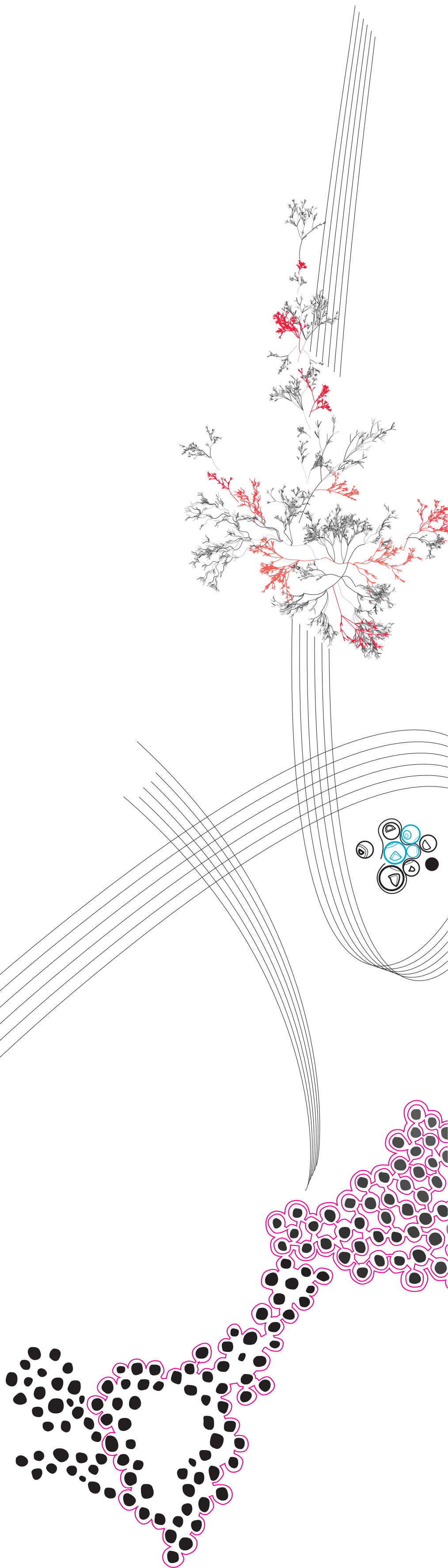

BSc Thesis Applied Mathematics

Energy efficient scheduling in data centers

Margriet Eijken

Supervisor: Ruben Hoeksma and Maria Vlasiou

January, 2023

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

UNIVERSITY OF TWENTE.

Preface

I want to thank every one close to me that supported me during this assignment. Special thanks to my supervisors Ruben Hoeksma and Maria Vlasiou for providing me with guidance and feedback.

Energy efficient scheduling in data centers

Margriet Eijken*

January, 2023

Abstract

This paper considers the problem of energy efficiency in data centers. This subject was chosen because a lot big part of the global energy consumption comes from data centers [23]. It would be beneficial for the data centers to implement algorithms that reduce the energy consumption. This would also be beneficial for every one because less energy would be used which is better for the environment. In this research some of the literature regarding this subject is categorized per server type, arrival assumption and type algorithm. Also, an online deterministic algorithm considering heterogeneous servers is chosen to investigate further. This is done by implementing the algorithm in Python and multiple simulations are run.

Keywords: Scheduling, Energy efficiency, Data center, Speed scaling, Power down, Poisson, Homogeneous, Heterogeneous

1 Introduction

Almost every one nowadays uses the internet. Some use it for work others for recreational purposes. One thing is certain we cannot function without internet. Most of these computational tasks given by people are processed in data centers and these data centers use a lot of energy when doing this.
In 2018 around 1% of global energy consumption was attributed to data centers [23] While there has been a strong increase in demand for data center services, the energy consumption has not grown much [16]. This is partly due to existing energy efficient algorithms. Probably, the use of data-center services will only increase the coming years and the improvement of these energy efficient algorithms will help to keep this energy consumption from increasing to absurd amounts.
Also, for data centers it would be beneficial to make use of more energy efficient algorithms simply because electricity is the relatively highest expense of the data center [2][1]. So the less energy they use the more profit they can make. The prices of electricity have also been rising significantly from mid-2020 until now and this trend could continue [3]. This is even more of an incentive for the data centers to use or make these energy efficient algorithms.
The last reason why my research is of importance to everyone is that it will be better for the environment if data centers use less energy. Because data centers are such large energy consumers, decreasing their energy consumption would also significantly help decrease the global energy consumption. Which would overall be better for the environment and thus be in everyone's interest.

*Email: m.eijken@student.utwente.nl

1.1 My contribution

The first part of my research is a literature review of papers that I believe can be useful in the research of energy efficient scheduling in data centers. These papers are categorized in section 2 by their server type and because each model considers some kind of server(s) all papers will be discussed here. The papers are also categorized in section 3 per their arrival assumption if they have one and in section 4 the papers are categorized per type algorithm. Then in section 5 the second part of my research will be explained, this is the practical part. For my experiment I the algorithm from [7] to look into further and I did this by implementing it in Python and running multiple simulations with different arrival patterns. Finally, the paper is concluded in section 6 where my conclusions are summarized.

2 Server type

In this section all papers will be mentioned that I looked into because each of them considers either homogeneous, heterogeneous servers or a single server. The difference between homogeneous and heterogeneous servers is that homogeneous servers are all considered equal, which means they have the same characteristics regarding job processing and heterogeneous servers are considered different per server or server type. From this categorization I noticed that most of the literature I found was considering the single server case. This is quite logical because it all started with one device, one server, that could process assignments, so the single server case has been around longer than the multiple server case. For the multiple server case the amount of papers considering the homogeneous and heterogeneous servers is fairly divided.

2.1 Homogeneous

In this section all papers are discussed that consider homogeneous servers or something that can be resembled as homogeneous servers, like homogeneous nodes.

Dynamic power allocation in server farms a real time optimization approach
This paper first formally defined the average power minimization problem as an optimization problem. Then, because this problem is hardly tractable a reformulation is given as a tractable optimal control problem that can be solved using the two-stage real-time optimization approach. For this optimal control problem a sub-optimal dynamic power allocation policy is proposed.
The server farm that is considered consists of M homogeneous servers all sharing workload and power supply. Incoming jobs are queued at an available server according to the task assignment policy. This task assignment policy is that an incoming job will go to the least busy server. This is allowed because of the assumption that the jobs are first split into equally sized task units and therefore all jobs have the same size. Simulations are done to compare the proposed algorithm with optimal static power allocation. The optimal static is better as workload rate changes become faster and more abrupt. However, under a condition called slowly varying job arrival rate the proposed algorithm performs better than the optimal static solution. This slowly varying arrival rate is typically satisfied in server farms. [5] Why the choice was made for homogeneous servers is not completely clear to me but I think it is because of simplicity.

Dynamic Right-Sizing for Power-Proportional Data Centers
This paper investigates a new online power down algorithm based on the optimal offline

solution, which exhibits a "lazy" structure when viewed in reverse time. The new algorithm called Lazy Capacity Provisioning(LCP) has similarly to the optimal solution a "lazy" structure, this means that it stays within upper and lower bounds. However, LCP does this moving forward in time instead of backwards like the optimal solution does. This new algorithm is proven to be 3-competitive under arbitrary workloads, general delay cost and general energy cost models provided that they result in a convex operating cost. However, in practice the algorithm seems to be nearly optimal which is also backed up by the experiments that were performed with two real load traces. [20] The model that is discussed is also very general and very similar to the one discussed in [7] except that here the assumption of homogeneous servers is made. This new algorithm is based on Receding Horizon Control which is a 'well-known' online policy and this could be beneficial because there is more research done on this policy. Why the choice is made for homogeneous servers is not clear.

Min$_c$ Heterogeneous concentration policy for energy aware scheduling of jobs with resource contention
In this paper it is assumed that there are j different type of jobs and m homogeneous servers are available. Now the problem that is addressed is to which server each incoming job has to go to. So, the problem is job allocation. Min$_c$ is an utilization aware algorithm which allocates job j to the machine with minimum concentration of jobs of the same type at time r_j, where r_j is the release time of job j. According to the simulation results in this paper this is the best algorithm to minimize the power consumption. Utilization aware means that there is some CPU utilization information available. Also, two different types of algorithms are investigated in this paper namely, energy aware and knowledge free, which works with power consumption information and no information about applications and resources respectively. [8] It is interesting that a utilization aware algorithm is optimal for energy minimization as you would probably think that it would be an energy aware algorithm. Also, many different algorithms are investigated here which is not something most papers do. Why the choice is made for homogeneous servers is not clear.

Minimizing energy on homogeneous processors with shared memory
In this paper the energy efficient task scheduling problem is considered with multiple homogeneous cores and the main memory power shared. It is assumed that the number of tasks is larger than the number of cores and each task has to be assigned to a single core. Because this problem is NP-hard, an approximation algorithm is designed in this paper to obtain a near-optimal performance. Before the approximation algorithm is given, an optimal polynomial time algorithm is presented where the assignment of tasks to cores is given. It is also assumed that each core has an individual voltage supply and the speed changes in a continuous fashion. The choice of speed is part of the algorithm and is determined for a block of tasks. All tasks are available from the beginning thus the algorithm is offline and a schedule is considered feasible if all tasks are finished before their deadlines. In addition, the optimal algorithm is given for the single core case to base the approximation algorithm on. [10] Why the choice is made for homogeneous servers is not clear.

Optimality analysis of energy-performance trade-off for server farm management
The objective of the paper is to analytically address optimizing the Energy Response time Product (ERP) metric in server farms. The ERP is given by $ERP^\pi = \mathbf{E}[P^\pi] \cdot \mathbf{E}[T^\pi]$ where $\mathbf{E}[P^\pi]$ is the long-run average power consumed under a control policy π and $\mathbf{E}[T^\pi]$ is the mean customer response time under a control policy π. This metric implies that a reduction in mean response time from 2 s to 1 s is a lot better than a reduction from 1001 s to 1000

s, which is not something that another popular metric called ERWS can do even though it is more realistic. The results in this paper are some optimal power down policies for a single server with Poisson arrivals and some near optimal power down policies with Poisson arrivals with unknown arrival rate. Also, for the multiple homogeneous server case with known Poisson arrivals some near optimal power down policies are given and finally this was also done for the case considering a time-varying arrival pattern. In addition, some guidelines are given such that an efficient server farm management policy can be chosen more easily. [13] These guidelines could be very convenient in practice. The policies that are researched here are not too complicated but effective, backed up by analysis and experiments. The single server case is here used as a set up to the multi server case that is why I put this paper under the homogeneous server category and not the single server category. Why they chose here for homogeneous servers is not completely clear to me but I think it is because of simplicity.

Structural properties and exact analysis of energy-aware multiserver queueing systems with setup times
The first thing that is done in the paper is several key structural properties are derived belonging to optimal policies under linear cost functions. Then two specific policies having these properties were analysed. These policies are bulk setup and staggered threshold. More detail about these policies will be given in section 4.1. An exact analysis was performed for these policies using the Recursive Renewal Reward technique. Also some experiments were done with the two different policies to find out the best number of servers to always keep on. [22] This paper gives a good insight in the staggered threshold and bulk setup policies, with both analysis and experiments to back up results. Why the choice is made for homogeneous servers is not clear.

2.2 Heterogeneous

In this section all papers considering heterogeneous server will be discussed. What is interesting to see here is that if a cloud environment is considered it will be highly likely that heterogeneous servers are assumed because that is that structure of a cloud environment. So, it is almost certain that when a cloud environment is discussed, heterogeneous servers are assumed.

A green policy to schedule tasks in a distributed cloud
In this paper a new green policy is given for scheduling tasks in a volunteer cloud environment. To solve this optimization problem one algorithm that is considered is called the Alternating Direction Method of Multipliers. A distributed method of scheduling of tasks is almost necessary because of the complexity of the volunteer cloud. It is large scale, servers can leave and join the volunteer cloud at any time without warning and each device has different properties which determines what tasks it can and cannot process. The green policy is tested against a task greedy policy and from the numerical results can be concluded that the green policy does save energy without having significant losses in other performance parameters which are not related to energy consumption. [27] Cloud computing could be really useful as it uses devices that already exist and have some memory left for other services. So no need for new servers. The choice for heterogeneous servers was made because they wanted to research a volunteer cloud system which consists of different devices which can be represented as heterogeneous 'servers'.

A Dynamic and Energy Efficient Greedy Scheduling Algorithm for Cloud Data Centers
In this paper an algorithm is proposed to schedule tasks dynamically with higher energy efficiency. This algorithm is based on an integer programming optimization problem. [26] It is not really possible to conclude anything from this paper because it is so poorly written. This also makes me hesitant about trusting that they did the simulations correctly and thus whether to rely on these results. Why they make use of heterogeneous servers is not clear from the paper but it is probably because it is an characteristic of a cloud data center.

A Lightweight Optimal Scheduling Algorithm for Energy-Efficient and Real-Time Cloud Services
In this paper the algorithm that is presented is an expansion of an algorithm that was created in their previous work based on multiprocessors. The new algorithm presented here has an improved time complexity from $\Omega(N^3 logN)$ to $O(N^2)$ in comparison to their old algorithm. This improvement could be made by providing a different method for constructing a new structure of the flow network and designing a solver. From the experimental results can be seen that this new algorithm performs better regarding time complexity and energy efficiency, compared to their old algorithm. [28] This flow network representation is different from most papers and could be interesting to look into more. I am actually not completely sure they assume heterogeneous servers but I think they do because in their related work they talk about a similar model with heterogeneous servers and a cloud environment typically deals with heterogeneous servers. However, some simulations were done to find out the number of active servers that are necessary to process all tasks in time, which is typically something for homogeneous servers.

Algorithms for energy conservation in heterogeneous data centers
In this paper, it is studied how a data center with heterogeneous servers can dynamically be right-sized to minimize the energy consumption. A deterministic online algorithm is presented that has a competitive ratio of $2d$ and a randomized version of the algorithm is presented which has a competitive ratio of $1.58d$, where d denotes the number of server types. This paper discusses the following problem, a data center is considered with d different server types and there are m_j servers of each type j. Each server type has its own switching cost and operating denoted by β_j and l_j respectively. These costs are ordered like $l_1 > \ldots > l_d$ and $\beta_1 < \ldots < \beta_d$ which means each server type has unique operating and switching costs. A finite time horizon is considered with time slots $t \in \{1, \ldots, T\}$ and during each time slot a job volume of $\lambda_t \in \mathbf{N}_0$ arrives and has to be processed completely. Each server can process one job per time slot independent of server type. [7] The exact details of this algorithm are discussed in section 5 as it is the subject of my experiment. It is a very clear paper which explains the algorithms very elaborate. They have found the optimal online deterministic algorithm for their assumptions so this could be a great basis to compare the same algorithm with slightly different assumptions. In this paper the choice for heterogeneous servers was very conscious. They chose for heterogeneous servers because in reality most data centers consist of heterogeneous servers. Data centers can have old servers, new servers and different types of servers that all have different operating and switching costs. Another reason that is given for researching heterogeneous servers is that there has not been a lot of research regarding this case.

