CHALLENGE 2

Giuseppe Di Poce 2072371
Mario Edoardo Pandolfo 1835189
Gabriele Pelliccioni 1838084

Dipartimento di Informatica, Sapienza Università di Roma
Data Science-I3S Master

May 30, 2023

1 Dispatching Algorithm

A dispatching algorithm, in the specific case of a data center, is a formal computational procedure or set of rules used to efficiently allocate and schedule tasks, or jobs on the available computing resources, within the data center environment. It aims to optimize resource utilization, minimize response time, and ensure fair distribution of workload.

1.1 Formal Description

Dispatching Algorithm: Adapted version of LWL

Input: Arrived Job
Order the Job tasks from the minimum to the maximum in terms of CPU usage
Poll all servers for their status. Save their current unfinished work in a local copy.
for each task of the Job **do**
  Select the servers with least number of tasks with equal or higher priority than the one of the current task to dispatch
  From the subset of servers select the server with the minimum unfinished work (ties are solved uniformly at random).
  Send the task to the selected server.
  Update the local copy of the servers status.
end for

1.2 Observations and motivations

Since we wanted to implement a preemptive scheduler, LWL is not the most optimal type of dispatcher because servers with the least unfinished work are not guarantee to run the arriving task as fast as it could run on other servers due to the time slot mechanism.

Our main idea to solve this problem was to add a first control in which the dispatcher search for the servers with the minimum number of tasks with priority equal or higher than the one of the current dispatching task. On this subset of servers then, as in the standard version of LWL, our algorithm choose the server with the least remaining *unfinished work* (ties are solved uniformly at random), trying to maintain an uniform distribution of the *working load* among the servers.

In the standard LWL the poll of the servers status is done for each task of the arrived job, while in our algorithm the poll is done only one time right after the job arrives.
This is possible because we are in the assumption that:

- There is only one dispatcher.
- There is no delay for message exchange among the dispatcher and the servers.

Thanks to these assumptions it is possible for the dispatcher to compute exactly the unfinished work of the servers each time that we send a task (tasks of the same job are sent basically at the same time).
This modification of the LWL algorithm reduces the number of needed messages among the dispatcher and the servers, indeed only one message for polling is needed for each job.

The total average messages load $\overline{L}$ for our dispatcher is given by:

$$\overline{L} = \frac{N_{messages}}{N_T} = \frac{2(N_S \cdot N_J) + \sum_{j=1}^{N_J} N_{Tj}}{N_T} = \frac{2(N_S \cdot N_J) + N_T}{N_T} = 2 \cdot \frac{N_S \cdot N_J}{N_T} + 1$$

While for the standard LWL is:

$$\overline{L} = \frac{N_{messages}}{N_T} = \frac{(2N_S + 1) \cdot \sum_{j=1}^{N_J} N_{Tj}}{N_T} = \frac{(2N_S + 1) \cdot N_T}{N_T} = 2N_S + 1$$

where :
N_S = number of Servers; N_J = number of Jobs; N_{Tj} = number of tasks of the j^{th} job; N_T = number of Tasks

Notice that in our dispatcher the $\overline{L}$ still decrease as N_T increases, while in the standard implementation of LWL it remain constant to $2N_S + 1$.

Finally, in our version of LWL we order the tasks of a job by their CPU usage (in ascending order): by sending each time the task from the lowest to the largest CPU usage we distribute more evenly the load to the servers.
From our simulations it seems that this ordering policy has some good effects on the $\overline{S}$ (Job slow down) and on $\overline{R}$ (Response Time). The reason could be that the dispatcher sends ordered batches of data to the servers (for jobs with a large number of tasks).

2 Scheduling Algorithm

As Scheduling algorithm we decided to implement Fixed-priority pre-emptive scheduling with aging (FPS). It is commonly used in real-time systems to manage the execution of processes on computing resources. It operates based on process priorities and allows for preemption, which means a higher-priority task can interrupt the execution of a lower-priority task at any time, but thanks to aging the priority the last one are going to be eventually executed.

