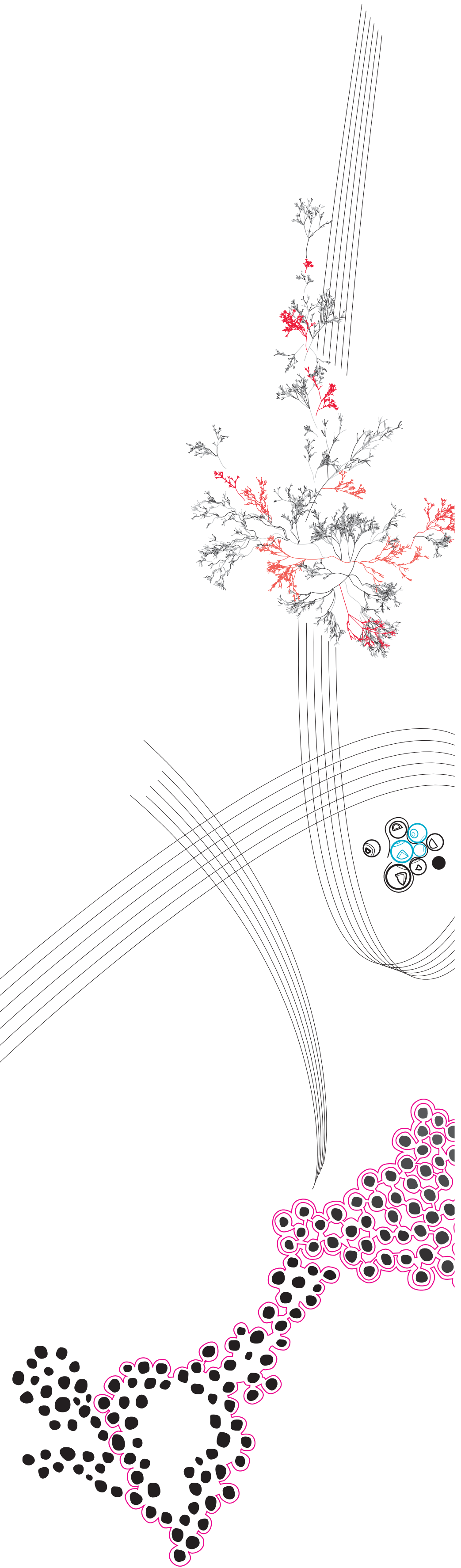


BSc Thesis Applied Mathematics

Online scheduling of tasks to minimize maximum processor temperature

Floor van Maarschalkerwaart

Supervisors: M. de Graaf and B. Ozceylan

July 1, 2022

Department of Applied Mathematics
Faculty of Electrical Engineering,
Mathematics and Computer Science

UNIVERSITY OF TWENTE.

Preface

This report is part of my Bachelor's assignment of Applied Mathematics at the University of Twente. I would like to thank Maurtis de Graaf and Baver Ozceylan for their help with my assignment. Thank you for being so involved and for thinking along whenever I got stuck or lost motivation. It really helped to know I could come by or send an e-mail at any moment. After a little dip during the middle of the process I did find my motivation again and I really enjoyed learning a lot about this subject.

Online scheduling of tasks to minimize maximum processor temperature

Floor van Maarschalkerwaart*

July 1, 2022

Contents

1 Introduction **2**

2 Definitions and models **3**
2.1 Problem definition . . . 3
2.2 Thermal model . . . 3
2.3 System definition offline problem . . . 5
2.4 System definition online problem . . . 6

3 Optimal allocation **7**
3.1 Policies . . . 7
3.2 Optimal allocation offline problem . . . 8
3.3 Optimal allocation online problem . . . 9
3.4 Analytic results . . . 10
3.4.1 Output offline problem . . . 10
3.4.2 Output online problem . . . 10
3.4.3 Queuing model . . . 11

4 Simulation **12**
4.1 Discretization . . . 12
4.2 Simulation offline problem . . . 12
4.3 Simulation online problem . . . 13
4.4 Verification . . . 15
4.4.1 Output offline job problem . . . 15
4.4.2 Output online job problem . . . 16
4.4.3 Queuing model . . . 16
4.5 Simulation results . . . 17

5 Discussion **19**

6 Conclusion **21**

7 Bibliography **23**

8 Appendix A **24**

9 Appendix B **25**

10 Appendix C **29**

11 Appendix D **33**

*Email: f.vanmaarschalkerwaart@student.utwente.nl

1 Introduction

Thermal management is becoming more and more important since high temperatures can have a lot of consequences for electronic devices. High temperatures shorten the lifespan of devices, decrease the reliability and can also decrease the effective operating speed. Large variations in the temperature also decrease the reliability even further. It is hard however to control the power consumption (high power consumption is a large cause of high processor temperature) of these devices while they are operating. Active cooling mechanisms often consume too much power for mobile devices due to their battery dependent nature and are often not suited for the environments where these devices are used, usually subject to vibrations and shock. Therefore the focus of thermal management of electronic devices lies with passive cooling mechanisms. [9] [13] In our research we will focus on one such passive cooling mechanism, namely idle time scheduling which idles the processor for a time to let it cool down. When idling the processor it will have some time to cool down, but idling it for too long will cause the system to reduce performance. Therefore, a balance between performance and processor temperature has to be found. More specifically, we will research how the algorithm for optimal temperature aware-scheduling of offline jobs from the paper by B. Ozceylan et al. [9] behaves when applied to online scheduling. Offline scheduling is a type of decision-making where a set schedule is made before it is executed. With online scheduling, the schedule is made as the system is running. This could be for example that a new schedule is made every few time units but also that a new schedule is made when the system has reached a certain temperature.

There has already been quite some research for thermal-aware management for offline systems. Some examples include a resource management framework [6], using idle time [10] or task scheduling [5] to minimize the maximum temperature while still meeting the time constraints of certain jobs. One of the first papers analyzing online heuristics for thermal management is by Yao et al. [13]. The paper provides a formal analysis of the minimum-energy scheduling problem and gives a simple model in which each job is to be executed between its arrival time and deadline by a single variable-speed processor. It then proposes an offline algorithm and analyses some online heuristics. The paper is innovative in the sense that it is one of the first papers on saving energy of electronic devices, especially in the form of a scheduling problem. They also state that an online algorithm cannot in general construct an optimal energy schedule, even for a simple problem with only two tasks. In [3] the paper by Yao et al. [13] is described as initiating the theoretical study of speed scaling policies to manage energy. As this innovative paper is only from 1995, we can acknowledge that the field of (stochastic) online scheduling is still relatively new. There are already quite a few results but very many problems remain open [11]. The paper by T. Vredeveld [11] combines online and stochastic scheduling into the stochastic online scheduling (S0S) model. In this model, jobs arrive in an online manner and as soon as this job arrives, the scheduler only learns about the probability distribution of the workload and not the actual workload. The paper by N. Bansal et al. [3] proposes an online algorithm which always puts out a feasible schedule, which means that for each task i with workload w, at least w work is done on this task after its arrival time and before its deadline. They show that no deterministic online algorithm can have a better competitive ratio than their algorithm. The competitive ratio is the ratio between the performance of the online algorithm and the offline version of the algorithm. There are already some more online algorithms for specific systems/fields. Examples include the field of edge computing [7] and multiple clock domain chip multi-processors [2]. The paper by Q. Bashir et al. [4] proposes an online scheduling technique to avoid thermal emergencies in multiprocessor systems. Their technique performs load balancing based on dynamic temperature measurement at a fixed ambient temperature. Their simulation results show that their technique reduces the overall temperature up to 5%.

The goal of *this* paper is to apply the algorithm from [9] into this setting to have an online version of this algorithm that minimizes the maximum temperature while meeting the performance constraints by scheduling idle time. We will then implement this algorithm into a simulation to see the effects it has on the temperature. Our research question is thus as follows: *"How can we schedule online jobs to minimize the maximum processor temperature of electronic devices while still meeting deadlines of jobs?"*

We show how the policy proposed in [9] can be implemented as an online scheduling method to

minimize the maximum processor temperature while still meeting the deadlines of jobs with respect to other policies. We show this in a system with one type of job with the same (known) workload and deadline and a certain arrival process. In this situation, this policy can decrease the temperature with up to 44% with respect to other policies.

2 Definitions and models

In this section we will show the models and system definitions we used. We will start by stating our problem definition and summarizing the thermal model and offline system of [9]. Afterwards we will define our online system.

2.1 Problem definition

In this bachelor's thesis, we will consider a system with a processor for which we can schedule idle time and we consider one type of job arriving to the system. With this we mean that every job arriving to the system has the same (known) workload and time before it must be finished (also known as the deadline). However, we do not know when each job will arrive, we only know the arrival rate of its Poisson arrival process. We also assume the processor can process only one job at the same time. When a job arrives while another is already in the system, this means it will be rejected. When a job enters the system, the system starts processing it and the processor temperature will increase. Once a job leaves the system, the processor will cool down until a new job arrives. The goal of the *optimal* policy from [9] is to schedule and assign the resources in a certain way to minimize the maximum processor temperature.

As input variables of the algorithm we use one type of job. The algorithm will yield a schedule that shows how many resources should be assigned at each moment. We use this schedule together with an initial temperature as input of our simulation to get the temperature over time as output of our simulation. A schematic overview of the input and output can be found in Figure 1.

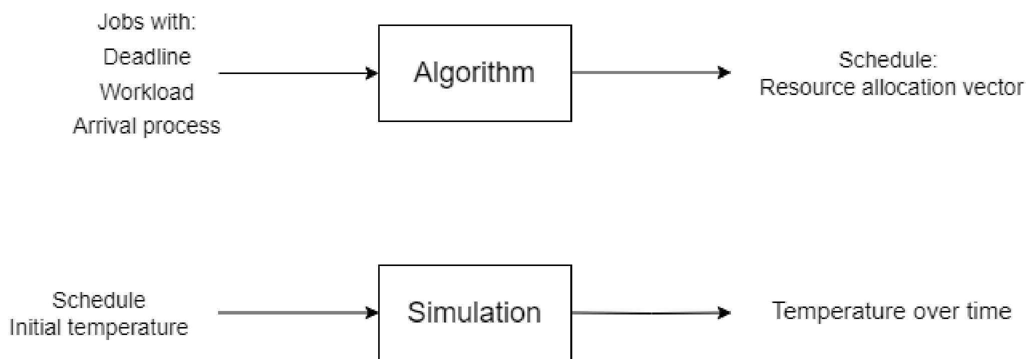


Figure 1: Schematic overview of input and output of our algorithm and simulation

2.2 Thermal model

We will briefly explain the thermal model we used. This model is the same as in [9], upon which this research is building. Therefore we shall not discuss it in great detail but it is important since it introduces some variables and symbols we will use in our system definition.

The heat dissipation in a system is directly related to the assigned resources. There are two heat sources in a processor unit. One reflecting the switching activities in the processor, which depends on the active resources of the processor. The second source relates to the leakage current in the processor, which depends on the temperature of the system. But as stated in [9] this source has a very low effect compared to the first one so we focus on the heat source reflecting the switching activities:

$Q_D(t) = \alpha x(t)$, where $0 \leq x(t) \leq 1$ is the normalized amount of active resources (also known as processor utilization) and $\alpha > 0$ is a system dependent parameter. [9]

To describe the change in temperature when a certain utilization $x(t)$ is applied, [9] uses the following dynamic heat transfer equation:

$$\dot{T} = -\frac{1}{\tau}T(t) + \frac{1}{\tau}T_a + \frac{\alpha}{\tau}x(t) \tag{1}$$

where $\dot{T}$ is the first-order derivative of the temperature T, $\tau > 0$ a thermal system dependent time constant and T_a is the ambient temperature dependent on the environment. We assume T_a is constant over time.

Afterwards, [9] defines the *output* as the normalized temperature difference $y(t) = \frac{1}{\alpha}T(t) - \frac{1}{\alpha}T_a$. Now we can rewrite Eq. (1) as follows:

$$\dot{y}(t) = -\frac{1}{\tau}y(t) + \frac{1}{\tau}x(t) \tag{2}$$

From [9], the solution to this first-order DE is

$$y(t) = \frac{1}{\tau}\int_{t_0}^{t} e^{-\frac{t-\sigma}{\tau}}x(\sigma)d\sigma + y(t_0)e^{-\frac{t-t_0}{\tau}}, \qquad \forall t \geq t_0 \tag{3}$$

where [9] takes the initial value of $y(t)$, namely $y_0 = y(t_0) \in [0, 1]$. Then $0 \leq y(t) \leq 1$ always since $x(t)$ is normalized. And thus we have $T_a \leq T(t) \leq T_a + \alpha$.

Having established the thermal model, [9] makes two observations to ease notation. This will help us with the analysis and simulation later. Firs, suppose $x(t)$ is constant in a time interval $[\delta_a, \delta_b]$ where $0 \leq \delta_a \leq \delta_b$, so $x(t) = x(\delta_a),\ \forall t \in [\delta_a, \delta_b]$. Then we can write $y(t)$ using Eq. (3) as follows:

$$y(t) = x(\delta_a)(1 - e^{-\frac{t-\delta_a}{\tau}}) + y(\delta_a)e^{-\frac{t-\sigma}{\tau}}. \tag{4}$$

Second, if $y(t)$ is constant in this time interval such that $y(t) = y(\delta_a),\ \forall t \in [\delta_a, \delta_b]$ then we can write $x(t)$ using Eq. (2) as:

$$x(t) = y(t) = y(\delta_a),\ \forall t \in [\delta_a, \delta_b]. \tag{5}$$

To illustrate the behavior of $y(t)$ and $T(t)$ for different input $x(t)$ we have made an example based on the theory in [9]. Suppose $\alpha = 25$, $\tau = 0.6s$, $T_a = 10°C$ and $T_0 = 17.5°C$. Then Figure 2 shows a sample input $x(t)$ and the corresponding output $y(t)$ and temperature $T(t)$.

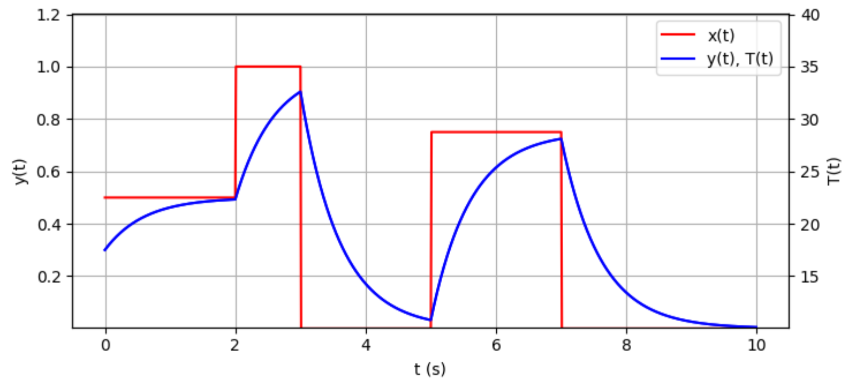


Figure 2: Example of the behaviour of the output of the system with varying input

2.3 System definition offline problem

System definition for multiple offline jobs

We will start by summarizing the offline model of [9] so we can base our online system off of this later.