An approach to reduce energy consumption and performance losses on heterogeneous servers using power capping
In this paper a heterogeneous microserver appliance is considered which is not exactly a data center but the methods used in the paper could be scaled to be applied to a data center.

Power capping is not explicitly a method of saving energy because it means that a fixed total amount of power is used and this is distributed over the available servers. However, if it is combined with some energy saving algorithm the same amount of power is used while more jobs can be processed so it is then beneficial to use such an algorithm. In this case the basic power management algorithm consists of a power capping procedure and an energy saving procedure, these run simultaneously. Then two power distribution algorithms are proposed to improve the energy efficiency of the basic power management algorithm. One greedy algorithm and an exact optimization algorithm using a Mixed Integer Linear Programming (MILP) solver are proposed. However, the MILP solver is not suitable for the power capping procedure as this method has to act as fast as possible when the power budget is exceeded and this MILP solver is not fast enough for this. So, the exact optimization algorithm with the MILP solver will only be applied to the energy saving procedure of the algorithm and the greedy algorithm will be applied to the power capping procedure. To evaluate these algorithms also two simple power capping algorithms are defined called random and simple. Several simulations are done and from the results can be concluded that the proposed algorithms improve energy efficiency compared to no power capping and the simple and random power capping algorithms. Also the performance losses were decreased significantly in comparison to the random and simple power capping. [11] The choice of heterogeneous nodes was made because of the specific system that was chosen to investigate .

Energy efficient scheduling via partial shutdown
In this paper a collection of new problems regarding energy saving in data centers is defined and are called machine activation problems. It is assumed that there is a collection of m machines with each having activation cost a_i for machine i and there is a collection of n jobs that have to be processed. The processing time of job j on machine i takes $p_{i,j}$ time. Different from the standard scheduling models is that here it is assumed that there is an activation cost budget A and we would like to select a subset S of the machines to activate such that $a(S) \leq A$ and find a schedule minimizing the makespan, which is the total time required to execute the jobs via the schedule. Also, the setting with uniformly related machines is considered which means that the processing time of job j depends on the machine i and is $p_{i,j} = p_j/s_i$. The results that are given are several approximation algorithms with different competitive ratios for both the related and unrelated machines case. Also a version of the problem is considered where not all jobs have to be processed but each job j that is processed has benefit π_j and at least total benefit Π has to be obtained. [17] The choice for heterogeneous servers was made because the problem that is discussed here is based upon more studied basic problems like the uniform machine scheduling problem in which they assume heterogeneous servers.

Energy efficient scheduling in data centers
The goal of the paper is to reduce capital as well as operational costs of a tiered data center by implementing energy efficient scheduling algorithms. A tiered data center will typically consist of a load dispatcher which will direct an incoming job to the appropriate tier and on each tier there will be servers that function similarly. In this paper several strategies are given to tackle the dynamic server provisioning problem for deferrable jobs, these are jobs that do not have to be processed by one specific server. Two optimization problems are defined namely the joint minimization of the capital, operational and switching costs and the minimization of only the operational and switching cost, the respective algorithms are called CapexOpexMin and OpexMin. These problems are solved with a technique called Model Predictive Control also know as Receding Horizon Control, this method is also used to create an algorithm in [20]. A discrete time model is considered. The capital costs are

mainly influenced by the peak server provisioning cost, so reducing these costs will directly decrease the capital costs. Several simulations were done with these algorithms and they were compared to each other and their optimal offline adversary. The algorithms were evaluated based on Cumulative Electricity Cost (CEC), Cumulative Renewable Consumption (CRC) and Cumulative Number of Switchings(CNS). The results summarized are that the CapexOpexMin algorithm gives a reasonable balance between the three parameters mentioned before. [24] This paper's main focus seemed to be on a realistic scenario so that explains that the choice was made for a tiered data center which has heterogeneous servers.

Energy-aware dispatching in parallel queues with on-off energy consumption
A queue assignment problem is considered in this paper where each queue can resemble a different server type. The servers can also be turned off to save energy. The results that are given in this paper is a dispatching policy based on the first policy iteration principle. The first policy iteration principle is that a first or basic policy is chosen at the start and this policy is improved at every iteration and it will converge to the optimal policy. For this problem the basic policy that was chosen allocates the jobs to the servers such that the arrivals in each queue are independent Poisson processes and each queue is an independent M/G/1 queue. There are two two-queue examples given to demonstrate how the policy works and that their policy has improved the results in comparison with a basic policy and a myopic policy, a policy is myopic if it does not consider the effects of future jobs. This policy can work not only for Poisson arrivals but also for other arrival processes. [25]I think that it would be useful if some more experiments were done to see how much better this policy performs over another policy. In this paper the choice for heterogeneous servers was made because the energy aware dispatching problem especially arises with these kind of servers. That is because you need a reason to choose a certain server for a job.

2.3 Single server

In this section all papers regarding single servers will be discussed. What was most interesting I observed in this category is that most single server models use a speed scaling algorithm as a solution for energy minimization. This is not unexpected because if you have a single server then turning it on and off, like in a power down algorithm, might not be as beneficial as changing the speed of the server. Sometimes however the server has not the option to run with different speeds so then a power down algorithm is the only option to reduce energy consumption.

A scheduling model for reduced CPU energy
In this paper a simple model is considered namely a single variable speed processor and each incoming job has to be processed between its arrival time and deadline. An offline algorithm is given that computes the optimal schedule for any set of jobs and the only restriction for the power consumption function is that it has to be convex. Then the online case is considered and the average rate heuristic is designed for this. How this heuristic works is discussed in section 4.2. Some more analysis is done for the average rate heuristic which includes but is not limited to an analysis of the competitive ratio of the algorithm for different power consumption functions. [32] It is also mentioned that some simulations have been done but there are no details about these simulations other than some conclusions that could be drawn from them. I am not sure if this is flaw of the paper or if the details are not mentioned because they are not useful for the reader. I belief that there could be several reasons the details of the simulations are not mentioned namely, it is a conference paper

and such details are then not too interesting. This paper is one of the older ones regarding this subject, it is from 1995 and maybe at that time running this kind of simulation could only be done on a certain kind of device so why mention this if it is set for every one or they simply did not think that it was interesting to mention the simulation details. Another possibility for not mentioning these details is that they do not know what they are, as said in the acknowledgements someone else performed the simulations for them. However, despite not knowing the details of the simulation I do not think that the quality of the paper is harmed by this. In this paper they chose for a single server model because the paper does not consider data centers, it is about a single device like a computer that has to process jobs.

Algorithms for power savings
In this paper two algorithms are designed one dynamic speed scaling algorithm without a sleep state(DSS-NS) and one dynamic speed scaling algorithm with a sleep state(DSS-S). When the system is in this sleep state it will consume less power but a fixed amount of energy is required to transition back to the on state. The power is here a convex function of the speed denoted by P(s), such that P(s)/s is also a convex function. Both an off- and online algorithm are given for the DSS-S problem where the offline algorithm has a competitive ratio of 3. The online algorithm for the DSS-S problem makes use of an online algorithm for DSS-NS. The assumption is made that the online algorithm for DSS-NS is monotonic, which means that it only increases the speed at the arrival of a job. The AVR algorithm discussed in [32] is actually the only competitive algorithm for DSS-NS right now and it is also a monotonic algorithm. Both algorithms are analysed and the results are some upper bounds for the competitive ratios. [15] In this paper servers or processors are not really discussed but they discuss a system and it is left in the middle whether this system consists of one or multiple servers. Because one system is discussed in this paper it fits best in the category of a single server because they consider the one or multiple servers as a single system.

Energy efficient online deadline scheduling
In this paper the classic online deadline scheduling problem is considered but with a slight adaptation that the speed of the server is limited. The speed of the server can be adjusted to any value in $[0,T]$ with T being the fixed maximum speed capacity of the server. It is assumed that preemption is allowed. But with the assumption of a limited server speed it is possible that the system becomes overloaded, which means that it might not finish all jobs before their deadline. To solve this problem a job selection algorithm called FSA (Full Speed Admission) is proposed and is combined with the speed scaling algorithm called OAT (Optimal Available, at most T). These algorithms will be further discussed in section 4.2. Furthermore, some analysis is done on the FSA algorithm and the FSA(OAT) algorithm and some examples are given how the FSA and FSA(OAT) algorithm can be applied to other cases and what the competitive ratios would be for those cases. [19] FSA(OAT) is another speed scaling algorithm but it takes into account a more realistic assumption of the server having a maximum speed. The competitive ratios are not too great but the case that is considered is closer to reality in my opinion. We should also keep in mind that a competitive ratio is a worst case scenario but there are no numerical results regarding this algorithm so no conclusion can be drawn about the average performance. In this paper the choice to consider a single server was made because their algorithm is based on a previous research where a single server is considered with varying unbounded speed.

Energy efficient algorithms
This paper is kind of a summary of many different energy efficient algorithms. Most of the

algorithms work for single server systems and some of the methods that are used are power down and speed scaling. First two-state systems are considered, these are systems that are either in an active state where they can process jobs or in a sleep state where no jobs can be processed and no energy is consumed. The objective of the considered algorithms is to minimize energy consumption so the time delay that arises when going from a lower power state to a higher power state is not taken into account. Apart from the power down and speed scaling algorithms, that will be discussed in 4.1 and 4.2 respectively, multiple algorithms are mentioned regarding the balancing of minimum response time and energy saving. One algorithm is a solution to an offline power allocation problem and it is explained that the online problem is a lot more complicated. Also a simple online strategy is mentioned where jobs are processed in batches, it is explained that the computationally expensive scheduling decisions only have to be made every now and then, which is better. Then the Shortest Remaining Processing Time job allocation policy is discussed and that it is used in combination with a speed scaling algorithm based on job count. There is also a small section dedicated to papers that discuss algorithms that consider multiple server systems. Furthermore, there is touched upon the subject of wireless networks where the network topologies are considered as well as the subject of data aggregation. For these subjects some more algorithms are mentioned including the minimum spanning tree. [6] This paper is very clear, elaborate and discusses many different algorithms and their complexities. This paper could be a great starting point when researching energy efficient scheduling algorithms. In this paper they chose mainly to investigate algorithms which consider single servers, this is just because most available literature is still about the single server problem.

On optimal policies for energy-aware servers
This paper provides an elaborate analysis of a single server system where the server's energy state can be dynamically changed. This analysis is done with the use of queueing theoretic tools and results. The system that is considered has four energy states namely, off, setup, busy and idle. The system will only move from off to setup if a number of k jobs have accumulated in the queue, where k can be chosen by the operator. The system will move from idle to off after spending spending a certain amount of time being idle, this amount of time is exponentially distributed with rate α, where α can be chosen by the operator. The server spends an exponentially distributed time γ in setup before it can begin processing jobs in the busy state, where the job processing times are exponentially distributed with rate μ. This system is modelled as a continuous time Markov chain. [21] I think it is a very elaborate analysis which could be helpful in finding an optimal policy. In this paper the choice was made for the single server assumption because they want a broader analysis including more metrics of the single server system and more cost functions examined. This is also a solid basis for creating an optimal policy for a multiple server system which they talk about briefly.

On the Gittins index for multistage jobs
This paper gives a more direct method on how to compute the Gittins index for sequential multistage jobs. However, the paper itself also already says that usually the Gittins indices are hard to compute so this is probably not the most useful paper. Also, in this paper the objective is to minimize the mean delay so it has not really anything to do with energy efficiency. However, this paper could maybe function as a basis for finding an energy efficient policy as many papers consider the M/G/1 queue when finding energy efficient policies. [4] They talk about a single server queueing system in this paper because they research the Gittins index policy which is known to be optimal for the M/G/1 queue minimizing the mean delay.

Online speed scaling based on active job count to minimize flow plus energy
This model considers online scheduling algorithms that have the objective to minimize the flow time as well as the energy consumption. Flow time is the time it takes to process a job, so the time from arrival until it being finished. First the simple speed scaling algorithm called AJC is analysed when coupled with an arbitrary sleep management algorithm, for the case of multiple sleep states. The AJC algorithm works based on the Active Job Count and changes speed discretely. Then the sleep management algorithm IdleLonger is introduced and it is analysed when combined with an arbitrary speed scaling algorithm for both the single and multiple sleep state case. Lastly, AJC and IdleLonger are combined to tackle the problem of a server with a maximum speed and multiple sleep states. It is shown this combined algorithm is $O(1)$ competitive for flow plus energy. [18] This paper is very elaborate and clear in my opinion. Why the choice for the single server assumption was made is not clear. They do not discuss data centers so the single server assumption is not unjustified.

Optimal power allocation in server farms
Power allocation is similar to speed scaling because the more power is given to a server the higher the speed of that server will be. In this paper the power to frequency relationship in a single server is thoroughly researched. They found that this relationship is either linear or cubic. These power to frequency relationships can also be useful for other speed scaling algorithms. In this paper the focus lays mainly on the power-to-frequency relationship, with this relationship and other important factors influencing the effect of power allocation, a queuing theoretic model is made. The model predicts the mean response time considering all these different factors and allows us to determine the optimal power allocation policy. Also three different power allocation schemes are tested namely, PowMax, PowMed and PowMin which means running the servers at their highest, intermediate or lowest power level respectively. The conclusion is that for different scenarios a different power level is optimal. [14]

Polynomial-time algorithms for minimum energy scheduling
The problem that is considered in this paper is an offline deadline scheduling problem with a single server and preemptions are allowed. They investigate whether there is a solution to this problem that can be solved in polynomial time. It is found that this is indeed the case and they show step by step how this polynomial time algorithm is created. The basis of their algorithm is minimum gap scheduling which means that the number of idle periods is minimized, this is part of a power down algorithm. This minimum gap scheduling is done with dynamic programming but with a special inversion method. [9] It is a very elaborate paper but also very complicated to me, but I did not come across a lot of papers that consider gap scheduling so it interesting to read about this different approach. They chose for the single server assumption because that is what was done in the open problem they solved.

Power-aware Speed Scaling in Processor Sharing Systems
This paper extends the stochastic analysis of dynamic speed scaling. The focus lays on the M/GI/1 queue under Processor Sharing (PS) scheduling. PS scheduling means that all jobs in the system are processed simultaneously and all jobs receive the same fraction of the capacity available, so all jobs are processed at the same rate. The reason why the focus lays on PS is because it is a tractable model of current scheduling policies in CPUs, web servers, routers, etc. according to the paper. Furthermore, there are three different speed scaling algorithms researched in this paper each one being a little more dynamic than the last one. They try to find out how much the energy efficiency improves with each dynamic

improvement. This can help with finding the best dynamic speed scaling algorithm. The performance metric considered in this paper is $\mathbb{E}[T] + \mathbb{E}[E]/\beta'$, where $\mathbb{E}[T]$ is the expected response time of a job, $\mathbb{E}[E]$ is the expected energy spent on that job and β' represents the relative cost of delay. The results that are given are bounds on the performance of dynamic speed scaling and for the speeds used by the optimal dynamic speed scaling scheme, for the last case also asymptotics are given. From these results can be concluded that a simple power down(gated static provisioning) scheme is almost as good as the optimal dynamic speed scaling scheme. This is also backed up by numerical experiments, but from these experiments it can be seen that an optimal dynamic speed scaling scheme has the benefit of improved robustness to mis-estimation of workload parameters and bursty traffic. Also, a connection was made between the optimal stochastic policy and results form the worst-case community. [31] They use the single server assumption because they consider the M/GI/1 PS queue and this is a simple model. This paper is based on research they have done previously and for that research they used the same model.