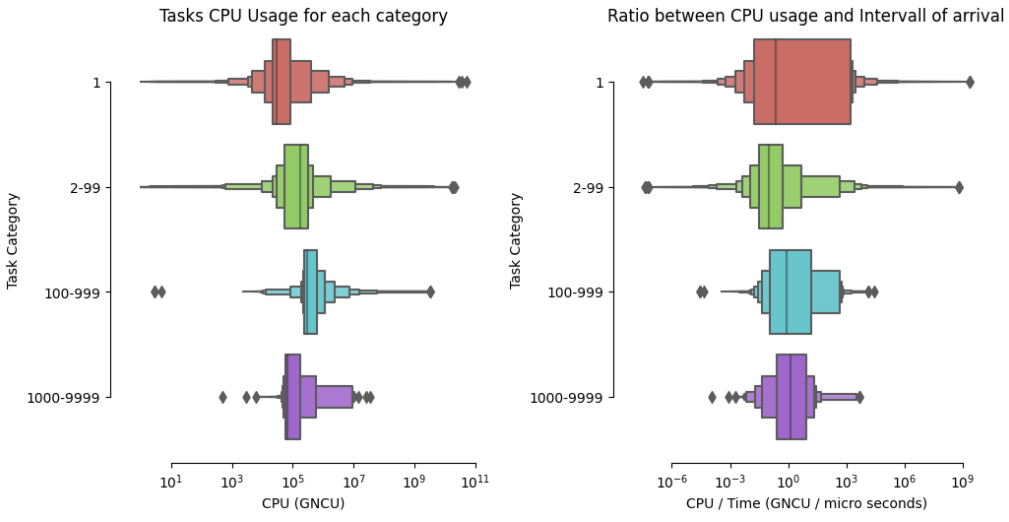
2.1 Formal Description

Scheduling Algorithm: Fixed-priority pre-emptive scheduling with aging

---

repeat
  Take the task with higher priority from the queue
  Give to the selected task a time slot of CPU usage
  while time slot is not ended **or** task is not ended **or** a task with higher priority is not arrived **do**
    Run the selected task
  end while
  if task has ended **then**
    Communicate the end of the task
  else
    Add the task back in queue
  end if
  if time slot is ended **then**
    Age all tasks priority by 1
  end if
until There are task in queue

2.2 Observations and motivations



In both plots we have divided the tasks in 4 main groups based on the number of tasks of their job: it can be seen that in both cases distributions become wider as the number of tasks per job decreases. The left side plot highlights that tasks coming from jobs with low number of tasks are the most CPU-hungry, while the right side plot highlights that those tasks are also the more more inclined to generate queues.
Further analysis shows that this 'trend' also occurs for memory usage and distribution of tasks (most numerous and most frequent)

Our approach to solve the problem was trying to reduce the response time by using a pre-emptive scheduler, which guarantees that no task hogs the CPU for any time longer than a given time: this type of scheduler assigns to the executing task a specific time interval (a time slot) for CPU utilisation, after which it can stop the current task and give the resources to the next one in queue.

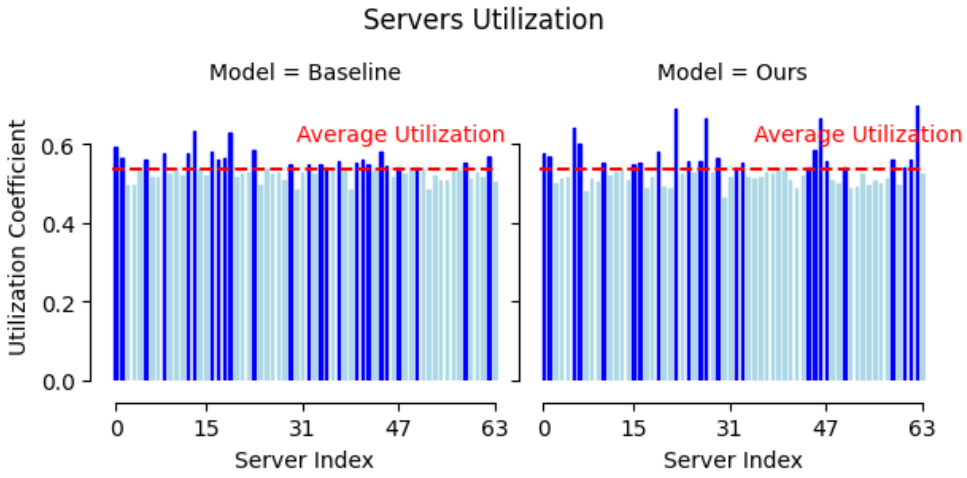
The choice of the time slot value is crucial because it sets the rhythm of the queue: in our simulation we have set its value taking into account the median (more robust with respect to outliers) of the task CPU usage for jobs with one task divided by the server processing power μ.

Based on the results of our EDA, we have chosen to optimize the scheduler to the founded "trend": our algorithm gives more priority to jobs with lower number of tasks. However, this scheduling scheme is particularly vulnerable to *starvation*: since priority is given to most numerous and frequent tasks, the lower-priority tasks could wait an indefinite amount of time. One common method of arbitrating this situation, that we have implemented, is **aging**, which gradually increments (in our case by one) the priority of waiting tasks, ensuring that they will all eventually execute.

