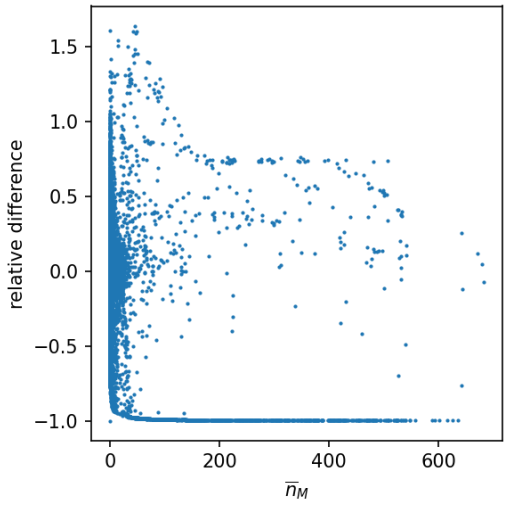


Figure 4.3: The value of $\overline{n}_M$ is plotted against the relative difference between $\overline{w}_M\overline{x}_M$ and $\overline{n}_M$ for the benchmark `bodytrack`. The total execution time of 960 ms is divided into 960000 time intervals of each 1000 ms and every point corresponds to a specific interval.

Little's law in the DRAM

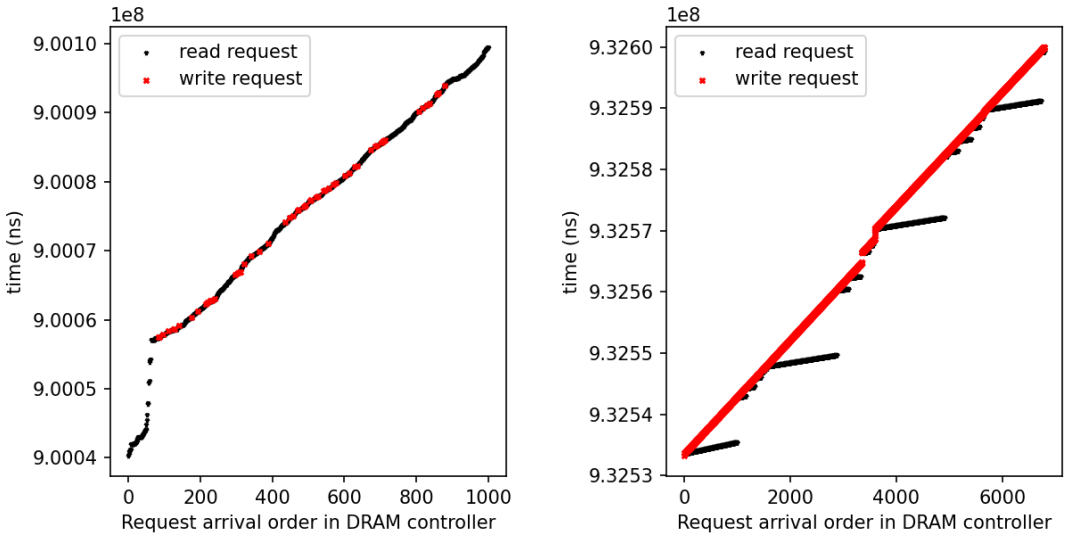
This section analyses whether Little's law holds in the DRAM during benchmark execution. Figure 4.3 shows the relative difference

$$\frac{\overline{x}_M\overline{w}_M - \overline{n}_M}{\overline{n}_M}$$

between $\overline{w}_M\overline{x}_M$ and $\overline{n}_M$, where $\overline{w}_M$, $\overline{x}_M$ and $\overline{n}_M$ denote respectively the combined average waiting time, throughput and average number of requests in the DRAM. The `bodytrack` benchmark was used to generate the plot and the total execution time of 960 ms was divided into 960000 time intervals of 1000 ms. Every point corresponds to a specific interval. According to Little's law, all points should be at $y = 0$, so Little's law does not hold for the DRAM during benchmark execution.

Figure 4.4 shows why Little's law does not hold for the DRAM controller. In the figure, the time of request creation by the core is plotted against the order in which the requests arrive in the DRAM controller. Figure 4.4a seems to be more in line which what you would assume: the later a request is created by the core, the later it arrives in the DRAM controller. However, Figure 4.4b contradicts this assumption.

To better understand what happens in Figure 4.4b, a detailed and zoomed-in version of the figure is shown in Figure 4.5. This figure suggests that a certain priority system is in place: the write requests have higher priority than the read requests. This means that some component in the computer chooses to delay the read requests, and thus prioritizes the write requests over the read requests.



(a) No priority. (b) Priority

Figure 4.4: The relation is shown between the time at which requests are created in the core and the order in which they arrive at the DRAM controller. The relation is shown for two different time intervals of the `bodytrack` benchmark.

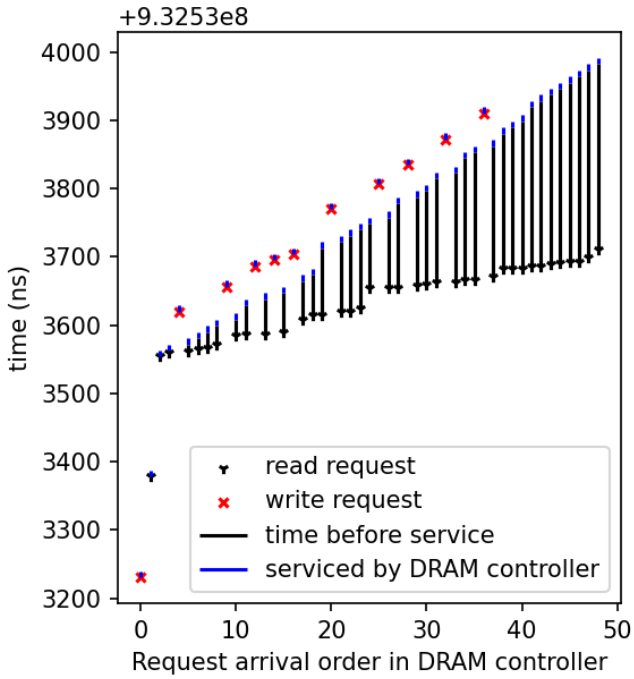
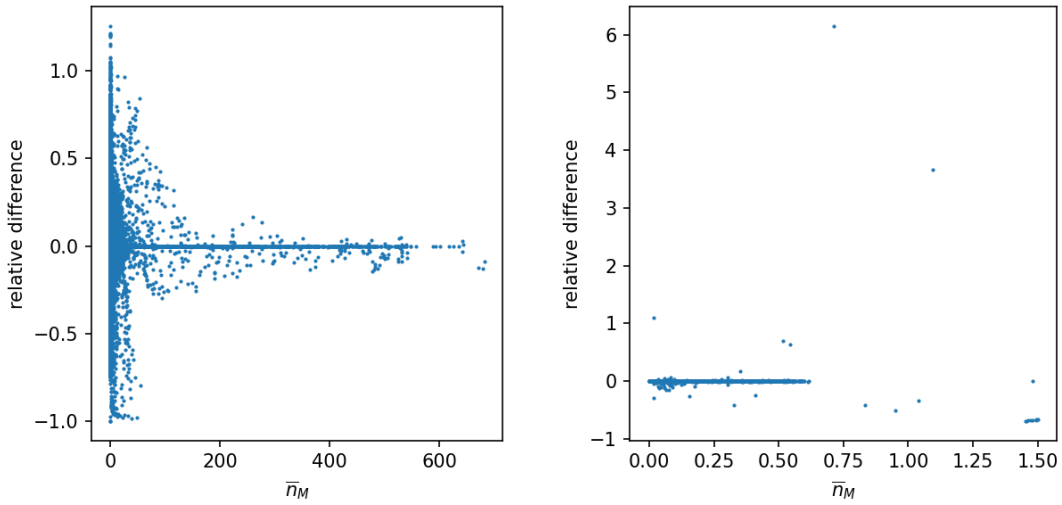