Scheduling for reduced CPU energy
In this paper different speed scaling algorithms are tested by running simulations. The speed scaling algorithms are further discussed in section 4.2. The algorithms that are simulated are not too interesting and their conclusion is also very general that speed scaling is beneficial and that energy efficiency would greatly increase if there are better workload prediction models available. [29] This paper really focuses on the CPU and its properties and thus this paper might not be too interesting if data centers are considered. The choice for a single server was made because they wanted to investigate the CPU.

Stochastic analysis of power-aware scheduling [30]
This paper is a successor of the paper [31]. It discusses almost completely the same things including the same results except the focus is more on the gated static provisioning scheme and the dynamic speed scaling. Also, some proofs are omitted. So, for further details about the paper see the paragraph regarding "Power-aware speed scaling in processor sharing systems".

3 Assumption arrival process

Most papers have the assumption of a Poisson process for the arrivals. Another possibility is assuming a Markov modulated Poisson process or some other distribution for the arrival process. And the easiest option is that just some random number of jobs arrive. What I observed from this categorization is that all the papers that I came across and have an arrival distribution are papers that do not consider heterogeneous servers. Which I found surprising. I think a reason behind this is that most papers considering heterogeneous servers try to model a situation as close to reality as possible. And most of them use then data from the real world for simulations and therefore do not need an arrival assumption.

3.1 Poisson

In this section all papers assuming a Poisson arrival process for the jobs are mentioned.

Energy-aware dispatching in parallel queues with on-off energy consumption
The results that are given in this paper is a dispatching policy based on the first policy iteration principle. For this problem the basic policy that was chosen allocates the jobs to the servers such that the arrivals in each queue are independent Poisson processes. This is

where the assumption of Poisson arrivals comes from. This policy can work not only for Poisson arrivals but also for other arrival processes. [25]

On optimal policies for energy-aware servers [21]
This paper chose for Poisson arrivals because it is also very often used in other papers regarding this subject.

On the Gittins index for multistage jobs
It is quite logical this paper chose for Poisson arrivals because they make use of the following fact. The Gittins index policy is optimal in a tandem queue with Poisson arrivals and fullly flexible and collaborative servers minimizing(among all nonanticipating control policies) (i) the mean total delay and the mean total number of jobs in any case; (ii) the mean holding costs for the special case; when the holding costs remain the same in each station and there are no switching costs nor any switching delays and if the service times in all stations belong to NBUE. Let NBUE refer to the family of service time distributions with the New-Better-than-Used-in-Expectation property, i.e., M(a) M(0) for all a 0. [4]

Optimal power allocation in server farms
The assumption of Poisson arrivals is only used for for the open loop configuration which means that jobs arrive from outside system and leave after its finished. The closed loop configuration is also considered, in this case there is always the same number of jobs in the system and thus no arrival assumption is necessary for this case. [14]

Optimality analysis of energy-performance trade-off for server farm management
Here the assumption of a Poisson arrival process is made for analytical tractability. In the first part of this paper they assume a known fixed arrival rate and in the second part of the paper they assume an unknown time varying arrival rate. [13]

Power-aware Speed Scaling in Processor Sharing Systems
This paper is in this category because it discusses an M/GI/1 queue under Processor Sharing. [31]

Stochastic analysis of power-aware scheduling[30]
This paper is a successor of the paper [31]. So, for further details about the paper see the paragraph regarding "Power-aware speed scaling in processor sharing systems".

Structural properties and exact analysis of energy-aware multiserver queueing systems with setup times
The model that is studied is an M/M/C queue where each server can be powered down without taking any time and can be powered up taking an exponentially distributed time. [22]

3.2 Not Poisson

Not often a different arrival assumption is made than Poisson arrivals. I think this is the case because Poisson arrivals are very well studied and a lot about them is known. However, the paper that have an arrival assumption different from Poisson are mentioned here and as you can see I did not find a lot of them.

Dynamic power allocation in server farms a real time optimization approach
In this paper a Markov Modulated Poisson Process (MMPP) is used to describe the time-varying job arrival process. This choice was made because it qualitatively models the time-varying arrival rate and is still analytically tractable. The server farm that is considered consists of M homogeneous servers all sharing workload and power supply. Incoming jobs are queued at an available server according to the task assignment policy. [5]

4 Type algorithm

In this section the I distinguished the three most common types of algorithms that I encountered. These were the power down, speed scaling and power allocation algorithms. Now, for some categories you might miss one of the papers that I discussed before, then the algorithm was not really worth mentioning again in one of these categories.

4.1 Power down

A power down algorithm or right-sizing a data center is a very simple tactic to decrease the energy consumption. This tactic is especially used in data centers with simple servers, that is servers that can only be turned on or off. The downside of powering down idle servers is that there is a big chance that it has to be powered up again at some point and this usually comes with some cost or delay. In these algorithms there is often a threshold of time before a server is powered down to make sure that it is worth it.

Algorithms for energy conservation in heterogeneous data centers
In this paper, it is studied how a data center with heterogeneous servers can dynamically be right-sized to minimize the energy consumption. [7] The exact details of this algorithm are discussed in section 5 as it is the subject of my experiment.

Algorithms for power savings
In this paper two algorithms are designed one dynamic speed scaling algorithm without a sleep state(DSS-NS) and one dynamic speed scaling algorithm with a sleep state(DSS-S). For the situation with a sleep state a power down algorithm is considered. This power down algorithm works as follows, if the server is in the active state it stays in the active state while there are jobs to process. When there are no more jobs the system becomes idle and remains idle as long as possible until it has to become active again to be able to process all jobs by their deadline at with speed s_{crit}. s_{crit} denotes the critical speed which is the optimal speed if the switching costs were zero. [15]

Dynamic Right-Sizing for Power-Proportional Data Centers
The proposed online algorithm LCP(w) works as follows. Let $X^{LCP(w)} = (x_0^{LCP(w)}, \ldots, x_T^{LCP(w)})$ denote the vector of active server under LCP(w), where w is the prediction window. This vector can be calculated using the following forward recurrence relation:

$$x_\tau^{LCP(w)} = 0 \qquad \tau \leq 0 \qquad x_\tau^{LCP(w)} = (x_{\tau-1}^{LCP(w)})_{x_\tau^{L,w}}^{x_\tau^{U,w}} \qquad \tau \geq 1$$

Where $x_\tau^{L,w}$ and $x_\tau^{U,w}$ are the lower and upper bound respectively.

Energy-aware dispatching in parallel queues with on-off energy consumption
The choice for a power down algorithm is made because not all server and network compo-

nents have the ability to change there processing speed and sometimes a simple power down algorithm can already provide most of the energy savings possible. [25]

Energy efficient scheduling via partial shutdown [17]
This paper discusses an algorithm which is not exactly a power down algorithm but I would categorize it as one. Instead of powering down idle servers they choose a subset of the available servers to power up and each server has its own activation cost. They cannot activate all machines because of an activation cost budget.

Energy efficient algorithms
This paper is kind of a summary or many different energy efficient algorithms. Most of them work for single server systems and some of the methods that are used are power down and speed scaling. This paper discusses a few simple power down algorithms considering two state systems and a couple algorithms which consider a system with multiple lower power states. The simple power down algorithms discussed are a deterministic, after a set amount of time in the idle period the server is powered down if the idle period continues. A randomized algorithm, the time spend in an idle state before powering down is determined according to a probability distribution. Also, one that considers a stochastic setting where the duration of the idle periods is determined by a known probability distribution, in that case the optimal algorithm is the deterministic algorithm. Another power down algorithm is given which is called Lower-Envelope and it is basically a generalization of the deterministic algorithm mentioned before. The last power down algorithm that is mentioned is also a generalization but of the stochastic case given earlier. [6]

Online speed scaling based on active job count to minimize flow plus energy
In this paper the speed scaling algorithm AJC is combined with the power down algorithm IdleLonger. The IdleLonger algorithm works as follows, when the server is in its working state then if $n(t) > 0$ the server will stay in its working state continuing to process jobs, else the server is switched to its idle state. When the server is in its idle state then $t' \leq t$ is the last time the server was in its working state. If the inactive flow over $[t', t]$ equals $(t-t')\sigma$ then the server is switched to its working state, else if $(t-t')\sigma = \omega$ then the server is switched to its sleep state. Otherwise remain in idle state. And finally if the server is in its sleep state then if the inactive flow over $[t', t]$ equals ω the server is switched to its working state. Otherwise stay in the sleep state. The switching cost is denoted by ω. The inactive flow refers to the flow accumulated due to new jobs arriving and not being processed. [18]

Structural properties and exact analysis of energy-aware multiserver queueing systems with setup times
In this paper two policies are considered called bulk setup and staggered threshold. The bulk setup policy and staggered threshold policy both have two decision variables namely C_s and k which denote the number of servers which always stay on and the threshold variable such that the dynamic servers behave in a specific manner respectively. This means for the bulk setup policy that when there are $C_s + 2k$ jobs in the system and the second server is off then all dynamic servers will be put into setup and when the first server is turned on the rest will be turned off again. For the staggered threshold policy this means that when there are $C_s + 2k$ jobs in the system and no dynamic servers are turned on then only two servers will go into setup. The choice for a power down algorithm is based on the already existing literature. [22]

4.2 Speed scaling

Speed scaling algorithm uses the property of the adjustable speed setting. Not for all servers is this possible, in that case usually a power down algorithm is considered.

A scheduling model for reduced CPU energy
This paper presented the Average Rate Heuristic. This heuristic works as follows, each job j has density $d_j = \frac{R_j}{b_j - a_j}$ where R_j is the required number of CPU cycles and b_j and a_j are the deadline and arrival time of job j respectively. What the average rate heuristic does, is that it sets the processor speed at $s(t) = \sum_j d_j(t)$ and it can be checked easily that this creates a feasible schedule. [32] This speed scaling algorithm is not too complicated and can be a good basis for developing other possibly more complex speed scaling algorithms. In this paper they wanted to research the realistic case of a computer with a variable speed processor and then it makes a lot of sense to use its speed adjusting abilities in the algorithm.

Algorithms for power savings
In this paper two algorithms are researched namely a dynamic speed scaling algorithm without sleep state and a dynamic speed scaling algorithm with sleep state. The second algorithm with the sleep state is a power down algorithm, but this one is different from most because it is combined with speed scaling. They assume a convex power function and all jobs have to be finished between their release time and deadline for it to be a feasible schedule. For the offline DSS-NS case the algorithm from [32] is used. [15]

Energy efficient online deadline scheduling
In this paper a new job allocation algorithm is proposed called FSA and it is combined with the speed scaling algorithm OAT. OAT chooses the minimum between T and the speed used by the OA algorithm. The OA algorithm is also discussed in [32]. How FSA works is that it will admit a job J for processing if when using the maximum speed it can complete job J and all of the remaining work of the already admitted jobs. FSA(OAT) is another speed scaling algorithm but it takes into account a more realistic assumption of the server having a maximum speed. It is also different from a simple speed scaling algorithm because it is combined with a job allocation algorithm. [19]

Energy efficient algorithms
This paper is kind of a summary or many different energy efficient algorithms. Most of them work for single server systems and some of the methods that are used are power down and speed scaling. This paper discusses multiple speed scaling algorithms including the offline algorithm YDS and the online algorithms Average Rate and Optimal Available. These algorithms are analysed in [32], with the YDS algorithm being the offline algorithm given in [32]. Another online algorithm called BKP is discussed which approximates the optimal speeds of YDS using the knowledge from the jobs that already arrived. These algorithms are all solutions to the deadline scheduling problem. Then some adjusted versions of the deadline scheduling problem are considered. The first extension being the situation where server speeds are bounded and for this the paper explains how the YDS algorithm can be adapted, here [19] is also shortly mentioned for the online case. The second extension considers temperature minimization where it is mentioned that the YDS and BKP algorithms actually have favorable properties for this. The last extension regarding this subject considers systems with a sleep state, which means no energy will be consumed only in that state. For this case [15] is mentioned as an offline and online solution. For this same case [9] and [12] are also mentioned even though they do not consider speed scaling. [6] The algorithm

researched in [19] is also shortly discussed.

Minimizing energy on homogeneous processors with shared memory
The algorithm discussed in this paper is actually for the largest part a job allocation algorithm but a part of it is a speed scaling algorithm. The speed can be calculated by dividing the sum of the workload of the tasks over the length of the interval. In this paper it is assumed that the power function is convex and each core has an individual voltage supply for which the speed changes are in a continuous fashion. [10]

Online speed scaling based on active job count to minimize flow plus energy
In this paper first the AJC algorithm is discussed. AJC works as follows, at any time t the job with the shortest remaining work is chosen to be processed at the speed $(n(t) + \sigma)^{1/\alpha}$. Here $n(t)$ is the number of active jobs at time t, the static power is denoted by σ and s^{α} denotes the dynamic speed with α being a constant $\alpha > 1$. The algorithm sets the speed based on the active job count and speed changes are only implemented at job arrival or completion, this is different from previous work which changes the speed continuously over time. Then they discuss a power down algorithm called IdleLonger and combine the two algorithms to get a better competitive ratio. The IdleLonger algorithm is further discussed in section 4.1 [18]

Power-aware Speed Scaling in Processor Sharing Systems
In this paper they researched three different speed scaling algorithms each one being a little more dynamic than the last one. They try to find out how much the energy efficiency improves with each dynamic improvement. This can help with finding the best dynamic speed scaling algorithm. The following algorithms are researched in this paper. Static provisioning: one constant speed is used by the server throughout, Gated static provisioning: the server is powered down when there are no jobs to process and Dynamic speed scaling: the server speed is adapted to the number of job arrivals in the system. It is assumed that the server can have any speed and there are no switching costs. It is found that even with these ideal assumptions the dynamic speed scaling algorithm is not that much better than an optimal gated static provisioning algorithm. However, the dynamic speed scaling algorithm does have increased robustness to bursty traffic and mis-estimation of workload parameters. [31]

Scheduling for reduced CPU energy
Three different speed scaling algorithms are simulated. These are the unbounded-delay perfect-future (OPT), bounded-delay limited-future (FUTURE), and bounded-delay limited-past (PAST). These algorithms adjust the CPU clock speed at the same time that scheduling decisions are made With the goal of decreasing time spend in the idle state while retaining interactive response. The first algorithm discussed called OPT is an offline algorithm so it needs perfect future knowledge. The OPT algorithm stretches out all the run times to fill the idle times, not considering the off-time. This is not too interesting because you have to know the future and it creates many delays, but it is optimal in energy efficiency and therefore will be a good baseline. FUTURE is the second algorithm discussed and is similar to OPT except that it uses only a small part of the future. With this algorithm there are no delays allowed and the energy efficiency is close to optimal but it still uses knowledge of the future so it is still unpractical. The last algorithm discussed is called PAST and it looks at a small part of the past to predict the next window. This works quite well and it even approaches OPT as the interval between speed adjustments is lengthened, but a long adjustment interval also compromises the performance. Therefore a balance between the

energy efficiency and performance has to be determined. [29]

Stochastic analysis of power-aware scheduling [30]
This paper is a successor of the paper [31]. So, for further details about the speed scaling algorithms in this paper, see the paragraph regarding "Power-aware speed scaling in processor sharing systems".

