Kill Smart, Run Fast: Using Job Termination for Resource Efficiency in Data Centers

Adityo Anggraito
adityo.anggraito@unive.it
University 'Ca' Foscari' of Venice
Dep. of Environmental Sciences, Informatics and Statistics
Venice, Italy

Rostislav Razumchik
rrazumchik@frccsc.ru
Federal Research Center "Computer Science and Control" of the Russian Academy of Sciences
Moscow, Russia

Andrea Marin
marin@unive.it
University 'Ca' Foscari' of Venice
Dep. of Environmental Sciences, Informatics and Statistics
Venice, Italy

Abstract

For the last few years, job scheduling in large data centers has been attracting many research interests, especially because of the larger scales of these infrastructures and the drastic change in the workload characteristics introduced by AI jobs. The high variability in the resource demands (in terms of memory, number of cores, GPUs, etc.) makes the scheduling design extremely complicated, and the maximum achievable throughput is limited with respect to the amount of available resources. This is due to the head-of-line blocking (HOL), i.e., the scheduler has to wait for the termination of some jobs to make space for a job demanding many resources. In this paper, we study a killing policy that serves the jobs in a nearly First-Come First-Served (FCFS) order, i.e., the scheduler preempts the jobs in service to make space for a large job. In order to make the analysis realistic, we assume that the preempted jobs must be restarted once they enter again in service. Our results are summarized as follows: (i) we show that we can achieve higher throughput than standard FCFS beside the waste of computational resources due to the restart policy, (ii) we study a queueing model that allows us to understand the trade-off among throughput, fairness and energy waste in a simplified scenario, (iii) we use discrete event simulation to show that the killing policy allows for higher throughput and lower expected response times of jobs with workload characteristics derived from the traces, released by Google, of the Borg scheduler with respect to FCFS.

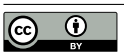
CCS Concepts

• **Computing methodologies** → *Modeling and simulation*; • **Mathematics of computing** → **Queueing theory**; • **Theory of computation** → *Scheduling algorithms.*

Keywords

Datacenter job scheduling; preemption-restart; multiserver-job queueing systems

ACM Reference Format:
Adityo Anggraito, Rostislav Razumchik, and Andrea Marin. 2026. Kill Smart, Run Fast: Using Job Termination for Resource Efficiency in Data Centers. In *Proceedings of the 17th ACM/SPEC International Conference on Performance Engineering (ICPE '26), May 04–08, 2026, Florence, Italy*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3777884.3796995



ICPE '26, Florence, Italy

ACM ISBN 979-8-4007-2325-4/2026/05
https://doi.org/10.1145/3777884.3796995

1 Introduction

Resource allocation in large-scale data centers is a crucial problem in contemporary computer infrastructures. Recent developments in artificial intelligence, such as the wide introduction of large language models, have drastically changed the workload characteristics of data centers. For example, the analysis of Alibaba traces performed in [23] reveals four important characteristics of the large-scale data center workloads: *spatial imbalance*, i.e., heterogeneous resource utilization across machines and requests, *temporal imbalance*, i.e., high differences among the jobs' service times, *imbalanced proportion of multi-dimensional resources*, and, finally, *imbalanced resource demands and runtime statistics* (duration and task number) between online service and offline batch jobs. Similar observations arise from the analysis of Google data center traces as recently reported in [7, 33].

The primary consequence of such workload characteristics is an extremely low achievable utilization of the infrastructure due to what is known under the name of *head of line blocking* (HOL): suppose a job at the head of the line of the waiting jobs requires a large amount of resources that are currently unavailable. If behind this job there is a job requiring an amount of resources that are currently available, the scheduler must face an important decision: should the smaller job overtake the large job or wait in line? In the second case, we observe HOL. While the first option seems more appealing, it can create a very important *unfairness* with respect to large jobs[9] which are, usually, those paying higher fees.

In the literature, a scheduler is called *throughput optimal* if it can approach asymptotically the maximum resource utilization while the workload intensity increases. Why is approaching this property extremely important in practice? Suppose that a data center implements a throughput-optimal scheduler that can handle a workload whose maximum intensity is λ_{ideal}. This means that no other scheduler can handle traffic with higher intensity, i.e., any other scheduler can handle only a traffic intensity $\lambda_{max} \leq \lambda_{ideal}$. In particular, if the scheduler is not throughput-optimal, then $\lambda_{max} < \lambda_{ideal}$. What can we say about the user experience, i.e., the system's average response time? Here, it is important to note that, in general, it is not true that the average response time with the throughput optimal scheduler is always not worse than that with the non-optimal one. However, queueing theory tells us that any infinite capacity system has an average response time that tends to diverge as the traffic intensity approaches its maximum,

i.e., λ_{max} or λ_{ideal}. Since $\lambda_{ideal} > \lambda_{max}$, we can conclude that there must exist $\lambda^* < \lambda_{max} < \lambda_{ideal}$ such that, for all traffic intensities higher than λ^* (i.e. in $(\lambda^*, \lambda_{max})$), the average response time with the optimal scheduler is lower than that with the non-optimal one. This reasoning can be extended in the comparison of schedulers: *If the two schedulers, say $S^{(1)}$ and $S^{(2)}$, have maximum workload intensities $\lambda^{(1)}_{max} < \lambda^{(2)}_{max}$, respectively, then, in heavy traffic (for $S^{(1)}$ scheduler), the expected response time with $S^{(2)}$ is lower than that with $S^{(1)}$.*

In order to achieve throughput optimality or good approximations, several strategies have been proposed in the literature. A family of schedulers known as *backfilling* [24] requires jobs to declare their service times, and overtaking is allowed if it does not affect the waiting time of the jobs in the head of the waiting line. This is a widely deployed strategy in nowadays data centers, and it is implemented in most practical schedulers (e.g., SLURM [34]). However, knowing the jobs' service times is not an easy request, and in many cases, it just moves the problem of the underutilization of the resources to the end-users, who now have to estimate accurately their jobs' service times.

Several works have also explored relying on system predictions instead of using user estimates. Tsafrir et al. [31] proposed a backfilling approach based on runtime predictions inferred automatically from historical execution data. While such prediction-based methods can reduce the burden on users, their practical deployment raises several challenges. In particular, inaccurate predictions—especially underestimations—may lead to premature job termination, which can be unacceptable in production environments. Moreover, these approaches typically rely on maintaining and processing historical user data, which may incur computational and memory overheads at scale. Interestingly, prior studies [27, 35] have also observed that inaccurate user estimates might even be beneficial to the system by implicitly smoothing scheduling decisions. As a result, prediction-based backfilling techniques have seen limited adoption in practice.

Preemption is another technique that can be used to approach throughput optimality. This is used, e.g., by the scheduler *ServerFilling* [15, 17]. Intuitively, the scheduler can remove running jobs whenever a better way to fill the servers is discovered, thanks to the most recent arrivals. One assumption required to reach throughput optimality is that preemption can be performed in zero time and without a *waste of computation*. This can be extremely difficult in many practical scenarios.

In general, the efficiency of schedulers has been recently studied within the framework of queueing theory, as recommended in [19], with the introduction of the multiserver-job queueing model (MJQM). In this queueing system, jobs belong to a certain class, which is associated with a deterministic demand of resources (e.g., CPUs) and a random service time. Resources are acquired and released simultaneously. A job can enter service only if all the demanded resources are available. The model is extremely useful because it captures the problem of HOL, which appears in real data centers. HOL makes the queueing model non-work conserving (see Def. 3.1) and therefore different from traditional queueing models.

Our contribution. In this work, we analytically study for the first time the preemption-restart policy for multiserver-job systems. In simple words, under certain conditions, our scheduler decides that a better occupation of the resources is possible, and hence it stops the execution of some jobs in service. These jobs are then rescheduled for service and restart from the beginning. Two aspects of this policy should be highlighted: (i) the work performed on a preempted job is lost and hence must be redone; (ii) the model resembles the truly preemption-restart policy and not the preemption-resample policy, which, in general, tends to underestimate the system's workload (see Section 3). The main result of this work is that, differently from most conventional (work-conserving) queueing models, *preemption-restart can improve the maximum throughput reachable by multiserver-job queueing models with respect to the First-Come First-Served policy at the cost of the energy wasted for the restart operations.* We mathematically prove this statement for a simplified scenario, in which we consider two classes of jobs, one requiring a single server and the other requiring all servers. But the hypothesis is also tested by simulation on the real-world dataset.

Structure of the paper. The paper is structured as follows. Section 2 discusses some related works, and in Section 3, we introduce the problem and the notation used throughout the paper. Section 4 introduces the problem in a simplified scenario and provides the analytical results in saturation. These results will be extended in Section 5, where we carry out a thoughtful simulation study to support the insights obtained with the analytical results in the simplified scenario. Finally, Section 6 concludes the paper.

2 Related Work

Works on preemption policies have a long history. One of the earliest formal treatments of preemptive service disciplines can be traced to [14]. This paper introduces analytical models for queues in which service can be interrupted and later resumed or restarted, depending on the policy. Gaver establishes foundational results for systems with priority classes and systems with *preemptive-resume* or *preemptive-repeat (restart)* disciplines. Building on this, Jaiswal [20] studied the *preemptive-resume* policy, in which preempted jobs resume service from the interruption point and the portion of service already completed is preserved. The paper derived the steady-state generating function of the queue length and the Laplace transform of the busy-period distribution.

Chang [8] extended this line of work to priority queues. In the paper, preemptive policies are classified into three disciplines, depending on how jobs are treated upon returning to service. Along with the *preemptive-resume* policy, the *preemptive-repeat* discipline has two variants. The *preemptive-repeat-identical* policy, also known as the *preemptive-restart* policy, assumes that interrupted jobs restart with the same required service duration as before interruption. The *preemptive-repeat-different* policy, or *preemptive-resample*, restarts jobs with a new service time sampled from the same distribution. Chang provided a systematic method to obtain the stationary distributions of queue size, waiting time, and busy period for all priority classes.

More recent works have provided new perspectives on preemptive policies. Roy et al. [28] showed that resetting a job with a new service time can significantly affect the average waiting time, even without priority classes. Fralix et al. [11] studied all three

disciplines—resume, resample, and restart—under a *Last-In–First-Out* (LIFO) service rule. Their work demonstrated how a natural coupling procedure can establish a recursive approach for calculating the busy period, which can then be expressed via its Laplace–Stieltjes transform.