Figure 4.5: A detailed and zoomed in version of Figure 4.4b.



(a) Only read requests.

(b) Only write requests.

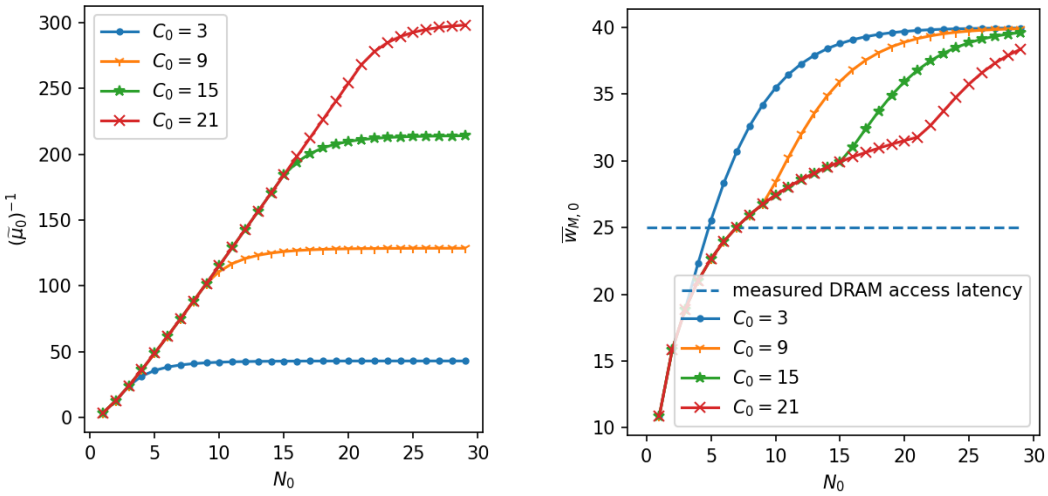
Figure 4.6: The value of $\overline{n}_M$ is plotted against the relative difference between $\overline{w}_M\overline{x}_M$ and $\overline{n}_M$ for read requests (left) and write requests (right). The total execution time of 960 ms is divided into 960000 time intervals of each 1000 ns and every point corresponds to a specific interval.

There are many reasons why a DRAM read request could be delayed. One of the possible reasons is write-read consistency: If a write and read request use the same memory address and the write instruction is executed in the core before the read instruction, then the write request is prioritized.

Distinguishing the requests by type results in Figure 4.6. From the figure can be deduced that Little's law seems to hold better when separating the requests by type. As can be seen, the law still doesn't hold for all intervals. The following reasons are violations of Little's law and thus can explain why Little's law doesn't hold for all intervals:

- A change of priority system occurs during the interval, e.g. all requests are treated equally from $0 - 500$ ns, while the write requests receive priority from $500 - 1000$ ns.
- The DRAM controller is not in equilibrium: more requests are arriving than departing.

To summarise, Little's law only holds for either read or write requests on intervals in equilibrium at which there is no change in the priority system. However, the model does not distinguish between read and write requests, so Little's law does not hold. This implies that we satisfy at most two criteria.



(a) The value of the service time $\frac{1}{\widetilde{\mu}_0}$.

(b) The DRAM controller waiting time $\overline{w}_{M,0}$.

Figure 4.7: The value of the core 0 service time $\frac{1}{\widetilde{\mu}_0}$ on the left and the value of the core 0 waiting time $\overline{w}_{M,0}$ for certain values of N_0, C_0 and $\widetilde{\mu}_0(N_0, C_0)$ on the right.

4.2 Model parameter analysis

This section discusses the influence of the model parameters on the DRAM throughput and waiting time.

4.2.1 The model parameters

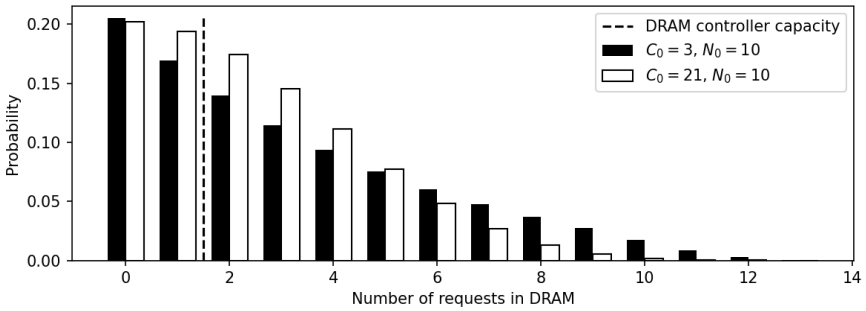
The service time $\frac{1}{\mu_M}$ and capacity C_M of the DRAM controllers are fixed and depend only on the configuration of the system. Since the configuration from Section 2.2 will be used, we must have that

$$\mu_M = \frac{1}{\text{DRAM controller service time}} = \frac{1}{9,010} = 0.111$$