4.3 Power allocation

Whether power allocation should be a separate category can be debated. Power determines at what speed a server will run in most cases so it can also be said that power allocation is basically speed scaling. However, I made it a separate category because there are multiple papers which explicitly talk about power allocation and it is a little bit different from speed scaling. When people talk about power allocation it is usually assumed that there some power available and it has to be distributed over the servers which is not completely the same problem as considered with speed scaling.

Optimal power allocation in server farms
Three different power allocation schemes are tested namely, PowMax, PowMed and PowMin which means running the servers at their highest, intermediate or lowest power level respectively. The conclusion is that for different scenarios a different power level is optimal.

Dynamic power allocation in server farms a real time optimization approach
In this paper a dynamic power allocation scheme is developed and compared to a static power allocation scheme. Then an RTO approach is used to solve the average power minimization problem over a dynamic power allocation scheme. Then by simulations it is shown that under the condition called slowly varying job arrival rate, typically satisfied in server farms, the dynamic power allocation scheme significantly improves energy efficiency compared to the static scheme with the allowable mean delay. The optimal static is better as workload rate changes become faster and more abrupt. [5]

5 The experiment

I chose to further research the algorithm from the paper "Algorithms for energy conservation in heterogeneous data centers" [7]. The decision was made to examine a model with heterogeneous servers. The following problem formulation was given by the paper.

We consider a data center with d different server types. There are m_j servers of type j. Each server has an active state where it is able to process jobs, and an inactive state where no energy is consumed. Powering up a server of type j (i.e., switching from the inactive into the active state) incurs a cost of β_j (called switching cost); powering down does not cost anything. We consider a finite time horizon consisting of the time slots $\{1, \dots, T\}$. For each time slot $t \in \{1, \dots, T\}$, jobs of total volume $\lambda_t \in \mathbb{N}_0$ arrive and have to be processed during the time slot. In practice, the job volume might be predicted from history or it is assumed that the jobs arriving during a time slot have to be finished in the next time slot. In our model, all servers have the same computational power and can process a job volume of 1 per time slot. Hence, there must be at least λ_t active servers at time t to process the arriving jobs. We consider a basic setting where the operating cost of a server of type j is load- and time-independent and denoted by $l_j \in \mathbb{R}_{\geq 0}$. Hence, an active server incurs a

constant but type-dependent operating cost per time slot.

A schedule X is a sequence $x_1, \ldots, x_T$ with $x_t = (x_{t,1}, \ldots, x_{t,d})$ where each $x_{t,j}$ indicates the number of active servers of type j during time slot t. At the beginning and the end of the considered time horizon all servers are shut down, i.e., $x_0 = x_{T+1} = (0, \ldots, 0)$. A schedule is called feasible if there are enough active servers to process the arriving jobs and if there are not more active servers than available, i.e., $\sum_{j=1}^{d} x_{t,j} \lambda_t$ and $x_{t,j} \in \{0, 1, \ldots, m_j\}$ for all $t \in \{1, \ldots, T\}$ and $j \in \{1, \ldots, d\}$. The cost of a feasible schedule is defined by $C(X) := \sum_{t=1}^{T} (\sum_{j=1}^{d} l_j x_{t,j} + \sum_{j=1}^{d} \beta_j (x_{t,j} - x_{t-1,j})^+)$ where $(x)^+ := max(x, 0)$.
The switching cost is only paid for powering up. However, this is not a restriction, since all servers are inactive at the beginning and end of the workload. Thus the cost of powering down can be folded into the cost of powering up. A problem instance is specified by the tuple $I = (T, d, \mathbf{m}, \boldsymbol{\beta}, \mathbf{l}, \boldsymbol{\Lambda})$ where $\mathbf{m} = (m_1, \ldots, m_d), \boldsymbol{\beta} = (\beta_1, \ldots, \beta_d), \mathbf{l} = (l_1, \ldots, l_d)$ and $\boldsymbol{\Lambda} = (\lambda_1, \ldots, \lambda_T)$. The task is to find a schedule with minimum cost. We focus on the central case without inefficient server types. A server type j is called inefficient if there is another server type $j' \neq j$ with both smaller (or equal) operating and switching costs, i.e., $l_j \geq l'_j$ and $\beta_j \geq \beta'_j$. This assumption is natural because a better server type with a lower operating cost usually has a higher switching cost. An inefficient server of type j is only powered up, if all servers of all types j' with $\beta'_j \leq \beta_j$ and $l'_j \leq l_j$ are already running. Therefore, excluding inefficient servers is not a relevant restriction in practice. [7]

For this problem a deterministic online power down algorithm was constructed. This algorithm is 2d-competitive. The algorithm works as follows, first an optimal schedule is computed for the problem instance ending at the current time slot. Then the operating costs of the active servers are compared to the operating costs of the servers in the optimal schedule and if the operating costs of the active servers are higher then the operating costs of the servers in the optimal schedule then servers with higher operating costs are replaced with servers with lower operating costs. Also, a server is powered down after a certain amount of time not being used dependent on its operating and switching costs. The pseudocode given by [7] can also be seen in figure 1.

Algorithm 1 Algorithm $\mathcal{A}$.

Input: $\mathcal{I} = (T, d, \boldsymbol{m}, \boldsymbol{\beta}, \boldsymbol{l}, \Lambda)$ with $\Lambda = (\lambda_1, \ldots, \lambda_T)$
Output: $X^{\mathcal{A}} = (x_1^{\mathcal{A}}, \ldots, x_T^{\mathcal{A}})$
1: Set $e_k := 0$ for all $k \in [d]$
2: **for** $t := 1$ **to** T **do**
3: Calculate $\hat{X}^t$ such that $\hat{y}^t_{t',k} \geq \hat{y}^{t-1}_{t',k}$ for all $t' \in [t]$ and $k \in [m]$
4: **for** $k := 1$ **to** m **do**
5: **if** $y^{\mathcal{A}}_{t-1,k} < \hat{y}^t_{t,k}$ **or** $t \geq e_k$ **then**
6: $y^{\mathcal{A}}_{t,k} := \hat{y}^t_{t,k}$
7: $e_k := t + \bar{t}_{y^{\mathcal{A}}_{t,k}}$ where $\bar{t}_0 := 0$
8: **else**
9: $y^{\mathcal{A}}_{t,k} := y^{\mathcal{A}}_{t-1,k}$
10: $e_k := \max\{e_k, t + \bar{t}_{\hat{y}^t_{t,k}}\}$ where $\bar{t}_0 := 0$
11: Set $x^{\mathcal{A}}_{t,j} := |\{k \in [m] \mid y^{\mathcal{A}}_{t,k} = j\}|$ for all $j \in [d]$

FIGURE 1: Pseudocode of the algorithm

The optimal schedule can be found by using a dynamic program. This dynamic program was not given in the paper so I had to make it myself and it works as follows. The stages are each time slot t, the possible states are the decisions that could have been made in the previous time slot and these are all combinations of servers s.t. $\sum_{j=1}^{d} x_{t-1,j} \geq \lambda_{t-1}$, expect for the case when $t = 1$ then the only possibility is no servers are active because $x_0 = x_{T+1} = (0, \ldots, 0)$ is given. The possible decisions during time slot t are denoted by all combinations of server s.t. $\sum_{j=1}^{d} x_{t,j} \geq \lambda_t$, except for the case when $t = T$ then $\sum_{j=1}^{d} x_{t,j} = \lambda_t$ must hold. Also, $0 \leq x_{t,j} \leq m_j$ must be true at all times. Now the optimal schedule can be found by calculating the optimal decision during each stage for each state with $f_t(x_{t-1}) = min\{\text{cost at stage } t + f_{t+1}(x_t)\}$. The cost at stage t is denoted by $\sum_{j=1}^{d} l_j x_{t,j} + \sum_{j=1}^{d} \beta_j (x_{t,j} - x_{t-1,j})^+ \text{where} (x)^+ := max(x, 0)$.

Now when having the algorithm from the paper and the dynamic program for the optimal schedule I implemented this in Python and ran several simulations, the code can be found in Appendix Python code. The results will be discussed in the next section.

5.1 Results

I ran several simulations, most of them with the same m, l, β, d, t and a variable input of Λ. For the instance that took the shortest amount of time I also tried a few larger instances to see what influence the time span has.

Before I could run the simulations with the variable Λ I first had to find out what the largest number of servers was that I could run my program with within a reasonable time frame(2 hours). By trial and error I found out that $m = (5, 5, 4, 4, 4)$ is the maximum amount of servers I could use in my program and it would be finished within two hours. The choice for l and β was not something I thought about thoroughly because the effect of different l and β is not what I wanted to investigate, but this could be part of further research. I did a number of 35 simulations all with a different Λ.

In Appendix simulation results you can see the complete results I gathered from the simulations. More info can be found there regarding the simulations like the schedules that were made by the algorithm.

There are a few interesting observations that can be made from the results. The most interesting case I experimented with was a fixed amount of total jobs that arrived during the complete time span, these instances are colored green in figure 2 and have a total job arrival of 330 which is half of the total amount possible with these servers and this time span.

Actually the instance with the highest increase in cost in percentage is from the green set. It has a significantly higher cost increase than the second highest cost increase which are respectively 16,2% and 13,9%. The arrival pattern with the largest cost increase is the one with the maximum amount of jobs per time slot arriving for the first half of the time and the second half only no job arrivals. It is interesting that if the order of job arrivals for this instance is exactly the other way around then the schedule from the paper algorithm has the same cost as the optimal schedule.

The cost of the optimal schedule being the same as the cost of the paper schedule only occurred for this one instance, this instance can be seen at the bottom of figure 2. It makes sense that there is no cost increase for this instance because if at the first time slot no jobs arrive there is no benefit in turning on servers. Even if jobs arrive in the next time slot it is not beneficial to turn them on before this time slot because there will be unnecessary operational costs during that time slot. Also for this case when there are jobs arriving there is only one possible way to do this because all servers have to be turned on and this is automatically the same schedule as the optimal schedule. I also tried another instance that has a similar structure, this instance and the resulting schedule can be seen in figure 3. This instance does have a significant cost increase of 10,78% and that is because it actually acts the same as an uniform instance when jobs start arriving. When we look at the uniform instance with 11 jobs arriving you can see that the schedule is the same as the one in figure 3, it only shifted.

Structure of input	Input Lambda	Total jobs arrived	Average job arrival per time slot	Cost paper schedule	Cost optimal schedule	Cost increase in percentage
First half all jobs then no jobs arriving	(22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)	330	11	1290	1110	16,22
Alternating max number jobs and zeros	(22, 22, 22, 22, 22, 22, 22, 22, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 22, 22, 22, 22, 22, 22, 22)	330	11	1321	1185	11,48
Lower numbers left then higher numbers then zeros	(6, 11, 5, 7, 9, 13, 11, 13, 11, 9, 10, 7, 19, 13, 15, 20, 13, 21, 15, 17, 17, 18, 16, 20, 14, 0, 0, 0, 0, 0)	330	11	1021	921	10,86
Random numbers between 6 and 16	(7, 9, 15, 15, 12, 14, 10, 7, 9, 15, 9, 9, 7, 9, 9, 13, 14, 15, 13, 10, 8, 10, 14, 14, 6, 13, 15, 11, 11, 7)	330	11	892	815	9,45
Higher numbers left then lower numbers then zeros	(19, 13, 15, 20, 13, 21, 15, 17, 17, 18, 16, 20, 14, 6, 11, 5, 7, 9, 13, 11, 13, 11, 9, 10, 7, 0, 0, 0, 0, 0)	330	11	1003	925	8,43
Alternating max number jobs and zeros	(22, 22, 22, 22, 22, 0, 0, 0, 0, 0, 22, 22, 22, 22, 22, 0, 0, 0, 0, 0, 22, 22, 22, 22, 22, 0, 0, 0, 0, 0)	330	11	1352	1252	7,99
Random numbers between 0 and 22	(9, 10, 19, 15, 15, 15, 13, 8, 16, 11, 7, 2, 20, 7, 1, 20, 4, 19, 6, 10, 9, 16, 4, 16, 7, 7, 1, 8, 15, 20)	330	11	1117	1049	6,48
Lower numbers left of zeros higher numbers right of zeros	(6, 11, 5, 7, 9, 13, 11, 13, 11, 9, 10, 7, 0, 0, 0, 0, 0, 19, 13, 15, 20, 13, 21, 15, 17, 17, 18, 16, 20, 14)	330	11	1021	961	6,24
Uniform	(11,)*30	330	11	726	686	5,83
Higher numbers left of zeros lower numbers right of zeros	(19, 13, 15, 20, 13, 21, 15, 17, 17, 18, 16, 20, 14, 0, 0, 0, 0, 0, 6, 11, 5, 7, 9, 13, 11, 13, 11, 9, 10, 7)	330	11	1003	961	4,37
Zeros then lower numbers then higher numbers	(0, 0, 0, 0, 0, 6, 11, 5, 7, 9, 13, 11, 13, 11, 9, 10, 7, 19, 13, 15, 20, 13, 21, 15, 17, 17, 18, 16, 20, 14)	330	11	961	921	4,34
Alternating max and min number jobs	(0,22,)*15	330	11	1712	1656	3,38
Zeros then higher numbers then lower numbers	(0, 0, 0, 0, 0, 19, 13, 15, 20, 13, 21, 15, 17, 17, 18, 16, 20, 14, 6, 11, 5, 7, 9, 13, 11, 13, 11, 9, 10, 7)	330	11	943	925	1,95
No jobs first half, all jobs second half	(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22)	330	11	1110	1110	0,00

FIGURE 2: Results total job arrival 330

Structure of input	Input Lambda	Schedule paper
No jobs first half, half job capacity second half	(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11)	[[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [5, 5, 0, 0, 1], [0, 0, 3, 4, 4], [0, 0, 3, 4, 4], [0, 0, 3, 4, 4], [0, 0, 3, 4, 4], [0, 0, 3, 4, 4], [0, 0, 3, 4, 4], [0, 0, 3, 4, 4], [0, 0, 3, 4, 4], [0, 0, 3, 4, 4], [0, 0, 3, 4, 4], [0, 0, 3, 4, 4], [0, 0, 3, 4, 4], [0, 0, 3, 4, 4], [0, 0, 3, 4, 4]]

FIGURE 3: Instance structure similar to the one with no cost increase

What is also interesting is that four uniform instances that I tested are in the top ten of highest cost increase. It is clear that a uniform instance with low job arrivals has a larger cost increase than a uniform instance with higher job arrivals, this can be seen from 4. This is very logical because for the first time slot higher server types will be used because the optimal schedule for one time slot uses higher server types. For example, if there are 21 job arrivals then all except one server will be used and this one will be the lowest server type for one time slot and for further time slots the one unused server will be the highest server type and the switching cost of this server during the previous time slot will be lost. Now another situation is that only one job arrives during the first time slot then again the highest type server will be turned on and for later time slots it will be turned off, the

lowest server type will be turned on and the switching cost of the higher server type will be lost. So, the same thing happens but with 21 jobs arriving the overall cost will be a lot higher then when only 1 job arrives so the relative difference in cost increase is higher for a smaller amount of jobs arriving than when more jobs arrive.