We have a set of jobs $n \in \{1, 2, ..., N\}$ becoming available at time $t = 0$ with known deadline vector $d = [d_1, d_2, ..., d_N]$ and workload vector $w = [w_1, w_2, ..., w_N]$ which are defined as follows:

- w_n is the time needed for the system to process job n at full system resources (i.e. $x(t) = 1$).
- d_n is the time before which job n has to be completed.

Assume jobs are ordered by increasing deadlines so $d_1 \leq d_2 \leq ... \leq d_n$ and processed in this order. The minimum cumulative required resources needed to meet all deadlines coming before a given time t is

$$F_w(t) = \sum_{n \in \{n | d_n \leq t\}} w_n, \qquad \forall t \geq 0 \tag{6}$$

And the cumulative resources assigned up to a given time t is $F_x(t) = \int_0^t x(\sigma) d\sigma$.

We can now form the following constraints for the system:

- The assigned resources are at least as large as the minimum cumulative required resources at time t:
  $F_x(t) \geq F_w(t) \;\; \forall t \in [0, d_N]$.
- We cannot assign more resources if there is no more remaining workload:
  $F_x(t) \leq F_w(d_N), \;\; \forall t \in [0, d_N]$.
- Since $0 \leq x(t) \leq 1$ we have
  $0 \leq F_x(t) \leq t, \;\; \forall t \in [0, d_N]$.

Summarizing, the feasibility conditions for the input are as follows: $\min\{F_w(d_N), t\} \geq F_x(t) \geq F_w(t), \;\; \forall t \in [0, d_N]$. This means the system has a solution only if $F_w(t) \leq t \;\; \forall t \in [0, d_N]$. From here on, we will focus on the interval $[0, d_N]$ and assume the above conditions are satisfied.

To illustrate the relation between these constraints we have made an example based on the theory from [9]. Suppose we have five jobs in the form of $F_w(t)$ and a sample input $x(t)$ in the form of $F_x(t)$. This set contains $N = 5$ jobs with workload vector $w = [1, 2, 0.5, 3, 1]$ and deadline vector $d = [2, 4, 6, 8, 10]$. Then Figure 3 show the relation between the constraints.

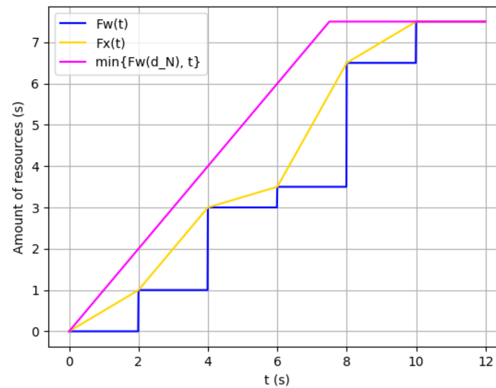


Figure 3: An example illustrating the relation between constraints

To ease some notation, [9] defines the remaining minimum required amount of resources at time t and for a given deadline d_n where $t < d_n$ as follows:

$$\lambda(t; d_n) = F_w(d_n) - F_x(t) \tag{7}$$

This is the required amount of resources at time t to meet d_n. If it is negative, the deadline is already satisfied. We have $\max \lambda(t; d_n) = d_n - t$ since $F_w(t) \leq t$. These are the total amount of available resources. We can define its complement as follows: $\lambda^c(t; d_n) = d_n - t - \lambda(t; d_n)$ and since $y(t) \in [0, 1]$ we can define the complement of $y(t)$ as $y^c(t) = 1 - y(t)$ and this is also in the interval [0,1].

Finally, we have the multiple job allocation problem. Given w, d and y_0 our problem is to find the input x(t) that minimizes $\max y(t)$:

Multiple offline job allocation problem: (8)

$$\min_{x(t)} \max_{t \in [0, d_N]} y(t) \tag{9}$$

$$\text{s.t. } y(t) = \frac{1}{\tau} \int_0^t e^{-\frac{t-\sigma}{\tau}} x(\sigma) d\sigma + y_0 e^{-\frac{t}{\tau}} \tag{10}$$

$$F_x(d_k) \geq F_w(d_k), \quad \forall k \in \{1, 2, ..., N-1\} \tag{11}$$

$$F_x(d_N) = F_w(d_N) \tag{12}$$

$$0 \leq x(t) \leq 1, \quad \forall t \in [0, d_N] \tag{13}$$

System definition for a single offline job

Later, we will describe how [9] minimizes the maximum temperature for multiple offline jobs. However, in order to do this and to later generalize this to an online setting we first simplify the model to one offline job, in the same way as in [9]. We consider a single job becoming available at $t = 0$ with a workload w and a deadline d.

Here we have

$$F_w(t) = \begin{cases} w & \text{if } t \geq d \\ 0, & \text{if } t < d \end{cases} \tag{14}$$

and the deadline constraint

$$F_x(d) = w. \tag{15}$$

Given w, d and y_0 our problem is to find the input x(t) that minimizes max y(t):

Single offline job allocation problem: (16)

$$\min_{x(t)} \max_{t \in [0, d]} y(t) \tag{17}$$

$$\text{s.t. } y(t) = \frac{1}{\tau} \int_0^t e^{-\frac{t-\sigma}{\tau}} x(\sigma) d\sigma + y_0 e^{-\frac{t}{\tau}} \tag{18}$$

$$\int_0^d x(t) dt = w \tag{19}$$

$$0 \leq x(t) \leq 1 \;\; \forall t \in [0, d] \tag{20}$$

2.4 System definition online problem

Now, it is time to extend the idea for the offline problem to the online problem. In this case we will consider multiple jobs $n = 1, 2, ...$ arriving according to a Poisson arrival process with arrival rate λ. Each job has a deadline, which is the time after the arrival of the job by which it must be completed. This is different than in the offline model since there the deadline was equal to the time-point by which the job must be completed. For example in the online problem, if a job arrives at $t = r$ and

has deadline d, it must be completed by $t = r + d$. Each job also has a workload, which is the amount of (normalized) resources that is needed to complete the job. For example, if a job arrives at $t = r$, has workload w and the system immediately processes this job at full resources, it will be finished by $t = r + w$. Recall that we will consider only one type of job for this thesis. With this we mean that all jobs arriving to the system have the same workload w and deadline d. We assume $d \geq w > 0$ and that job n cannot be processed by the system before $t = r_n$, with r_n being the actual arrival time of job n. Finally, we assume that the capacity of the system is equal to one. This means there can only be one job in the system at a time. So if a job is already in the system and a new job arrives, it will be deleted.

An example of this problem is a locked door that can be unlocked by a card reader. There can only be one card read at a time, and the workload and deadline of each card to be read is the same.

There are very few analytical expressions for measuring certain quantities (such as the expected time until a new arrival at an arbitrary time) of this kind of queuing system namely an M/D/1 queue with a capacity of c. In our case $c = 1$. We will discuss the reasons of this in section 3.4.2. We have found an article giving an explicit solution for the mean queue length and the average waiting time [8]. Unfortunately this was not exactly what we were looking for as we are looking for expressions such as the expected number of processed jobs at a certain time-point. This makes it difficult to construct a realistic model for this multiple online job allocation problem. However, we can make a single online job allocation problem based on the single offline job allocation problem. As soon as a job arrives at $t = r$, we know its deadline and the system can choose to process it in a certain way. The problem for this single job is then similar to the single offline job allocation problem but instead of the job becoming available at $t = 0$ it becomes available at $t = r$. Since all jobs arriving to the system while this job is still being processed are deleted, we can now treat this as the single offline job problem allocation problem.

Given w, d and y_0 our problem is to find the input x(t) that minimizes max y(t):

Single online job allocation problem: (21)

$$\min_{x(t)} \max_{t \in [r, r+d]} y(t) \tag{22}$$

$$\text{s.t. } y(t) = \frac{1}{\tau} \int_r^t e^{-\frac{t-\sigma}{\tau}} x(\sigma) d\sigma + y(r) e^{-\frac{t-r}{\tau}} \tag{23}$$

$$\int_r^{r+d} x(t) dt = w \tag{24}$$

$$0 \leq x(t) \leq 1 \quad \forall t \in [r, r+d] \tag{25}$$

For the multiple online job allocation problem, we can repeat the *optimal* policy for each job as soon as it enters the system.

3 Optimal allocation

In this section we will shortly discuss how the algorithm for the offline problems as detailed in [9] optimally allocates the resources to minimize the maximum temperature. Afterwards we show how we implemented this algorithm into our online problem. Lastly, we will do some analysis on how the temperature should behave in our system when this algorithm is applied so that we can compare it to our simulated results later.

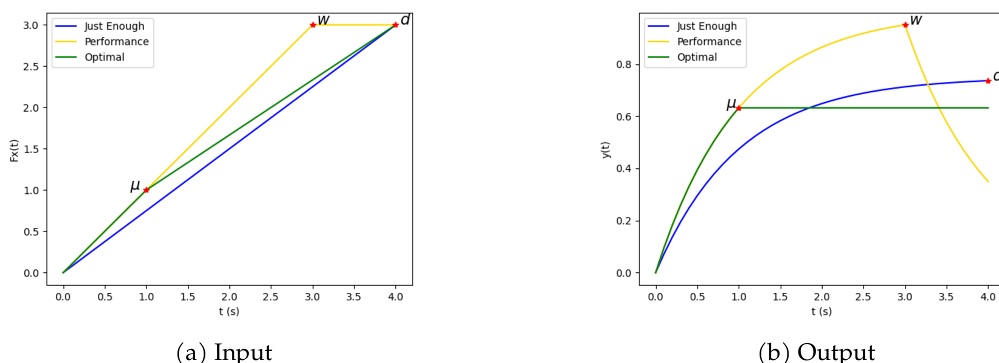
3.1 Policies

The algorithm we will implement in our system for online jobs is a policy on how to assign the resources of the system. As stated, we will apply the algorithm as detailed in [9], which introduces the *just enough* policy and the *performance* policy and then proposes its own *optimal* policy which is a combination of the former two. We will summarize these policies and their effects on the temperature in this section.

Performance policy: This policy processes a job immediately at full resources. When a job enters the system at time $t = 0$ with workload w and deadline d the *performance* policy will immediately process it at full resources and finish it at the earliest possible moment, which is actually $t = w$. This means that $x(t) = 1 \ \forall t \in [0, w)$ and $x(t) = 0$ otherwise. For this policy, deadlines set by the user are not relevant as the policy immediately processes the job and finishes it as soon as possible, regardless of the deadline. The maximum value of $y(t)$ will then occur at $t = w$ and $\max y(t) = 1 + (y_0 - 1)e^{-\frac{w}{tau}}$ because of Eq. (4).

Just enough policy: This policy assigns the resources such that the job is completed exactly at the deadline and not before. Once a job n enters the system at time $t = 0$ with workload w and deadline d the *just enough* policy will immediately process it, but with the least amount of resources such that it is completed exactly at $t = d$. This means that $x(t) = \frac{w}{d} \ \ \forall t \in [0, d)$ and $x(t) = 0$ otherwise so that $\int_0^d x(t) = \int_0^d w/d = w$, therefore the job is finished. The maximum value of $y(t)$ will then occur at $t = d$ and $\max y(t) = \frac{w}{d} + (y_0 - \frac{w}{d})e^{-\frac{d}{\tau}}$. This maximum temperature is lower than the maximum temperature of the *performance* policy. However, this policy does not solve the resource allocation problem (16) optimally because its utilization is not effective when the temperature is low. Therefore [9] proposes the optimal solution as described below.

Optimal: When a new job arrives, the optimal policy behaves like the *performance* policy until it reaches a steady-state temperature at a time-point μ, and behaves like the *just enough* policy afterwards. This will make sure that the temperature stays constant at this steady-state temperature instead of increasing and minimizes the maximum temperature as can be seen in Figure 4.



(a) Input (b) Output

Figure 4: Illustration of three different policies for the single job case with $y_0 = 0$.

3.2 Optimal allocation offline problem

Optimal allocation for a single offline job

The solution to the problem with a single offline job we will discuss here is proposed and proven to be optimal in [9]. This solution distinguishes whether the system is in a heating, steady or cooling state and behaves like the *performance* policy until a steady-state temperature is reached at a certain time $t = \mu$ and then behaves like the *just enough* policy. This means the system applies full or no utilization until it reaches $t = \mu$ (i.e. $x(t) = 1 \ \ \forall t \in [0, \mu)$ or $x(t) = 0 \ \ \forall t \in [0, \mu)$) and applies minimal utilization afterwards (i.e. $x(t) = (\textit{remaining workload}) \ / \ (\textit{remaining time until deadline}) \ \ \forall t \in [\mu, d)$).

We will repeat the definition of these states and how the steady-state temperature can be computed from [9] since we will use this in our solution to the online problem.

The three system states are defined in [9] as follows:

- The system is in a heating state when $y(t) < \frac{\lambda(t;d)}{d-t}$.

- The system is in a stable state when $y(t) = \frac{\lambda(t;d)}{d-t}$.
- The system is in a cooling state when $y(t) > \frac{\lambda(t;d)}{d-t}$.

So when $t = 0$, the system is in a heating state when $y_0 < \frac{w}{d}$, a stable state when $y_0 = \frac{w}{d}$ and a cooling state when $y_0 > \frac{w}{d}$.

Suppose there is one job with deadline d arriving to an empty system at $t = 0$. At time $t \in [0, d)$, if the system is in a heating state, the steady-state temperature $y_{ss}(t; d)$ is reached after applying full utilization for a certain time period. This steady-state temperature is proven in [9] to be as follows:

$$y_{ss}(t; d) = 1 - \frac{\lambda^c(t; d)}{\tau W_0(\frac{e^{\frac{d-t}{\tau}}}{\tau} \frac{\lambda^c(t;d)}{y^c(t)})}, \tag{26}$$

where W_0 denotes the real value of the principal branch of the *Lambert W function* which is defined to solve $x = ze^z$ as $z = W(x)$.

Suppose there is one job with deadline d arriving to an empty system at $t = 0$. At time $t \in [0, d)$, if the system is in a cooling state, the steady-state temperature $y_{ss}(t; d)$ is reached after applying zero utilization for a certain time period. This steady-state temperature is proven in [9] to be as follows:

$$y_{ss}(t; d) = \frac{\lambda(t; d)}{\tau W_0(\frac{e^{\frac{d-t}{\tau}}}{\tau} \frac{\lambda(t;d)}{y(t)})}. \tag{27}$$

The algorithm in [9] determines the moment the steady-state temperature is reached dependent on what state it is in and assigns the resources accordingly. If it is in a heating state it will apply full utilization until this moment and it will behave like the *just enough* policy afterwards. If it is in a cooling state it will apply zero utilization until this moment and it will behave like the *just enough* policy afterwards.