So far, these results have considered only single-server systems. An important early survey of service-interruption models is given by [22] that reviews queueing systems subject to various types of interruption and different modeling (resume/restart). The work identifies major gaps in the literature — notably, the limited study of preemption disciplines in multiserver settings, as studied in our work. One modern baseline for preemptive scheduling in multiserver systems is the *ServerFilling* policy. Grosof et al. [15, 17] introduced ServerFilling for the multiserver-job model, where jobs require multiple servers concurrently. Unlike traditional FCFS or simple packing policies, ServerFilling seeks to keep all servers busy whenever possible and provides the first known bounds on mean response time in heavy-traffic conditions. The policy uses *preemptive-resume* method and assumes that all job classes have server requirements that are powers of two, which enables throughput-optimality under this condition.

Another closely related line of work considers scheduling policies that uses preemptive-restart mechanisms and operate without prior knowledge of job service times. Harchol-Balter [18] proposed TAGS (Task Assignment with unknown duration), a policy designed for heavy-tailed workloads in distributed systems. In TAGS, arriving jobs are dispatched to individual servers with increasing limits. If a job does not complete before the limit at a given server, it is killed and restarted at another server with a higher limit. This mechanism effectively implements a *preemptive-restart* discipline without requiring service-time information. TAGS prioritizes short jobs while allowing long jobs to progress gradually, leading to relatively uniform slowdown across different job sizes. However, TAGS may incur substantial wasted work due to repeated restarts and can suffer from underutilization of servers with higher limits, with larger jobs being disproportionately penalized.

Fei et al. [10] extended the TAGS framework in the KOALA-C resource management system, targeting integrated multicluster and multicloud environments. KOALA-C introduces two TAGS-based policies to mitigate the limitations of the original approach. TAGS-chain organizes servers into chains with increasing runtime limits, where jobs are restarted along the chain upon exceeding their current limit. To improve load distribution, jobs are assigned to chains either uniformly at random or in a round-robin fashion. Despite this, higher-limit servers within each chain may still remain underutilized. TAGS-sets further addresses this issue by grouping servers into sets: early sets contain many servers with low limits, while later sets contain fewer servers with higher limits. This design improves utilization balance while preserving the core TAGS principle.

However, while TAGS operates in a multiserver environment, each job is executed on a single server at a time, and thus the model does not consider jobs requiring multiple servers simultaneously, as in the multiserver-job settings.

3 Background

In this section, we review the main results for MJQM and introduce the notation used throughout this work.

We consider a MJQM with infinite buffer capacity and N servers. Customers arrive according to independent Poisson processes and belong to a certain class m, where $1 \le m \le M$. The arrival rate of class m is λ_m, and the aggregated arrival rate is $\lambda = \sum_{m=1}^{M} \lambda_m$. We use p_m to denote the ratio λ_m/λ, i.e., the proportion of jobs of class m. Each class requires a deterministic amount of servers n_m for a random service time with mean μ_m^{-1}. The required servers must be obtained simultaneously and are also released simultaneously. We assume the servers to be identical and with unitary speed.

Given a certain occupancy of the servers and the buffer, the scheduler decides at each instant if any change should be applied to this configuration.

A key property of schedulers that characterizes most well-known queueing systems is that they are work-conserving.

Definition 3.1 (Work-conserving scheduling policy). A scheduler is work-conserving if it does not create or destroy work and if there cannot be jobs in the waiting line while there are idle resources.

MJQM with FCFS scheduler is an example of a non work conserving scheduler. In fact, since resources must be allocated simultaneously, there are several cases where we have idle servers and jobs in the buffer that cannot take advantage of these spare resources. This phenomenon has been deeply investigated in [16, 26].

Therefore, given an arrival rate λ, the total work arriving at the system is $\lambda \sum_{m=1}^{M} p_m n_m \mu_m^{-1}$, which must be strictly lower than N. This allows us to derive the stability condition for an optimal, work-conserving scheduler:

$$\lambda < \lambda_{ideal} = \frac{N}{\sum_{m=1}^{M} p_m n_m \mu_m^{-1}} .$$

Since for MJQM, schedulers may be not work-conserving (and, indeed, most are not), the stability condition is more restrictive than $\lambda < \lambda_{ideal}$. We use λ_{max} to denote the maximum arrival rate for a MJQM. λ_{max} is known explicitly only for a few cases, even for a simple discipline like FCFS; in general, it is limited to the case of two classes only. In [26], the authors present an analysis of the stability region for two-class MJQM systems in the one-or-all scenario using the saturation technique. This work was extended in [3], which provides exact closed-form solutions for several key performance metrics. Moreover, it relaxes the one-or-all assumption by allowing large jobs to require fewer than N servers, instead specifying a server requirement, T, that is divisible by N. Further analysis was carried out in [2, 5], extending the stability analysis to non-exponential service distributions, including heavy-tailed distributions such as Pareto. Additional extensions were presented in [4], where the matrix geometric technique was applied to Cox-2 distributions, which have heavier tails than exponential distributions. This work not only provides stability analysis but also gives closed-form matrix expressions for several key system performance measures.

A scheduler is said *preemptive* if it can remove jobs from the service room and put them in the buffer before their natural completion. In the literature, we have three policies to restore the computation: (i) *resume*, i.e., the work done on the job is not wasted and once

restored the job continues as if it were not interrupted; (ii) *resample*, i.e., the service time is resampled from its distribution; (iii) *restart*, i.e., the work on the job must be restarted but it keeps the same service time that it had the first time it entered the system.

Preemption-resume requires the scheduler to save the state of the job and restore it when needed. This is assumed to be done in a negligible time so that no work power is wasted. However, for jobs in the cloud, this operation might be prohibitive [29]. *Preemption-resample* is a policy that makes the system tractable in many scenarios since the model state must account just for the number (and possibly order) of the preempted jobs in the buffer and not their service times. However, even for exponentially distributed service times, it changes the workload since the jobs that are preempted are usually the longest ones, and once they return to work, they are distributed as all other jobs. In fact, preemption-resume and preemption-resample are indistinguishable for exponentially distributed service times [32], proving that we are not accounting for the waste of work done. It is also difficult to motivate the fact that the same job entering in service twice faces two *independent* service times. *Preemption-restart* is more realistic than preemption-resample for most systems; however, it poses more difficulties in its analysis because we must be sure that once a preempted job is put back in service, its service time must not be lower than the amount of work we already did on the same job before the preemption. Therefore, the state of the system must account for this. In this work, we will focus on this latter policy implemented in the MJQM.

Hereafter, we use the usual acronyms for probabilistic analysis: random variable (r.v.), probability density function (pdf), cumulative density function (CDF), independent and identically distributed (i.i.d.).

4 Understanding the Problem via a Simplified Scenario: Analytical Results

Let us consider the MJQM with an infinite number of servers (i.e., $N = \infty$) and only two classes of jobs: *small* jobs that require only one server and *big* jobs that require all servers. Notice that, differently from common queueing systems with infinite servers, this model is not unconditionally stable because of the demand of the large class [5]. We assume independent service times, exponentially distributed for small jobs with mean μ_s^{-1}, and arbitrarily distributed with mean μ_b^{-1} for big jobs. The only assumption we impose on the arrival process is that each job independently belongs to the small class with the probability p_s, and to the large class with the probability $p_b = 1 - p_s$.

If we consider a FCFS scheduler, we can visualize two alternating phases: (1) big jobs are being served, and we serve a geometrically distributed sequence of big jobs; (2) small jobs are in service and when a new small job arrives, it either enters in service immediately if there is no big job waiting in the buffer, or it waits in the queue in FCFS order. The sequence of small jobs that we serve is also geometrically distributed.

4.1 Killing policy

Observe that the loss of computational power in FCFS is due to the HOL. In this simplified scenario, we have that the head of the waiting line is occupied by a big job followed by a small one, and there is at least one small job in service.

A naive killing policy. The easiest way to address the problem of reducing the waste of resources is to kill all small jobs in service when their number is less than or equal to a fixed integer $\nu > 0$. These jobs are sent back to the waiting line, and their service will be restarted once they are back in service. Unfortunately, this policy does not work in practice. Consider a saturation scenario with an infinite many waiting jobs and $\nu = 1$, i.e., when the system is serving small jobs, the last one is killed and it will join the next sequence of small jobs. Since all small jobs in a sequence enter in service simultaneously, the killed one is the one with the largest service time. This will enter in service after the departure of the big job that it created space for. The process repeats, but we can observe that, *according to this naive policy, in saturation, we postpone the service of the longest job indefinitely*. In practice, in heavy-load, this policy creates an unfairness against small jobs with long service times that can be preempted many times before being served.

Note that this policy with preemption/restart scenario can, in principle, be applied in other schedulers, e.g., Server-Filling. But in the latter case, some small jobs will still suffer from the same problem, i.e., excessively long delays due to many preemptions.

Our killing policy. In order to avoid the problems of the naive policy, we introduce a first rule for our policy: *never kill the same job more than once*. This avoids the scenario of extreme unfairness toward small jobs with long service times. However, this is not enough to design a smart policy. In fact, if we kill a small job, this must be the longest of its batch, and it has a high probability to be the longest also of the next batch of small jobs. As a consequence, in many sequences of small jobs, we will not preempt any job and thus HOL will still play a major role in wasting computational resources. Therefore, we introduce a second rule: *preempted jobs will be served together after at most every K big job services*. Therefore, we have a non-preemptable session for the preempted small jobs either when all servers are idle or after the service of every K big jobs. After serving the preempted jobs, we reset the counter of the big jobs. This policy has two advantages. Firstly, it groups together the longest small jobs. Secondly, all the phases of service of small jobs (with the exception of those containing only preempted jobs) will be served by using preemption as a method to limit the impact of HOL. We will explain the extension of this policy to an arbitrary number of classes in Section 5.