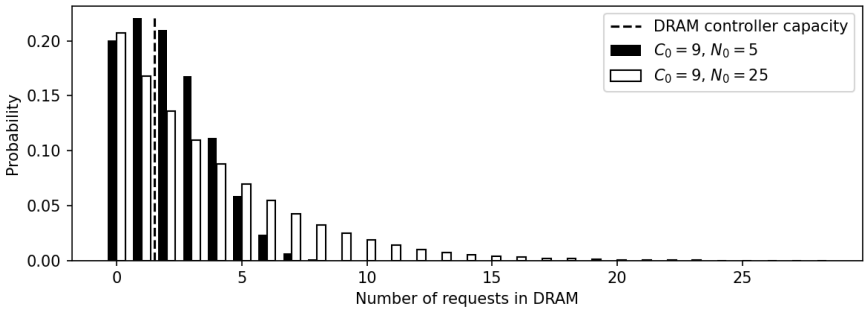
Furthermore, the number of requests that can be served simultaneously must be equal to

$$C_M = 1.$$

The remaining parameters are C_i, N_i, μ_i for $0 \leq i \leq M-1$. We will use the value of μ_i to fulfill the throughput criterion, which means that the value of N_i and C_i can be used to satisfy the waiting time criterion. We will use $\widetilde{\mu}_i(N_i, C_i)$ to denote the value of μ_i such that the throughput criterion of core-i-requests is satisfied and $NC_i(\mu_i) \in \mathbb{N}^2$ to denote the set of combinations of N_i and C_i for which the waiting time criterion is satisfied. We may use $\widetilde{\mu}_i$ when the values of N_i and C_i are clear from the context.



(a) A varying value of C_0.



(b) A varying value of N_0.

Figure 4.8: The probability of observing a certain number of requests in the DRAM for a varying value of C_0 (above) and N_0 (below).

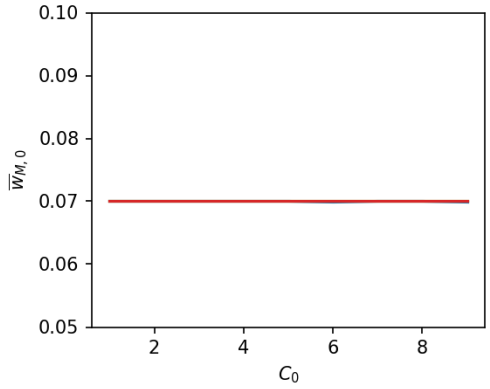
Figure 4.7a shows for different values of C_0 and N_0 the value of the service time $(\widetilde{\mu}_0(N_0, C_0))^{-1}$. Figure 4.7b shows for C_0, N_0 and $\widetilde{\mu}_0(C_0, N_0)$ the DRAM controller waiting time. It can be observed that increasing the value of N_0 or decreasing the value of C_0, while keeping the DRAM controller throughput the same, will result in a higher DRAM controller waiting time. Moreover, it can be observed that for a desired (measured) waiting time of 25 ns the set $NC_0(\widetilde{\mu}_0)$ is non-empty, while for a desired waiting time of 100 ns the set $NC_0(\widetilde{\mu}_0)$ is empty.

We will start by discussing what causes the waiting time to change when the throughput stays the same, and then we will discuss which values of N_0 and C_0 are best to choose.

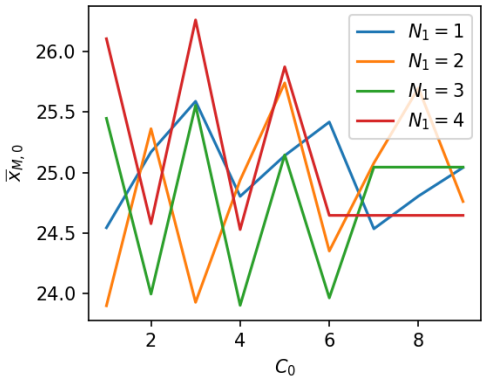
The influence of N_0 and C_0

The change in waiting time by a change in N_0 and C_0 (when keeping the throughput the same) can best be explained by Figure 4.8.

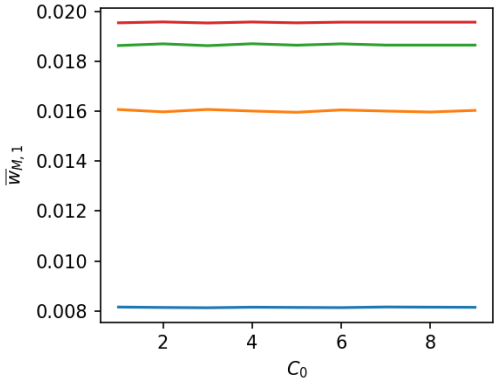
Figures 4.8a and 4.8b show how the probability of observing a certain number of requests in the DRAM controller changes for a varying value of respectively C_0 and N_0. It can thus be observed that an increase of N_0 or a decrease of C_0 increases the variance



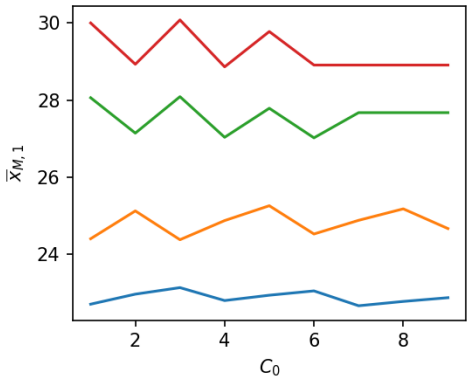
(a) Waiting time of core-0-requests.



(b) Waiting time of core-1-requests.



(c) Average number of core-1-requests.



(d) Throughput of core-1-requests.

Figure 4.9: For a varying value of C_0 are the best values of μ_0 and N_0 found such that core 0 satisfies the throughput and waiting time criteria. In the top are the achieved throughput and waiting time of core 0 requests plotted when the desired throughput was 0.07 and the desired waiting time was 25 ns. In the plots below is shown for fixed values of N_1 how the varying value of C_0 (and the corresponding values of μ_0 and N_0) influences the throughput and waiting time of core-1-requests in the DRAM controller.

of the probability distribution. Note that widening the probability distribution on the right side of the dashed line increases the average waiting time, because the probability mass on that side corresponds to the queueing delay (the time a request has to wait before being served).

Choosing N_0 and C_0

In order to find the best value of $N_0, C_0 \in NC_i(\widetilde{\mu}_i(N_0, C_0))$ it is important to know how each combination of N_0 and C_0 influences the throughput and waiting time of the requests of the other cores.