Optimal allocation for multiple offline jobs

When generalizing their solution to a multiple job problem, [9] looks for a certain, optimal division point. This division point divides the jobs into two sub-intervals: jobs with a deadline before this point and jobs with a deadline after this point. This division point d_v is proven in [9] to be found by the following rule:

$$d_v = \arg\max_{n \in \{1,\dots,N\}} \{y_{ss}(0, d_n)\}. \tag{28}$$

For the multiple offline job case, [9] applies the algorithm for the single offline job to the first interval and use the input given by this algorithm up to the division point. Afterwards, they look for a new division point and repeat the same process until the last job is processed. This division point yielding the optimal input is proven to be the deadline of the job with the maximum steady-state temperature. This solution to the multiple job allocation problem is also proven in [9] to be an optimal solution.

3.3 Optimal allocation online problem

Our proposed solution will follow the same idea as the one proposed in [9]. Once a job arrives, it will behave like the *performance* policy until the steady-state temperature is reached and like the *just enough* policy afterwards and repeat this for every job. We will start by applying the *performance* policy and the *just enough* policy to the system separately and see what the effects are. Afterwards we can combine these policies to determine our optimal policy. When applying the *optimal* policy on multiple offline jobs, [9] groups the jobs in an optimal way to find the steady-state temperature per group instead of per job. The fact that we don't do this will probably result in a higher variance than [9] since the input will switch values more often, which means the output will switch from heating or cooling more often and remain constant for shorter amounts of time.

3.4 Analytic results

In this section we will do some mathematical analysis on how the output should behave in our system when applying the *optimal* policy and how our queuing system should behave in terms of busy and idle time.

3.4.1 Output offline problem

In this section we will determine some expressions for the temperature of the offline single job system when applying the *optimal* policy.

Suppose our system has initial value $y_0 \leq \frac{w}{d}$. Since the algorithm applies a constant utilization of 1 in the interval $[0, \mu)$ we can apply Eq. (4) as follows to that interval:

$$y(t) = 1(1 - e^{-\frac{t-0}{\tau}}) + y(0)e^{-\frac{t-0}{\tau}}) = 1 - (1 - y_0)e^{-\frac{t}{\tau}}, \;\; \forall t \in [0, \mu). \tag{29}$$

Suppose our system has initial value $y_0 > \frac{w}{d}$. Since the algorithm applies a constant utilization of 0 in the interval $[0, \mu)$ we can apply Eq. (4) as follows to that interval:

$$y(t) = 0(1 - e^{-\frac{t-0}{\tau}}) + y(0)e^{-\frac{t-0}{\tau}}) = y_0 e^{-\frac{t}{\tau}}, \;\; \forall t \in [0, \mu). \tag{30}$$

Since the solution applies a constant utilization of $x(\mu) = (w - \mu)/(d - \mu)$ in the interval $[\mu, d)$, we can apply Eq. 4 as follows to that interval:

$$y(t) = x(\mu)(1 - e^{-\frac{t-\mu}{\tau}}) + y(\mu)e^{-\frac{t-\mu}{\tau}}, \;\; \forall t \in [\mu, d). \tag{31}$$

and we can determine $y(\mu)$ from that as follows:

$$(1 - e^{-\frac{t-\mu}{\tau}})y(\mu) = x(\mu)(1 - e^{-\frac{t-\mu}{\tau}}).$$

So

$$y(\mu) = x(\mu) = \frac{w - \mu}{d - \mu}. \tag{32}$$

Now, Eq. (**??**) gives us the following:

$$y(t) = x(\mu)(1 - e^{-\frac{t-\mu}{\tau}}) + x(\mu)e^{-\frac{t-\mu}{\tau}} = x(\mu)(1 - e^{-\frac{t-\mu}{\tau}} + e^{-\frac{t-\mu}{\tau}}) = x(\mu), \;\; \forall t \in [\mu, d). \tag{33}$$

3.4.2 Output online problem

We will determine expressions for the output of our online system in a steady-state analytically. We will determine these expressions for the *just enough* and *performance* policies separately.

Since the assigned resources are either 0, 1 or w/d and always constant in a certain interval, we can use Eq. (4) to compute the expected output of our system. In a system with the *just enough* or *performance* policy, there are two possible states the system can be in. For the *just enough* policy this is either minimal utilization ($x(t) = w/d$) or no utilization ($x(t) = 0$) and for the *performance* policy this is either maximal utilization ($x(t) = 1$) or no utilization ($x(t) = 0$). We can calculate the expected output at each time-point of these states since the assigned resources are constant while the system is in that state.

Just enough
Let $y_{J,0}$ denote the expected output of the system applying the *just enough* policy at the end of the idle period (so when a new job has just arrived) and let $y_{J,1}$ denote the expected output of this system at the end of the busy period (so when a job has just finished processing). Suppose we are currently at

time $t = r_1 + d$, and a job that arrived at $t = r_1$ with workload w and deadline d has just finished processing. Using Eq. (4) and the fact that the output at $t = r_1$ is equal to the output of the system at the end of an idle period $y_{J,0}$, we get the following for the output at this time:

$$y_{J,1} = x(r_1)(1 - e^{-\frac{r_1+d-r_1}{\tau}}) + e^{-\frac{r_1+d-r_1}{\tau}} y_{J,0} = \frac{w}{d}(1 - e^{-\frac{d}{\tau}}) + e^{-\frac{d}{\tau}} y_{J,0}. \tag{34}$$

We now want to know the expected time between the finishing of a job and the arrival of a new job to find the expected output at the end of an idle period. Suppose the jobs arrive according to a Poisson process with arrival rate λ. Unfortunately, we cannot say that at any time t the expected time until a new job arrives is $\frac{1}{\lambda}$. This is because of two reasons. First, it is true that on average λ jobs per time unit arrive to the system but a proportion of those arrivals finds the system filled to capacity and leave. Therefore the rate at which customers actually enter the system is not equal to λ. Second, it is not appropriate to model this system as a birth-death process since the service times no longer have their memorylessness property. [12]

There are very few analytical expressions for the measuring quantities of this kind of queuing system. Unfortunately, we could not find a satisfactory analytical expression for the time between the finishing of a job and the arrival of a new job. Therefore we decided to find an approximate value for the time between the finishing of a job and the arrival of a new job by running a simulation. Let this value be ρ_J for the *just enough* policy and ρ_P for the *performance policy*. Then at time $t = r_1 + d$ the expected time until a new job arrives is ρ_J for the *just enough* policy. Now the expected output at this time (the end of the idle period) is:

$$y_{J,0} = x(r_1 + d)(1 - e^{-\frac{\rho_J}{\tau}}) + y_{J,1} e^{-\frac{\rho_J}{\tau}} = y_{J,1} e^{-\frac{\rho_J}{\tau}} \tag{35}$$

since the utilization during the idle period is zero per definition.

Let $\alpha_J = e^{-\frac{\rho_J}{\tau}}$ and $\beta_J = e^{-\frac{d}{\tau}}$. We can now substitute Eq. (35) into Eq. (34) to solve the system of equations:

$$y_{J,1} = \frac{w}{d}(1 - \beta_J) + \alpha_J \beta_J y_{J,1} \tag{36}$$

$$\text{so } y_{J,1} = \frac{1 - \beta_J}{1 - \alpha_J \beta_J} \frac{w}{d} \tag{37}$$

$$\text{and } y_{J,0} = \frac{\alpha_J(1 - \beta_J)}{1 - \alpha_J \beta_J} \frac{w}{d}. \tag{38}$$

Performance

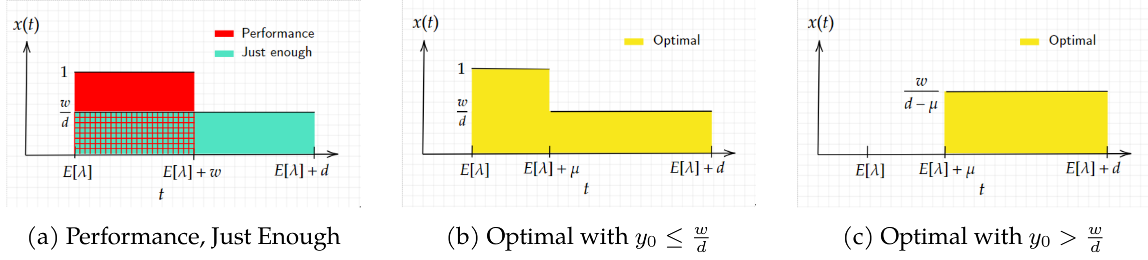
Let $y_{P,0}$ denote the expected output of the system applying the *performance* policy at the end of the idle period (so when a new job has just arrived) and let $y_{P,1}$ denote the expected output of this system at the end of the busy period (so when a job has just finished processing). We can easily find these values by replacing d with w in Eqs. (37) and (38) since the performance policy immediately starts processing with full utilization for the duration of the workload, regardless of the deadline. Let $\alpha_P = e^{-\frac{w}{\tau}}$ and $\beta_P = e^{-\frac{\rho_P}{\tau}}$ Therefore we get:

$$y_{P,1} = \frac{1 - \beta}{1 - \alpha_P \beta_P} \tag{39}$$

$$\text{and } y_{P,0} = \frac{\alpha_P(1 - \beta_P)}{1 - \alpha_P \beta_P}. \tag{40}$$

3.4.3 Queuing model

We will analytically determine the ratio our system is busy or idle analytically for all three policies separately. The proportion of time the system applying the *just enough* policy is busy is $\frac{w}{d}$ multiplied by d and divided by the expectation of the arrival rate plus the deadline, as can be seen in Figure 5a.



(a) Performance, Just Enough (b) Optimal with $y_0 \leq \frac{w}{d}$ (c) Optimal with $y_0 > \frac{w}{d}$

Figure 5: Workload distribution over time for three different policies

The busy ratio of our *just enough* system is $\pi_1 = \frac{w}{E[\lambda]+d}$ and the ratio our system is idle is $\pi_0 = 1 - \pi_1 = \frac{E[\lambda]+d-w}{E[\lambda]+d}$. The ratio's of our *performance* policy are then the same as that of the *just enough* policy but we replace d by w, which can aslo be seen in Figure 5a. If we process a job at a single utilization level - which is the case for the *just enough* and *performance* policies - we can revert back to the M/D/1 model since our workload is fixed. However, for the *optimal* policy we have a variable utilization level so in general there is no simple queuing model that describes it. Suppose the temperature of the system when a job enters the system is equal to y_0. The ratio's of the *optimal* policy are a different for when the system is in a heating ($y_0 \leq \frac{w}{d}$) or in a cooling state ($y_0 > \frac{w}{d}$), as can be seen in Figures 5b and 5c. When the system is in a heating state, the system does the same amount of work in the same amount of time as the *just enough* policy. However, when the system is in a cooling state, the system does the same amount of work in a smaller amount of time, namely $d - \mu$. In the heating state the busy and idle ratio's of the *optimal* policy are therefore the same as for the *just enough* policy and for the ratio's of the cooling state we replace d by $d - \mu$. We will leave these separate and we will not combine them due to the variable utilization level.

4 Simulation

In this section we will shortly discuss how we set up the different simulations and show some pseudo-code of the simulation for the online problem. Afterwards, we will verify our simulation and show some results. However, we will start with a brief explanation on how we discretized our system.

4.1 Discretization

In our model $x(t)$ and $y(t)$ are described as continuous functions of t. However in our simulation, x and y will become vectors x and y respectively. We can iterate over x and y at integer time-points, but not in between those. Therefore we chose to simulate in milliseconds, so the time between x[k] and x[k+1] is equal to one millisecond, the same holds for y. We have to be careful in how we calculate y from x. For example, say we have y[0] = 0 and we assign x[i] = 1 for $i \in \{0, 1, 2\}$. In that same iteration-loop, we cannot assign y[i] = y[0]$e^{-\frac{t}{\tau}}$+ x[0]$(1 - e^{-\frac{t}{\tau}})$ for $i \in \{0, 1, 2\}$ since then we overwrite y[0] to be equal to y[0] + 1 = 1. Thus, we would start the y-calculation one iteration later. We would also have to go one iteration further because when we do the same when assigning x[i] for $i \in \{3, 4, 5\}$, the iterations would always skip one value of y, in this case y[3]. Thus, if the iterations over x go from $i = \{k, .., n\}$ the iterations over y would go from $i = \{k + 1, ..., n + 1\}$. This was not necessary for the single offline job allocation problem as the calculations were very simple but it was for the multiple online job simulation.

4.2 Simulation offline problem

We started by simulating the solution to the problem for one offline job arriving to an empty system. The implementation of the functions $F_w(t), F_x(t)$ and $\lambda(t; d)$ (Eqs. (14), (15) and (7) respectively) were very straight-forward. For the implementation of Eq. (3) we actually used Eqs. (29), (30) and (31) in our simulation resulting in the following pseudo-code:

```
if y0 <= workload / deadline:
```

```
        if t < mu:
            y[t] = 1 - (1 - y_0) * exp(-t/tau)
        else:
            y[t] = x[mu] * (1 - exp(-t/tau)) + y[mu] * exp(-t/tau)
    else:
        if t < mu:
            y[t] = y_0 * exp(-t/tau)
        else:
            y[t] = x[mu] * (1 - exp(-t/tau)) + y[mu] * exp(-t/tau)
```

Finally, we implemented the algorithm as detailed in [9] which was pretty straight-forward. The algorithm would yield a vector x as optimal input. The code can be found in Appendix A.

4.3 Simulation online problem

For our online simulation we used the Python library DiscreteEventSimulation from the 2020 project of Module 8 of Applied Mathematics at the University of Twente [1]. This library contains some basic classes and functions so we can build our own discrete events. We started by simulating the *performance* and *just enough* policies separately and finally combined them into the *optimal* policy. In this section we will first discuss the pseudo-code of those two policies and afterwards we will discuss the pseudo-code of the simulation implementing the *optimal* policy. The actual code of the *just enough* and *performance* policies can be found in Appendix B. Most of the code of the *just enough* and *performance* policies is the same, except that in the *performance* policy the deadline is set equal to the workload. In the pseudo-code, "Time" is used to denote the time-point at which the calculations are being done.

The pseudocode for when a job arrives is as follows:

```
Arrival
    If Time == 0:
        Start "start processing"
    If the number of jobs in the system > 0:
        Delete job
    If the number of jobs in the system == 0:
        If Time > 0:
            For t in [prevDeadline, Time]:
                y[t] = exp(-(t - prevDeadline)/tau) * prevTempArrival
        prevArrival = Time
        prevTempDeadline = y[Time]
        Start "start processing"
    Schedule "arrival"  at Time + exponential time with rate "arrivalRate"
```

We treat "Time ==0" separately here as to not get a division by 0 and since we give the system an initial value for y[0] the temperature is already "saved".