Example 4.1 (Trace of model with $K = 3$ and $\nu = 1$). Let us consider the example depicted in Figure 1. We explain the policy using the idea of a cycle of $K = 3$ jobs. The cycle begins with the departure of the last small job of a non-preemptable batch of small jobs. Therefore, there is no preempted job waiting for service. The first big job enters in service, followed by the first batch of small jobs entering service (Phase 1). It consists of 3 jobs among which the two smallest are completed, while the longest one is preempted and stored in the orange box (waiting room). In Phase 2, another big job is in service, and at its completion, the batch with 4 small jobs enters service. One small job is preempted and stored in the orange box. Now, we have Phase 3. After the third big job service completion, the scheduler puts into service all preempted jobs together with

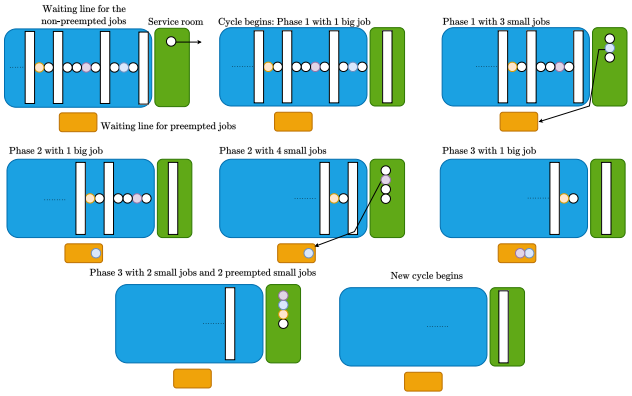


Figure 1: Example of preemption-restart policy with $K = 3$ and $\nu = 1$

the third batch of small jobs. This phase contains the preempted jobs; therefore, no further preemption takes place (even for the jobs that are served for the first time). The new cycle can begin.

4.2 Modeling assumptions and limitations

In this work, we introduce two models for the analysis of the killing policy: a simulation model used to study systems with arbitrary number of classes and obtain estimates of the jobs' waiting times, and an analytical model used to study the stability region of the killing policy in an exact manner. First, we introduce the assumptions on the simulation model used in Section 5, then we add the assumption required by the analytical model.

The simulation model relies on the following assumptions: The model has an arbitrary number of classes and their arrival processes and service times are independent. Each class is also characterized by the required number of cores which is deterministic. The service times are independent but their distributions depend on the class, while the arrival processes are independent Poisson streams. We assume that the server has only one resource type, namely the cores, and jobs' requests are not elastic, i.e., if a class requires n cores it can enter in service only if there are at least n available cores. The killing process employ a preemption-restart policy, i.e., the preempted job maintains memory of its original service time once it returns in service. Moreover, the preemption is assumed to be instantaneous, i.e., without switching delays.

When we consider the analytical model, we have more assumptions with respect to the simulation model. First, we consider only two classes of jobs: small jobs require 1 server and big jobs require all servers. The number of server is very large, i.e., we can put in service any finite amount of small jobs in parallel. Service times are exponentially distributed for small jobs while they are general for big jobs. Quite interestingly, since we propose a saturation based stability analysis, the results are independent of the distributions of the inter-arrival times, as long as the class of the customers is chosen according to a Bernoulli trial.

While the simulation model's assumptions are rather general, the analytical models' ones are much more restrictive to allow the derivation of mathematically rigorous results. These results play two roles: (i) they mathematically support the counter-intuitive result that preemption-restart improves the system's stability region instead of decreasing it and (ii) it helps the intuition behind this phenomenon.

In practice, real data centers may exhibit more than two job classes with non-exponential service times, and/or non-negligible preemption overheads. Furthermore, the one-or-all configuration is a restrictive workload model, but not without some relevance. Indeed, in database management systems (DBMS) [13], tasks such as database updates and queries can be effectively modeled as big and small jobs, respectively. To achieve the so-called conflict-serializability, DB updates must not run concurrently with other updates or read-only queries [1]. This requirement means that DB updates can be represented as big jobs, that occupy all available cores to prevent other simultaneous operations. Conversely, read-only queries, typically more frequent and faster [12], can be modeled as small jobs that are processed concurrently. Extending the analysis to more general service time distributions and more than two classes of jobs is left for future work.

4.3 Analysis of the stability region of the system with preemption-restart

In order to derive the stability region of the system with two classes, we resort to the *saturation assumption* presented in [6]. This method allows us to explicitly obtain the stability condition, and this will be insensitive to the distribution of inter-arrival times as long as we have independent class probabilities. In simple terms, we assume the system has an infinitely long waiting line and compute the throughput of the system in this scenario. This will be the maximum arrival intensity that the system can handle. Recall that the event that generates the preemption is *ν small jobs in service with the head of line occupied by a big job*. If, at the departure of a big job, there are not more than ν jobs followed by a big job, these do not enter in service.

The behavior of the system consists of K phases. We iterate exactly $K-1$ times the service of a big job, followed by a (possibly empty) batch of small jobs where preemption can occur. Then, we serve the K-th big job followed by the preempted jobs together with the last batch of small jobs, and the cycle begins again. The moment in time when the last job of the last batch leaves the system is a renewal epoch, i.e., the system loses its history and restarts again. Therefore, the throughput of one cycle is indeed the throughput of the entire process, and by [6] it is the maximum arrival rate that can be handled by the system.

Let S be the r.v. representing the number of small jobs in a sequence, i.e.:

$$Pr\{S = n\} = p_b p_s^n \,, \tag{1}$$

where p_b and p_s are the frequencies of big and small jobs, respectively, and $n \geq 0$. Since we consider that the sequence of small jobs can have length 0, the sequence of big jobs is deterministically equal to 1. For example, this means that two consecutive big jobs are seen as two sequences of big jobs of length 1, with in between a sequence of small jobs with length 0.

Let T_s be the random variable describing the length of a period of service of small jobs that have never been preempted. Notice that, since we have infinite servers, all small jobs enter in service

simultaneously. We have:

$$Pr\{T_s \le x|S\} = \begin{cases} 1 & \text{if } S \le \nu \\ \sum_{j=S-\nu}^{S} \binom{S}{j} (1-e^{-\mu_s x})^j (e^{-\mu_s x})^{S-j} & \text{if } S > \nu \end{cases}, \quad (2)$$

where the expression for the case $S > 1$ is the CDF of the $(S-\nu)$th order statistics of S i.i.d. exponential random variables with parameter μ_s.

By deconditioning, we obtain the following lemma whose proof is reported in the Appendix.

Lemma 4.2. *The CDF of the duration of service of a batch of small jobs is:*

$$Pr\{T_s \le x\} = 1 - \left(\frac{p_s}{p_b e^{\mu_s x} + p_s}\right)^{\nu+1} \quad (3)$$

and its mean is:

$$E[T_s] = -\frac{1}{\mu_s}\left(\sum_{j=1}^{\nu} \frac{p_s^j}{j} + \log p_b\right) \quad (4)$$

Since each phase of service of big jobs consists of exactly one job, we clearly have $E[T_b] = \mu_b^{-1}$. It remains to compute the length of the final phase where the preempted jobs are served together with non-preempted jobs.

Let us now characterize the statistics of the longest preempted job in a batch. Let J be the r.v. modeling its service time, taking into account that if we do not preempt any job because $S = 0$, we consider this length to be 0. Then we have:

$$Pr\{J \le x|S\} = \begin{cases} 1 & \text{if } S = 0 \\ (1-e^{-\mu_s x})^S & \text{if } S > 0 \end{cases}.$$

By deconditioning, we obtain:

$$Pr\{J \le x\} = 1 - \frac{p_s}{e^{\mu x} p_b + p_s}.$$

Now, let $J_1, J_2, \ldots, J_{K-1}$ be the i.i.d. random variables modeling the durations of the batch-longest $K-1$ killed jobs. The distribution of the length of the phase of service of the killed jobs T_J is the maximum of $J_1, \ldots, J_{K-1}$ and the maximum of the length of the jobs of the Kth cycle of service of small jobs. Notice that this maximum is also distributed as J. Therefore, we have:

$$Pr\{T_J \le x\} = \left(1 - \frac{p_s}{e^{\mu_s x} p_b + p_s}\right)^K.$$

We can now state Lemma 4.3 whose proof is reported in the appendix.

Lemma 4.3. *The expectation of T_J is:*

$$E[T_J] = \frac{1}{\mu_s}\left(\sum_{j=1}^{K-1} \frac{1-p_b^j}{j} - \log p_b\right).$$

At this point, we can obtain the stability condition of the MJQM system with preemption-restart.

Theorem 4.4 (Stability of MJQM with preemption-restart). *The stability condition of the MJQM system with preemption and restart is:*

$$\lambda < \frac{1}{p_b}\frac{K}{E[T_C]}, \quad (5)$$

where:

$$E[T_C] = (K-1)E[T_s] + \frac{K}{\mu_b} + E[T_J].$$

Proof. The expected length of a cycle of the renewal process is $E[T_C] = (K-1)E[T_s] + K\mu_b^{-1} + E[T_J]$, since we have $K-1$ phases of service of small jobs with preemption, K services of big jobs, and, finally, one phase of service for the preempted jobs. During this renewal cycle, the system has served exactly K big jobs. The expected number of small jobs is coupled to this quantity by the properties of the arrival process. Therefore, the throughput of big jobs is $K/E[T_C]$ and, by [6], this is also the maximum arrival rate for the big jobs. Therefore

$$\lambda p_b < \frac{K}{E[T_C]},$$

which simply brings us to the statement of the theorem. □

4.4 Performance indices

In order to assess the performance of our killing policy, we introduce some metrics to complement our stability analysis and use the MJQM system with FCFS as a benchmark for comparison. The first index that we use is clearly the maximum workload intensity that can be handled by the system, namely λ_{max}^{FCFS} and $\lambda_{max}^{K,\nu}$ for FCFS and our killing policy, respectively. By Theorem 4.3:

$$\lambda_{max}^{K,\nu} = \frac{1}{p_b}\frac{K}{E[T_C]},$$

where we recall that $E[T_C]$ depends both on ν and K. For FCFS, we can derive the expression from [26]:

$$\lambda_{max}^{FCFS} = \frac{1}{p_b}\left(\frac{1}{\mu_b} - \frac{\log(p_b)}{\mu_s}\right)^{-1}.$$

Let W^{FCFS} and $W^{K,\nu}$ be the rate at which we waste computational power for FCFS and our killing policy, respectively. We assume, as a unit of measure, that each unit of time of computation requires the same amount of energy. Clearly, $W_J^{FCFS} = 0$, and we have:

$$W^{K,\nu} = \lim_{t\to\infty} \frac{\text{Wasted computational energy in } [0,t)}{t}.$$

We can evaluate explicitly $W_J^{K,\nu}$ in saturation during a renewal cycle and extend it to the whole system.