The influence of different combinations of N_0 and C_0 on core-1-requests is shown in Figure 4.9. Although the waiting time in Figure 4.9b is not exactly equal to 25 ns, it is close enough to say that the waiting time criterion is satisfied. As can be observed, for a fixed value of N_1 and different values of N_0, C_0 and $\widetilde{\mu}_i(N_0, C_0)$, the throughput and waiting time of core-1-requests in the DRAM controllers stay constant. This implies that any value within $(N_i, C_i) \in NC_i(\widetilde{\mu}_i(N_i, C_i))$ can be chosen. Due to the fact that the MVA algorithm from Section 3.3 has time complexity which is linear in N_i, a low value N_i is better. Due to the fact that a low value of C_i requires a low value of N_i to satisfy the waiting time criterion (Figure 4.7b), the best approach would be to let $C_i = 1$ for all $0 \leq i \leq M - 1$.

In conclusion, the value of μ_i will be used to satisfy the throughput criterion, the value of C_i is set equal to 1, and an appropriate value of N_i is found to satisfy the waiting time criterion.

4.3 Benchmark execution data to model mapping

This section describes the two steps in determining model parameters from isolated benchmark execution data. The first step includes partitioning the total time interval into smaller intervals. The sub-intervals with the same throughput and access latency can then be clustered into one group. The second step concerns the calculation of the model parameters for each cluster.

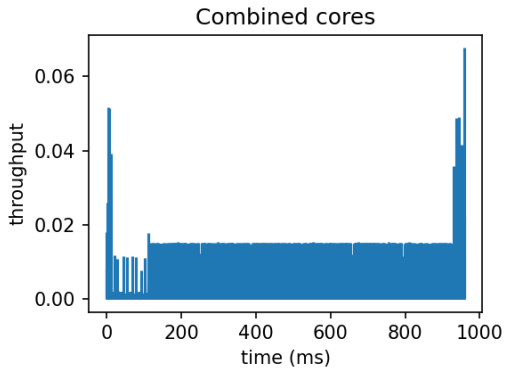


Figure 4.10: A plot of the average DRAM controller throughput of the cores combined during the execution of `bodytrack`.

4.3.1 Retrieving model parameters

The first step concerns partitioning the total time into sub-intervals that have approximately the same DRAM controller throughput and access latency. Figure 4.10 illustrates the importance of this step. As can be observed, the benchmark progresses through various stages during its execution. This means that it would be inaccurate to model the

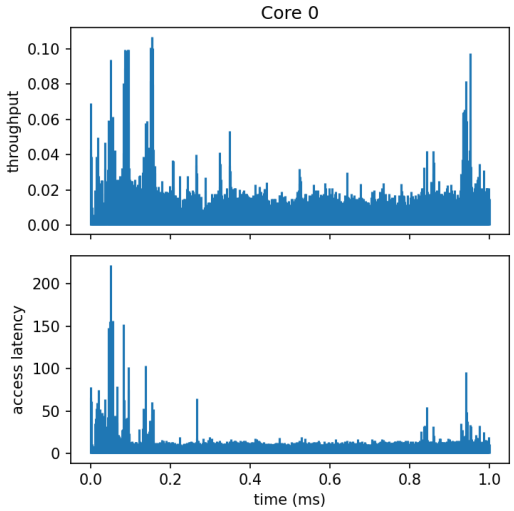
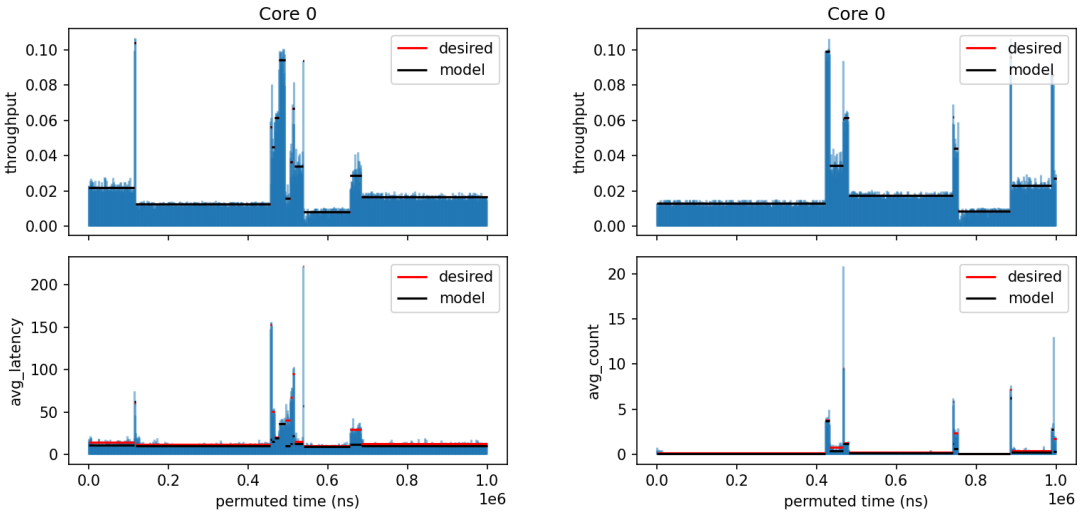


Figure 4.11: Throughput and average latency plot of the interval $0 - 1ms$.



(a) Average latency.

(b) Average number.

Figure 4.12: The desired (average) values and the values obtained by the model are plotted against a permutation of the time. The waiting time (left) and average number criteria (right) were aimed at satisfying (additional to the throughput criterion).

benchmark with fixed model parameters. You essentially want to change the model parameters during execution, but a closed queueing network does not allow a change of parameters. We can solve this problem by partitioning the total time interval, such that the throughput and access latency stay roughly constant during each interval in

the partition. The following method will be used in determining the partition:

1. Divide the total time into intervals of length `stepsize` (we choose 1000 ns for now).
2. Use k-means clustering in order to group the sub-intervals by the throughput and waiting time (or average number).
3. Then all sub-intervals belonging to the same cluster can be modeled with the same model parameters.

It then suffices to only calculate the model parameters for each cluster, instead of for each sub-interval of 1000 ns. The following approach can then be used in order to determine the model parameters:

1. The value of C_i is initialized at 1 for all cores i and the value of N_j is initialized to 1 for all cores j having a non-zero throughput.
2. The value of $\widetilde{\mu}_i(N_i, C_i)$ is determined by applying the function `scipy.optimize.root` to the function f, where f outputs the difference between the throughput measured by the MVA algorithm and the real DRAM controller throughput.
3. If the waiting time is too low, then the value of N_i is increased by 1.
4. Step 2 and 3 are repeated until the desired waiting time (the real DRAM controller access latency) is achieved.