The pseudo-code for the starting of processing is as follows:

```
Start processing (just enough and performance)
    x[Time] = workload / deadline
    For t in [Time + 1, Time + deadline - 1]:
        x[t] = workload / deadline
        y[t] = exp(-(t - prevArrival)/tau) * prevTemp
            + ( 1 - exp(-(t - prevDeadline)/tau)) * workload / deadline
    y[Time + deadline] = exp(-(t - prevArrival)/tau) * prevTemp
            + ( 1 - exp(-(t - prevDeadline)/tau)) * workload / deadline
    prevTemp = y[Time + deadline]
    Schedule "end processing" at Time + deadline
```

We take x[Time] and y[Time + deadline] out of the loop so that we don't overwrite the initial temperature at the moment a job arrives with a wrong temperature and we don't have an "extra" value for x, as described in section 4.1. The code above is for the *just enough* policy. If we replace deadline by workload we will get the *performance* policy.

The last type of event left to code is the "end processing" event:

```
End processing
    prevDeadline = Time
    If Time > timeToEndSimulation:
        Stop Simulation
```

We start our simulation by triggering an arrival at Time = 0.

To be able to simulate the *optimal* policy, we needed to add a function that calculates the steady-state temperature for each job at the moment it arrives so we know when to switch from policies. Next to that, the "start processing" event needed to be altered to implement that switching. The code of the *optimal* policy can be found in Appendix C.

The pseudo-code for the calculation of the steady-state temperature yss is as follows:

```
yss(y0):
    if y0 <= workload / deadline
        yc = 1 - y0
        lambc = deadline - workload
        yss = 1 - (lambc / (tau * (lambertW(exp(deadline/tau))/tau * lambc/yc).real))
    else:
        lamb = workload
        yss = lamb / (tau * (lambertW(exp(workload/tau))/tau * lamb/y0).real)
```

And the pseudo-code for the "start processing" event was altered as follows:

```
Start processing (optimal)
    workDone = 0
    yss = yss(y[Time])
    if y[Time] <= workload / deadline
        x[Time] = 1
        For t in [Time + 1, Time + deadline - 1]:
            if not steadyState:
                x[t] = 1
                y[t] = exp(-(t - prevArrival)/tau) * prevTemp
                    + ( 1 - exp(-(t - prevDeadline)/tau))
                    if y[t] >= yss:
                        steadyState = True
                        mu = t
                        workDone = t - prevArrival
                        steadyStateTemp = y[t]
                        Quit for loop
        y[Time + deadline] = exp(-(t - prevArrival)/tau) * prevTemp
                    + ( 1 - exp(-(t - prevDeadline)/tau))
        else:
            x[Time] = 0
            For t in [Time + 1, Time + deadline - 1]:
                if not steadyState:
                x[t] = 0
                y[t] = exp(-(t - prevArrival)/tau) * prevTemp
                if y[t] <= yss:
                        steadyState = True
                        mu = t
```

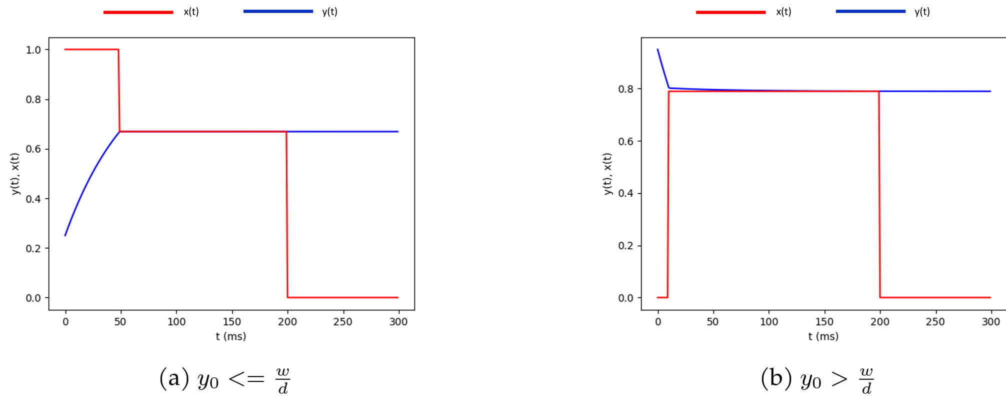
```
                workDone = 0
                steadyStateTemp = y[t]
                Quit for loop
        y[Time + deadline] = exp(-(t - prevArrival)/tau) * prevTemp

    if steadyState:
        x[mu] = (workload - workDone) / (prevArrival + deadline - mu)
        For t in [mu + 1, Time + deadline - 1]:
            x[t] = (workload - workDone) / (prevArrival + deadline - mu)
            y[t] = exp(-(t - prevArrival)/tau) * steadyStateTemp
                + ( 1 - exp(-(t - prevDeadline)/tau)) * (workload - workDone) /
                (prevArrival + deadline - mu)
        y[Time + deadline] = exp(-(t - prevArrival)/tau) * steadyStateTemp
                + ( 1 - exp(-(t - prevDeadline)/tau)) * (workload - workDone) /
                (prevArrival + deadline - mu)

prevTemp = y[Time + deadline]
steadyState = False
Schedule "end processing" at Time + deadline
```

4.4 Verification

In this section, we will compare the analytic expressions we determined in section 3.4 to the values our simulation gives to verify our simulation. We will start by verifying if our simulation implementing the *optimal* policy for a single job matches that of [9] and our expectation from section 3.4.1. Afterwards we will analytically compute a few values to verify that our simulation for multiple online jobs represents the reality in some way. Here, we will analyze the two main aspects of our simulation: the output and the queuing model.

4.4.1 Output offline job problem

Here, we will check if our simulation implementing the *optimal* policy for a single offline job does what we want it to do so we know we have a good basis for the online simulation. We used one job with $w = 150ms$, $d = 200ms$, $\tau = 60ms$ and two different initial values for $y(t)$, one above and one below $\frac{w}{d}$. Figure 6a shows the output for $y_0 = 0.25 \leq \frac{w}{d}$ and Figure 6b shows the output for $y_0 = 0.95 > \frac{w}{d}$.



(a) $y_0 <= \frac{w}{d}$

(b) $y_0 > \frac{w}{d}$

Figure 6: Output of single offline job with different initial values of $y(t)$

As we can see, the input starts at 0 or 1 and the output sinks or rises respectively, as we expect from Eqs. (29) and (30. When the steady-state is reached the input rises or drops to the value the output has at that time-point, as we expect from Eqs. (32) and (33).

4.4.2 Output online job problem

Before we verify our simulation with the online system, we will verify our simulation with a system with one periodic type of job (with workload w and deadline d), this means we are looking at a D/D/1 queue with a capacity of one. Let's say this job arrives every r time units. We can then calculate the output at the end of the idle and busy states exactly and compare this to what values our simulation gives. We do not need to use ρ_1 or ρ_2 here since we know the time between the finishing of a job and a new job arriving as this is discrete and equal to $r - d$. If the errors of those simulation values are small we can be assured that the basis of our simulation is satisfactory. We get the following parameters for Eqs. (37) and (38) when we apply the *just enough* policy: $\alpha_J = e^{-\frac{d}{\tau}}$ and $\beta_J = e^{-\frac{r-d}{\tau}}$. All these vaulues are known since we can choose our parameters. When applying the performance policy, we replace d by w. We take our parameter values as in Table 1. We simulated for 100000 time units and took the mean of the outputs at the end of the idle period and at the end of the bsy period as our simulation value. We used Eqs. (37), (38), (39) and (40) to determine the analytic values. The analytically calculated values and simulated values for the different outputs of the different policies and the errors can be found in Table 2.

Table 1: Values used to compare the analytical results to the simulation results.

Parameter	symbol	unit	value
Workload	w	ms	40
Deadline	d	ms	70
Time between arrivals	r	ms	100
System dependent parameter	τ	ms	200

Table 2: Analytical results versus simulation results of a D/D/1 queue with capacity of one.

Quantity	Analytical ($\approx$)	Simulation ($\approx$)	Absolute error	Relative error
$y_{J,1}$	0.4289	0.4264	0.0025%	0.58%
$y_{J,0}$	0.3691	0.3666	0.25%	0.68%
$y_{P,1}$	0.4607	0.4531	0.0076%	1.67%
$y_{P,0}$	0.3413	0.3353	0.60%	1.76%

The errors are all very small, so we can be satisfied with the basis of our simulation.

We can now fill in our parameters to verify our simulation for the online system. We will verify the output $y(t)$ of the *just enough* and *performance* simulations separately. If these give us satisfactory results, then we can use the same way of calculating the output for the *optimal* policy. We take the values from Table 3. Using these values and simulating over 100.000 time-points, our simulation gave us an approximate value for ρ, which is also displayed in Table 3. The values used for the parameters are summarized in Table 3. We simulated for 100000 time units and took the mean of the outputs at the end of the idle period and at the end of the busy period as our simulation values. The analytically calculated values and simulated values for the different outputs of the different policies and the errors can be found in Table 4, where we again used Eqs. (37), (38), (39) and (40) to determine the analytic values.

We can see that the errors are all very low, so we can conclude that the output-calculation of our simulation is representing the reality quite well.

4.4.3 Queuing model

To verify that our queuing model is working properly, we can compute the ratio the system is idle or busy analytically and compare this to the results of our simulation. We have different expressions for the busy and idle ratio's of the *optimal* policy when the system is in a heating or a cooling state as stated in section 3.4.3. The amount of time the system is in a heating or cooling state is dependent

Table 3: Values used to compare the analytical results to the simulation results.

Parameter	symbol	unit	value
Workload	w	ms	40
Deadline	d	ms	70
Arrival rate	λ	1/(expected ms between arrivals)	1/50
System dependent parameter	τ	ms	2000
Expected time between arrivals just enough	ρ_J	ms	51.3277
Expected time between arrivals performance	ρ_P	ms	51.7961

Table 4: Analytical results versus simulation results.

Quantity	Analytical ($\approx$)	Simulation ($\approx$)	Absolute error	Relative error
$y_{J,1}$	0.3339	0.3274	0.65%	1.9%
$y_{J,0}$	0.3254	0.3189	0.66%	2.0%
$y_{P,1}$	0.4414	0.4288	1.3%	2.9%
$y_{P,0}$	0.4301	0.4176	1.3%	2.9%

on basically all of the different parameters of our system, however we have seen in our simulation that when $y_0 = 0$ and using the parameters of Table 3, the system does not reach the cooling state. Therefore, we will only compute and compare the ratio's of the heating state. To determine the busy ratio in our simulation we can simply sum up all elements of our resource vector and divide that by its length. The idle ratio is then 1 - busy ratio. The analytically computed values, the simulated values and the errors can be found in Table 5. Here, the analytical values are determined by the ratio's as described in section 3.4.3 and again using the values of Table 3.

Table 5: Analytical results versus simulation results.

Quantity	Analytical	Simulation ($\approx$)	Absolute error	Relative error
Busy ratio Just Enough	1/3	0.3297	0.36%	1.1%
Idle ratio Just Enough	2/3	0.3703	0.36%	0.54%
Busy ratio Performance	4/9	0.4359	1.0%	2.3%
Idle ratio Performance	5/9	0.5641	0.85%	1.53%
Busy ratio Optimal	1/3	0.3272	0.61%	1.8%
Idle ratio Optimal	2/3	0.6728	0.61%	0.92%

These errors are all low enough to conclude that our queuing model is also representing the reality quite well.

4.5 Simulation results

In Table 6 the results of a simulation with $y_0 = 0$, $\tau = 2000ms$, $d = 70ms$, $\lambda = 1/50$ and $w = 20, 40, 60ms$ can be found. For all of these simulations we used a simulation time of 10.000ms. We have displayed the mean output and the mean output of the system when a job departs for all policies and the mean steady-state output for the *optimal* policy. All of these outputs are taken over the steady-state, excluding the warm-up or cool-down period. In Table 7 the absolute and relative differences between the simulated departure times of the *just enough* and *optimal* policies can be found. Here, the relative difference is calculated as follows: (*just enough output - optimal output*) / *just enough output*. In Table 8 the results of simulations with the same parameters but $\tau = 200ms$ can be found. When using $\tau = 200ms$, the error of the mean arrival output of the *performance* policy rises to 5% to 6%. However all other errors remain below 5% and the arrival output is the least interesting to us. When using this same τ but setting $\lambda = 1/80$ the errors are very low. We specifically look at the departure times of each job because this is always the maximum output for the *just enough* and *performance* policies, therefore the goal is to minimize this output.

The following observations can be made about these three figures.

Table 6: Analytical results versus simulation results.

Workload	20	40	60
Just enough			
Mean Output	0.1606	0.3349	0.5618
Mean Departure Output	0.1629	0.3380	0.5707
Performance			
Mean Output	0.2662	0.4904	0.5177
Mean Departure Output	0.2734	0.4984	0.5895
Optimal			
Mean Output	0.1457	0.3138	0.4595
Mean Departure Output	0.1480	0.3180	0.4670
Mean Steady-State Output	0.1480	0.3180	0.4670

Table 7: Difference departure output Just Enough and Optimal from Table 6

Workload	Absolute Difference	Relative difference
20	1.5%	9.21%
40	2.0%	5.93%
60	10.4%	18.17%

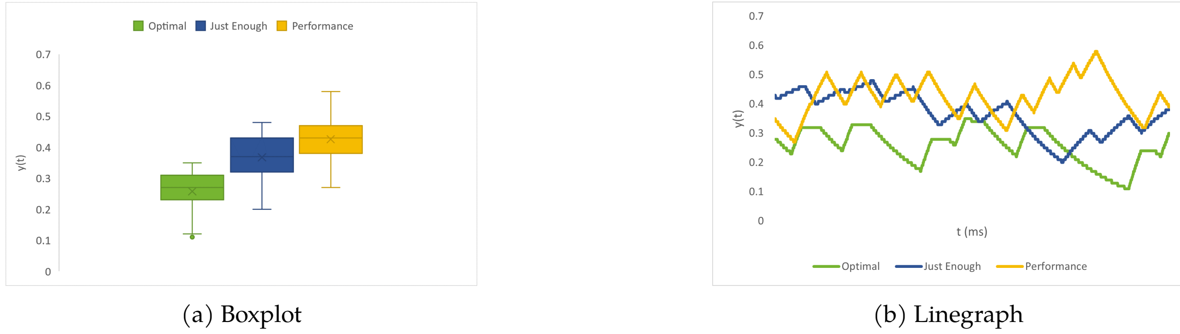
Table 8: Departure output Just Enough and Optimal for $\tau = 200$ and various workloads.