Lemma 4.5 (Wasted computational power). *In saturation, the wasted computational power is given by:*

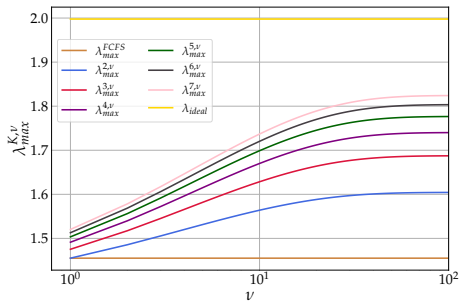
$$W^{K,\nu} = \frac{(K-1)\nu E[T_s]}{E[T_C]}. \quad (6)$$

The proof is given in the appendix.

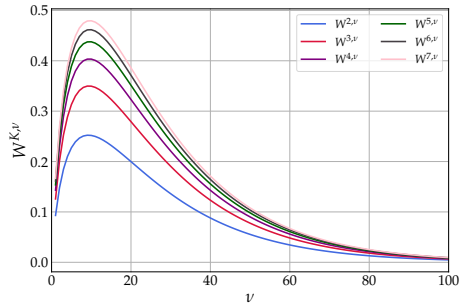
Finally, we introduce a fairness index. Consider the expected amount of time that preempted jobs have to wait before entering in service with respect to those that are not preempted. We define $E[D_J^K]$ as the ratio between this accumulated delay and the total number of small jobs served, i.e.:

$$D_{Js}^{K,\nu} = \lim_{t\to\infty} \frac{\text{Extra delay of preempted jobs in } [0,t)}{\lambda p_s t}.$$

Similarly to the previous cases, we study the index under the saturation assumption. The following Lemma gives a closed-form expression for this index.

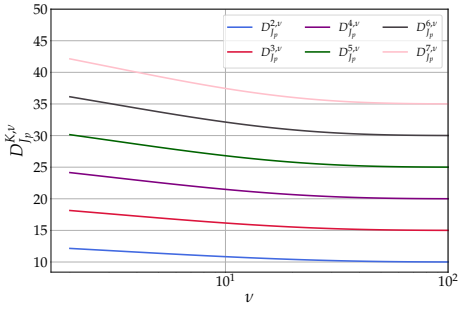


(a) Stability region

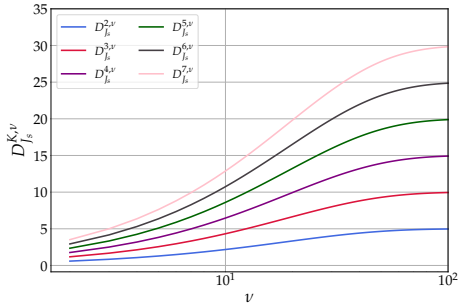


(b) Wasted power

Figure 2: Analysis of stability and wasted power with $p_s = 0.95$, $\mu_s = 0.8$, and $\mu_b = 0.1$



(a) Extra Delay for preempted jobs



(b) Extra Delay for small jobs

Figure 3: Delay analysis with $p_s = 0.95$, $\mu_s = 0.8$, and $\mu_b = 0.1$

Lemma 4.6 (Extra delay for small jobs). *In saturation, the expected extra delay for the small jobs can be computed as:*

$$D_{J_S}^{K,\nu} = \frac{K-1}{K}(1-p_s^{\nu})\left(\frac{K-2}{2}(E[T_s]+\mu_b^{-1})+\mu_b^{-1}\right) + \frac{K-1}{K}\frac{p_b}{p_s}\nu E[T_s]\,. \quad (7)$$

The proof is given in the appendix.

Lemma 4.7 (Extra delay for preempted jobs). *In saturation, the expected extra delay for the preempted jobs can be computed as:*

$$D_{J_P}^{K,\nu} = \frac{K-2}{2}(E[T_s]+\mu_b^{-1})+\mu_b^{-1}+\frac{p_b}{p_s(1-p_s^{\nu})}\nu E[T_s]\,. \quad (8)$$

The proof follows the lines of Lemma 4.6.
Clearly, for FCFS, there is no extra delay.

4.5 Illustrative example

Let us consider a system under heavy load that we study with the saturation assumptions presented in this section. The arrival process has 95% of small jobs ($p_s = 0.95$, $p_b = 0.05$) and $\mu_s^{-1} = 1.25$ and $\mu_b^{-1} = 10$. Figure 2a shows the relation between ν and the maximum workload intensity for various values of K. We observe that higher values of K and ν give larger stability regions. Figure 2b shows the power cost of preemption-restart. First, notice that when ν is very small or very big, the wasted power is limited because either we preempt only a few jobs or because we preempt so many jobs that the duration of a preemptable service batch is very small. Furthermore, we observe that higher values of K increase the fraction of sequences of small jobs that suffer preemption, and hence we have higher power waste. The (wrong) conclusion one may be tempted to draw from these figures is: do not start the service for any small job ($\nu \to \infty$) for $K-1$ sequences and let K be as big as possible. In this way, one does not waste power and obtains the maximum stability region. Actually, for the saturated scenario with two classes, this policy would be an approximation of the scheduling discipline MSF and, in practice, may not be a good idea. In fact, MSF in heavy-load leads the system to behave in a bimodal manner, i.e., serving long sequences of big jobs followed by long sequences of small jobs.

This intuition is formally captured by Figure 3a that reports the extra delay experienced by preempted jobs under our killing policy for different values of K and ν. We observe that increasing K leads to a larger average extra delay, which corresponds to reduced fairness: preempted jobs must wait through more service cycles before restarting. In contrast, increasing ν generally reduces the average extra delay per preempted job. A larger ν increases the likelihood that the sequence of small jobs in service is shorter than ν, reducing the chance that a preempted small job has to wait longer for the non-preempted jobs to finish service, even in its own batch. In extreme cases where ν is very high, small jobs may be preempted almost immediately upon entering service (i.e., $Pr\{S > \nu\}$ becomes small), thereby minimizing wasted power and reducing the extra delay. Figure 3b shows a different behavior when

examining the extra delay for all small jobs, including the non-preempted ones. While preempted jobs individually wait less, their waiting time becomes a larger fraction of the cycle because the number of preempted jobs grows as we increase ν, which reflects a trade-off in fairness: reducing absolute delay per job may still increase the relative impact they collectively have on the system.

In conclusion, this analysis shows a clear trade-off that the system administrator can face: on the one hand, a more aggressive preemption policy toward small jobs can lead to better performance at a relatively low energy cost; on the other hand, the unfairness toward jobs requiring a small amount of resources can soon become intolerable.

4.6 Simulation experiments for the non-saturated scenario

To complement the analytical results, we also perform a simulation-based[1] analysis—particularly to evaluate performance metrics that cannot be obtained analytically, such as the response time. We use the same parameter configuration as in Figure 2 and 3. We set the number of servers to a sufficiently large value so that the simulation closely approximates the analytical results when the workload intensity is high. In all experiments, we use $N = 2048$.

One thing to notice is that under low-load conditions, preempted jobs will experience longer waiting times before being restarted. This occurs because the low arrival rate of large jobs extends the duration required to complete K cycles of large-job service phases. Consequently, the waiting time for preempted jobs may become excessively long. To mitigate this effect, we have a secondary condition that allows the restart of preempted jobs: when the system becomes completely idle (i.e., no jobs remain in either the buffer or the service area), the service of preempted jobs can begin.

Figure 4a shows that the response time under our proposed policy is consistently lower than that of FCFS. The asymptotic line follows the stability region illustrated in Figure 2a. The same figure also compares the performance when varying the parameters K and ν. Increasing ν generally yields a greater improvement in response time than increasing K. However, when ν exceeds the average sequence length of small jobs, i.e., p_s/p_b, the improvement diminishes. For instance, if the average small-job sequence length is 19, setting $\nu = 100$ provides little benefit, since only about 19 small jobs are likely to be preempted in one sequence.

Figure 4a also provides intuition for selecting optimal parameters: the best performance is achieved when the total number of jobs killed within a cycle of length K is approximately equal to the total number of servers, i.e., $\nu(K-1) \approx N$. Furthermore, Figure 4i demonstrates that our policy continues to outperform FCFS even when job service times follow longer-tailed distributions, such as the Bounded Pareto distribution. The only drawback occurs under low-load conditions, where FCFS performs slightly better—although it becomes unstable more quickly than our killing policy as the load increases.

Figure 4b compares response times under low and medium-load conditions. We observe that FCFS slightly outperforms our killing policy when the load is low; a similar trend is visible in the medium load. This occurs because infrequent job arrivals increase the waiting time for preempted jobs, as the system takes longer to complete K large-job service cycles (i.e., $K-1$ preemption cycles). However, as the load increases (see the right-hand side of Figure 4b), FCFS becomes unstable more quickly than our policy. This behavior is confirmed in Figure 4c, which shows that at high loads, our killing policy clearly outperforms FCFS.

In addition to overall response time, we also measure the response times of small and large jobs separately. Since the mean service times differ between classes, Figure 4g focuses on their respective waiting times. Our policy reduces the waiting time of large jobs while slightly increasing that of small jobs. When $\nu(K-1)$ is large—close to N—we observe that at medium loads, small jobs experience higher waiting times under our policy than under FCFS.

The gap between the waiting times of small and large jobs widens as either K or ν increases. At first glance, this might suggest that our policy is unfair to small jobs. However, under FCFS, especially at medium load, lower overall response times are achieved by allowing large jobs to suffer from HOL blocking. Since large jobs contribute a significantly larger fraction of the total system load, FCFS can actually be considered less fair overall.

