Performance Metrics for Scheduling Algorithms in Deadline-Constrained Data Transfer

Takashi Kurimoto
National Institute of Informatics
Tokyo, Japan
tkurimoto@nii.ac.jp

Kohei Shiomoto
Tokyo City University
Tokyo, Japan
shiomoto@tcu.ac.jp

***Abstract*—In this paper, we propose a performance metric in order to evaluate the performance of deadline-aware scheduling methods.By using linear programming to find the ideal off-line scheduling and evaluating the performance of real-time scheduling methods based on ideal scheduling, it is possible to clearly identify the differences from the ideal scheduling.This paper proposes a performance metric that focuses on four evaluation metrics and formulates liner programming to determine optimal off-line scheduling for a baseline for evaluating online scheduling algorithm. It also examines optimal scheduling for both known and unknown job occurrence patterns. Additionally, we report the performance evaluation of EDF algorithm, a representative deadline-aware scheduling method, using this metric.**

***Index Terms*—deadline-aware data transfer, scheduling algorithm, EDF, ILP**

I. Introduction

In recent years, the widespread adoption of IoT services and cloud applications has created a growing demand for efficiently processing diverse jobs in environments where limited computing resources are shared. Particularly in edge computing and cloud operations, there has been an increase in services—such as industrial control systems, sensor data processing, and batch-type analytical tasks—that are premised on being completed within a certain period of time. These services may tolerate some processing delays; however, if deadlines are exceeded, the results may become meaningless or lead to quality degradation or system failures. Therefore, whether a job can be completed within its deadline becomes a decisive factor that affects the reliability and value of the service. Deliver of output data are also required to completed within its deadline (In the following, we refer to such data transfers as deadline-aware data transfers.)

One example of such an application is scientific computing workflows. For instance, the Zwicky Transient Facility [1] generates approximately 1.3 GB of uncompressed data every 45 seconds, which is transmitted to HPC (High-Performance Computing) nodes via a network. These nodes process the data through multiple pipelines to generate alerts, which must be produced within a strict deadline in order to allow for additional observations later the same night. In this case, both the data transfer between nodes and the processing pipelines on the HPC nodes must meet their respective deadlines.

Another application example is the data processing workflows of large-scale cloud service providers' search engines. In web search applications, a new search index must be propagated across data centers every 24 hours. If the transfer of search indexes is delayed, the quality of search responses degraded, so the data must be transferred in accordance with its deadline.

Several scheduling algorithms have been proposed to process deadline-aware data transfers. The Earliest Deadline First (EDF) algorithm [2], [3] being the most representative. And there are several extensions [4]. Evaluations of these deadline-aware algorithms have primarily focused on comparisons with EDF. There are not enough evaluation in comparing performance against ideal scheduling outcomes.

To satisfy all requirements of data transfer deadlines, it is necessary to accurately understand future conditions and optimally scheduling data transfer across the time axis. This problem can be formulated as an optimization problem in cases where the data arrival patterns are known in advance. On the other hand, when the data arrival patterns are not known beforehand, the optimization problem must be solved repeatedly whenever new data arrives. However, since there is no knowledge of the future when solving the optimization problem, the solution obtained may not be optimal as future conditions change.

In this paper, we formulate the problem as an optimization problem in cases where data arrival patterns are known in advance and derive its solution. We also propose a method for applying this approach to cases where the arrival pattern is not known beforehand. Furthermore, we propose the performance metrics using these findings to evaluate the real-time scheduling algorithm. Also, we apply this performance metrics to EDF algorithms.

The rest of this paper is organized as follows. In Section II, we clarify the deadline-aware data transfer scheduling models and its performance metrics. In Section III, we present the mathematical formulation of deadline-aware data transfer scheduling. In Section IV, we resolve a optimization problem for the case that the job arrival pattern within the busy period is known in advance. In Section V, we resolve a optimization problem for the case that the job arrival pattern within the busy period is not known in advance and propose a performance metric for evaluating the deadline-aware scheduling algorithm. In Section VI, we evaluate the performance of EDF algorithm with proposed performance metrics. Section VII concludes this

paper by summarizing the findings.