Workload	20	40	60
Just enough	0.1861	0.3955	0.5976
Performance	0.3071	0.5177	0.6168
Optimal	0.1653	0.3448	0.5628
Relative Difference JE - O	8.0%	12.8%	5.82%
Relative Difference P - O	44.3%	33.4%	8.75%

- First of all, we notice that for the *optimal* policy, the mean steady-state output is (almost) equal to the mean departure output for every case.
- The mean departure output of the *optimal* policy is always lower than that of the other policies.
- In the case of $\tau = 2000ms$, the difference between the *optimal* and the *just enough* policy is biggest for a high workload of $w = 60ms$ and lowest for a medium workload of $w = 40ms$. In the case of $\tau = 200ms$ the difference is actually the biggest for a medium workload of $w = 40ms$.
- In both cases of $\tau = 200ms, 2000ms$, the difference between the *performance* and the *just enough* policy is biggest for a low workload of $w = 20ms$ and smallest for a high workload of $w = 60ms$. In the case of $\tau = 2000ms$ the relative difference in the mean departure output for $w = 20ms$ is 45.9%.
- The differences for the case of $\tau = 200ms$ lie much closer together than the differences for $\tau = 2000ms$.
- We can also see that in any case, a higher workload means a higher output.

In Figures 9 to 12 in Appendix D multiple box-plots and line-graphs can be found. For various workloads and initial values for $y(t)$ we have plotted the simulated results of the different policies in the same plot. The parameters that were the same for every simulation are $\tau = 200ms$, $d = 70ms$ and $\lambda = 1/50$ and every time we have simulated over 10.000ms. In Figures 9 to 11 the results of the simulations with initial value $y_0 = 0$ and a low workload $w = 20ms$, medium workload $w = 40ms$ and a high workload $w = 60ms$ can be found. We have made one box-plot and line-graph for each workload over a smaller time-interval towards the end of the simulation so we can see the mean and variance over the steady-state and so that we can look at the differences between the policies clearer. This time-interval is about 1.000ms. The same is done for initial value $y_0 = 0.9$ and workloads $w = 20ms, 40ms$

in Figures 12 to 13. The box-plot and line-graph of initial value $y_0 = 0$ and workload $w = 40ms$ can also be found below in Figure 7.



(a) Boxplot (b) Linegraph

Figure 7: Results of the simulation with workload $w = 40ms$, $\tau = 200ms$, $y_0 = 0$, $d = 70ms$, $\lambda = 1/50$

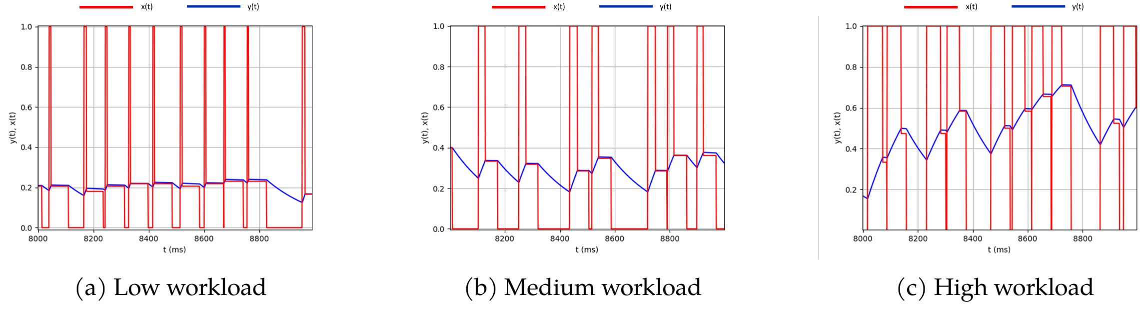
- We can see that the mean of the *optimal* policy is at least as low as the mean of the *just enough* policy and the maximum value of the *optimal* value is also as least as low as the maximum value of the *just enough* policy for every simulation.
- We can see that the mean and departure values of the *optimal* policy over the steady-state are quite a bit lower than those of the *just enough* policy for every simulation.
- The variance of the *performance* policy is usually the largest. The variance of the *just enough* and *optimal* policies lie quite close to one another except that in Figures 9a and 13a the variance of the *just enough* policy is a bit smaller.
- The *optimal* policy usually has less outliers, increases less when a new job arrives and decreases less when a new job departs.
- In the case of initial value $y_0 = 0.9$ we see that the difference between the *optimal* and *just enough* policy is smaller than when $y_0 = 0$.
- The difference between the *optimal* and *just enough* policies in a steady state seems the largest for a medium workload in most cases.

We have plotted the input $x(t)$ and output $y(t)$ of the *optimal* policy for various workloads over the time-interval approximately $t \in [8000, 9000]$ in Figure 8 to see how the output behaves. It is displayed a bit bigger in Appendix D, Figure 14. We have again plotted for a low workload $w = 20$, medium workload $w = 40$ and a high workload $w = 60$ and $y_0 = 0$, $\tau = 200$, $d = 70$ and $\lambda = 1/50$. We can see that the input $x(t)$ shoots up to 1 when a new job n arrives, then falls to somewhere just below, at or just above the current output at a certain time-point ($t = \mu_n$) and then falls down to 0 when the job is finished. We can see that the output $y(t)$ rises when the input $x(t) = 1$, stays approximately constant from $t = \mu_n$ until the job is finished and then falls down until a new job arrives.

5 Discussion

In this section we will make some observations and state some conclusions that can be made from the results in section 4.5. First of all, we have seen that playing with the parameters can cause very different values and it is important to keep looking at the errors of our output with respect to the analytically determined expected output and to think about what types of configurations of parameters are "logical" in the real world.

We have seen that the mean steady-state output of the *optimal* policy is very close to to the mean departure output over the steady-state. This means the simulation does what we expect it to: when



(a) Low workload (b) Medium workload (c) High workload

Figure 8: Relation of input x(t) and output y(t) for different workloads

a job arrives and the steady-state output is reached, the output will remain approximately the same until that job leaves the system. We can also see that in any case, a higher workload means a higher output. This is also what we expect, since the system will have to do more work in the same amount of time, and this means a higher temperature.

When looking at the steady-state outputs in Figures 9 - 11, we can clearly see that the *optimal* policy has lower mean and departure outputs than both the *performance* and the *just enough* policy when $y_0 = 0.9$ than when $y_0 = 0$. As for the case with a high initial temperature, wee see in the Figures **??** - 13 that the *optimal* policy has lower mean and departure outputs than the other two policies but the difference with the *just enough* policy is smaller. As stated in [9] the *just enough* policy does not assign the resources efficiently for *low* temperatures. Therefore it is not surprising that the differences between the policies are smaller when the initial temperature is high.

We can see in all the different steady-state simulations that the *optimal* policy produces a lower output than the *just enough* policy, both in the mean output and the mean departure output. That this difference is the largest for the medium workload is a logical consequence. For a high workload, all of the policies will behave more like the *performance* policy, since the closer the workload gets to the deadline, the more the *just enough* policy will behave like the *performance* policy and thus the hybrid *optimal* policy will do the same. This causes the differences between all policies to become smaller, this intuition is confirmed by the smaller relative differences in Table 8 and the behaviour of $x(t)$ and $y(t)$ during our simulation. We can see this behaviour in Figure 14c, as the *optimal* policy behaves like the *performance* policy for a long time, after which it behave like the *just enough* policy for a shorter time. For a low workload, the *optimal* policy will behave more like the *just enough* policy, since the system will have to process less workload in the same amount of time so it can spread the work more evenly. This is again confirmed by the smaller relative difference between the *just enough* and the *optimal* policy in Table 8 and can also be seen in Figure 14a, we see the system behaves like the *performance* policy for a very short time and then shoots down to behave like the *just enough* policy for quite long. That is why the differences between the *optimal* and *just enough* policies are smaller for a low workload. The relative difference between the *optimal* and *performance* policy are biggest for a small workload since this is when the *optimal* policy behaves least like the *performance* policy. Finally, we can see in 14b that for a medium workload, the time the system spends as the *performance* or *just enough* is less skewed to one or the other. In the case of $\tau = 200$, it is quite odd that the relative difference between the mean departure outputs of the *optimal* and *just enough* policies is the smallest for $w = 40$, as we can see in Table 7. This probably has to do with the value for τ since when this is 200, the relative difference is the highest for $w = 40$, as we expect.

That the variance of the *performance* policy is usually the largest is in line with our expectations, since this input shoots up to its maximum value very quickly when a job arrives and falls down at the earliest possible moment, so the temperature rises high and falls down quickly as well. We would have expected the *optimal* policy to usually have a smaller variance than the *just enough* policy, as the former aims to stay at a somewhat constant value. In [9] this expectation is confirmed as they state that a reduction in the variance of the temperature profile is a side-effect of the *optimal* policy. The fact that we don't see that in our results is likely the result of not grouping jobs together in an optimal way to

find the steady-state temperature per group of jobs instead of per job.

The behaviour of the input and output for different workloads in Figure 14 as we have described in section 4.5 is exactly what we expect from the system. The system behaves like the *performance* policy when a job arrives, then the input $x(t)$ shoots down to behave like the *just enough* policy until the job is finished. We can see that the value $x(t)$ takes on when the steady-state temperature is reached is approximately equal to the steady-state temperature, which is in line with Eq. (5) for when we have a constant output $y(t)$.

Finally, in general the output of the *optimal* policy does what we hope it will do: it minimizes the maximum processor temperature by smartly assigning the resources, with respect to the *just enough* and *performance* policies. We have seen that the *optimal* policy can decrease the maximum temperature in terms of $y(t)$ with up to 18.17% with respect to the *just enough* policy and up to 45.9% with respect to the *performance* policy.

There are still some improvements left in our implementation of the *optimal* policy into an online scheduling algorithm. First of all, in this thesis we have implemented the *optimal* policy into quite a specific case of online scheduling with only one type of job and a capacity of only one. This could be extended to become a system with more types of jobs (where each type of job has the same workload, deadline and arrival rate) and/or where the workload and deadlines are stochastic as well. Further research could also include researching the effect the *optimal* policy has on such a system that has a larger capacity, maybe with more jobs that can be processed at the same time and/or that the jobs can be put on hold for a certain time, so they are not immediately deleted if they arrive to the system and find the system full. Furthermore, the *just enough* and *optimal* policy finish a job exactly before it's deadline and not earlier. An arising question is how many jobs are rejected by the *optimal* policy in a system with a capacity. If we set a constraint on how many jobs can be rejected we expect the *optimal* policy would then skew more to the *performance* policy, as the latter finishes a job as soon as possible.

We have modeled an M/D/1 queue with a capacity of one. Even though there are many real-life systems that can be modeled as this kind of queuing system there are not many known analytical expressions for values such as the expected time until a new job arrives when the last one has just finished. This made it more difficult to perform an analysis on this queuing system when applying the *optimal* policy since we would have to know this value. We made do with a value taken from our simulations but it would be nice for further research to find an exact value for this. Further research could include how the *optimal* policy would effect the temperature in a M/M/1 queue with a capacity of one or a M/D/1 queue with infinite capacity because for these systems there are known analytical expressions for values such as the expected time until the next arrival.

The way we have implemented the *optimal* policy is that it finds the steady-state temperature per job, while the offline multiple job algorithm in [9] groups the jobs in an optimal way and finds the steady-state temperature of this group of jobs. In further research, it could be interesting to see how that can be implemented into an online setting. This would probably cause the system to have a more constant overall value since the input will not have to jump to different values as much and thus the output will not either. This would cause a smaller variance in the output which would mean the policy was more reliable. Lastly, we have not looked at how the leakage current which we shortly described in section 2.2 affects our output, this could cause the actual temperature of the system to be different from how we calculated it in terms of $y(t)$.
We have not shown that this *optimal* policy is actually optimal like they do in [9], but as is said in [13], an online algorithm cannot in general construct an optimal energy schedule. It could be interesting to try however, in the future. Determining the competitive ratio would be a good start at least.

6 Conclusion

To conclude, we have shown that the *optimal* policy can be implemented as an online scheduling method to minimize the maximum processor temperature with respect to the *just enough* and the

optimal policies while still meeting the deadlines of jobs. In fact, we have shown that the *optimal* policy decreases the temperature with up to 18% with respect to the *just enough* policy and up to 44% with respect to the *performance* policy. The *optimal* policy is mainly useful when the workload of a job is medium with respect to its deadline, so that the ratio is around 1/2. This is shown in a system with one type of job that has a known workload and deadline but with a stochastic arrival process and where there can only be one job in the system at a time. This M/D/1 queuing system with a capacity of one is a more difficult system than anticipated, as there do not exist analytical expressions for certain values we need. Further research would be needed to find these.
The research could be expanded and generalized to encompass a more general system This could encompass different types of jobs with a variable workload and/or deadline, a larger capacity in the form of more jobs being processed at the same time and/or the queuing of jobs that find the system busy. The research could also be extended to find a way to decrease the variance of the *optimal* policy even more.

7 Bibliography

[1] Discrete event simulation in python. *University of Twente*, 2020.

[2] Amirali Shayan Arani. Online thermal-aware scheduling for multiple clock domain cmps. In *2007 IEEE International SOC Conference*, pages 137–140, 2007.

[3] N. Bansal, T Kimbrel, and K Pruhs. Speed scaling to manage energy and temperature. *Journal of the ACM*, 54(1):1–39, 2007.

[4] Qaisar Bashir, Muhammad Naeem Shehzad, Muhammad Naeem Awais, Sobia Baig, Muhammad Ghaffar Dogar, and Aamir Rashid. An online temperature-aware scheduling technique to avoid thermal emergencies in multiprocessor systems. *Computers Electrical Engineering*, 70:83–98, 2018.

[5] Thidapat Chantem, Robert P. Dick, and X. Sharon Hu. Temperature-aware scheduling and assignment for hard real-time applications on mpsocs. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '08, page 288–293, New York, NY, USA, 2008. Association for Computing Machinery.

[6] Youngmoon Lee, Hoon Sung Chwa, Kang G. Shin, and Shige Wang. Thermal-aware resource management for embedded real-time systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2857–2868, 2018.

[7] J. Meng, H. Tan, X. Y. Li, Z. Han, and B. Li. "online deadline-aware task dispatching and scheduling in edge computing. *n IEEE Transactions on Parallel and Distributed Systems*, 31(6):1270–1286, june 2020.

[8] Jean-Marie Garcia Olivier Brun. Analytical solution of finite capacity m/d/1 queues. *Journal of Applied Probability*, 37(4), December 2000.

[9] Baver Ozceylan, Boudewijn R. Haverkort, Maurits de Graaf, and Marco E. T. Gerards. Minimizing the maximum processor temperature by temperature aware scheduling of real-time tasks, (article has been accepted for inclusion in a future issue of the journal IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEM, as of june 2022).

[10] Pratyush Panda and Lothar Thiele. Cool shapers: Shaping real-time tasks for improved thermal guarantees. In *Proceedings - Design Automation Conference*, pages 468–473, 06 2011.