The original time plots for the interval $0-1$ ms are shown in Figure 4.11 and the result of the approach for one core is shown in Figure 4.12. In the figure, the sub-intervals that belong to the same cluster are arranged consecutively. This means that the x-axis does not show the real-time, but some permutation of the time. The waiting time and average number criteria are tried satisfying in respectively Figure 4.12a and 4.12b.

It is noteworthy to mention that in some cases the desired waiting time cannot be reached, because, for each throughput, there is a limit for the maximum waiting time. This can be observed from Figure 4.7b, where it can be seen that the waiting time curve flattens for large values of N_0.

Similar results can be obtained for a benchmark running on multiple cores. Since we only care about the response time of a benchmark and not about the response times of the individual cores, we can combine the throughput of the cores. Figures 4.13 and 4.14 show the throughput for respectively the cores being handled separately and combined. It illustrates how combining the throughput of the cores causes a better match between the desired and model data. Therefore, from now on, we will combine the cores of a benchmark and thus view a benchmark as a single source of DRAM requests.

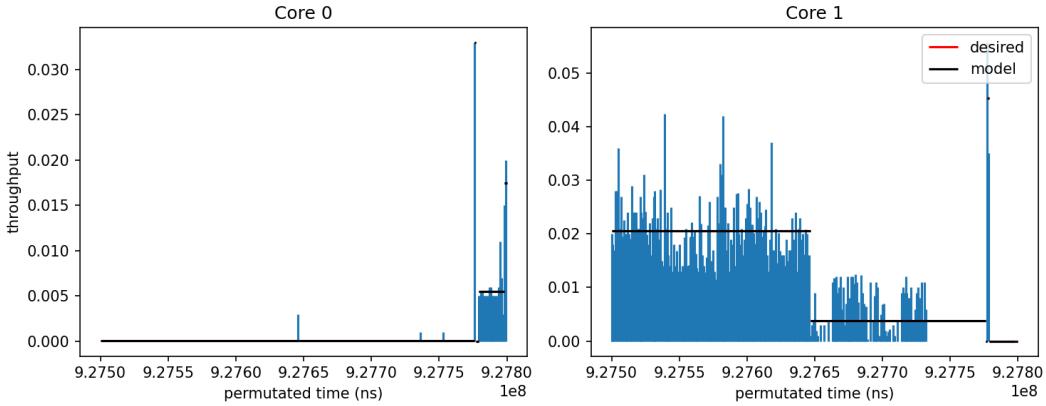


Figure 4.13: Cores handled separately.

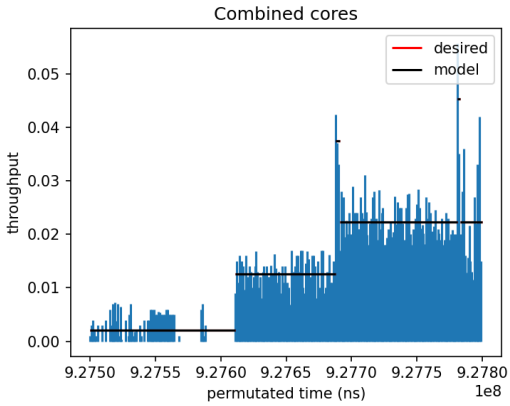


Figure 4.14: Cores handled combined.

4.4 Other mathematical models

This chapter showed that there are two problems that could not be solved by closed queueing networks: changing parameters and a priority system. The first problem could partly be solved by conditioning on the value of the throughput and waiting time (Figure 4.12) and determining different model parameters for each set of intervals in the partition. The second problem could not be solved so easily.

In the field of queueing theory, there is much more theory available, and these problems can possibly be solved by switching to the queueing discipline of non-preemptive priority queues [1] and by letting the arrival process be a Markov-modulated Poisson process [5]. Sadly, the closed queueing network doesn't allow priority queues or Markov-modulated Poisson processes, but we will still briefly discuss them in this section because they can be a topic of interest for further research.

4.4.1 Non-preemptive priority

In non-preemptive priority queues, there is a hierarchy of importance between the different types of customers. Without loss of generality, we assume type 1 customers have the highest priority, then type 2 customers, and so on. When a type i customer arrives at a service center, it is more important than a type $j > i$ customer, so the service of a type i customer must start before the service of a type $j > i$ customer starts. However, the non-preemptive rule implies that a type i can only cut in line and he cannot interrupt the service of a type $j > i$ customer.

Notice how this is similar to a DRAM controller, because the write requests can be given priority over the read requests, but it should be noted that the priority system in a DRAM controller depends on the dependency between the instructions.

4.4.2 Markov modulated Poisson processes

In Markov-modulated Poisson processes, the arrival rate of customers is not fixed but depends on some underlying and independent Markov chain. This means that it can possibly be used to model a benchmark that progresses through different stages of the execution. Each of the stages then corresponds to a state of the underlying Markov chain. The disadvantage of this approach is that in reality, the duration of the different stages depends on the current state of the system. For example, if a task in isolated execution has a DRAM throughput of 1 and it spends 100 ns in stage i of the execution, then the core might spend 200 ns in stage i when it only receives a throughput of 0.5 due to the parallel execution of other tasks. However, this does not mean that Markov-modulated Poisson processes are doomed to fail, but it only illustrates that benchmarks do not have a fixed underlying Markov chain, but the underlying Markov chain depends on which benchmarks are run in parallel.

5 Multi-benchmark execution and model accurateness

Using the approach from Section 4.3, the model parameters can be determined for each benchmark. These parameters can then be used to model the parallel execution of a benchmark. The method from Section 4.3.1 makes use of two hyperparameters: the stepsize and number of clusters. The entire execution of a benchmark is divided into sub-intervals of length `stepsize`. The sub-intervals are then clustered in `n_clusters` according to their DRAM throughput and access latency. Thus, a lower stepsize and a higher number of clusters give a more accurate model. Analysis of the hyperparameters on the accurateness of modeling parallel benchmark execution is discussed in Section 5.1, while the accurateness of the model for the benchmarks in the PARSEC benchmarks suite is discussed in Section 5.2.