To capture both performance and fairness, we introduce the weighted overall response time, defined as:

$$E[T_W] = \sum_{m=1}^{M} \frac{\rho_m}{\rho} E[T_m],$$

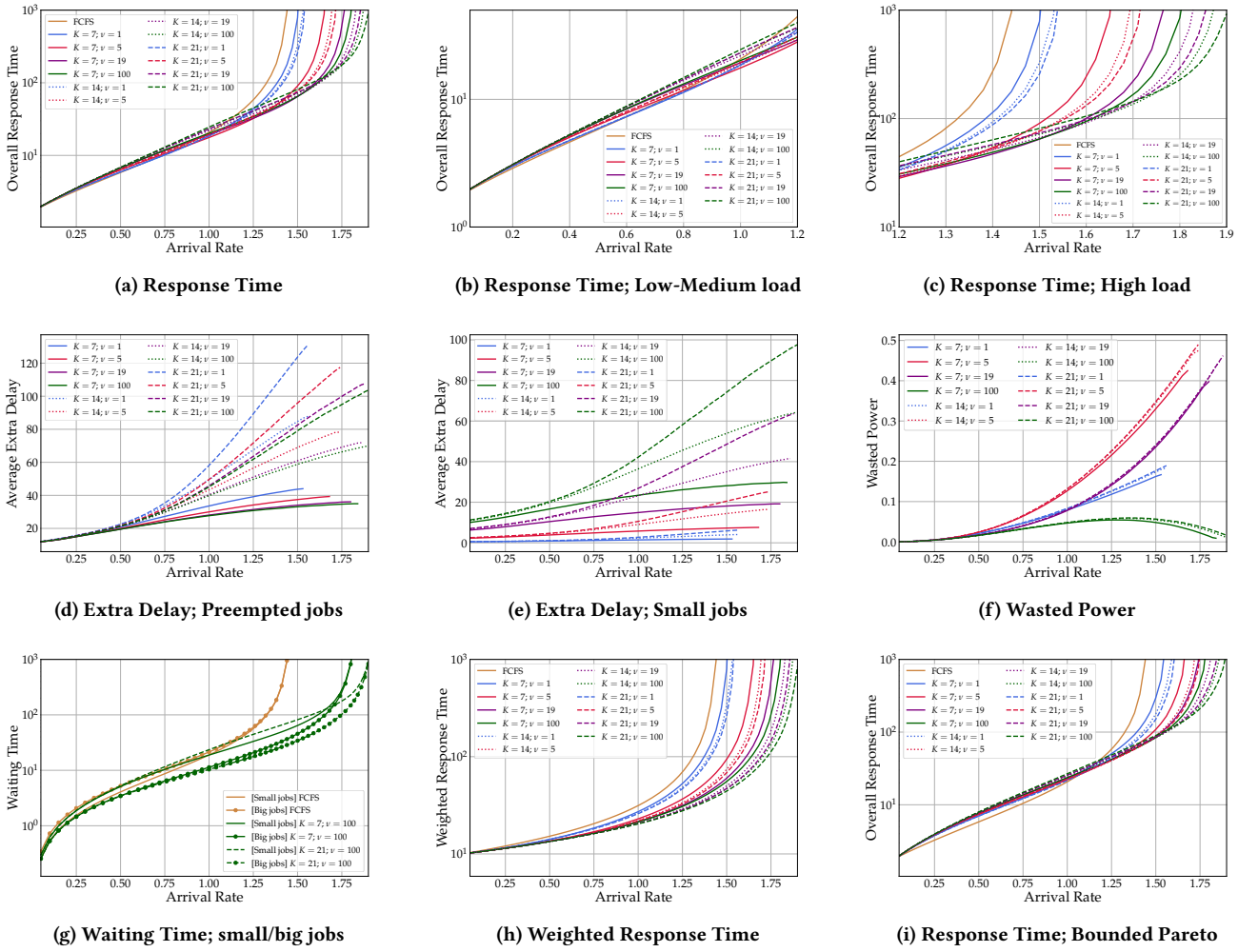
where $\rho_m = \frac{n_m p_m}{\mu_m}$ is the system load contributed by class-m jobs, derived from the product of the server need, average service time, and the probability of that class, and $\rho = \sum_{m=1}^{M} \rho_m$ is the total system load of all M classes. Here, the weight assigned to each class corresponds to its contribution to the total system load. A policy achieves a better weighted response time if it prevents large jobs, which contribute most to the load, from excessive HOL blocking.

Figure 4h compares the weighted overall response times of our policy (under various parameter settings) with those of FCFS. As shown in the figure, under medium load conditions, all configurations of our policy outperform FCFS. This consistent advantage persists as we increase K and ν, demonstrating improved weighted response times across a range of load levels.

We now turn to the impact on the preempted jobs themselves. Figure 4d shows their extra delay, exhibiting the same trends observed analytically in Figure 3a: lower ν increases delay since preemptions occur later in the small-job sequence, while higher K increases delay by extending the number of service phases before restarts. However, Figure 4e shows that the normalized delay (per all small jobs) increases with ν, emphasizing the trade-off between choosing the right value of ν, which has a different impact both ways.

Finally, Figure 4f shows the wasted power resulting from our policy. Consistent with the analytical results in Figure 2b, increasing K leads to a slight rise in wasted power due to the additional preemption cycles before restarts occur. In contrast, increasing ν does not necessarily increase wasted power. In fact, higher wasted power tends to occur when ν lies between 1 and p_s/p_b (the average length of a small-job sequence). When ν is very large, nearly all small jobs in a batch are killed immediately, resulting in less power wasted on partially served preempted jobs. Conversely, for intermediate ν, more service time is expended on small jobs before they are terminated, increasing the overall power wasted in the system.

[1] https://github.com/UniVe-NeDS-Lab/mjqm-simulator



(a) Response Time **(b) Response Time; Low-Medium load** **(c) Response Time; High load**

(d) Extra Delay; Preempted jobs **(e) Extra Delay; Small jobs** **(f) Wasted Power**

(g) Waiting Time; small/big jobs **(h) Weighted Response Time** **(i) Response Time; Bounded Pareto**

Figure 4: Analysis of the system through simulation with $p_s = 0.95$, $\mu_s = 0.8$, $\mu_b = 0.1$ and $N = 2048$

5 Extension to systems with an arbitrary number of classes

In this section, we aim to define a general killing policy applicable to systems with more than two job classes. Since we now consider a finite number of servers, the total number of servers occupied by the jobs terminated within a working cycle must not exceed the total number of available servers, i.e., $N \geq K \cdot \nu$. This ensures that all preempted jobs can be restarted simultaneously. We continue with identifying the conditions under which job termination (or "killing") can be initiated:

(i) The number of servers occupied by the job(s) selected for termination must be not greater than ν. In one-or-all scenarios, this corresponds to the number of small jobs currently in service.

(ii) The size of the job at the head of the line (HOL) must be bigger than the total number of servers currently occupied. In other words, we kill some jobs in service only if, as a consequence of this operation, the job at the head of the line can enter in service and the utilization of the resources is improved. We use the job size—or equivalently, server occupancy—as the reference metric in determining the value or priority of killing.

After a killing operation is performed, the servers are expected to be idle, allowing the HOL job to enter service. Additional subsequent jobs may also be admitted, provided they fit within the available server capacity. This relaxes the strict one-or-all assumption, as the HOL job may not necessarily occupy all servers. Once we have executed $K - 1$ killing operations, all previously terminated jobs can be restarted. We first reintroduce the killed jobs into service, followed by any subsequent jobs that can fit within the remaining server capacity. All these jobs begin service simultaneously. Importantly, the restarted jobs—and any additional jobs that start together with them—cannot be terminated. We also use the idle-system condition previously explained to mitigate the potentially longer waiting time that the preempted jobs could receive due to the longer time needed to reach $K - 1$ killing phases.

After this restart phase, the system once again permits killing operations, effectively resetting the process with zero preempted jobs. The pseudo-code can be seen in Algorithm 1.

Algorithm 1 Extended Policy for Multi-Class Systems

```
while event < n_events do
    if event = arrival then
        add job to buffer
    else if event = departure then
        busy_servers ← busy_servers − job.size
        if job is in non_preemptable_jobs then
            non_preemptable ← non_preemptable − 1
        end if
    end if
    if jobs_can_enter then
        put jobs from buffer into service
    end if
    if killing_permitted and busy_servers ≤ ν and kill_cycle < K − 1 and HOL_job_size > idle_servers and HOL_job_size > busy_servers then
        preempt all jobs in service
        put HOL_job into service
        kill_cycle ← kill_cycle + 1
    end if
    if kill_cycle = K − 1 or idle_system then
        jobs_can_enter ← false
        killing_permitted ← false
        if idle_servers < preempted_jobs_size then
            continue
        end if
        for each job in preempted_jobs do
            put the job into service
            non_preemptable ← non_preemptable + 1
        end for
        while idle_servers > 0 and buffer not empty do
            put the job from buffer into service
            non_preemptable ← non_preemptable + 1
        end while
        kill_cycle ← 0
        jobs_can_enter ← true
    end if
    if non_preemptable = 0 then
        killing_permitted ← true
    end if
    event ← event + 1
end while
```

5.1 Simulation study with Google Borg dataset

To evaluate the effectiveness of our extended policy, we use real workload traces from Cell B and Cell E of the Google Borg dataset [30, 33] and compare its performance against FCFS. The experiment setup and the explaination of the Google Borg cells used fpr the simulation can be seen in the Appendix.