Structure of input	Input Lambda	Total jobs arrived	Average job arrival per time slot	Cost paper schedule	Cost optimal schedule	Run time in seconds	Run time in minutes	Cost increase in percentage
	(2,)*30	60	2	82	72	6.857	114,28	13,89
Uniform	(5,)*30	150	5	233	209	6.496	108,27	11,48
	(9,)*30	270	9	537	498	3.642	60,70	7,83
	(10,)*30	300	10	634	592	6.895	114,92	7,09
Uniform	(11,)*30	330	11	726	686	1.932	32,21	5,83
	(15,)*30	450	15	1170	1146	151	2,52	2,09

FIGURE 4: Only uniform arrival instances

The effect of the l and β choices probably determines how many time slots differ from the optimal schedule or at least this is the case for a uniform instance. What I mean by this is that now for the uniform instances only the first time slot is different and the rest of schedule is the same for the remaining time. This means that only for one time slot certain servers are beneficial to use but if the operating or switching costs are adjusted it could be that using these servers for two or more time slots can still be beneficial before switching to more long term server. This will of course influence the cost increase and I think that higher cost increases will be seen with, for example a higher difference in switching costs.

If the instance is uniform than a longer time span means less cost increase in percentage because the schedule converges until the schedule of each time slot is the same so the difference between the optimal schedule and the paper schedule will be in the first few slot and the total cost difference will be the same.

I also experimented with several instances where the pattern is alternating, so $\Lambda = (a, b,) * 15$ with $a < 11$ and $b > 11$, and the other way around so $\Lambda = (b, a,) * 15$. The results from these simulations can be seen in figure 5.

I discovered that the $\Lambda = (b, a,) * 15$ so the one starting with the higher number, has less of a cost increase than the other way around. This makes sense because if at $t = 1$ 20 jobs arrive then there are not a lot of different possibilities of which and how many servers to turn on because most of them have to be on, when the maximum number of servers is 22. So, because most servers have to be turned on the difference in costs between the different possibilities will not be too great. Then, for the next time slot less jobs arrive so the necessary servers are already on and only some will be turned off and for the time slot after that, the same servers will be used as in the first time slot so the schedule for the odd and the even time slots will be the same respectively. However, this does not work if first a small amount of jobs arrive because for one time slot it will be cheaper to use different servers for a small amount of jobs and after the first time slot the schedule will go back to same pattern as for the $\Lambda = (b, a,) * 15$ schedule.

Structure of input	Input Lambda	Total jobs arrived	Average job arrival per time slot	Cost paper schedule	Cost optimal schedule	Cost increase in percentage
Alternating low and high arrivals	(5,18,)*15	345	11,5	1312	1298	1,08
Alternating high and low arrivals	(18,5,)*15	345	11,5	1304	1298	0,46
Alternating low and high arrivals	(10,15,)*15	375	12,5	1074	1042	3,07
	(9,16,)*15	375	12,5	1155	1128	2,39
	(8,17,)*15	375	12,5	1237	1214	1,89
Alternating high and low arrivals	(17,8,)*15	375	12,5	1218	1214	0,33
	(19,7,)*15	375	12,5	1409	1393	1,15
Alternating low and high arrivals	(7,19,)*15	375	12,5	1427	1393	2,44
Longer time span alternating high and low arrivals	(19,7,)*30	390	13	2774	2743	1,13
	(19,7,)*50	1300	13	4594	4543	1,12
	(19,7,)*100	2600	13	9144	9043	1,12

FIGURE 5: Alternating instances shaped like (a,b, ..., a,b)

Unfortunately I do not think a lot of solid conclusions can be drawn from my results because the difference in cost increase is overall not too big. In figure 6 it can be seen that three quarters of the data points are in the zero to 8% cost increase range. I believe that a lot more simulations are necessary to be able to really say something about the quality of the algorithm. However, another explanation for these relative cost increases being so low is that the algorithm functions really well, but like I said a lot more simulations are necessary to be able to claim this.

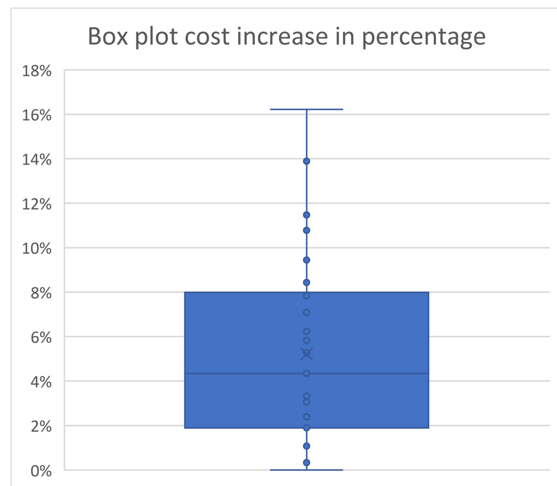

FIGURE 6: Boxplot with data points being relative cost increase

6 Conclusions

What can be concluded from the literature review is the following. By categorizing the papers that I came across and found relevant to the study of energy efficient scheduling in data centers I concluded the following. When categorizing per server type I saw that most papers still consider the single server situation. This is probably the case because because this situation has been around the longest and it is a solid base for the multiple server case. When categorizing the papers by their arrival assumption I found that if they have an arrival assumption it is probably a Poisson process. Also, all the papers that I found having an arrival assumption do not consider heterogeneous servers. One of the reasons that I came up with is that most papers considering heterogeneous servers want to consider a model that is as close to reality as possible and therefore use real data in their simulations which does not require an arrival assumption. Lastly from categorizing per type algorithm I only discovered that for most cases when a speed scaling algorithm is considered, the model uses a single server. Combining this with the conclusion drawn from the server type category, it is fairly safe to say that a speed scaling algorithm usually considers the single server case and the other way around. But we do have to keep in mind that I did not exhaust the literature regarding the subject, so my conclusions should be taken with a grain of salt.

Now, for the conclusion from my experiment. From [7] it is known that the algorithm is 2d competitive which means that the cost of the schedule made by the algorithm is at most 2d higher than the cost of the optimal schedule. For my case that would mean a cost increase of at most 1000% and I have not even gotten a little bit close to that with a maximum of 16,22% cost increase. So, I am wondering what kind of arrival pattern is necessary for that. It is also possible that the key is not the arrival pattern but the number of servers or the operating and/or switching cost of the servers. But this is not something I looked into due to time constraints. However, my results could also indicate that the algorithm is in practice a lot better than the upper bound of 2d. Which is definitely possible because the [7] does not mention the average case and they do not have experimental results. But to find out which of these explanations is correct I think a lot more simulations have to be done, also with possibly more situations that are closer to reality.

References

[1] Data centers – what are the costs of ownership?, Oct 2013. Accessed on November 4, 2022.

[2] Costs of a data center, 2019. Accessed on November 4, 2022.

[3] Euro area energy pricesdecember 2022 data - 1996-2021 historical - january forecast, Dec 2022. Accessed on November 4, 2022.

[4] Samuli Aalto and Ziv Scully. On the gittins index for multistage jobs. *Queueing Systems*, 102:353–371, 12 2022.

[5] Mohammadreza Aghajani, Luca Parolini, and Bruno Sinopoli. Dynamic power allocation in server farms: A real time optimization approach. pages 3790–3795. Institute of Electrical and Electronics Engineers Inc., 2010.

[6] Susanne Albers. Energy-efficient algorithms. *Communications of the ACM*, 53:86–96, 5 2010.

[7] Susanne Albers and Jens Quedenfeld. Algorithms for energy conservation in heterogeneous data centers. *Theoretical Computer Science*, 896:111–131, 12 2021.

[8] F. A. Armenta-Cano, A. Tchernykh, J. M. Cortes-Mendoza, R. Yahyapour, A. Yu Drozdov, P. Bouvry, D. Kliazovich, A. Avetisyan, and S. Nesmachnow. Min_c: Heterogeneous concentration policy for energy-aware scheduling of jobs with resource contention. *Programming and Computer Software*, 43:204–215, 5 2017.

[9] Philippe Baptiste, Marek Chrobak, and Christoph Dürr. Polynomial-time algorithms for minimum energy scheduling. *ACM Transactions on Algorithms*, 8, 7 2012.

[10] Vincent Chau, Chi Kit Ken Fong, Shengxin Liu, Elaine Yinling Wang, and Yong Zhang. Minimizing energy on homogeneous processors with shared memory. *Theoretical Computer Science*, 866:160–170, 4 2021.

[11] Tomasz Ciesielczyk, Alberto Cabrera, Ariel Oleksiak, Wojciech Piątek, Grzegorz Waligóra, Francisco Almeida, and Vicente Blanco. An approach to reduce energy consumption and performance losses on heterogeneous servers using power capping. *Journal of Scheduling*, 24:489–505, 10 2021.

[12] Erik D. Demaine, Mohammad Ghodsi, Mohammadtaghi Hajiaghayi, Amin S. Sayedi-Roshkhar, and Morteza Zadimoghaddam. Scheduling to minimize gaps and power consumption. *Journal of Scheduling*, 16:151–160, 4 2013.

[13] Anshul Gandhi, Varun Gupta, Mor Harchol-Balter, and Michael A. Kozuch. Optimality analysis of energy-performance trade-off for server farm management. volume 67, pages 1155–1171, 11 2010.

[14] Anshul Gandhi, Mor Harchol-Balter, Rajarshi Das, and Charles Lefurgy. *Optimal Power Allocation in Server Farms*. Association for Computing Machinery, 2009.

[15] Sandy Irani, Sandeep Shukla, and Rajesh Gupta. Algorithms for power savings. *ACM Trans. Algorithms*, 3(4):41–es, nov 2007.

[16] George Kamiya. Data centres and data transmission networks – analysis, Sep 2022. Accessed on November 4, 2022.

[17] Samir Khuller, Jian Li, and Barna Saha. Energy efficient scheduling via partial shutdown. *CoRR*, abs/0912.1329, 2009.

[18] Tak Wah Lam, Lap Kei Lee, Isaac K.K. To, and Prudence W.H. Wong. Online speed scaling based on active job count to minimize flow plus energy. *Algorithmica*, 65:605–633, 3 2013.

[19] Lap-Kei Lee, Prudence W H Wong, H L Chan, W T Chan, T W Lam, L K Lee, K S Mak, and P Wong. Energy efficient online deadline scheduling, 2007.

[20] Minghong Lin, Adam Wierman, Lachlan L.H. Andrew, and Eno Thereska. Dynamic right-sizing for power-proportional data centers. *IEEE/ACM Transactions on Networking*, 21:1378–1391, 2013.

[21] V. J. Maccio and D. G. Down. On optimal policies for energy-aware servers. *Performance Evaluation*, 90:36–52, 6 2015.

[22] V. J. Maccio and D. G. Down. Structural properties and exact analysis of energy-aware multiserver queueing systems with setup times. *Performance Evaluation*, 121-122:48–66, 5 2018.

[23] Eric Masanet, Arman Shehabi, Nuoa Lei, Sarah Smith, and Jonathan Koomey. Recalibrating global data center energy-use estimates. *Science*, 367(6481):984–986, 2020.

[24] Debdeep Paul, Wen De Zhong, and Sanjay K. Bose. Energy efficient scheduling in data centers. volume 2015-September, pages 5948–5953. Institute of Electrical and Electronics Engineers Inc., 9 2015.

[25] Aleksi Penttinen, Esa Hyytiä, and Samuli Aalto. Energy-aware dispatching in parallel queues with on-off energy consumption. 2011.

[26] Mrudula Sarvabhatla, Swapnasudha Konda, Chandra Sekhar Vorugunti, and M. M.Naresh Babu. A dynamic and energy efficient greedy scheduling algorithm for cloud data centers. volume 2018-January, pages 47–52. Institute of Electrical and Electronics Engineers Inc., 4 2018.

[27] Stefano Sebastio and Giorgio Gnecco. A green policy to schedule tasks in a distributed cloud. *Optimization Letters*, 12, 2018.

[28] Joohyung Sun and Hyeonjoong Cho. A lightweight optimal scheduling algorithm for energy-efficient and real-time cloud services. *IEEE Access*, 10:5697–5714, 2022.

[29] Mark Weiser, Brent Welch, Alan Demers, and Scott Shenker. *Scheduling for Reduced CPU Energy*, pages 449–471. Springer US, Boston, MA, 1996.

[30] Adam Wierman, Lachlan L.H. Andrew, and Ao Tang. Stochastic analysis of power-aware scheduling. pages 1278–1283, 2008.

[31] Adam Wierman, Lachlan L.H. Andrew, and Ao Tang. Power-aware speed scaling in processor sharing systems. pages 2007–2015, 2009.

[32] Frances Yao, Alan Demers, and Scott Shenker. A scheduling model for reduced cpu energy. pages 374–382. IEEE, 1995.