5.1 hyperparameters analysis

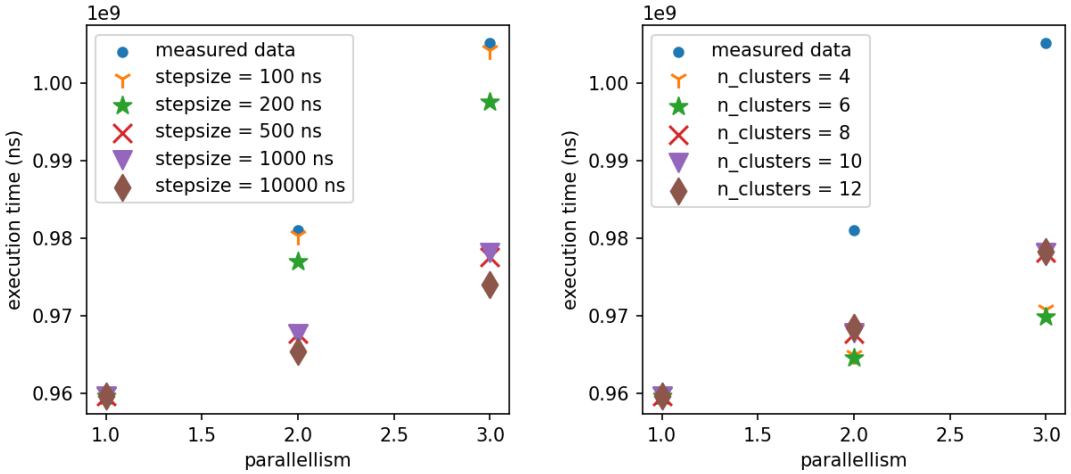
There are two hyperparameters used in determining the model parameters: `n_clusters` (the number of clusters) and `stepsize`. This section discusses the influence of the hyperparameters on the accuracy of the modeled parallel execution time. The accuracy analysis is done for the `bodytrack` and `streamcluster` benchmarks.

5.1.1 The `bodytrack` benchmark

Figures 5.1a and 5.1b illustrate how respectively a change in `stepsize` and `n_clusters` influences the accuracy of the model for the `bodytrack` benchmark.

This figure shows that the modeled execution time is increasing when the stepsize is decreased or the number of clusters is increased. Furthermore, it can be observed that increasing the number of clusters beyond 8 does not seem to affect the modeled execution time. Lastly, the modeled execution time seems to converge to the measured execution time when the stepsize is decreased.

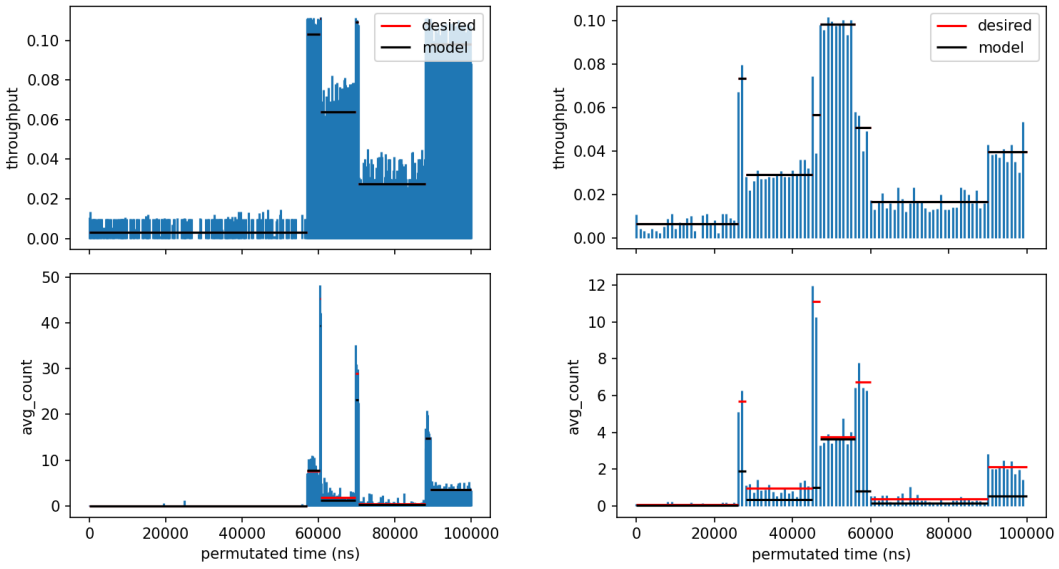
Figure 5.2 can help to explain the difference caused by the different stepsizes. The figure shows that for a stepsize of 100 ns, both the throughput and average number criteria can be satisfied, while only the throughput criterion can be satisfied when a stepsize of 1000 ns is chosen. Notice that for a stepsize of 1000 ns, the modeled average number of requests in DRAM is often significantly lower than the desired (measured) average number of requests in DRAM. This means that the `bodytrack` benchmark slows down other benchmarks less when it is modeled with a stepsize of 1000 ns than when it is modeled with a stepsize of 100 ns.



(a) A varying stepsize.

(b) A varying number of clusters.

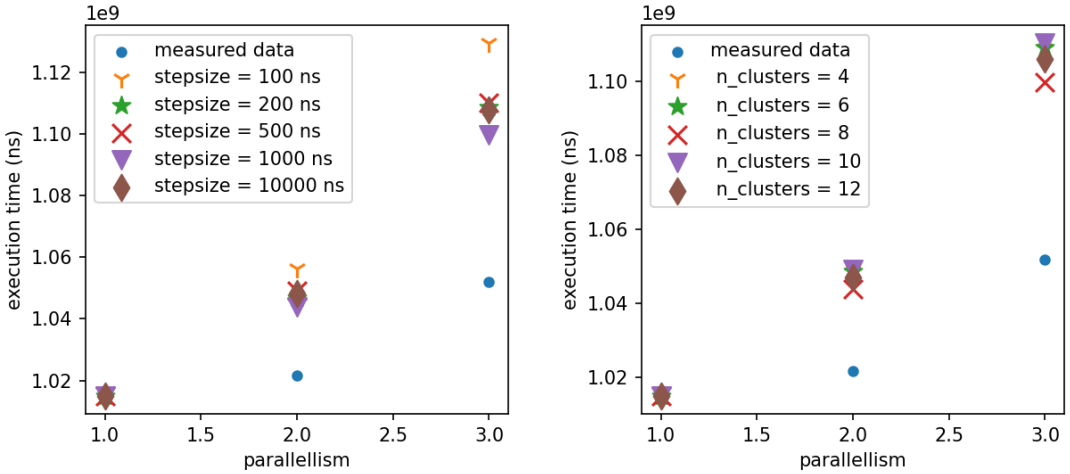
Figure 5.1: The influence of different stepsizes (left) and a different number of clusters (right) on the accuracy of the parallel execution of 1, 2 or 3 different instances of the `bodytrack` benchmark. The number of clusters for the left figure equals 8, while the stepsize for the right figure equals 1000 ns.