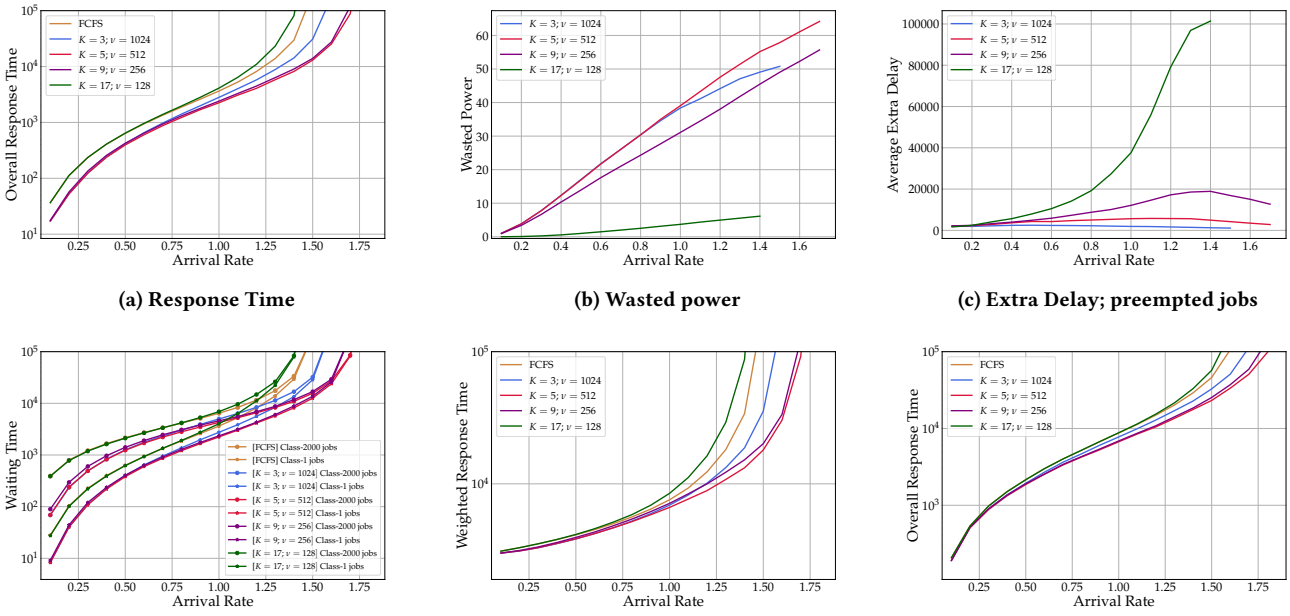
Figure 5a presents the overall response times obtained for Cell B. We observe consistent improvements in three out of the four configurations tested. The greatest gains occur when K is set to a moderate value; however, when K becomes too large, the performance of our policy deteriorates and eventually falls below that of FCFS. This suggests that excessively large K values—especially when combined with relatively small ν—reduce the benefits of preemption and increase system overhead. Figure 6a however shows that all tested configurations in Cell E gains improvement over FCFS. The consistent feature is that the configuration with high value of K always perform the worst among all the other configurations. Furthermore, Figure 5f and 6f indicates that the extended policy remains effective even under workloads with long-tailed service time distributions, such as those modeled by the Bounded Pareto distribution. In these experiments, we find that these settings still yield clear performance improvements over FCFS, despite the increased variability in service times. This suggests that the advantages of our approach are not limited to light-tailed or synthetic settings but extend to more realistic, highly variable workloads as well.

Examining the waiting times for the smallest and largest job classes (in terms of server demand) in Figure 5d and 6d, we note that our policy does not achieve the same improvements for large jobs as observed earlier in Figure 4g. This difference can be explained by the relatively low number of preemptions occurring in the Google Borg trace simulations compared to the synthetic two-class "one-or-all" scenarios. In fact, when looking at the experiment's results, jobs from the smallest class—those requesting a single server—are preempted less than 1% of the time. In both cells, the class that experienced the highest number of preemptions are those that has a very low service rate. For example, in cell E, the class that experienced the most preemptions are class-180 jobs that has an average service time of almost 20000 seconds. Nevertheless, the result shows that even for those classes, the waiting times under our policy are not significantly worse than under FCFS.

These findings are summarized in Figure 5e and 6e, which reports the weighted response times. The trends observed here align closely with those in Figure 5a and 6a, indicating that the configurations performing best in terms of raw response time are also the best under the weighted response time. We believe that the job class distributions play a major role in determining the performance of our policy. This observation highlights opportunities for further improvement through the design of more adaptive or class-aware preemption conditions in systems with more than two job classes.

Finally, Figure 5b illustrates the wasted power observed in our Cell B simulations. Notably, the three configurations that outperform FCFS also exhibit higher levels of wasted power. Similarly for Cell E, 6b shows that the two configurations that has the higher values of ν exhibit higher wasted power than the two configurations with lower values of ν. This shows that higher ν means that some jobs that ended up being preempted has already received so much work that the waste due to getting killed become pretty high. This also suggests a clear trade-off between achieving better performance—both in response time and stability—and incurring higher wasted power due to the need to restart jobs that had already received partial service prior to preemption.

Figure 5c and 6c further examines the extra delay experienced by preempted jobs. As expected, increasing K causes these delays to grow, since preempted jobs must wait through more large-job



(a) Response Time **(b) Wasted power** **(c) Extra Delay; preempted jobs**

(d) Waiting Time for smallest/biggest class **(e) Weighted Response Time** **(f) Response Time; Bounded Pareto**

Figure 5: Analysis of the system through simulation with real traces from Cell B

service phases before resuming execution. Importantly, the configurations that achieve the strongest improvements in response time and stability do not correspond to the ones producing the largest penalties for preempted jobs. For example, parameter settings with very high K and small ν result in significantly worse extra delay, yet do not provide the best performance. These results introduce an additional nuance to the policy trade-off: not only between performance and wasted power, but also in terms of fairness to the preempted jobs.

6 Conclusion

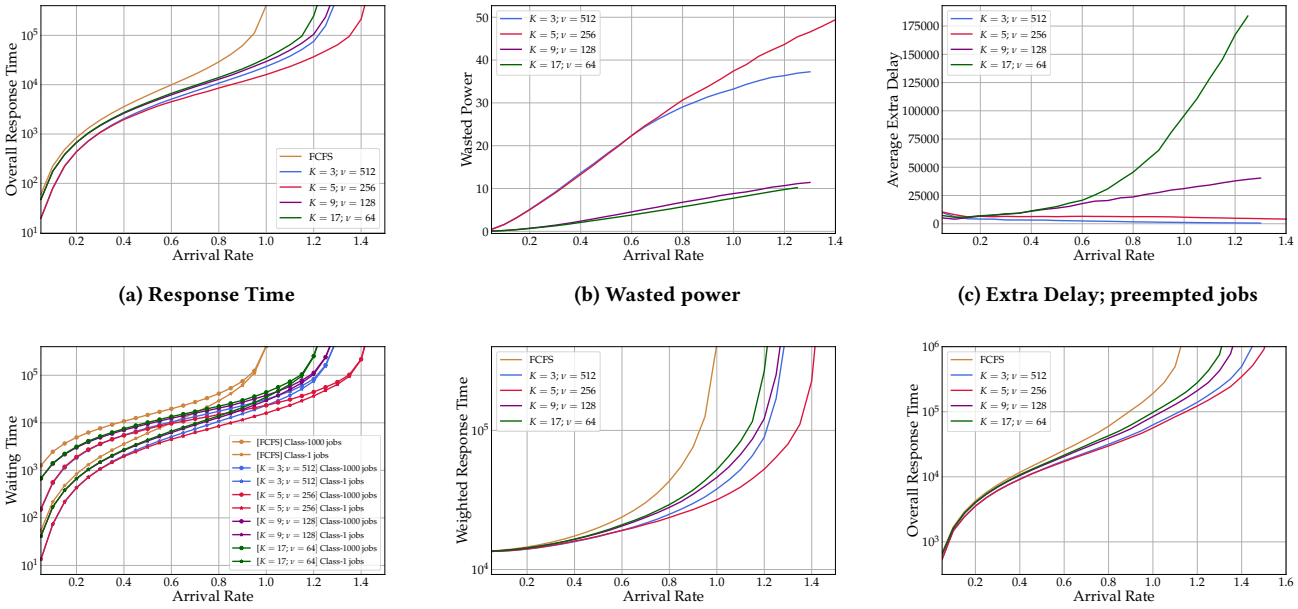
Policies based on preemption-restart have been known for quite a long time in the context of priority systems. Their key advantage is that they are simple to implement, both from the hardware and software points of view. In these systems, preemption-restart is used to satisfy priority requirements, but the restart mechanism negatively affects the maximum throughput reachable by the system.

To the best of our knowledge, in this paper, we have proposed for the first time a preemption-restart policy for multiserver-job systems, and we have surprisingly observed that we can improve the maximum throughput with respect to FCFS as long as the work wasted on preempted jobs does not exceed the service waste caused by HOL blocking. This has been shown both mathematically in a simple scenario and by discrete event simulation on a real dataset. Furthermore, we observed that energy waste is tolerable and is always bounded. As a directed consequence of preemption, in the heavy-load scenarios, the user's experience, measured by the average response time, is also improved compared with the FCFS scheduling. The general explanation of these counterintuitive results lies in the non-work-conserving nature of the multiserver-job systems.

While FCFS is still widely used in high-performance computing for batch job scheduling, many data centers employ more sophisticated policies. For instance, SLURM, a widely used scheduler in large clusters, relies on *backfilling* policy and requires users to declare job runtimes [21, 34]. Thanks to this knowledge, they can reorder the jobs to maximize the system utilization. In this work, we aimed to study the impact of preemption-restart when service times are unknown, and we compared its performance with FCFS that shares the same knowledge on the workload.

In this way, we have been able to prove the benefits of preemption-restart rigorously, although for a simplified scenario (two classes with exponentially distributed service time and an arbitrary arrival process). Clearly, it could be interesting to understand if policies belonging to the family of backfilling can find the same benefits of FCFS from preemption-restart. One interesting future work is a cost-based analysis contrasting preemptive scheduling with backfilling, particularly in terms of the value of runtime estimates.

We think that this paper also opens other future research directions. Firstly, the analytical results should be generalized for general distributions of the service processes, e.g., for deterministic service times or some other tractable distribution. Then it would be of interest to devise more sophisticated heuristics for the extended policy in systems with an arbitrary number of classes. The proposed killing policy may also be relevant for latency-sensitive service environments where long tasks can block shorter ones. For instance, in GPU-based inference systems such as in Large Language Model (LLM), long-running requests can create head-of-line blocking for



(a) Response Time **(b) Wasted power** **(c) Extra Delay; preempted jobs**

(d) Waiting Time for smallest/biggest class **(e) Weighted Response Time** **(f) Response Time; Bounded Pareto**

Figure 6: Analysis of the system through simulation with real traces from Cell E

smaller requests when preemption is not available or is limited. While our model does not explicitly capture these type of systems, the underlying motivation of reducing HOL blocking under limited job-size information is still relevant. A detailed evaluation in LLM settings is left for future work.

Finally, in the proposed "killing" policy, it is tempting to interchange the role of small and big jobs, i.e., to consider the dual policy: when a small job arrives, it preempts a big job, and once K big jobs are accumulated, they are served without preemption. In this way, we expect to improve the response time of the large majority of jobs, although the stability region may suffer a reduction.