[11] Tjark Vredeveld. Stochastic online scheduling. *Computer Science - Research and Development*, 27:1–7, 01 2010.

[12] Wayne L. Winston. *Operations Research, Applications and Algorithms*. Thomson. 4 edition, 2004.

[13] F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced cpu energy. In *Proceedings of IEEE 36th Annual Foundations of Computer Science*, pages 374–382, 1995.

8 Appendix A

```
"""
 Created by Floor van Maarschalkerwaart
 This is a simulation for a single job with a known deadline and workload
 We apply the optimal policy

"""

import random
import numpy as np
import math
from scipy.special import lambertw
from matplotlib import pyplot as plt

# Initializing
dt = 1  # 10 milliseconds
t0 = 0  # initial time
y0 = 0.25  # standardized initial output
interval = np.arange(0, 300, dt)
n = len(interval)
d = int(200)  # deadline of job
w = 150  # workload of job
tau = 60  # 600 milliseconds
x = np.zeros((n,))  # initializing
t = int(0)  # current time
integermu = 0


# Minimum cumulative required resources to meet all deadlines coming before a time t
def Fw(t):
    global d, w
    if t >= d:
        func_val = w
    else:
        func_val = 0
    return func_val


# Cumulative resources assigned up to a given time t
def Fx(t):
    subarrayx = x[0:t]  # goes from index 0 to index t-1
    return np.sum(subarrayx)


# Remaining minimum required amount of resources at a time t for a deadline dn
def lamb(t, dn):
    lam = int(Fw(dn) - Fx(t))
    return lam


# Function to calculate output
def y(t):
    global integermu

    if y0 <= w / d:
        if integermu >= t:
            func_val = 1 - (1-y0)*math.exp(-t/tau)
        else:
            func_val = x[integermu] * (1-math.exp(-t/tau)) + y(integermu) * math.exp(-
    t/tau)
    else:
        if integermu >= t:
            func_val = y0 * math.exp(-t/tau)
        else:
            func_val = x[integermu] * (1-math.exp(-t/tau)) + y(integermu) * math.exp(-
    t/tau)
    return func_val


```

```
# Algorithm for a single job
def SJ(w, d, y0):
    global x, integermu
    if y0 <= w/d:
        mu = d - tau * (lambertw(math.exp(d/tau)/tau * (d-w)/(1-y0))).real
        integermu = int(mu)
        if integermu < 0:
            integermu = 0
        for t in np.arange(0, integermu):
            x[t] = 1  # Voor t = 0 tot mu
        w = w - integermu
    else:
        mu = d - tau * (lambertw(math.exp(d/tau)/tau * w/y0)).real
        integermu = int(mu)
        for t in np.arange(0, integermu):
            x[t] = 0  # Voor t = 0 tot mu
    for t in np.arange(integermu, d):
        x[t] = w/(d-integermu)  # Voor t=mu tot d
    return x


# Input
ltd = lamb(t, d)
y_input = y0
finalx = SJ(ltd, d - t, y_input)

temperature = np.zeros((n,))

for i in np.arange(0, n):
    temperature[i] = y(i)

print(finalx, temperature)

xaxis = np.arange(0, n)

plt.plot(xaxis, temperature, 0.5, color='red')
plt.xlabel('t (ms)')
plt.ylabel('y(t)')
plt.show()
```

9 Appendix B

```
"""
 Created by Floor van Maarschalkerwaart
 This is a simulation for multiple jobs with the same known deadline and workload
    arriving according to a Poisson arrival process
 We apply the just enough policy
 This code can be altered to apply the performance policy, as indicated at line 126

"""
import random
import math
import numpy as np
import DiscreteEventSimulation as DES
import sys
import matplotlib.pyplot as plt
np.set_printoptions(threshold=sys.maxsize)


# Function that performs some actions when a job starts processing
def startService(Time):
    global serviceRate, x, deadline

    # Allocation of resources and computation of output
    x[Time] = serviceRate/deadline
    for i in np.arange(Time + 1, Time+deadline):
        x[i] = serviceRate/deadline
```

```
26          temperature[i, 0] = math.exp(-(i - currSimTime1) / tau) * temperature[
    currSimTime1, 0] + (
27                          1 - math.exp(-(i - currSimTime1) / tau)) * serviceRate /
    deadline
28
29      temperature[Time + deadline, 0] = math.exp(-(Time + deadline - currSimTime1) / tau
    ) * temperature[currSimTime1, 0] + (
30                          1 - math.exp(-(Time + deadline - currSimTime1) / tau)) *
    serviceRate / deadline
31
32      # Trigger endservice event
33      DES.insertEvent(EndService(Time + deadline))
34
35
36  # Class that performs some actions when a job arrives
37  class Arrival(DES.Event):
38      def description(self):
39          return 'arrival'
40
41      def execute(self):
42          global numberOfCustomers, numberOfArrivals, temperature, x, currSimTime2,
    rejected, currSimTime1
43          global prevArrival, tau, deadline
44
45          # Take this out of loop to not overwrite initial values
46          if self.Time == 0:
47              numberOfCustomers += 1
48              numberOfArrivals += 1
49              currSimTime1 = self.Time
50              startService(self.Time)
51
52          # Reject arriving job if there is already a job in the system
53          if self.Time <= currSimTime2:
54              rejected += 1
55          else:
56              if numberOfCustomers > 0:
57                  rejected += 1
58
59              if numberOfCustomers == 0:
60                  # Update counters
61                  numberOfCustomers += 1
62                  numberOfArrivals += 1
63
64                  # Save information about temperature
65                  if self.Time != 0:
66                      for i in np.arange(currSimTime2, self.Time + 1):
67                          temperature[i, 0] = math.exp(-(i - currSimTime2) / tau) *
    temperature[currSimTime2 - 1, 0]
68
69                      temperature[self.Time, 1] = 1  # 1 = arrival
70                      temperature[self.Time, 2] = numberOfArrivals  # Save number of job
71
72                      interArrival[self.Time] = self.Time - currSimTime2  # Time between
     deadline and arrival
73
74                  # Save current time
75                  currSimTime1 = self.Time
76
77                  # Trigger start service event
78                  startService(self.Time)
79
80          # Trigger next arrival
81          prevArrival = int(random.expovariate(arrivalRate))
82          DES.insertEvent(Arrival(self.Time + prevArrival))
83
84
85  # Class that performs some actions when a job has finished processing
86  class EndService(DES.Event):
87      def description(self):
88          return 'end service'
89
```

```
90     def execute(self):
91         global numberOfCustomers, currSimTime1, currSimTime2, numberOfArrivals,
    prevArrival, tau, deadline
92 
93         temperature[self.Time, 1] = 2  # 2 = departure of job
94         temperature[self.Time, 2] = numberOfArrivals  # Save number of job
95 
96         numberOfCustomers -= 1
97 
98         currSimTime2 = self.Time
99 
100        DES.stopSimulation = self.Time > timeToEndSimulation
101 
102 
103 # Initializing
104 numberOfArrivals = 0
105 numberOfCustomers = 0
106 initialTemperature = 0
107 currSimTime1 = 0
108 currSimTime2 = 0
109 rejected = 0
110 epsilon = 0.01
111 timeToEndSimulation = 10000
112 matrix_length = timeToEndSimulation + 1000  # To make sure there is room for all data
113 x = np.zeros((matrix_length,))  # Our resource vector
114 interArrival = np.zeros((matrix_length,))  # Our steady-state temperature vector
115 
116 # Initialize temperature matrix
117 # Column 1 is temperature; 2 is arrival, steady-state point or departure; 3 is number
    of job
118 temperature = np.zeros((matrix_length, 3))
119 temperature[0, 0] = initialTemperature  # Set initial temperature to 0
120 temperature[0, 1] = 1  # Classify first Arrival
121 temperature[0, 2] = 1  # First job
122 
123 # Set our parameters
124 arrivalRate = 1/50
125 serviceRate = 40
126 deadline = 70  # When we set deadline = serviceRate we should get the performance
    policy
127 tau = 200  # 200 milliseconds
128 prevArrival = 0
129 
130 # Trigger first arrival at time = 0 and run simulation
131 DES.insertEvent(Arrival(0))
132 DES.runSimulation()
133 
134 
135 ###########################
136 # Everything below is the computation of values to verify the simulation and/or
    collect results
137 ###########################
138 
139 
140 # Print assigned resources and temperature matrix
141 print(f'Assigned resources are {x}')
142 print(f'Temperature matrix is {temperature}')
143 print(f'Number of rejected jobs is {rejected} out of {numberOfArrivals + rejected}')
144 
145 # Identify indeces of non-arrival temperatures
146 index_arrival = np.where(temperature[0:matrix_length, 1] != 1)
147 # Identify indeces of non-deadline temperatures
148 index_deadline = np.where(temperature[0:matrix_length, 1] != 2)
149 # Identify time-points where we did not save any data
150 index_zero = np.where(temperature[0:matrix_length, 1] == 0)
151 
152 # Make a matrix of only arrival temperatures
153 arrivalTemperatures = np.delete(temperature, index_arrival, axis=0)
154 arrivalLength = len(arrivalTemperatures)
155 # Make a matrix of only departure temperatures
156 departureTemperatures = np.delete(temperature, index_deadline, axis=0)
```

```
157 departureLength = len(departureTemperatures)
158 # Make a matrix of all temperatures
159 allTemperatures = np.delete(temperature, index_zero, axis=0)
160 allLength = len(allTemperatures)
161 # Delete zero values
162 interzero = np.where(interArrival[0:matrix_length] == 0)
163 interArrival = np.delete(interArrival, interzero, axis=0)
164
165 # Save as csv file
166 temperature_final = np.around(temperature[0:timeToEndSimulation, 0], decimals=2)  # *
    alpha + T_a
167 temperature_final.tofile('temperature_JE.csv', sep=',')
168
169 # Plot temperature over time
170 plt.plot(np.arange(0, timeToEndSimulation), temperature[0:timeToEndSimulation, 0],
    0.2, color='blue')
171 plt.xlabel('t (s)')
172 plt.ylabel('y(t)')
173 plt.grid()
174 plt.show()
175
176 # Verify model
177 # Verify queueing simulation by checking idle/busy formula's
178 actualUtilization = sum(x[0:timeToEndSimulation])/len(x[0:timeToEndSimulation])
179 actualIdleTime = (len(x[0:timeToEndSimulation])-sum(x[0:timeToEndSimulation]))/len(x
    [0:timeToEndSimulation])
180
181 expectedUtilization = serviceRate / (deadline + 1/arrivalRate)
182 expectedIdleTime = (deadline + 1/arrivalRate - serviceRate)/(deadline + 1/arrivalRate)
183
184 print(f'System is utilized {actualUtilization} of the time while we expect {
    expectedUtilization}')
185 print(f'absolute error is {100*abs(expectedUtilization-actualUtilization)}, rel error
    is {100*abs(expectedUtilization-actualUtilization)/expectedUtilization}.')
186 print(f'System is idle {actualIdleTime} of the time while we expect {expectedIdleTime}
    ')
187 print(f'absolute error is {100*abs(expectedIdleTime-actualIdleTime)}, rel error is
    {100*abs(expectedIdleTime-actualIdleTime)/expectedIdleTime}')
188
189 # Verify temp calculation
190 interArrivalTime = np.mean(interArrival)
191 beta = math.exp(-deadline/tau)
192 alpha = math.exp(-interArrivalTime/tau)
193 expectedTempBusy = ((1 - beta)/(1 - alpha*beta)) * serviceRate/deadline
194 expectedTempIdle = alpha*expectedTempBusy
195
196 meanTemp = np.mean(allTemperatures[0:allLength, 0])
197 meanTempArrival = np.mean(arrivalTemperatures[0:arrivalLength, 0])
198 meanTempDeparture = np.mean(departureTemperatures[0:departureLength, 0])
199 maxTemp = max(allTemperatures[0:allLength, 0])
200
201 print(f'Actual mean temperature is {meanTemp}')
202 print(f'Mean arrival temperature is {meanTempArrival} while we expected {
    expectedTempIdle}')
203 print(f'absolute error is {abs(meanTempArrival - expectedTempIdle)*100}, rel error is
    {100 * abs(meanTempArrival - expectedTempIdle)/expectedTempIdle}')
204 print(f'Mean departure temperature is {meanTempDeparture} while we expected {
    expectedTempBusy}')
205 print(f'absolute error is {abs(meanTempDeparture- expectedTempBusy)*100}, rel error is
     {100*abs(meanTempDeparture- expectedTempBusy)/expectedTempBusy}')
206 print(f'Mean interarrival time is {np.mean(interArrival)}')
207 print(f'Max temperature is {maxTemp}')
208
209 print(f'Mean temp over steady state is {np.mean(temperature[6000:timeToEndSimulation,
    0])}')
210 print(f'Mean departure temp over steady state is {np.mean(departureTemperatures[int
    ((2*departureLength/3)):departureLength, 0])}')
```

10 Appendix C

```
"""
 Created by Floor van Maarschalkerwaart
 This is a simulation for multiple jobs with the same known deadline and workload
    arriving according to a Poisson arrival process
 We apply the optimal policy

"""
import random
import math
import numpy as np
import DiscreteEventSimulation as DES
import sys
import matplotlib.pyplot as plt
from scipy.special import lambertw

np.set_printoptions(threshold=sys.maxsize)


# Function to calculate steady-state output
def yss(y0):
    global tau, serviceRate, deadline
    y = y0
    if y0 <= serviceRate / deadline:
        yc = 1 - y
        lambc = deadline - serviceRate
        yss = 1 - (lambc / (tau * (lambertw((math.exp(deadline / tau) / tau) * (lambc
    / yc))).real))
    else:
        lamb = serviceRate
        yss = lamb / (tau * (lambertw((math.exp(deadline / tau) / tau) * (lamb / y))).
    real)
    return yss


# Function that performs some actions when a job starts processing
def startService(Time):
    global serviceRate, x, deadline, steadyState, tau, temperature, steadyStateTime,
    steadyTemps

    # Initialize / reset some variables
    workDone = 0
    sstemp = yss(temperature[Time, 0])
    steadyTemps[Time] = sstemp

    # Allocation of resources and computation of output
    if temperature[Time, 0] <= serviceRate/deadline:
        x[Time] = 1
        for i in np.arange(Time + 1, Time + deadline):
            if not steadyState:
                x[i] = 1
                temperature[i, 0] = math.exp(-(i - currSimTime1) / tau) * temperature[
    currSimTime1, 0] + (
                        1 - math.exp(-(i - currSimTime1) / tau)) * 1
                if temperature[i, 0] >= sstemp:
                    steadyState = True
                    steadyStateTime = i
                    temperature[i, 1] = 3
                    workDone = i - currSimTime1
        temperature[Time+deadline, 0] = math.exp(-(Time + deadline - currSimTime1) /
    tau) * temperature[currSimTime1, 0] + (
                1 - math.exp(-(Time + deadline - currSimTime1) / tau)) * 1
    else:
        x[Time] = 0
        for i in np.arange(Time + 1, Time + deadline):
            if not steadyState:
                x[i] = 0
                temperature[i, 0] = math.exp(-(i - currSimTime1) / tau) * temperature[
    currSimTime1, 0]
```