(a) Stepsize is 100 ns.

(b) Stepsize is 1000 ns

Figure 5.2: The influence of the stepsize on satisfying the throughput (above) and average number (below) criteria in the first 100000 ns of the `bodytrack` benchmark. The sub-intervals are grouped together based on their cluster, so the x-axis shows some permutation of the time.



(a) A varying stepsize.

(b) A varying number of clusters.

Figure 5.3: The influence of different stepsizes (left) and a different number of clusters (right) on the accuracy of the parallel execution of 1, 2 or 3 different instances of the `streamcluster` benchmark. The number of clusters for the left figure equals 8, while the stepsize for the right figure equals 1000 ns.

5.1.2 The `streamcluster` benchmark

Figures 5.3a and 5.3b illustrate how respectively a change in `stepsize` and `n_clusters` influences the accuracy of the model for the `streamcluster` benchmark.

Similar to Figure 5.1, Figure 5.3 shows that the modelled execution time is increasing when the stepsize is decreasing or the number of clusters is increasing. Furthermore, the figure suggests that the number of clusters barely influences the execution time. Contrary to Figure 5.1, the modelled execution time does not converge to the measured execution time.

In conclusion, from Figure 5.2 we can conclude that the model is most accurate when a stepsize of 100 ns is used. Moreover, the model is most accurate when the number of clusters is set to 8.

5.2 Model accuracy analysis

This section is devoted to determining the accuracy of the model for the 6 benchmarks in the PARSEC benchmark suite. From the previous section we were able to conclude that the model is most accurate when the stepsize is set equal to 100 ns and the number of clusters is set equal to 8. Therefore, the accuracy of the model will be determined for the model with those hyperparameters.

We will distinguish between two types of benchmarks: high and low DRAM-intensity benchmarks. The high DRAM-intensity benchmarks are the `bodytrack`, `streamcluster` and `dedup` benchmarks and they have an average DRAM utilization of $12 - 28\%$ (Table

Benchmark	Average DRAM utilization	Difference in isolated execution time
`bodytrack`	12%	0.3%
`dedup`	18%	0.033%
`streamcluster`	28%	0.04%
`blackscholes`	1%	0.5%
`fluidanimate`	1%	0.6%
`swaptions`	1%	0.4%

Table 5.1: The average DRAM utilization and difference in isolated execution time are shown for each of the benchmarks in the PARSEC benchmark suite. The 'difference in isolated execution time' is measured by simulating each of the benchmarks twice in isolation. The 'difference in isolated execution time' then refers to the difference in percentage between the two isolated execution times.

Benchmark	Simulated time increase	Modeled time increase
`bodytrack`	2,2%	2,1%
`dedup`	0,8%	1,3%
`streamcluster`	0,6%	4%

Table 5.2: The increase in execution time is shown between the isolated execution and the parallel execution of 2 instances of the same benchmark: The 'simulated time increase' refers to the increase in execution time measured by CoMeT, while the 'modeled time increase' refers to the increase in execution time measured by the model.

5.1). The low DRAM-intensity benchmarks are the other benchmarks. They have an average DRAM utilization of 1%. The 'difference in isolated execution' is shown for each of the benchmarks in Table 5.1. This value is calculated by simulating each benchmark twice in isolation and then calculating the difference in percentage between them. Although each benchmark is only simulated twice, it should still give an indication of how much the execution time of a benchmark can differ between different simulations.

5.2.1 High DRAM-intensity benchmarks

This section concerns the model accuracy analysis of the high DRAM-intensity benchmarks `bodytrack`, `streamcluster` and `dedup`. The difference in execution time between the isolated execution and the parallel execution of two instances of the same benchmark is shown in Table 5.2.

It should be noted that for all the benchmarks, the time increase is significantly higher than the 'difference in isolated execution time' (Table 5.1). This gives reason to believe that the results are reliable. The table seems to suggest that the 'modeled time increase' gives an upper bound for the 'simulated time increase'.

Benchmark	Measured time increase	Modeled time increase
`blackscholes`	1,1%	0.6%
`fluidanimate`	1,4%	1,4%
`swaptions`	0,7%	0,5%

Table 5.3: The increase in execution time is shown between the isolated execution and the parallel execution of two instances of the same benchmark: The 'simulated time increase' refers to the increase in execution time measured by CoMeT, while the 'modeled time increase' refers to the increase in execution time measured by the model.

5.2.2 Low DRAM-intensity benchmarks

This section concerns the model accuracy analysis of the low DRAM-intensity benchmarks `blackscholes`, `swaptions` and `fluidanimate`. Table 5.3 is generated by comparing the isolated execution with the parallel execution of 2 instances of the low DRAM-intensity benchmark and two instances of the `streamcluster` benchmark. Two instances of the `streamcluster` benchmark were added to the parallel execution due to the fact that the low DRAM-intensity benchmarks barely use any DRAM. This means that no significant increase in time would be noticed if only two of the low DRAM-intensity benchmarks were run in parallel.

Due to the high variance of the execution time of these benchmark (Table 5.1), it should be noted that these results may not be entirely trustworthy.

6 Conclusion

This thesis started by discussing the closed queueing networks and mean value analysis. The theory has been combined to create a simple model that can be used to predict the parallel execution time. The model was tested for six benchmarks of the PARSEC benchmark suite and for two of the benchmarks (the `bodytrack` and `fluidanimate` benchmarks), the model gave an accurate prediction of the parallel execution time. The model was less accurate for the other four benchmarks.

For the computer science part, I reached my main goal of creating a model and successfully calculating the model parameters from isolated execution data. I did not reach my second goal of implementing extensions for the model, because the analysis of the model took longer than expected.

For the mathematics part, I reached my main goal of discussing closed queueing networks and mean value analysis, but there was no time left to consider other types of distributions or other types of queueing networks. However, I discussed in Section 4.4 which other models from queueing theory can help in making the model more realistic.

6.1 Further steps

There are several ways the model can be extended or adjusted to increase the accuracy. The model can be extended by adding components, such as a memory unit inside the core. The parameter derivation can be adjusted by dividing the total time interval not in intervals of equal length, but in intervals of different lengths. Each interval can then be used to capture exactly one burst of DRAM requests. This could give better results, because, during each burst of DRAM requests, the priority system is unlikely to change. This could mean that Little's law holds for more sub-intervals. Furthermore, Markov modulated Poisson processes (Section 4.4.2) and non-preemptive priority (Section 4.4.1) can be used to create a model with changing model parameters and a priority system between the different types of requests.