References

[1] Rakesh Agrawal, Michael J. Carey, and Miron Livny. 1987. Concurrency Control Performance Modeling: Alternatives and Implications. *ACM Trans. on Database Systems* 12, 4 (1987), 609–654.

[2] Adityo Anggraito, Diletta Olliaro, Marco Ajmone Marsan, and Andrea Marin. 2024. Stability of the multiserver job queuing model with infinite resources. In *International Conference on Analytical and Stochastic Modeling Techniques and Applications*. Springer, 148–163.

[3] Adityo Anggraito, Diletta Olliaro, Andrea Marin, and Marco Ajmone Marsan. 2024. The Non-Saturated Multiserver Job Queuing Model with Two Job Classes: a Matrix Geometric Analysis. In *2024 32nd International Conference on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 1–8.

[4] Adityo Anggraito, Diletta Olliaro, Andrea Marin, and Marco Ajmone Marsan. 2025. The Multiserver Job Queuing Model with two job classes and Cox-2 service times. *Performance Evaluation* (2025), 102486.

[5] Adityo Anggraito, Diletta Olliaro, Marco Ajmone Marsan, and Andrea Marin. 2025. The Multiserver Job Queuing Model with big and small jobs: Stability in the case of infinite servers. *Performance Evaluation* 168 (2025), 102477.

[6] François Baccelli and Serguei Foss. 1995. On the saturation rule for the stability of queues. *Journal of Applied Probability* 32, 2 (1995), 494–507.

[7] Faisal Haque Bappy, Tariqul Islam, Tarannum Shaila Zaman, Raiful Hasan, and Carlos Caicedo. 2023. A deep dive into the Google cluster workload traces: Analyzing the application failure characteristics and user behaviors. In *2023 10th International Conference on Future Internet of Things and Cloud (FiCloud)*. IEEE, 103–108.

[8] Wei Chang. 1965. Preemptive priority queues. *Operations research* 13, 5 (1965), 820–827.

[9] Zhongrui Chen, Adityo Anggraito, Diletta Olliaro, Andrea Marin, Marco Ajmone Marsan, Benjamin Berg, and Isaac Grosof. 2025. Improving nonpreemptive multiserver job scheduling with quickswap. *Performance Evaluation* (2025), 102525.

[10] Lipu Fei, Bogdan Ghiţ, Alexandru Iosup, and Dick Epema. 2014. KOALA-C: A task allocator for integrated multicluster and multicloud environments. In *2014 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 57–65.

[11] Brian Fralix. 2024. On the time-dependent behavior of preemptive single-server queueing systems with Poisson arrivals. *Queueing Systems* 107, 1 (2024), 31–61.

[12] Florian Funke, Alfons Kemper, and Thomas Neumann. 2011. Benchmarking hybrid OLTP&OLAP database systems. In *Datenbanksysteme für Business, Technologie und Web (BTW)*. GI, Bonn, Germany, 390–409.

[13] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. 2009. *Database Systems: The Complete Book*. Pearson Prentice Hall, Saddle River, USA.

[14] Donald P Gaver Jr. 1962. A waiting line with interrupted service, including priorities. *Journal of the Royal Statistical Society Series B: Statistical Methodology* 24, 1 (1962), 73–90.

[15] Isaac Grosof, Mor Harchol-Balter, and Alan Scheller-Wolf. 2022. WCFS: A new framework for analyzing multiserver systems. *Queueing Systems* 102, 1 (2022), 143–174.

[16] Isaac Grosof, Mor Harchol-Balter, and Alan Scheller-Wolf. 2023. New stability results for multiserver-job models via product-form saturated systems. *ACM SIGMETRICS Performance Evaluation Review* 51, 2 (2023), 6–8.

[17] Isaac Grosof, Ziv Scully, Mor Harchol-Balter, and Alan Scheller-Wolf. 2022. Optimal scheduling in the multiserver-job model under heavy traffic. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 6, 3 (2022), 1–32.

[18] Mor Harchol-Balter. 2000. Task assignment with unknown duration. In *Proceedings 20th IEEE international conference on distributed computing systems*. IEEE, 214–224.

[19] Mor Harchol-Balter. 2022. The multiserver job queueing model. *Queueing Systems* 100, 3 (2022), 201–203.

[20] Narendra Kumar Jaiswal. 1961. Preemptive resume priority queue. *Operations Research* 9, 5 (1961), 732–742.

[21] Tim Jette, Morris A.and Wickberg. 2023. Architecture of the Slurm Workload Manager. In *Job Scheduling Strategies for Parallel Processing*. Springer Nature

m	$T_m^{(E)}$	$p_m^{(E)}$	$\tau_m^{(E)}[s]$	$T_m^{(B)}$	$p_m^{(B)}$	$\tau_m^{(B)}[s]$
1	1	9.3e-1	3.8e+0	1	8.4e-1	6.7e+0
2	2	6.3e-2	3.5e+0	2	5.0e-2	2.2e-1
3	3	8.2e-3	2.0e+1	3	6.5e-3	5.2e+0
4	4	3.2e-4	5.4e+1	4	2.4e-2	3.2e+0
5	5	5.6e-4	4.9e+1	5	8.8e-4	1.0e+2
6	6	1.2e-4	1.4e+1	6	1.5e-3	1.2e+2
7	7	1.2e-3	1.6e+0	9	8.6e-5	5.9e+0
8	10	3.2e-4	3.1e+1	10	5.7e-3	1.4e+1
9	11	1.4e-4	4.5e+1	11	3.4e-5	3.9e+3
10	12	6.7e-5	1.1e+1	14	4.0e-5	2.3e+2
11	15	2.2e-4	2.8e+2	15	3.9e-4	1.3e+1
12	20	6.3e-4	8.5e+1	16	7.8e-5	1.4e+4
13	25	4.9e-4	1.1e+2	20	6.9e-4	1.0e+2
14	30	3.4e-5	2.2e+0	30	2.4e-4	7.4e-1
15	40	1.2e-4	9.9e+3	35	2.2e-4	2.1e-1
16	43	8.9e-5	3.1e+0	38	5.8e-5	7.3e+1
17	48	3.5e-4	2.6e+0	50	7.2e-4	5.3e+1
18	50	1.0e-4	1.4e+1	98	8.8e-5	8.1e-1
19	100	1.6e-4	3.8e+2	99	7.6e-4	5.2e+0
20	180	3.4e-5	1.8e+4	100	6.9e-2	3.1e+0
21	200	3.7e-5	1.6e+1	120	3.6e-5	2.8e-1
22	400	3.1e-4	5.2e+1	200	8.0e-5	5.2e+3
23	457	5.2e-5	7.5e-1	256	1.4e-4	3.6e+3
24	500	5.0e-4	1.1e+1	500	2.1e-4	5.2e+1
25	929	3.9e-5	1.7e+1	795	3.8e-5	3.3e+0
26	1000	4.2e-5	2.0e+2	2000	1.3e-4	5.7e+2

Table 1: Original Google Data for Cell B and Cell E. n_m is the numbers of cores used by class m. p_m is the probability of class m. μ_m^{-1} is the average core holding time in seconds for class m.

Switzerland, Cham, Switzerland, 3–23.

[22] Achyutha Krishnamoorthy, Padinhare K Pramod, and Srinivas R Chakravarthy. 2014. Queues with interruptions: a survey. *Top* 22, 1 (2014), 290–320.

[23] Chengzhi Lu, Kejiang Ye, Guoyao Xu, Cheng-Zhong Xu, and Tongxin Bai. 2017. Imbalance in the cloud: An analysis on alibaba cluster trace. In *2017 IEEE International Conference on Big Data (Big Data)*. IEEE, 2884–2892.

[24] Ahuva W. Mu'alem and Dror G. Feitelson. 2002. Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling. *IEEE transactions on parallel and distributed systems* 12, 6 (2002), 529–543.

[25] Diletta Olliaro, Adityo Anggraito, Marco Ajmone Marsan, Simonetta Balsamo, and Andrea Marin. 2024. The impact of service demand variability on data center performance. *IEEE Transactions on Parallel and Distributed Systems* (2024).

[26] Diletta Olliaro, Marco Ajmone Marsan, Simonetta Balsamo, and Andrea Marin. 2023. The saturated multiserver job queuing model with two classes of jobs: Exact and approximate results. *Performance Evaluation* 162 (2023), 102370.

[27] Dejan Perkovic and Peter J Keleher. 2000. Randomization, speculation, and adaptation in batch schedulers. In *SC'00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*. IEEE, 7–7.

[28] Reshmi Roy, Arup Biswas, and Arnab Pal. 2024. Queues with resetting: a perspective. *Journal of Physics: Complexity* 5, 2 (2024), 021001.

[29] Ziv Scully. 2019. Open Problem—M/G/1 Scheduling with Preemption Delays. *Stochastic Systems* 9, 3 (2019), 311–312.

[30] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. 2020. Borg: the next generation. In *Proceedings of the fifteenth European conference on computer systems*. 1–14.

[31] Dan Tsafrir, Yoav Etsion, and Dror G Feitelson. 2007. Backfilling using system-generated predictions rather than user runtime estimates. *IEEE Transactions on Parallel and Distributed Systems* 18, 6 (2007), 789–803.

[32] Joris Walraevens, Bart Steyaert, and Herwig Bruneel. 2006. A preemptive repeat priority queue with resampling: Performance analysis. *Annals of Operations Research* 146, 1 (2006), 189–202.

[33] Mert Yildiz and Andrea Baiocchi. 2024. Data-driven workload generation based on google data center measurements. In *2024 IEEE 25th International Conference on High Performance Switching and Routing (HPSR)*. IEEE, 143–148.

[34] Andy B Yoo, Morris A Jette, and Mark Grondona. 2003. Slurm: Simple linux utility for resource management. In *Workshop on job scheduling strategies for parallel processing*. Springer, 44–60.

[35] Dmitry Zotkin and Peter J Keleher. 1999. Job-length estimation and performance in backfilling schedulers. In *Proceedings. The Eighth International Symposium on High Performance Distributed Computing (Cat. No. 99TH8469)*. IEEE, 236–243.