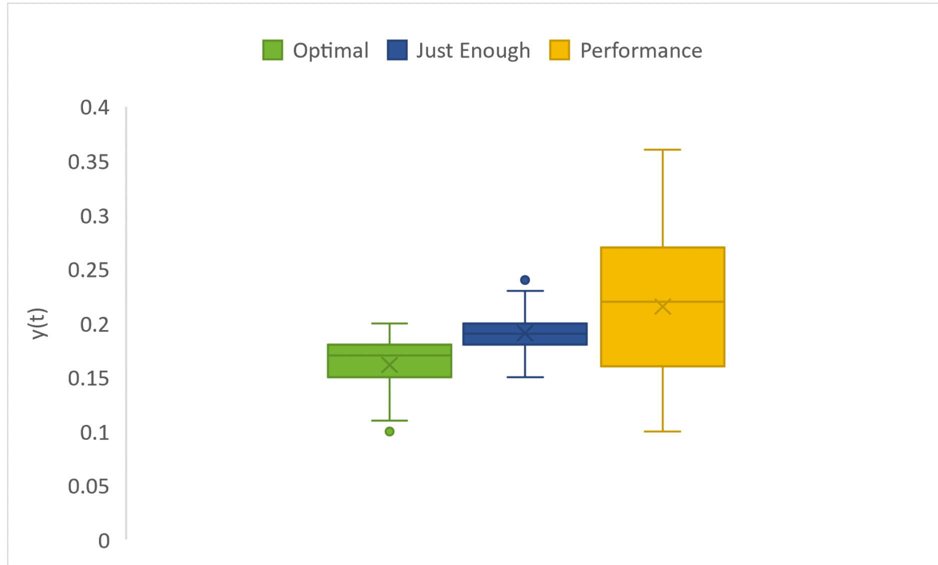
```
63                  if temperature[i, 0] <= sstemp:
64                      steadyState = True
65                      steadyStateTime = i
66                      temperature[i, 1] = 4
67                      workDone = 0
68          temperature[Time + deadline, 0] = math.exp(-(Time + deadline - currSimTime1) /
     tau) * temperature[
69              currSimTime1, 0]
70
71      if steadyState:
72          x[steadyStateTime] = (serviceRate - workDone) / (currSimTime1 + deadline -
    steadyStateTime)
73          for i in np.arange(steadyStateTime + 1, Time + deadline):
74              x[i] = (serviceRate - workDone) / (currSimTime1 + deadline -
    steadyStateTime)
75              if serviceRate - workDone > 0:
76                  temperature[i, 0] = math.exp(-(i - steadyStateTime) / tau) *
    temperature[steadyStateTime, 0] + (
77                            1 - math.exp(-(i - steadyStateTime) / tau)) * ((serviceRate
    - workDone) / (currSimTime1 + deadline - steadyStateTime))
78              else:
79                  temperature[i, 0] = math.exp(-(i - steadyStateTime) / tau) *
    temperature[steadyStateTime, 0]
80          temperature[Time + deadline, 0] = math.exp(-(Time + deadline - steadyStateTime
    ) / tau) * temperature[
81              steadyStateTime, 0] + (1 - math.exp(-(Time + deadline - steadyStateTime) /
     tau)) * ((serviceRate - workDone) / (currSimTime1 + deadline - steadyStateTime))
82
83      # Reset steady-state
84      steadyState = False
85
86      # Trigger endservice event
87      DES.insertEvent(EndService(Time + deadline))
88
89
90  # Class that performs some actions when a job arrives
91  class Arrival(DES.Event):
92      def description(self):
93          return 'arrival'
94
95      def execute(self):
96          global numberOfCustomers, numberOfArrivals, temperature, x, currSimTime2,
    rejected, currSimTime1
97          global steadyState, steadyStateTime, tau, deadline
98
99          # Take this out of loop to not overwrite initial values
100         if self.Time == 0:
101             numberOfCustomers += 1
102             numberOfArrivals += 1
103             currSimTime1 = self.Time
104             startService(self.Time)
105
106         # Reject arriving job if there is already a job in the system
107         if self.Time <= currSimTime2:
108             rejected += 1
109         else:
110             if numberOfCustomers > 0:
111                 rejected += 1
112
113             if numberOfCustomers == 0:
114                 # Update counters
115                 numberOfCustomers += 1
116                 numberOfArrivals += 1
117
118                 # Save information about temperature
119                 if self.Time != 0:
120                     for i in np.arange(currSimTime2, self.Time+1):
121                         temperature[i, 0] = math.exp(-(i - currSimTime2) / tau) *
    temperature[currSimTime2 - 1, 0]
122                     temperature[self.Time, 1] = 1  # 1 = arrival
123                     temperature[self.Time, 2] = numberOfArrivals  # Save number of job
```

```
124
125                 # Save current time
126                 currSimTime1 = self.Time
127
128                 # Trigger start service event
129                 startService(self.Time)
130
131         # Trigger next arrival
132         prevArrival = int(random.expovariate(arrivalRate))
133         DES.insertEvent(Arrival(self.Time + prevArrival))
134
135
136 # Class that performs some actions when a job has finished processing
137 class EndService(DES.Event):
138     def description(self):
139         return 'end service'
140
141     def execute(self):
142         global numberOfCustomers, currSimTime1, currSimTime2, numberOfArrivals, tau,
        deadline
143
144         temperature[self.Time, 1] = 2  # 2 = departure of job
145         temperature[self.Time, 2] = numberOfArrivals  # Save number of job
146
147         numberOfCustomers -= 1
148
149         currSimTime2 = self.Time
150
151         DES.stopSimulation = self.Time > timeToEndSimulation
152
153
154 # Set our parameters
155 arrivalRate = 1 / 50
156 serviceRate = 40
157 deadline = 70  # When we set deadline = serviceRate we should get the performance
        policy
158 tau = 200  #
159
160 # Initializing
161 numberOfArrivals = 0
162 numberOfCustomers = 0
163 initialTemperature = 0
164 currSimTime1 = 0
165 currSimTime2 = 0
166 rejected = 0
167 steadyState = False
168 steadyStateTime = 0
169 timeToEndSimulation = 10000
170 matrix_length = timeToEndSimulation + 1000  # To make sure there is room for all data
171 x = np.zeros((matrix_length,))  # Our resource vector
172 steadyTemps = np.zeros((matrix_length,))  # Our steady-state temperature vector
173
174 # Initialize temperature matrix
175 # Column 1 is temperature; 2 is arrival, steady-state point or departure; 3 is number
        of job
176 temperature = np.zeros((matrix_length, 3))
177 temperature[0, 0] = initialTemperature  # Set initial temperature to 0
178 temperature[0, 1] = 1  # Classify first Arrival
179 temperature[0, 2] = 1  # First job
180
181 # Trigger first arrival at time = 0 and run simulation
182 DES.insertEvent(Arrival(0))
183 DES.runSimulation()
184
185 ############################
186 # Everything below is the computation of values to verify the simulation and/or
        collect results
187 ############################
188
189 # Print assigned resources and temperature matrix
190 print(f'Assigned resources are {x}')
```

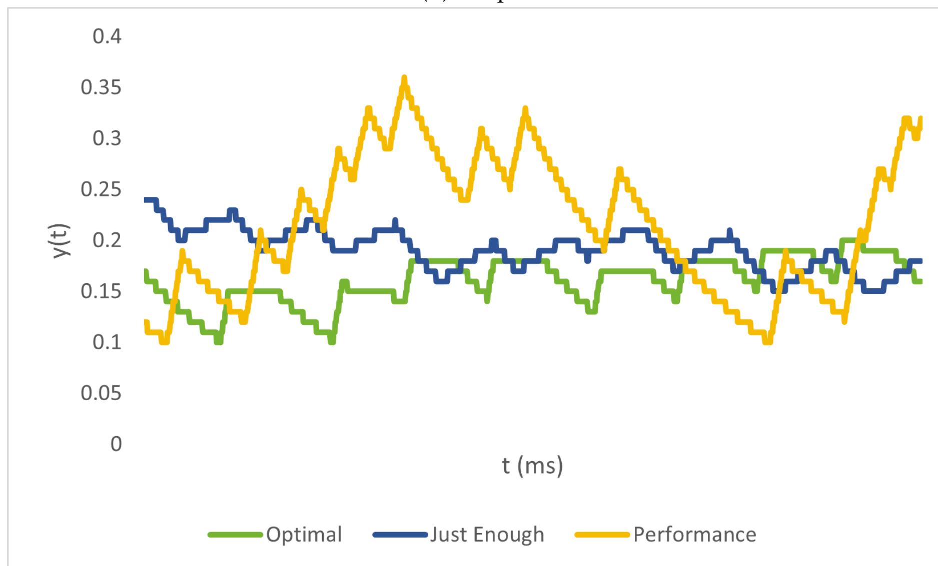
```
191 print(f'Temperature matrix is {temperature}')
192 print(f'Number of rejected jobs is {rejected} out of {numberOfArrivals + rejected}')
193 
194 # Identify indeces of non-arrival temperatures
195 index_arrival = np.where(temperature[0:matrix_length, 1] != 1)
196 # Identify indeces of non-deadline temperatures
197 index_deadline = np.where(temperature[0:matrix_length, 1] != 2)
198 # Identify time-points where we did not save any data
199 index_zero = np.where(temperature[0:matrix_length, 1] == 0)
200 index_zero_ss = np.where(steadyTemps[0:matrix_length] == 0)
201 
202 # Make a matrix of only arrival temperatures
203 arrivalTemperatures = np.delete(temperature, index_arrival, axis=0)
204 arrivalLength = len(arrivalTemperatures)
205 # Make a matrix of only departure temperatures
206 departureTemperatures = np.delete(temperature, index_deadline, axis=0)
207 departureLength = len(departureTemperatures)
208 # Make a matrix of all temperatures
209 allTemperatures = np.delete(temperature, index_zero, axis=0)
210 allLength = len(allTemperatures)
211 steadyTemps2 = np.delete(steadyTemps, index_zero_ss, axis =0)
212 steadyLength = len(steadyTemps2)
213 
214 # Save as csv file
215 temperature_final = np.around(temperature[0:timeToEndSimulation, 0], decimals=2)
216 temperature_final.tofile('temperature_o.csv', sep=',')
217 temperature_dep_final = np.around(departureTemperatures[0:departureLength, 0],
        decimals=2)
218 temperature_dep_final.tofile('temperature_o_dep.csv', sep=',')
219 
220 # Plot input and output over time
221 plt.plot(np.arange(8000, 9000), temperature[8000:9000, 0], 0.2, color='blue', label='y
        (t)')
222 plt.plot(np.arange(8000, 9000), x[8000:9000], 0.2, color='red', label='x(t)')
223 plt.xlabel('t (ms)')
224 plt.ylabel('y(t), x(t)')
225 plt.grid()
226 plt.show()
227 
228 # Plot only output over time
229 plt.plot(np.arange(0, timeToEndSimulation), temperature[0:timeToEndSimulation, 0],
        0.2, color='blue')
230 plt.xlabel('t (s)')
231 plt.ylabel('y(t)')
232 plt.grid()
233 plt.show()
234 
235 # Verify model
236 # Verify queueing simulation by checking idle/busy formula's
237 actualUtilization = sum(x[0:timeToEndSimulation]) / len(x[0:timeToEndSimulation])
238 actualIdleTime = (len(x[0:timeToEndSimulation]) - sum(x[0:timeToEndSimulation])) / len
        (x[0:timeToEndSimulation])
239 
240 expectedUtilization = serviceRate / (deadline + 1 / arrivalRate)
241 expectedIdleTime = (deadline + 1 / arrivalRate - serviceRate) / (deadline + 1 /
        arrivalRate)
242 
243 print(f'System is utilized {actualUtilization} of the time while we expect {
        expectedUtilization}')
244 print(
245     f'absolute error is {100 * abs(expectedUtilization - actualUtilization)}, rel
        error is {100 * abs(expectedUtilization - actualUtilization) / expectedUtilization
        }.')
246 print(f'System is idle {actualIdleTime} of the time while we expect {expectedIdleTime}
        ')
247 print(
248     f'absolute error is {100 * abs(expectedIdleTime - actualIdleTime)}, rel error is
        {100 * abs(expectedIdleTime - actualIdleTime) / expectedIdleTime}')
249 print(f'Actual mean temperature is {np.mean(allTemperatures[0:allLength, 0])}')
250 print(f'Mean arrival temperature is {np.mean(arrivalTemperatures[0:arrivalLength, 0])}
        ')
```

```
251 print(f'Mean departure temperature is {np.mean(departureTemperatures[0:departureLength
    , 0])}')
252 print(f'Mean temp over steady state is {np.mean(temperature[6000:timeToEndSimulation,
    0])}')
253 print(f'Mean departure temp over steady state is {np.mean(departureTemperatures[int
    ((2*departureLength/3)):departureLength, 0])}')
254 print(f'Mean steady state temperature is {np.mean(steadyTemps2[int((2*steadyLength/3))
    :steadyLength])}')
```