Appendix simulation results

Structure of input	Input Lambda	Schedule paper
No jobs first half, half job capacity second half	**(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11)**	**[[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [5, 5, 0, 0, 1], [0, 0, 3, 4, 4], [0, 0, 3, 4, 4], [0, 0, 3, 4, 4], [0, 0, 3, 4, 4], [0, 0, 3, 4, 4], [0, 0, 3, 4, 4], [0, 0, 3, 4, 4], [0, 0, 3, 4, 4], [0, 0, 3, 4, 4], [0, 0, 3, 4, 4], [0, 0, 3, 4, 4], [0, 0, 3, 4, 4], [0, 0, 3, 4, 4], [0, 0, 3, 4, 4]]**
Alternating low and high arrivals	(5,18,)*15	[[0, ,5, 0, 0, 0], [5, 5, 0, 4, 4], [0, 0, 0, 4, 4], [5, 5, 0, 4, 4], [0, 0, 0, 4, 4], [5, 5, 0, 4, 4], [0, 0, 0, 4, 4], [5, 5, 0, 4, 4], [0, 0, 0, 4, 4], [5, 5, 0, 4, 4], [0, 0, 0, 4, 4], [5, 5, 0, 4, 4], [0, 0, 0, 4, 4], [5, 5, 0, 4, 4], [0, 0, 0, 4, 4], [5, 5, 0, 4, 4], [0, 0, 0, 4, 4], [5, 5, 0, 4, 4], [0, 0, 0, 4, 4], [5, 5, 0, 4, 4], [0, 0, 0, 4, 4], [5, 5, 0, 4, 4], [0, 0, 0, 4, 4], [5, 5, 0, 4, 4], [0, 0, 0, 4, 4], [5, 5, 0, 4, 4], [0, 0, 0, 4, 4], [5, 5, 0, 4, 4], [0, 0, 0, 4, 4], [5, 5, 0, 4, 4]]
Alternating high and low arrivals	(18,5,)*15	**[[5, 5, 0, 4, 4], [0, 0, 0, 4, 4], [5, 5, 0, 4, 4], [0, 0, 0, 4, 4], [5, 5, 0, 4, 4], [0, 0, 0, 4, 4], [5, 5, 0, 4, 4], [0, 0, 0, 4, 4], [5, 5, 0, 4, 4], [0, 0, 0, 4, 4], [5, 5, 0, 4, 4], [0, 0, 0, 4, 4], [5, 5, 0, 4, 4], [0, 0, 0, 4, 4], [5, 5, 0, 4, 4], [0, 0, 0, 4, 4], [5, 5, 0, 4, 4], [0, 0, 0, 4, 4], [5, 5, 0, 4, 4], [0, 0, 0, 4, 4], [5, 5, 0, 4, 4], [0, 0, 0, 4, 4], [5, 5, 0, 4, 4], [0, 0, 0, 4, 4], [5, 5, 0, 4, 4], [0, 0, 0, 4, 4], [5, 5, 0, 4, 4], [0, 0, 0, 4, 4], [5, 5, 0, 4, 4], [0, 0, 0, 4, 4]]**
Alternating low and high arrivals	(10,15,)*15	[[5, 5, 0, 0, 0], [0, 5, 2, 4, 4], [0, 0, 2, 4, 4], [0, 5, 2, 4, 4], [0, 0, 2, 4, 4], [0, 5, 2, 4, 4], [0, 0, 2, 4, 4], [0, 5, 2, 4, 4], [0, 0, 2, 4, 4], [0, 5, 2, 4, 4], [0, 0, 2, 4, 4], [0, 5, 2, 4, 4], [0, 0, 2, 4, 4], [0, 5, 2, 4, 4], [0, 0, 2, 4, 4], [0, 5, 2, 4, 4], [0, 0, 2, 4, 4], [0, 5, 2, 4, 4], [0, 0, 2, 4, 4], [0, 5, 2, 4, 4], [0, 0, 2, 4, 4], [0, 5, 2, 4, 4], [0, 0, 2, 4, 4], [0, 5, 2, 4, 4], [0, 0, 2, 4, 4], [0, 5, 2, 4, 4], [0, 0, 2, 4, 4], [0, 5, 2, 4, 4], [0, 0, 2, 4, 4], [0, 5, 2, 4, 4]]
	(9,16,)*15	**[[4, 5, 0, 0, 0], [2, 5, 1, 4, 4], [0, 0, 1, 4, 4], [2, 5, 1, 4, 4], [0, 0, 1, 4, 4], [2, 5, 1, 4, 4], [0, 0, 1, 4, 4], [2, 5, 1, 4, 4], [0, 0, 1, 4, 4], [2, 5, 1, 4, 4], [0, 0, 1, 4, 4], [2, 5, 1, 4, 4], [0, 0, 1, 4, 4], [2, 5, 1, 4, 4], [0, 0, 1, 4, 4], [2, 5, 1, 4, 4], [0, 0, 1, 4, 4], [2, 5, 1, 4, 4], [0, 0, 1, 4, 4], [2, 5, 1, 4, 4], [0, 0, 1, 4, 4], [2, 5, 1, 4, 4], [0, 0, 1, 4, 4], [2, 5, 1, 4, 4], [0, 0, 1, 4, 4], [2, 5, 1, 4, 4], [0, 0, 1, 4, 4], [2, 5, 1, 4, 4], [0, 0, 1, 4, 4], [2, 5, 1, 4, 4]]**
	(8,17,)*15	[[3, 5, 0, 0, 0], [4, 5, 0, 4, 4], [0, 0, 0, 4, 4], [4, 5, 0, 4, 4], [0, 0, 0, 4, 4], [4, 5, 0, 4, 4], [0, 0, 0, 4, 4], [4, 5, 0, 4, 4], [0, 0, 0, 4, 4], [4, 5, 0, 4, 4], [0, 0, 0, 4, 4], [4, 5, 0, 4, 4], [0, 0, 0, 4, 4], [4, 5, 0, 4, 4], [0, 0, 0, 4, 4], [4, 5, 0, 4, 4], [0, 0, 0, 4, 4], [4, 5, 0, 4, 4], [0, 0, 0, 4, 4], [4, 5, 0, 4, 4], [0, 0, 0, 4, 4], [4, 5, 0, 4, 4], [0, 0, 0, 4, 4], [4, 5, 0, 4, 4], [0, 0, 0, 4, 4], [4, 5, 0, 4, 4], [0, 0, 0, 4, 4], [4, 5, 0, 4, 4], [0, 0, 0, 4, 4], [4, 5, 0, 4, 4]]
Alternating high and low arrivals	(17,8,)*15	**[[5, 5, 0, 3, 4], [0, 0, 0, 4, 4], [4, 5, 0, 4, 4], [0, 0, 0, 4, 4], [4, 5, 0, 4, 4], [0, 0, 0, 4, 4], [4, 5, 0, 4, 4], [0, 0, 0, 4, 4], [4, 5, 0, 4, 4], [0, 0, 0, 4, 4], [4, 5, 0, 4, 4], [0, 0, 0, 4, 4], [4, 5, 0, 4, 4], [0, 0, 0, 4, 4], [4, 5, 0, 4, 4], [0, 0, 0, 4, 4], [4, 5, 0, 4, 4], [0, 0, 0, 4, 4], [4, 5, 0, 4, 4], [0, 0, 0, 4, 4], [4, 5, 0, 4, 4], [0, 0, 0, 4, 4], [4, 5, 0, 4, 4], [0, 0, 0, 4, 4], [4, 5, 0, 4, 4], [0, 0, 0, 4, 4], [4, 5, 0, 4, 4], [0, 0, 0, 4, 4], [4, 5, 0, 4, 4], [0, 0, 0, 4, 4]]**
	(19,7,)*15	[[5, 5, 1, 4, 4], [0, 0, 0, 4, 4], [5, 5, 1, 4, 4], [0, 0, 0, 4, 4], [5, 5, 1, 4, 4], [0, 0, 0, 4, 4], [5, 5, 1, 4, 4], [0, 0, 0, 4, 4], [5, 5, 1, 4, 4], [0, 0, 0, 4, 4], [5, 5, 1, 4, 4], [0, 0, 0, 4, 4], [5, 5, 1, 4, 4], [0, 0, 0, 4, 4], [5, 5, 1, 4, 4], [0, 0, 0, 4, 4], [5, 5, 1, 4, 4], [0, 0, 0, 4, 4], [5, 5, 1, 4, 4], [0, 0, 0, 4, 4], [5, 5, 1, 4, 4], [0, 0, 0, 4, 4], [5, 5, 1, 4, 4], [0, 0, 0, 4, 4], [5, 5, 1, 4, 4], [0, 0, 0, 4, 4], [5, 5, 1, 4, 4], [0, 0, 0, 4, 4], [5, 5, 1, 4, 4], [0, 0, 0, 4, 4]]
Alternating low and high arrivals	(7,19,)*15	**[[2, 5, 0, 0, 0], [5, 5, 1, 4, 4], [0, 0, 0, 4, 4], [5, 5, 1, 4, 4], [0, 0, 0, 4, 4], [5, 5, 1, 4, 4], [0, 0, 0, 4, 4], [5, 5, 1, 4, 4], [0, 0, 0, 4, 4], [5, 5, 1, 4, 4], [0, 0, 0, 4, 4], [5, 5, 1, 4, 4], [0, 0, 0, 4, 4], [5, 5, 1, 4, 4], [0, 0, 0, 4, 4], [5, 5, 1, 4, 4], [0, 0, 0, 4, 4], [5, 5, 1, 4, 4], [0, 0, 0, 4, 4], [5, 5, 1, 4, 4], [0, 0, 0, 4, 4], [5, 5, 1, 4, 4], [0, 0, 0, 4, 4], [5, 5, 1, 4, 4], [0, 0, 0, 4, 4], [5, 5, 1, 4, 4], [0, 0, 0, 4, 4], [5, 5, 1, 4, 4], [0, 0, 0, 4, 4], [5, 5, 1, 4, 4]]**
Longer time span alternating high and low arrivals	(19,7,)*30	Same schedule as the one above only longer
	(19,7,)*50	**Same schedule as the one above only longer**
	(19,7,)*100	Same schedule as the one above only longer