Appendices

A Simulation

The simulation study was conducted using an ad-hoc C++ simulator. Input parameters were read from TOML files, and output results were stored in CSV format. Experiments were executed on a cluster node equipped with an Intel Xeon Gold 6148 CPU (20 cores) running at 2.40 GHz and 200 GB of ECC RAM. Storage was provided by a 30 TB NAS, and the system ran on a Nutanix hyperconverged architecture. Each configuration was simulated for 30 million events and repeated 30 times to compute the mean values and 95% confidence intervals. The values reported in the figures correspond to the mean of each metric over the repetitions. The code can be made available online upon publication.

A.1 Google Borg Dataset

The Google Borg data used in our simulations follows the processing described in [25]. The raw traces originate from [30] and were further refined in [33], where the number of cores used and the average holding times for each job were extracted. In our work, we treat the number of cores requested by a job as its class. Since the original trace contains a very large number of classes, we restrict our analysis to those with a frequency of at least 10^{-5}. Among the many clusters (cells) in the dataset, we use cell B and cell E. After filtering, the resulting both cells contains 26 classes, with core requirements ranging from 1 to 2000 for cell B, and 1 to 1000 for cell E. The holding times, meanwhile, span five orders of magnitude for both cells. Table 1 summarizes the characteristics of these job classes.

B Proof of results

B.1 Proof of Lemma 4.2

Proof. It is straightforward to check that for any $0 < a, b < 1$, $k \geq 2$ and $0 \leq m \leq k-1$ the following identity holds:

$$\sum_{n=k}^{\infty}(1-a)a^n\binom{n}{n-m}b^{n-m}(1-b)^m =$$

$$= \frac{(1-a)[a(1-b)]^m}{(1-ab)^{m+1}} - (1-a)\left(\frac{1-b}{b}\right)^m \sum_{n=m}^{k-1}(ab)^n\binom{n}{n-m}. \quad (9)$$

From (2), the geometric progression $\sum_{k=0}^{n} r^k = (1-r^{n+1})/(1-r)$, $0 < r < 1$, and the law of total probability, we have:

$$Pr\{T_s \le x\} = \sum_{n=0}^{\nu} Pr\{S=n\} + \sum_{n=\nu+1}^{\infty} Pr\{S=n\}\mathrm{P}(T_2 < x | S=n) =$$
$$= (1-p_s^{\nu+1}) + \sum_{n=\nu+1}^{\infty} p_b p_s^n \sum_{j=n-\nu}^{n} \binom{n}{j} (1-e^{-\mu_s x})^j (e^{-\mu_s x})^{n-j}.$$

By using (9), the second term of the right-hand side can be rewritten as:

$$\sum_{i=0}^{\nu} \frac{p_b (p_s e^{-\mu_s x})^i}{(1-p_s(1-e^{-\mu_s x}))^{i+1}} - \\ - \sum_{i=0}^{\nu} p_b \left(\frac{e^{-\mu_s x}}{1-e^{-\mu_s x}}\right)^i \sum_{n=i}^{\nu} (p_s(1-e^{-\mu_s x}))^n \binom{n}{n-i}.$$

By changing the order of summations in the second term, remembering the sum of geometric progression, we arrive at Eq. (3). Now, note that the mean of the $(n-\nu)$th order statistics of S i.i.d. exponential random variables with parameter μ_s is equal to $\frac{1}{\mu_s}\sum_{j=1}^{n-\nu}\frac{1}{n-j+1}$. Therefore, the mean of T_s can be obtained from the law of total expectation $E[T_s] = E[E[T_s|S]]$, by observing that

$$E[E[T_s|S]] = \sum_{n=0}^{\nu} Pr\{S=n\}\cdot 0 + \sum_{n=\nu+1}^{\infty} Pr\{S=n\}\frac{1}{\mu_s}(H_n - H_\nu),$$

where H_k denotes the kth harmonic number. By breaking the second term into two sums and performing the summation, remembering that the generating function for the harmonic numbers is equal to $\sum_{n=1}^{\infty} z^n H_n = -(1-z)^{-1}\log(1-z)$, we arrive at Eq. (4). □

B.2 Proof of Lemma 4.3

Proof. By applying the binomial theorem to $E[T_J] = \int_0^\infty (1 - Pr\{T_J \le x\})dx$ and invoking the result of the Lemma 3.2, we get

$$E[T_J] = \sum_{i=1}^{K} \binom{K}{i} (-1)^{i+1} \int_0^\infty Pr\{T_s > x\}dx =$$
$$= \sum_{i=1}^{K} \binom{K}{i} (-1)^i \frac{1}{\mu_s}\left(\sum_{j=1}^{i-1}\frac{p_s^j}{j} + \log p_b\right).$$

We obtain the two terms, which are further simplified to yield the statement of the lemma. Indeed, note that $\sum_{i=1}^{K}\binom{K}{i}(-1)^i = (1-1)^K - 1 = -1$ and thus the second term is simply $-\frac{1}{\mu_s}\log p_b$. For the first term, the change of the order of summations and the binomial theorem gives:

$$\frac{1}{\mu_s}\sum_{i=1}^{K}\binom{K}{i}(-1)^i\sum_{j=1}^{i-1}\frac{p_s^j}{j} = \frac{1}{\mu_s}\sum_{j=1}^{K-1}\frac{p_s^j}{j}\sum_{i=j+1}^{K}\binom{K}{i}(-1)^i =$$
$$= \frac{1}{\mu_s}\sum_{j=1}^{K-1}\frac{p_s^j}{j}\left((1-1)^K - \sum_{i=0}^{j}\binom{K}{i}(-1)^i\right) = \frac{1}{\mu_s}\sum_{j=1}^{K-1}\frac{(-p_s)^{j-1}}{j}\binom{K-1}{j}.$$

The latter expression can be simplified by noting that according to the hockey-stick identity $\binom{K-1}{j} = \sum_{i=j}^{K-1}\binom{i-1}{j-1}$, and therefore

$$\frac{1}{\mu_s}\sum_{j=1}^{K-1}\frac{(-p_s)^{j-1}}{j}\binom{K-1}{j} = \frac{1}{\mu_s}\sum_{j=1}^{K-1}\frac{(-p_s)^{j-1}}{j}\sum_{i=j}^{K-1}\binom{i-1}{j-1} =$$
$$= \frac{1}{\mu_s}\sum_{j=1}^{K-1}\frac{(-p_s)^{j-1}}{i}\sum_{i=j}^{K-1}\binom{i}{j} = -\frac{1}{\mu_s}\sum_{i=1}^{K-1}\frac{1}{i}\sum_{j=1}^{i}\binom{i}{j}(-p_s)^j =$$
$$= -\frac{1}{\mu_s}\sum_{i=1}^{K-1}\frac{(1-p_s)^i - 1}{i} = \frac{1}{\mu_s}\sum_{i=1}^{K-1}\frac{1-p_b i}{i}.$$

□

B.3 Proof of Lemma 4.5

Proof. Let us consider a single sequence of small jobs of length S in the queue, with the pdf of S defined by Eq. (1). If $S \le \nu$, then $T_s = 0$ and we do not waste any computational power because such batch of S jobs does not enter in service from the queue. Now, assume that the sequence of small jobs has more than ν jobs. The conditioned distribution of T_s is given by Eq. (2). Recall that each time K big jobs leave the system, the preempted jobs enter in service (and their service does not get preempted). Therefore, the total number of preempted batches of small jobs entering in service each time is $K-1$. Therefore, in saturation, on average, the wasted computational power is equal to:

$$W^{K,\nu} = \frac{K-1}{E[T_C]}\left(0\cdot Pr\{S\le\nu\} + \sum_{n=\nu+1}^{\infty}\nu\cdot E[T_s|S=n]Pr\{S=n\}\right),$$

□

which can be simplified to Eq. (6).

B.4 Proof of Lemma 4.6

We reason on the cycles consisting of K phases, as before. Consider the i-th batch of small jobs with $1 \le i < K$. The total extra delay accumulated by the preempted jobs is given by the sum of two terms:

- by the product of the preempted jobs and the processing time of the other shorter jobs in the same batch;
- by the product of the preempted jobs and the service of other $K-1-i$ sequence of a big job and a batch of small jobs.

Let us condition on the sequence of small jobs S. If $S \le \nu$, then the first term is 0 and we have only the second term, which is:

$$\sum_{n=1}^{\nu} n p_s^n p_b \left((K-1-i)(E[T_s] + \mu_b^{-1}) + \mu_b^{-1}\right)$$
$$= \frac{p_s}{p_b}(1-p_s^\nu(1+\upsilon p_b))\left((K-1-i)(E[T_s]+\mu_b^{-1}) + \mu_b^{-1}\right)$$

If $S > \nu$, we preempt exactly ν jobs, hence the second term is simply $\nu\left((K-1-i)(E[T_s]+\mu_b^{-1}) + \mu_b^{-1}\right)p_s^{\nu+1}$. As for the first term, we have to compute:

$$\sum_{n=\nu+1}^{\infty}\nu p s^n p_b E[T_s|S=n] = \nu E[E[T|S]] = \nu E[T_s],$$

as observed in the proof of Lemma 4.2. Summing up, we have:

$$ED_J^{K,v}KE[S] = \sum_{i=1}^{K-1} \left[\frac{p_s}{p_b}(1 - p_s^v(1 + vp_b)) \cdot \left((K-1-i)(E[T_s] + \mu_b^{-1}) + \mu_b^{-1}\right) + v\left((K-1-i)(E[T_s] + \mu_b^{-1}) + \mu_b^{-1}\right) p_s^{v+1} + vE[T_s] \right]$$

This can be rewritten as:

$$D_J^{K,v}KE[S] = \left(\sum_{i=1}^{K-1}(K-1-i)\right)\left(\frac{p_s}{p_b}(1-p_s^v(1+vp_b))(E[T_s]+\mu_b^{-1}) + v(E[T_s] + \mu_b^{-1})p_s^{v+1}\right) + (K-1)\frac{p_s}{p_b}\left(1 - ps^v(1+vp_b)\right)\mu_b^{-1} + (K-1)vp_s^{v+1}\mu_b^{-1} + (K-1)vE[T_s],$$

and further simplified to:

$$D_J^{K,v}KE[S] = \frac{(K-1)(K-2)}{2}\frac{p_s}{p_b}(1-p_s^v)(E[T_s] + \mu_b^{-1}) + (K-1)\frac{p_s}{p_b}(1-p_s^v)\mu_b^{-1} + (K-1)vE[T_s],$$

that easily leads to Eq. (7).