11 Appendix D

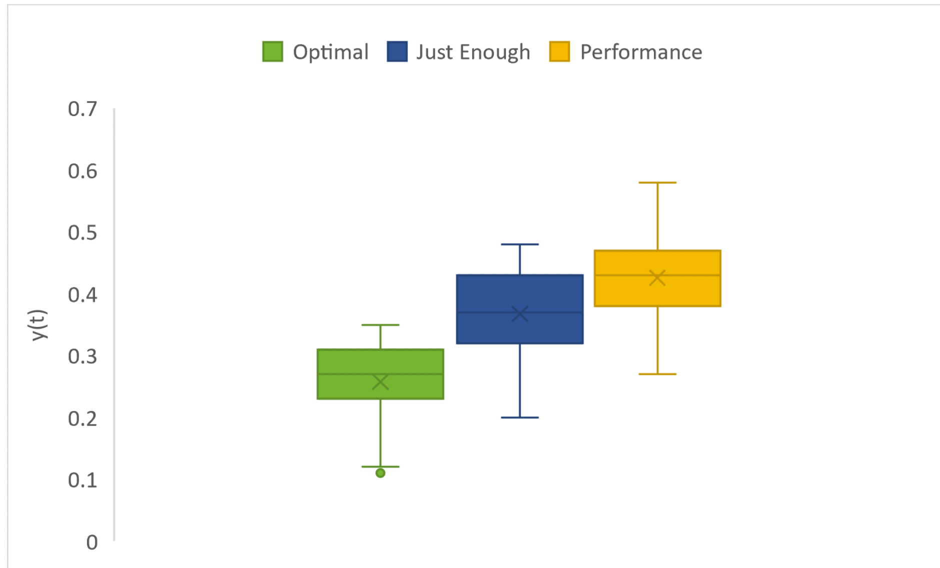


(a) Boxplot

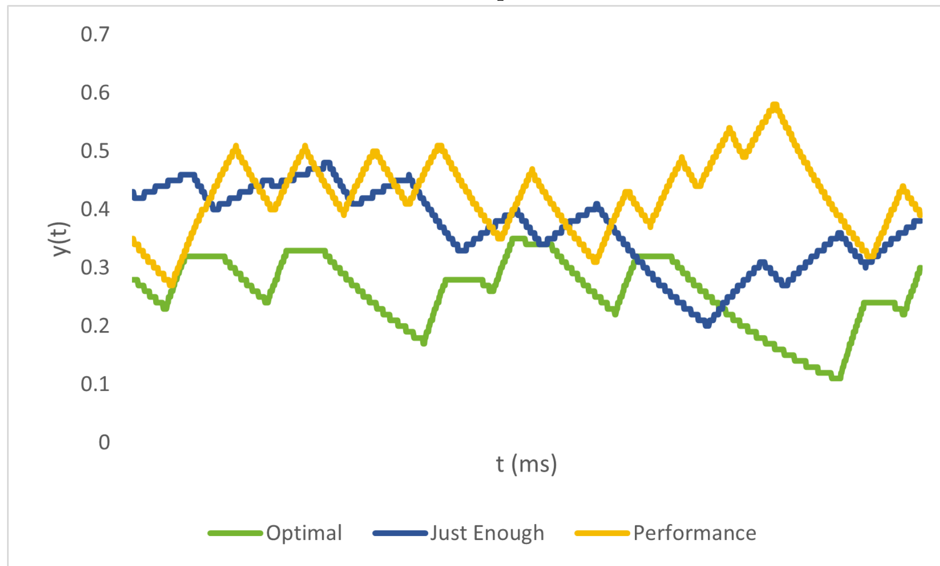


(b) Linegraph

Figure 9: Results of the simulation with low workload and low initial temperature, steady state

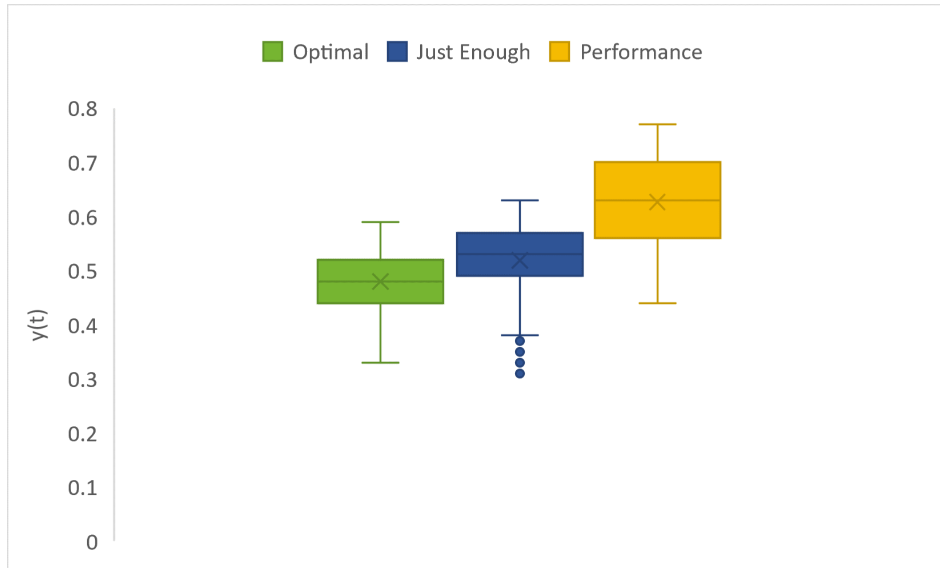


(a) Boxplot

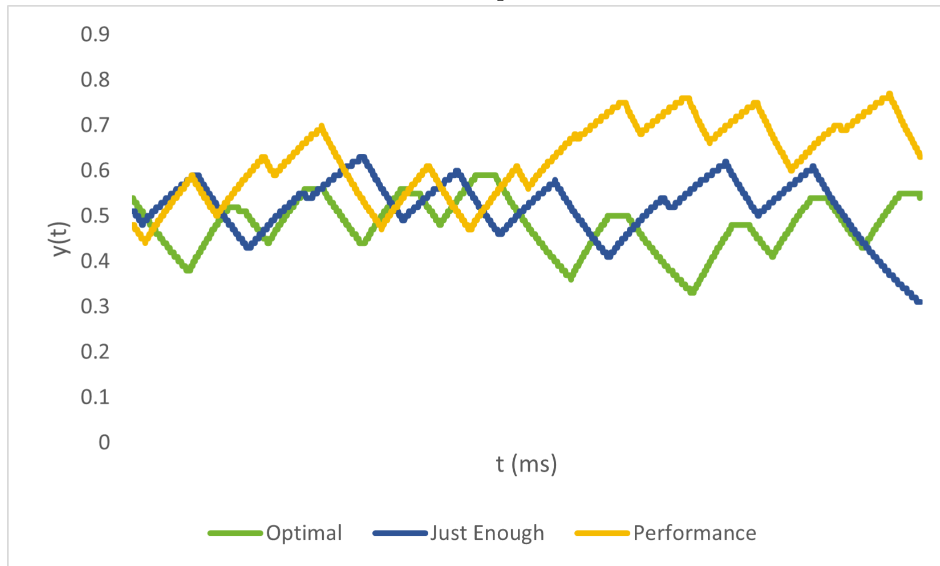


(b) Linegraph

Figure 10: Results of the simulation with medium workload and low initial temperature, steady state

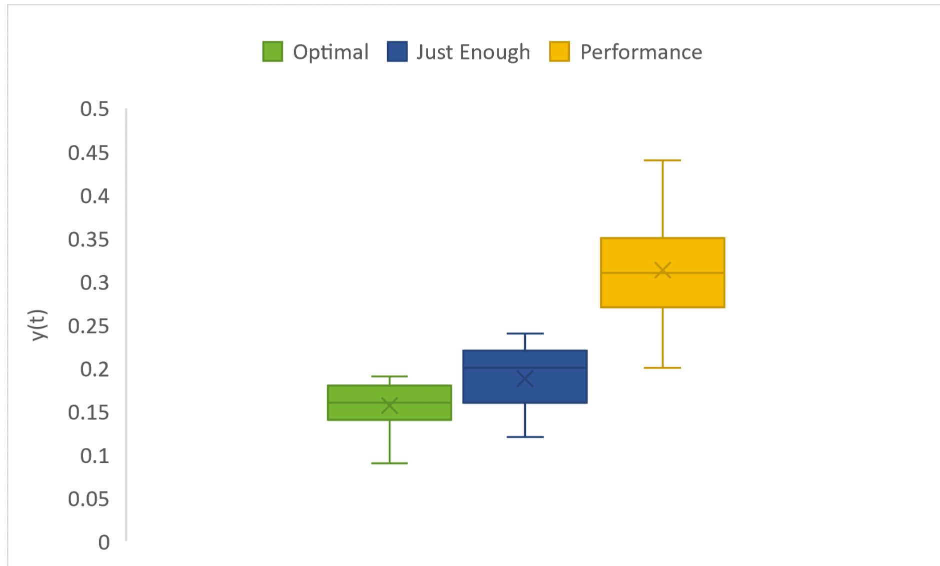


(a) Boxplot

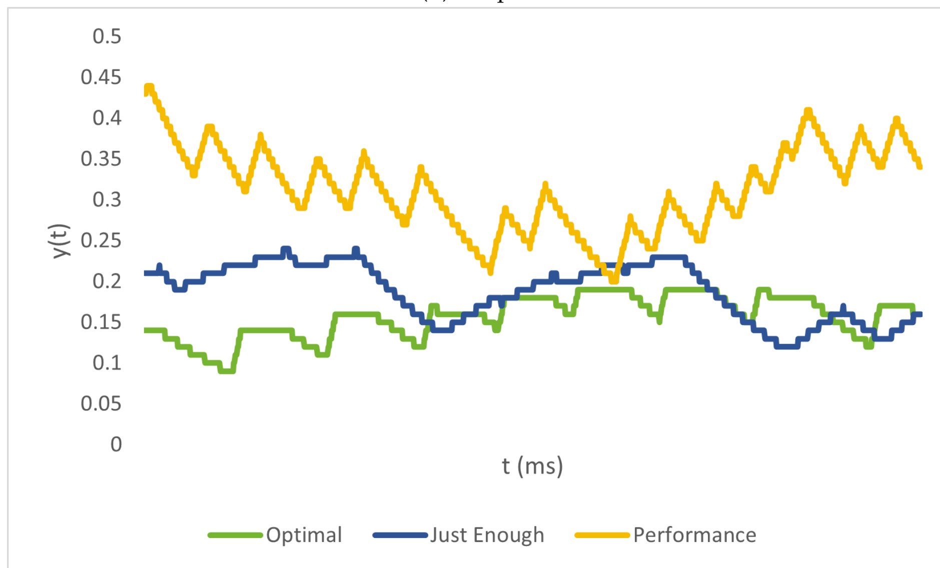


(b) Linegraph

Figure 11: Results of the simulation with high workload and low initial temperature, steady state

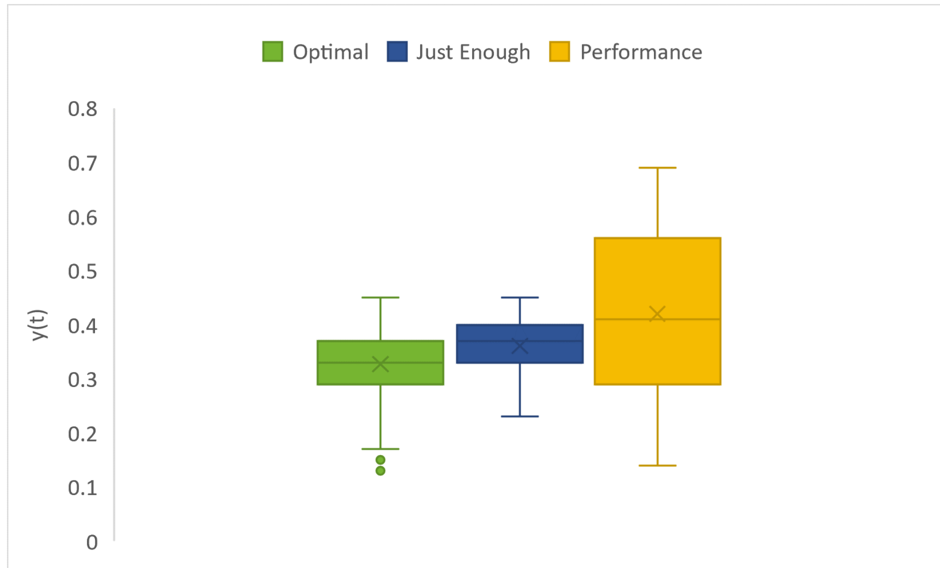


(a) Boxplot

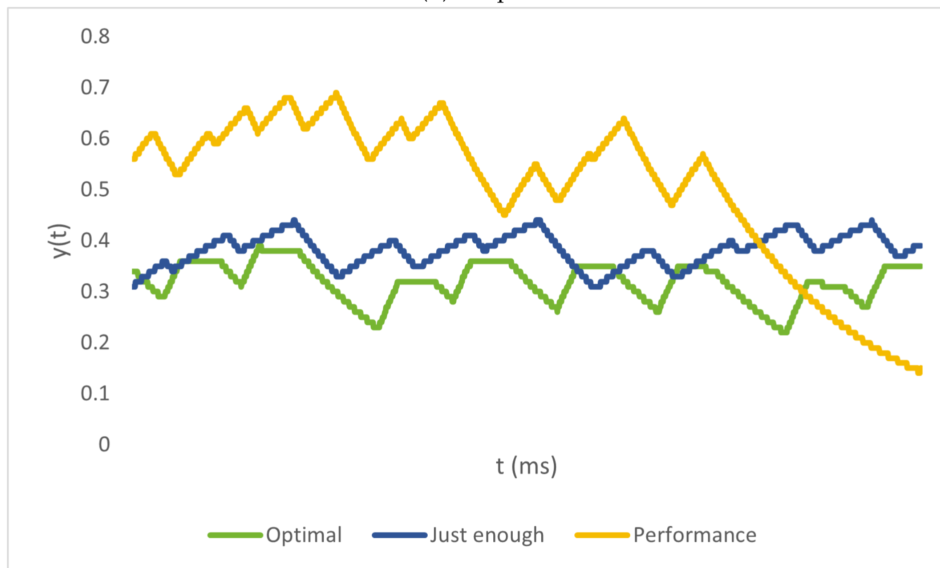


(b) Linegraph

Figure 12: Results of the simulation with low workload and high initial temperature, steady state

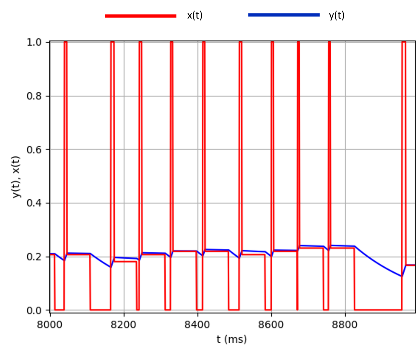


(a) Boxplot

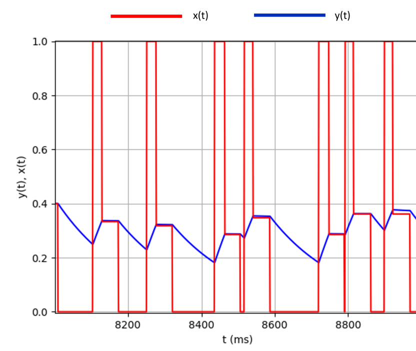


(b) Linegraph

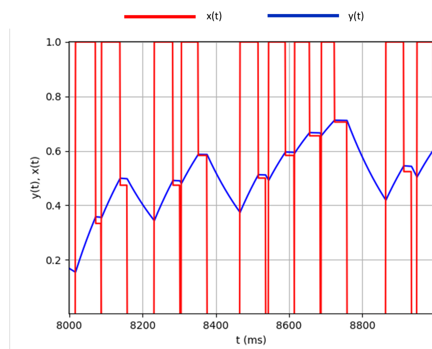
Figure 13: Results of the simulation with medium workload and high initial temperature, steady state



(a) Low workload



(b) Medium workload



(c) High workload

Figure 14: Relation of input $x(t)$ and output $y(t)$ for different workloads