FIGURE 7: Result of simulations only the schedules second half of simulations

Structure of input	Input Lambda	Schedule paper
Random numbers between 0 and 22	(21, 17, 16, 0, 20, 11, 12, 20, 22, 6, 12, 21, 18, 4, 3, 16, 16, 12, 2, 17, 5, 8, 22, 17, 3, 16, 6, 10, 13, 16)	[[5, 5, 3, 4, 4], [0, 5, 4, 4, 4], [0, 4, 4, 4, 4], [0, 0, 0, 4, 4], [5, 5, 2, 4, 4], [0, 0, 3, 4, 4], [0, 1, 3, 4, 4], [3, 5, 4, 4, 4], [5, 5, 4, 4, 4], [0, 0, 0, 4, 4], [0, 4, 0, 4, 4], [4, 5, 4, 4, 4], [1, 5, 4, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [5, 3, 0, 4, 4], [0, 4, 4, 4, 4], [0, 0, 4, 4, 4], [0, 0, 0, 4, 4], [4, 5, 0, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [5, 5, 4, 4, 4], [0, 5, 4, 4, 4], [0, 0, 0, 4, 4], [3, 5, 0, 4, 4], [0, 0, 0, 4, 4], [0, 2, 0, 4, 4], [0, 3, 2, 4, 4], [0, 4, 4, 4, 4]]
	(14, 18, 7, 10, 10, 19, 14, 13, 22, 11, 9, 11, 6, 17, 7, 9, 19, 19, 21, 3, 4, 19, 16, 17, 4, 2, 17, 10, 0, 21)	**[[5, 5, 0, 0, 4], [1, 5, 4, 4, 4], [0, 0, 0, 4, 4], [0, 2, 0, 4, 4], [0, 0, 2, 4, 4], [4, 5, 2, 4, 4], [0, 2, 4, 4, 4], [0, 1, 4, 4, 4], [5, 5, 4, 4, 4], [0, 0, 3, 4, 4], [0, 0, 1, 4, 4], [0, 2, 1, 4, 4], [0, 0, 0, 4, 4], [4, 5, 0, 4, 4], [0, 0, 0, 4, 4], [0, 1, 0, 4, 4], [5, 5, 1, 4, 4], [2, 5, 4, 4, 4], [4, 5, 4, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [5, 5, 1, 4, 4], [0, 4, 4, 4, 4], [0, 5, 4, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [5, 4, 0, 4, 4], [0, 0, 2, 4, 4], [0, 0, 0, 4, 4], [5, 5, 3, 4, 4]]**
Random numbers between 0 and 22 longer time span	(15, 21, 20, 9, 10, 0, 10, 14, 15, 16, 12, 20, 0, 16, 22, 1, 21, 1, 10, 3, 12, 9, 21, 12, 8, 18, 0, 2, 22, 2, 20, 10, 1, 16, 14, 20, 21, 4, 21, 19, 8, 20, 10, 5, 9, 20, 4, 18, 17, 4)	[[5, 5, 0, 1, 4], [4, 5, 4, 4, 4], [3, 5, 4, 4, 4], [0, 0, 1, 4, 4], [0, 1, 1, 4, 4], [0, 0, 0, 4, 4], [0, 2, 0, 4, 4], [0, 4, 2, 4, 4], [0, 3, 4, 4, 4], [0, 4, 4, 4, 4], [0, 0, 4, 4, 4], [3, 5, 4, 4, 4], [0, 0, 0, 4, 4], [3, 5, 0, 4, 4], [5, 5, 4, 4, 4], [0, 0, 0, 4, 4], [5, 5, 3, 4, 4], [0, 0, 0, 4, 4], [0, 2, 0, 4, 4], [0, 0, 0, 4, 4], [0, 4, 0, 4, 4], [0, 0, 1, 4, 4], [5, 5, 3, 4, 4], [0, 0, 4, 4, 4], [0, 0, 0, 4, 4], [5, 5, 0, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [5, 5, 4, 4, 4], [0, 0, 0, 4, 4], [5, 5, 2, 4, 4], [0, 0, 2, 4, 4], [0, 0, 0, 4, 4], [3, 5, 0, 4, 4], [0, 2, 4, 4, 4], [3, 5, 4, 4, 4], [4, 5, 4, 4, 4], [0, 0, 0, 4, 4], [5, 5, 3, 4, 4], [2, 5, 4, 4, 4], [0, 0, 0, 4, 4], [5, 5, 2, 4, 4], [0, 0, 2, 4, 4], [0, 0, 0, 4, 4], [0, 1, 0, 4, 4], [5, 5, 2, 4, 4], [0, 0, 0, 4, 4], [5, 5, 0, 4, 4], [0, 5, 4, 4, 4], [0, 0, 0, 4, 4]]
	(19, 4, 4, 10, 20, 7, 0, 20, 5, 14, 21, 10, 12, 7, 2, 0, 4, 16, 3, 21, 17, 12, 17, 2, 8, 7, 7, 11, 17, 18, 13, 21, 1, 10, 3, 22, 17, 9, 6, 3, 8, 18, 20, 14, 7, 2, 3, 17, 12, 4)	**[[5, 5, 1, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [1, 1, 0, 4, 4], [5, 5, 2, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [5, 5, 2, 4, 4], [0, 0, 0, 4, 4], [1, 5, 0, 4, 4], [4, 5, 4, 4, 4], [0, 0, 2, 4, 4], [0, 2, 2, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [5, 3, 0, 4, 4], [0, 0, 0, 4, 4], [5, 5, 3, 4, 4], [0, 5, 4, 4, 4], [0, 0, 4, 4, 4], [0, 5, 4, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [0, 3, 0, 4, 4], [1, 5, 3, 4, 4], [1, 5, 4, 4, 4], [0, 1, 4, 4, 4], [4, 5, 4, 4, 4], [0, 0, 0, 4, 4], [0, 2, 0, 4, 4], [0, 0, 0, 4, 4], [5, 5, 4, 4, 4], [0, 5, 4, 4, 4], [0, 0, 1, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [5, 5, 0, 4, 4], [3, 5, 4, 4, 4], [0, 2, 4, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [5, 4, 0, 4, 4], [0, 0, 4, 4, 4], [0, 0, 0, 4, 4]]**
Uniform	(5,)*30	[[0, 5, 0, 0, 0], [0, 0, 0, 1, 4], [0, 0, 0, 1, 4], ... , [0, 0, 0, 1, 4]]
	(2,)*30	**[[0, 2, 0, 0, 0], [0, 0, 0, 0, 2], [0, 0, 0, 0, 2], ... , [0, 0, 0, 0, 2]]**
	(9,)*30	[[4, 5, 0, 0, 0], [0, 0, 1, 4, 4], [0, 0, 1, 4, 4], ... , [0, 0, 1, 4, 4]]
	(10,)*30	**[[5, 5, 0, 0, 0], [0, 0, 2, 4, 4], [0, 0, 2, 4, 4], ... , [0, 0, 2, 4, 4]]**
	(15,)*30	[[5, 5, 0, 1, 4], [0, 3, 4, 4, 4], [0, 3, 4, 4, 4], ... , [0, 3, 4, 4, 4]]
Uniform	(11,)*30	**[[5, 5, 0, 0, 1], [0, 0, 3, 4, 4], [0, 0, 3, 4, 4], ... , [0, 0, 3, 4, 4]]**
Alternating max and min number jobs	(0,22,)*15	[[0, 0, 0, 0, 0], [5, 5, 4, 4, 4], [0, 0, 0, 4, 4], [5, 5, 4, 4, 4], [0, 0, 0, 4, 4], [5, 5, 4, 4, 4], [0, 0, 0, 4, 4], ... , [0, 0, 0, 4, 4], [5, 5, 4, 4, 4]]
Random numbers between 0 and 22	(9, 10, 19, 15, 15, 15, 13, 8, 16, 11, 7, 2, 20, 7, 1, 20, 4, 19, 6, 10, 9, 16, 4, 16, 7, 7, 1, 8, 15, 20)	**[[4, 5, 0, 0, 0], [0, 1, 1, 4, 4], [4, 5, 2, 4, 4], [0, 3, 4, 4, 4], [0, 3, 4, 4, 4], [0, 3, 4, 4, 4], [0, 1, 4, 4, 4], [0, 0, 0, 4, 4], [3, 5, 0, 4, 4], [0, 0, 3, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [5, 5, 2, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [5, 5, 2, 4, 4], [0, 0, 0, 4, 4], [5, 5, 1, 4, 4], [0, 0, 0, 4, 4], [0, 2, 0, 4, 4], [0, 0, 1, 4, 4], [2, 5, 1, 4, 4], [0, 0, 0, 4, 4], [3, 5, 0, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [2, 5, 0, 4, 4], [3, 5, 4, 4, 4]]**
Random numbers between 6 and 16	(7, 9, 15, 15, 12, 14, 10, 7, 9, 15, 9, 9, 7, 9, 9, 13, 14, 15, 13, 10, 8, 10, 14, 14, 6, 13, 15, 11, 11, 7)	[[2, 5, 0, 0, 0], [0, 2, 0, 3, 4], [1, 5, 1, 4, 4], [0, 3, 4, 4, 4], [0, 0, 4, 4, 4], [0, 2, 4, 4, 4], [0, 0, 2, 4, 4], [0, 0, 0, 4, 4], [0, 1, 0, 4, 4], [1, 5, 1, 4, 4], [0, 0, 1, 4, 4], [0, 0, 1, 4, 4], [0, 0, 0, 4, 4], [0, 1, 0, 4, 4], [0, 0, 1, 4, 4], [0, 4, 1, 4, 4], [0, 2, 4, 4, 4], [0, 3, 4, 4, 4], [0, 1, 4, 4, 4], [0, 0, 2, 4, 4], [0, 0, 0, 4, 4], [0, 2, 0, 4, 4], [0, 4, 2, 4, 4], [0, 2, 4, 4, 4], [0, 0, 0, 4, 4], [0, 5, 0, 4, 4], [0, 3, 4, 4, 4], [0, 0, 3, 4, 4], [0, 0, 3, 4, 4], [0, 0, 0, 4, 4]]
Lower numbers left of zeros higher numbers right of zeros	(6, 11, 5, 7, 9, 13, 11, 13, 11, 9, 10, 7, 0, 0, 0, 0, 19, 13, 15, 20, 13, 21, 15, 17, 17, 18, 16, 20, 14)	**[[1, 5, 0, 0, 0], [0, 5, 0, 2, 4], [0, 0, 0, 2, 4], [0, 0, 0, 3, 4], [0, 2, 0, 3, 4], [0, 4, 1, 4, 4], [0, 0, 3, 4, 4], [0, 2, 3, 4, 4], [0, 0, 3, 4, 4], [0, 0, 1, 4, 4], [0, 1, 1, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [5, 5, 1, 4, 4], [0, 1, 4, 4, 4], [0, 3, 4, 4, 4], [3, 5, 4, 4, 4], [0, 1, 4, 4, 4], [4, 5, 4, 4, 4], [0, 3, 4, 4, 4], [0, 5, 4, 4, 4], [0, 5, 4, 4, 4], [1, 5, 4, 4, 4], [0, 4, 4, 4, 4], [3, 5, 4, 4, 4], [0, 2, 4, 4, 4]]**
Lower numbers left then higher numbers then zeros	(6, 11, 5, 7, 9, 13, 11, 13, 11, 9, 10, 7, 19, 13, 15, 20, 13, 21, 15, 17, 17, 18, 16, 20, 14, 0, 0, 0, 0, 0)	[[1, 5, 0, 0, 0], [0, 5, 0, 2, 4], [0, 0, 0, 2, 4], [0, 0, 0, 3, 4], [0, 2, 0, 3, 4], [0, 4, 1, 4, 4], [0, 0, 3, 4, 4], [0, 2, 3, 4, 4], [0, 0, 3, 4, 4], [0, 0, 1, 4, 4], [0, 1, 1, 4, 4], [0, 0, 0, 4, 4], [5, 5, 1, 4, 4], [0, 1, 4, 4, 4], [0, 3, 4, 4, 4], [3, 5, 4, 4, 4], [0, 1, 4, 4, 4], [4, 5, 4, 4, 4], [0, 3, 4, 4, 4], [0, 5, 4, 4, 4], [0, 5, 4, 4, 4], [1, 5, 4, 4, 4], [0, 4, 4, 4, 4], [3, 5, 4, 4, 4], [0, 2, 4, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4]]
Higher numbers left of zeros lower numbers right of zeros	(19, 13, 15, 20, 13, 21, 15, 17, 17, 18, 16, 20, 14, 0, 0, 0, 0, 0, 6, 11, 5, 7, 9, 13, 11, 13, 11, 9, 10, 7)	**[[5, 5, 1, 4, 4], [0, 1, 4, 4, 4], [0, 3, 4, 4, 4], [3, 5, 4, 4, 4], [0, 1, 4, 4, 4], [4, 5, 4, 4, 4], [0, 3, 4, 4, 4], [0, 5, 4, 4, 4], [0, 5, 4, 4, 4], [1, 5, 4, 4, 4], [0, 4, 4, 4, 4], [3, 5, 4, 4, 4], [0, 2, 4, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [0, 3, 0, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [0, 1, 0, 4, 4], [0, 4, 1, 4, 4], [0, 0, 3, 4, 4], [0, 2, 3, 4, 4], [0, 0, 3, 4, 4], [0, 0, 1, 4, 4], [0, 1, 1, 4, 4], [0, 0, 0, 4, 4]]**
Higher numbers left then lower numbers then zeros	(19, 13, 15, 20, 13, 21, 15, 17, 17, 18, 16, 20, 14, 6, 11, 5, 7, 9, 13, 11, 13, 11, 9, 10, 7, 0, 0, 0, 0, 0)	[[5, 5, 1, 4, 4], [0, 1, 4, 4, 4], [0, 3, 4, 4, 4], [3, 5, 4, 4, 4], [0, 1, 4, 4, 4], [4, 5, 4, 4, 4], [0, 3, 4, 4, 4], [0, 5, 4, 4, 4], [0, 5, 4, 4, 4], [1, 5, 4, 4, 4], [0, 4, 4, 4, 4], [3, 5, 4, 4, 4], [0, 2, 4, 4, 4], [0, 0, 0, 4, 4], [0, 3, 0, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [0, 1, 0, 4, 4], [0, 4, 1, 4, 4], [0, 0, 3, 4, 4], [0, 2, 3, 4, 4], [0, 0, 3, 4, 4], [0, 0, 1, 4, 4], [0, 1, 1, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4]]
Zeros then higher numbers then lower numbers	(0, 0, 0, 0, 0, 19, 13, 15, 20, 13, 21, 15, 17, 17, 18, 16, 20, 14, 6, 11, 5, 7, 9, 13, 11, 13, 11, 9, 10, 7)	**[[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [5, 5, 1, 4, 4], [0, 1, 4, 4, 4], [0, 3, 4, 4, 4], [3, 5, 4, 4, 4], [0, 1, 4, 4, 4], [4, 5, 4, 4, 4], [0, 3, 4, 4, 4], [0, 5, 4, 4, 4], [0, 5, 4, 4, 4], [1, 5, 4, 4, 4], [0, 4, 4, 4, 4], [3, 5, 4, 4, 4], [0, 2, 4, 4, 4], [0, 0, 0, 4, 4], [0, 3, 0, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [0, 1, 0, 4, 4], [0, 4, 1, 4, 4], [0, 0, 3, 4, 4], [0, 2, 3, 4, 4], [0, 0, 3, 4, 4], [0, 0, 1, 4, 4], [0, 1, 1, 4, 4], [0, 0, 0, 4, 4]]**
Zeros then lower numbers then higher numbers	(0, 0, 0, 0, 0, 6, 11, 5, 7, 9, 13, 11, 13, 11, 9, 10, 7, 19, 13, 15, 20, 13, 21, 15, 17, 17, 18, 16, 20, 14)	[[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [1, 5, 0, 0, 0], [0, 5, 0, 2, 4], [0, 0, 0, 2, 4], [0, 0, 0, 3, 4], [0, 2, 0, 3, 4], [0, 4, 1, 4, 4], [0, 0, 3, 4, 4], [0, 2, 3, 4, 4], [0, 0, 3, 4, 4], [0, 0, 1, 4, 4], [0, 1, 1, 4, 4], [0, 0, 0, 4, 4], [5, 5, 1, 4, 4], [0, 1, 4, 4, 4], [0, 3, 4, 4, 4], [3, 5, 4, 4, 4], [0, 1, 4, 4, 4], [4, 5, 4, 4, 4], [0, 3, 4, 4, 4], [0, 5, 4, 4, 4], [0, 5, 4, 4, 4], [1, 5, 4, 4, 4], [0, 4, 4, 4, 4], [3, 5, 4, 4, 4], [0, 2, 4, 4, 4]]
First half all jobs then no jobs arriving	(22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)	**[[5, 5, 4, 4, 4], [5, 5, 4, 4, 4], [5, 5, 4, 4, 4], [5, 5, 4, 4, 4], [5, 5, 4, 4, 4], [5, 5, 4, 4, 4], [5, 5, 4, 4, 4], [5, 5, 4, 4, 4], [5, 5, 4, 4, 4], [5, 5, 4, 4, 4], [5, 5, 4, 4, 4], [5, 5, 4, 4, 4], [5, 5, 4, 4, 4], [5, 5, 4, 4, 4], [5, 5, 4, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4]]**
Alternating max number jobs and zeros	(22, 22, 22, 22, 22, 0, 0, 0, 0, 0, 22, 22, 22, 22, 22, 0, 0, 0, 0, 0, 22, 22, 22, 22, 22, 0, 0, 0, 0, 0)	[[5, 5, 4, 4, 4], [5, 5, 4, 4, 4], [5, 5, 4, 4, 4], [5, 5, 4, 4, 4], [5, 5, 4, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [5, 5, 4, 4, 4], [5, 5, 4, 4, 4], [5, 5, 4, 4, 4], [5, 5, 4, 4, 4], [5, 5, 4, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [5, 5, 4, 4, 4], [5, 5, 4, 4, 4], [5, 5, 4, 4, 4], [5, 5, 4, 4, 4], [5, 5, 4, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4]]
Alternating max number jobs and zeros	(22, 22, 22, 22, 22, 22, 22, 22, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 22, 22, 22, 22, 22, 22, 22, 22)	**[[5, 5, 4, 4, 4], [5, 5, 4, 4, 4], [5, 5, 4, 4, 4], [5, 5, 4, 4, 4], [5, 5, 4, 4, 4], [5, 5, 4, 4, 4], [5, 5, 4, 4, 4], [5, 5, 4, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [0, 0, 0, 4, 4], [5, 5, 4, 4, 4], [5, 5, 4, 4, 4], [5, 5, 4, 4, 4], [5, 5, 4, 4, 4], [5, 5, 4, 4, 4], [5, 5, 4, 4, 4], [5, 5, 4, 4, 4], [5, 5, 4, 4, 4]]**
No jobs first half, all jobs second half	(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22)	[[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [5, 5, 4, 4, 4], [5, 5, 4, 4, 4], [5, 5, 4, 4, 4], [5, 5, 4, 4, 4], [5, 5, 4, 4, 4], [5, 5, 4, 4, 4], [5, 5, 4, 4, 4], [5, 5, 4, 4, 4], [5, 5, 4, 4, 4], [5, 5, 4, 4, 4], [5, 5, 4, 4, 4], [5, 5, 4, 4, 4], [5, 5, 4, 4, 4], [5, 5, 4, 4, 4], [5, 5, 4, 4, 4]]