3 Plots and conclusions

Model	$\overline{R}$	$\overline{S}$	ρ	$\overline{L}$
Baseline	27602.9038 seconds	1241601.0121	0.5382	129.0 messages
Ours	2906.9292 seconds	117899.8347	0.5382	65.6750 messages

Table 3.1: Comparison among the two models in terms of Response Time($\overline{R}$), Job Slowdown ($\overline{S}$), Mean Utilization Coefficient (ρ) and mean message load ($\overline{L}$). Notice that decimal values are rounded to the fourth digit.



The left side plot about servers utilization shows that, due to the LWL dispatching policy, the weight of the task work is overall uniformly distributed among all servers (with higher peaks in our adaptation). Notice, however, that with our model, although there is equal average server utilisation with the baseline, the number of average messages per job is decreased from 129 to 65.68 (in this case halved because $N_T \simeq 2 \cdot N_J$). With our implementation we menage to decrease both $\overline{R}$ and $\overline{S}$ by 89.47%.

It seems that our implementation outperforms the baseline in terms of slowdown (right plot) both in average and in peak values. In the contrary it seems that for the response time (left plot) our implementation outperforms the baseline only for the average values, this tells us that our algorithm peaks are higher respect to the baseline ones. By looking at the response time distribution (above the eCCDF plot of the Response Time) we can notice that peaks are more frequent in the baseline rather than in our algorithm. This is important because it means that our implementation has a response time distribution more centered over the average values (where it performs the best), while the baseline has a second center of mass on the peaks values. Being those high peaks outliers, they could be caused by a specific interval of time in which there is starvation caused by an high traffic of different priority tasks (which some times happens for the given data). A possible way to solve this behavior is by implementing a more aggressive aging policy (conditioning the aging coefficient on how long a tasks waits in queue), which will make our scheduler handle this type of situations.

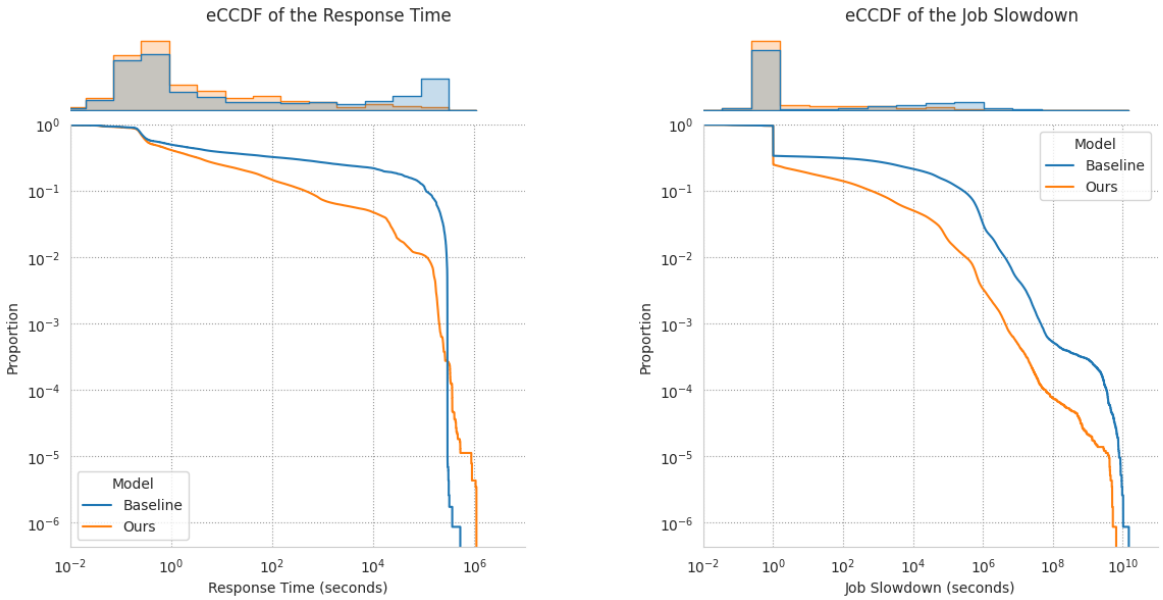


Fig 3.1: eCCDF of Response Time and Job slowdown. We decided to use a log scale on both axis in order to be able to detect the final behaviour of the response time, that was not visible without a logarithmic scaling on the y axis.