Although these suggestions can help improve the accuracy, I recommend starting with adjusting the service rate function of core i: ϕ_i. Currently, the function ϕ_i is given by

$$\phi_i(k) = \min\{C_i, k\}\,\mu_i,$$

but it should be noted that MVA supports any function ϕ_i and it is not even necessary for MVA to state how the total service rate ϕ_i is divided between the different customers at the service station. This follows from the fact that the value of $\phi_{i,k}$ is irrelevant for the invariant distribution of a closed queueing network (Equation 3.3).

A possible candidate for the functions $\phi_i(k)$ is given by

$$\phi_i(k) = \left(\frac{k}{N_i}\right)^{\alpha} \mu_i, \quad \alpha \in [0, \infty).$$

This function equals μ_i for $k = N_i$ and the value of α determines how dependent the core is on DRAM requests. When α is close to zero, the function is close to μ_i for small values of k, which means that the core does not need many requests to run at close to the maximum service rate (the total service rate of the core is not very dependent on the number of requests in the core). On the other hand, if α is very big, then the function is only running at close to the maximum service rate if k is close to N_i (the total service rate of the core is very dependent on the number of requests in the core). Thus, this function can differentiate between how dependent the core is on the number of requests.

Another possible suggestion would be to investigate how the value of C_i influences the parallel execution time. Currently, the value of C_i is set to 1, because this value does not influence the other cores. However, the value of C_i could still influence the parallel execution time. The following method can be then used to create a more accurate model for benchmark A:

1. Simulate benchmark A in isolation using CoMeT.
2. Simulate two instances of benchmark A in parallel using CoMeT.
3. Use the value of μ_i to satisfy the throughput criterion for the isolated execution.
4. Start with $C_i = 1$ and increase N_i until the waiting time criterion of the isolated execution is satisfied.
5. Increase C_i by one and repeat step 4 until the throughput criterion of the parallel execution of benchmark A is also satisfied.

Thus, this method also uses the parallel execution of benchmark A to create a more accurate model.

Bibliography

[1] Ivo Adan and Jacques Resing. Queueing systems. 2015. *URL: https://www. win. tue. nl/˜ iadan/queueing. pdf*, pages 89–98, 2003.

[2] 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.

[3] Christian Bienia. *Benchmarking modern multiprocessors*. Princeton University, 2011.

[4] Steven C Bruell, Gianfranco Balbo, and PV Afshari. Mean value analysis of mixed, multiple class bcmp networks with load dependent service stations. *Performance Evaluation*, 4(4):241–260, 1984.

[5] Wolfgang Fischer and Kathleen Meier-Hellstern. The markov-modulated poisson process (mmpp) cookbook. *Performance evaluation*, 18(2):149–171, 1993.

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

[7] Lokesh Siddhu, Rajesh Kedia, Shailja Pandey, Martin Rapp, Anuj Pathania, Jörg Henkel, and Preeti Ranjan Panda. Comet: An integrated interval thermal simulation toolchain for 2d, 2.5 d, and 3d processor-memory systems. *ACM Transactions on Architecture and Code Optimization (TACO)*, 19(3):1–25, 2022.

Populair summary

Determining the performance of a computer is not only a topic of interest for consumers but also for computer manufacturers and computer scientists. They may want to measure the performance of computer systems with slightly different specifications in order to accurately determine the influence of specific components on the performance of the entire computer system.

Benchmarking

Often, measuring the performance is done through benchmarking. Benchmarking is a method to determine the performance of a system by running resource-intensive computer programs (benchmarks). There are many different types of benchmarks available and most benchmarks are based on a single computational task a computer must be able to perform, e.g. image recognition, image compression, or pricing a portfolio of options.

Simulating benchmarks

It should be noted that the most accurate way of determining the performance of a system is by simulating the execution of a benchmark instead of by simply executing the benchmark. Although this is more accurate, it also takes a lot longer, e.g. a benchmark with an execution time of 1 second might take an hour to simulate. Furthermore, a computer scientist might not only want to simulate benchmark A and B in isolation, but he might also be interested in knowing what happens when A and B are simulated in parallel. This can take a very long time.

This thesis aims at solving this problem. To be more precise, this thesis has created a closed queueing network that tries to predict the parallel execution time of benchmarks A and B from the isolated execution of benchmarks A and B. For example, if benchmark A takes 1 second to execute and benchmark B takes 2 seconds to execute, then A and B take 1.05 and 2.08 seconds to execute in parallel. The goal of the thesis is illustrated in Figure 6.1 and it also shows the component of interference between the benchmarks: the shared memory unit.

Results

The model is applied to six different benchmarks of the PARSEC benchmark suite. For two of the benchmarks, the model could very accurately predict the parallel execution time, while the model was less accurate for the other four benchmarks.

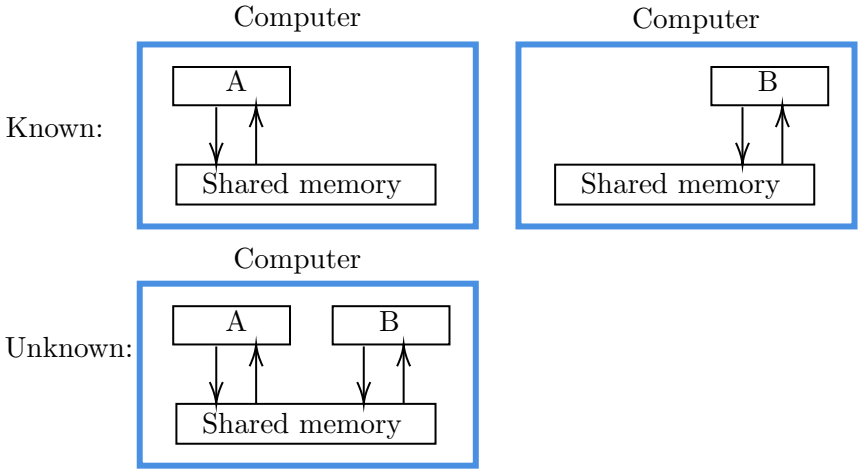


Figure 6.1: The goal of this thesis: predicting the parallel execution time of A and B from the isolated execution time of A and B.

Thus, the model is not very accurate, but it should be noted that the simplicity of the model enables anyone to extend or adjust the model in order to increase its accuracy.