Figure 8: Result of simulations only the schedules first half of simulations

Structure of input	Input Lambda	Total jobs arrived	Average job arrival per time slot	Cost paper schedule	Cost optimal schedule	Run time in seconds	Run time in minutes	Cost increase in percentage
Random numbers between 0 and 22	(21, 17, 16, 0, 20, 11, 12, 20, 22, 6, 12, 21, 18, 4, 3, 16, 16, 12, 2, 17, 5, 8, 22, 17, 3, 16, 6, 10, 13, 16)	382	12,7	1313	1274	1.011	16,85	3,06
	(14, 18, 7, 10, 10, 19, 14, 13, 22, 11, 9, 11, 6, 17, 7, 9, 19, 19, 21, 3, 4, 19, 16, 17, 4, 2, 17, 10, 0, 21)	369	12,3	1259	1196	1.470	24,50	5,27
Random numbers between 0 and 22 longer time span	(15, 21, 20, 9, 10, 0, 10, 14, 15, 16, 12, 20, 0, 16, 22, 1, 21, 1, 10, 3, 12, 9, 21, 12, 8, 18, 0, 2, 22, 2, 20, 10, 1, 16, 14, 20, 21, 4, 21, 19, 8, 20, 10, 5, 9, 20, 4, 18, 17, 4)	603	12,06	2159	2064	3.565	59,41	4,60
	(19, 4, 4, 10, 20, 7, 0, 20, 5, 14, 21, 10, 12, 7, 2, 0, 4, 16, 3, 21, 17, 12, 17, 2, 8, 7, 7, 11, 17, 18, 13, 21, 1, 10, 3, 22, 17, 9, 6, 3, 8, 18, 20, 14, 7, 2, 3, 17, 12, 4)			1768	1711	6.481	108,02	3,33
Uniform	(5,)*30	150	5	233	209	6.496	108,27	11,48
	(2,)*30	60	2	82	72	6.857	114,28	13,89
	(9,)*30	270	9	537	498	3.642	60,70	7,83
	(10,)*30	300	10	634	592	6.895	114,92	7,09
	(15,)*30	450		1170	1146	151	2,52	2,09
Uniform	(11,)*30	330	11	726	686	1.932	32,21	5,83
Alternating max and min number jobs	(0,22,)*15	330	11	1712	1656	2	0,03	3,38
Random numbers between 0 and 22	(9, 10, 19, 15, 15, 15, 13, 8, 16, 11, 7, 2, 20, 7, 1, 20, 4, 19, 6, 10, 9, 16, 4, 16, 7, 7, 1, 8, 15, 20)	330	11	1117	1049	1.633	27,22	6,48
Random numbers between 6 and 16	(7, 9, 15, 15, 12, 14, 10, 7, 9, 15, 9, 9, 7, 9, 9, 13, 14, 15, 13, 10, 8, 10, 14, 14, 6, 13, 15, 11, 11, 7)	330	11	892	815	2.208	36,80	9,45
Lower numbers left of zeros higher numbers right of zeros	(6, 11, 5, 7, 9, 13, 11, 13, 11, 9, 10, 7, 0, 0, 0, 0, 0, 19, 13, 15, 20, 13, 21, 15, 17, 17, 18, 16, 20, 14)	330	11	1021	961	3.366	56,10	6,24
Lower numbers left then higher numbers then zeros	(6, 11, 5, 7, 9, 13, 11, 13, 11, 9, 10, 7, 19, 13, 15, 20, 13, 21, 15, 17, 17, 18, 16, 20, 14, 0, 0, 0, 0, 0)	330	11	1021	921	2.233	37,22	10,86
Higher numbers left of zeros lower numbers right of zeros	(19, 13, 15, 20, 13, 21, 15, 17, 17, 18, 16, 20, 14, 0, 0, 0, 0, 0, 6, 11, 5, 7, 9, 13, 11, 13, 11, 9, 10, 7)	330	11	1003	961	1.788	29,81	4,37
Higher numbers left then lower numbers then zeros	(19, 13, 15, 20, 13, 21, 15, 17, 17, 18, 16, 20, 14, 6, 11, 5, 7, 9, 13, 11, 13, 11, 9, 10, 7, 0, 0, 0, 0, 0)	330	11	1003	925	1.305	21,75	8,43
Zeros then higher numbers then lower numbers	(0, 0, 0, 0, 0, 19, 13, 15, 20, 13, 21, 15, 17, 17, 18, 16, 20, 14, 6, 11, 5, 7, 9, 13, 11, 13, 11, 9, 10, 7)	330	11	943	925	2.438	40,64	1,95
Zeros then lower numbers then higher numbers	(0, 0, 0, 0, 0, 6, 11, 5, 7, 9, 13, 11, 13, 11, 9, 10, 7, 19, 13, 15, 20, 13, 21, 15, 17, 17, 18, 16, 20, 14)	330	11	961	921	3.879	64,65	4,34
First half all jobs then no jobs arriving	(22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)	330	11	1290	1110	1.682	28,03	16,22
Alternating max number jobs and zeros	(22, 22, 22, 22, 22, 0, 0, 0, 0, 0, 22, 22, 22, 22, 22, 0, 0, 0, 0, 0, 22, 22, 22, 22, 22, 0, 0, 0, 0, 0)	330	11	1352	1252	2.448	40,80	7,99
Alternating max number jobs and zeros	(22, 22, 22, 22, 22, 22, 22, 22, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 22, 22, 22, 22, 22, 22, 22, 22)	330	11	1321	1185	3.275	54,58	11,48
No jobs first half, all jobs second half	(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22)	330	11	1110	1110	4.999	83,32	0,00
No jobs first half, half job capacity second half	**(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11)**	**165**	**5,5**	**411**	**371**	**5.594**	**93,24**	**10,78**
Alternating low and high arrivals	(5,18,)*15	345	11,5	1312	1298	174	2,90	1,08
Alternating high and low arrivals	(18,5,)*15	345	11,5	1304	1298	175	2,92	0,46
Alternating low and high arrivals	(10,15,)*15	375	12,5	1074	1042	689	11,48	3,07
	(9,16,)*15	375	12,5	1155	1128	475	7,92	2,39
	(8,17,)*15	375	12,5	1237	1214	302	5,03	1,89
Alternating high and low arrivals	(17,8,)*15	375	12,5	1218	1214	301	5,02	0,33
	(19,7,)*15	375	12,5	1409	1393	71	1,19	1,15
Alternating low and high arrivals	(7,19,)*15	375	12,5	1427	1393	72	1,20	2,44
Longer time span alternating high and low arrivals	(19,7,)*30	390	13	2774	2743	293	4,88	1,13
	(19,7,)*50	1300	13	4594	4543	822	13,71	1,12
	(19,7,)*100	2600	13	9144	9043	3.191	53,19	1,12

FIGURE 9: Results of simulations without schedules

Appendix Python code

```
import random
import itertools
import time
import numpy as np
import math
from functools import cache

@cache
def ipossibilities(m,Lambda):
    Machineslist = [list(range(i + 1)) for i in m]
    Product = itertools.product(*Machineslist)
    filtered = filter(lambda L: sum(L) >= Lambda, Product)
    listi=[i for i in filtered]
    return listi

@cache
def xpossibilities(m,Lambda):
    Machineslist = [list(range(i + 1)) for i in m]
    Product = itertools.product(*Machineslist)
    filtered = filter(lambda L: sum(L) == Lambda, Product)
    listx=[x for x in filtered]
    return listx

@cache
def cost_at_time_t(d,x,i,l,beta):
    xi = np.array(x) - np.array(i)  # difference xi
    maxxi=[max(0,xi[f]) for f in range(d)]
    cost = np.array(l).transpose()@x + np.array(beta).transpose()@maxxi
    return cost

@cache
def sumparttuple(tuple,startsum,d):
    # initializing count
    count = 0
    for r in range(d-1,startsum-1,-1):
        # for loop
        i =tuple[r]
        count += i
    return count

def ytypeserverlanek(t,k,oschedule,d): #list of tuples
    if k in range(sum(oschedule[t-1])+1):
        j=d-1
        while j in range(d-1,-1,-1):
            if sumparttuple(oschedule[t-1],j,d)>=k:
                return j+1
            else:
                j=j-1
```

```
    else:
        return 0

def costpaperschedule(T,d,l,beta,schedule):
    cost = 0
    i=[0]*d
    x=schedule
    for t in range(T):
        xi = np.array(x[t]) - i  # difference xi
        maxxi=[max(0,xi[f]) for f in range(d)]
        cost = cost + np.array(l)@np.array(x[t]) + np.array(beta)@maxxi
        i=np.array(x[t])
    return cost


@cache
def paperschedule(T, d, m, beta, l, Lambda):
    st=time.time()
    tbar = np.divide(beta, l)
    tbarfinal = [0]
    for i in tbar:
        final = math.floor(i)
        tbarfinal.append(final)
    ypaper = {}            #make dictionary of sums because always same
    xpaper = {}
    xpaperfinal = []
    mtot=sum(m)
    e = np.zeros((mtot,), dtype=int)
    for t in range(1,T+1):
        #calculate optimal schedule for problem instance ending at time t
        oschedule =optschedule2(t, d, m, l, beta, Lambda) #=[] list of tuples
        yp = {}
        xp = []
        xpfinal = []
        for k in range(1,mtot+1):
            ytkopt=ytypeserverlanek(t,k,oschedule,d)
            if t==1:
                if t>=e[k-1] or 0 < ytkopt:
                    yp[k] = ytkopt
                    e[k-1] = t + tbarfinal[yp[k]]
                    xp.append(yp[k])
                else:
                    yp[k] = 0
                    e[k-1] = max(e[k], t + tbarfinal[yp[k]])
                    xp.append(yp[k])
                ypaper[t]=yp
            else:
                if t>=e[k-1] or ypaper[t-1][k] <ytkopt:
                    yp[k] = ytkopt
                    e[k-1] = t + tbarfinal[yp[k]]
```

```
                xp.append(yp[k])
            else:
                yp[k] = ypaper[t-1][k]
                e[k-1] = max(e[k-1], t + tbarfinal[yp[k]])
                xp.append(yp[k])
            ypaper[t] = yp
        xpfinal = [xp.count(j) for j in range(1, d + 1)]
        xpaperfinal.append(xpfinal)
    i = (0,) * d
    xpaperfinal.append(i)
    et=time.time()
    alltime=et-st
    return xpaperfinal, "cost␣paper␣schedule␣is",
    costpaperschedule(t,d,l,beta,xpaperfinal),
    "cost␣optimal␣schedule␣is",costpaperschedule(t,d,l,beta,oschedule),
    alltime


@cache
def fmin2(t,d,m,l,beta,Lambda): #only for t=T
    fdict={} # keys=i values=fmin(t,i)
             fdict[0 ... 0] will give the value of the schedule
    xdict={} # keys=i values=x for which
             xdict will give the schedule
    # calculate fmin(t,i) then fdict[i]=fmin(t,i)
    fdictpure={}
    nesteddictf={} # to collect all ft(i) per time step key=n
                   is time value=fdict
    nesteddictx={}
    xdictnu = {}
    n = t
    if t == 1:
        listx = xpossibilities(m, Lambda[n - 1])
#filter with paper restriction yhat_t>=yhat_t-1
        i = (0,) * d
        xdictnu = {}
        fdict = {}
        xdict = {}
        for x in listx:  # set(x) is here set(i) at
            fnietmin = cost_at_time_t(d, x, i, l, beta)
            xdictnu[x] = fnietmin
        xminvalue = min(xdictnu, key=lambda k: xdictnu[k])
# the key whose value is the smallest is xminvalue
        fminvalue = xdictnu[xminvalue]  # gives the smallest value
        temp = min(xdictnu.values())
        res = [key for key in xdictnu if xdictnu[key] == temp]
        for key in res:
            fdict[key] = xdictnu[key]
        xdict[i] = fdict  # dictionary with key=i and value=minx,
                          we can find here the x that belongs to fmin
```

```
            fdictpure[i] = fminvalue
            nesteddictf[n] = fdictpure
            nesteddictx[n] = xdict  # only after all i have been done
            return nesteddictx, nesteddictf
        else:
            listi = ipossibilities(m, Lambda[n - 2])
            listx = xpossibilities(m, Lambda[n -1])
            fdictpure={}
            for i in listi:  # list with available i
                for x in listx:
                    fdict = {}
                    fnietmin = cost_at_time_t(d,x, i, l, beta)
# for t=T
                    xdictnu[x] = fnietmin
                xminvalue = min(xdictnu, key=lambda k: xdictnu[k])
# the key whose value is the smallest is xminvalue
                fminvalue = xdictnu[xminvalue]  # gives the smallest value
                temp = min(xdictnu.values())
                res = [key for key in xdictnu if xdictnu[key] == temp]
                for key in res:
                    fdict[key] = xdictnu[key]
                xdict[i] = fdict  # dictionary with key=i and value=minx,
                                    we can find here the x that belongs to fmin
                fdictpure[i] = fminvalue
                nesteddictf[n]=fdictpure
            nesteddictx[n]=xdict # only after all i have been done

            n=n-1
            listinew = ipossibilities(m, Lambda[n - 2])
            while n > 1:
                fdict = {}
                xdict = {}
                fdictpure = {}
                listinew = ipossibilities(m, Lambda[n - 2])
                for i in listinew:
                    xdictnu = {}
                    for x in listi: #set(x) is here set(i) at n==t
                        fdict = {}
                        fnietmin = cost_at_time_t(d,x,i,l,beta) +
                                    nesteddictf[n+1][x]
                        xdictnu[x] = fnietmin
                    xminvalue = min(xdictnu, key=lambda k: xdictnu[k])
# the key whose value is the smallest is xminvalue
                    fminvalue = xdictnu[xminvalue] #gives the smallest value
                    temp = min(xdictnu.values())
                    res = [key for key in xdictnu if xdictnu[key] == temp]
                    for key in res:
                        fdict[key] = xdictnu[key]
                    xdict[i] = fdict  # dictionary with key=i and value=minx,
                                        we can find here the x that belongs to fmin
```

```
                    fdictpure[i] = fminvalue
                    nesteddictf[n] = fdictpure
                nesteddictx[n] = xdict  # only after all i have been done
                listi = listinew
                n = n - 1
            n=1

            i=(0,)*d
            xdictnu = {}
            fdict={}
            xdict={}
            fdictpure={}
            k = 1
            for x in listi:  # set(x) is here set(i) at
                fnietmin = cost_at_time_t(d,x, i, l, beta) + nesteddictf[n+1][x]
# for t
                xdictnu[x] = fnietmin
            xminvalue = min(xdictnu, key=lambda k: xdictnu[k])
# the key whose value is the smallest is xminvalue
            fminvalue = xdictnu[xminvalue]  # gives the smallest value
            temp = min(xdictnu.values())
            res = [key for key in xdictnu if xdictnu[key] == temp]
            for key in res:
                fdict[key] = xdictnu[key]
            xdict[i] = fdict  # dictionary with key=i and value=minx,
                                 we can find here the x that belongs to fmin
            fdictpure[i] = fminvalue
            nesteddictf[n] = fdictpure
            nesteddictx[n] = xdict  # only after all i have been done

    return (nesteddictx,nesteddictf)

@cache
def ytypeserverlanek2(t,k,oschedule,d): #list of tuples
    if k in range(sum(oschedule)+1):
        j=d-1
        while j in range(d-1,-1,-1):
            if sumparttuple(oschedule,j,d)>=k:
                return j+1
            else:
                j=j-1
    else:
        return 0

@cache
def optschedule2(t,d,m,l,beta,Lambda): #list with tuples
    summ=sum(m)
    xtj=(0,)*d #tuple of d zeros
    xzero=(0,)*d
    nesteddictx,nesteddictf=fmin2(t,d,m,l,beta,Lambda)
```

```
    schedule=[]
    for n in range(1,t+1):
        if t==1:
            xtjdict=nesteddictx[n][xtj] # is {(0, 0, 1): 34} or
                                        {(0, 1, 1): 27, (1, 0, 1): 27}
            res = [key for key in xtjdict if xtjdict[key] ==
                   nesteddictf[n][xtj]]  #oschedule is a tuple
            xtj = res[0] # we just choose the first x
            schedule.append(res[0]) # for t=1 it does not matter x will be
                        greater or equal to 0 always
        else:
            xtjdict = nesteddictx[n][xtj]  # is {(0, 0, 1): 34} or
                                            {(0, 1, 1): 27, (1, 0, 1): 27}
            res = [key for key in xtjdict if xtjdict[key] ==
                    nesteddictf[n][xtj]]   #oschedule is a tuple
            if len(xtjdict) == 1:
                schedule.append(res[0])
                xtj=res[0]
            else:
                v=1
                while v<=t:
                    x=res[v-1]
                    k=1
                    while k<=summ:
                        if k==summ and ytypeserverlanek2(t, k, x, d) >=
                            ytypeserverlanek2(t,k,
                            optschedule2(t-1,d,m,l,beta,Lambda)[n-1],d):
                            schedule.append(x)
                            xtj=x
                            k=summ+1
                            v=t+1
                        elif ytypeserverlanek2(t, k, x, d) >=
                            ytypeserverlanek2(t,k,
                            optschedule2(t-1,d,m,l,beta,Lambda)[n-1],d)  :
                            k=k+1
                        else:
                            k=summ+1
                            v=v+1

    i = (0,) * d
    schedule.append(i)
    costopt=nesteddictf[1][xzero]
    return schedule



if __name__ == "__main__":
    m = ( 5, 5, 4, 4,4)
    l=(5,4,3,2,1)
    beta=(1,2,4,5,6)
```

```
Lambda=(7,9,5,3,4)

##random numbers##
# num = 50
# start = 0
# end = 22
# res = []
# for j in range(num):
#       res.append(random.randint(start, end))
# print(res)
# Lambda = tuple(res)

###### random numbers sum i###
# i=330
# nums=[]
# total = i
# temp = []
# for i in range(29):
#       val = np.random.randint(6, 16)
#       temp.append(val)
#       total -= val
# temp.append(total)
# print(temp)
# Lambda=tuple(temp)
######

d = len(m)
t = len(Lambda)

print(paperschedule(t,d,m,beta,l,Lambda))
```