II. Deadline-Aware Data Transfer Scheduling

A. Scheduling Model Assumed in This Study

In this study, we consider a scheduling model for deadline-aware data transfer of jobs that arise within a computing system. Specifically, three key parameters must be considered in order to processing data transfer while taking deadlines into account:

- (1) Data arrival time
- (2) Processing time required for data transfer
- (3) Deadline for completing the data transfer

Figure 1 illustrates three representative scheduling models for deadline-aware data transfer. In this context, we assume a data arrival pattern involving three job data—J_1, J_2, and J_3—that arrive at times T_1, T_2, and T_3, respectively. The processing time required for data transfer are denoted as L_1, L_2, and L_3, and the corresponding deadlines as D_1, D_2, and D_3.

Figure 1(a) shows an example of scheduling **without preemption**. When job data J_1 arrives, its data transfer begins immediately, and no other job data is processed until J_1's transfer is complete. J_1 completes its transfer at time O_1, and since $O_1 < D_1$, the job meets its deadline. Next, there are two job data J_2 and J_3 and job data J_2 is selected for processing due to its earlier deadline. However, its transfer completes at time O_2, where $O_2 > D_2$, indicating a deadline miss. Subsequently, J_3 is processed and completes at time O_3, which satisfies the deadline condition since $O_3 < D_3$.

Figure 1(b) illustrates a scheduling where **preemption is allowed**. In this model, job data J_1 begins its data transfer first. However, when job J_2 arrives with a deadline earlier than that of J_1 ($D_2 < D_1$), the system preempts J_1 and immediately starts processing J_2. As a result, J_2 completes its transfer at O_2, satisfying the deadline condition ($O_2 < D_2$).

Next, among the remaining job data—J_1 (which is suspended) and J_3 (which is waiting)—J_3 is selected for transfer because it has the earlier deadline. J_3 completes at O_3, and since $O_3 < D_3$, the deadline is met. Finally, the remaining transfer of J_1 is resumed and completed at O_1, satisfying $O_1 < D_1$.

This example shows that for job data like J_1, which require a longer processing time and have a more relaxed deadline, preemption can help prioritize more urgent jobs and ensure that all deadlines are met.

Figure 1(c) presents a scheduling model based on **slot-based preemptive scheduling**. Unlike continuous-time processing, this model divides time into fixed-length intervals (slots), typically ranging from 100 milliseconds to several seconds. At the beginning of each slot, the scheduler selects an appropriate job data from the pending queue for processing within that slot. Because job data selection is repeated at every slot boundary, preemption becomes naturally supported. Once a job data is selected, a fixed-length segment of its data is extracted, encapsulated into a packet, and transmitted over the output link.

In this paper, we focus on the model shown in Figure 1(c), as it is assumed to be relatively straightforward to implement in practical systems.

B. Performance Metrics for Scheduling Policies

To evaluate the effectiveness of scheduling policies, we consider a reward/penalty-based framework. One scenario assumes that if a job's data transfer is not completed within its deadline, the processing becomes meaningless, yielding zero reward. In this case, the dead line is referred to as a *hard deadline* [5], emphasizing strict deadline requirements.

In contrast, in other scenarios, some reward may still be obtained even if the job's data transfer is only partially completed by the deadline. In this case, the deadline is referred to as a *soft deadline* [5], emphasizing loose deadline requirements.

In this study, we examine a total of four evaluation metrics—two variants under the hard deadline model and two under the soft deadline model—as outlined below:

- **Hard Deadline 1**: A unit reward is assigned to each job whose data transfer is successfully completed within its deadline. Accordingly, the performance metric is defined as the total number of such successfully completed jobs.
- **Hard Deadline 2**: If a job data transfer completes within its deadline, the total data transfer time (measured in slots) is counted as the reward. This metric evaluates the sum of the slot counts for all jobs that completed within their deadlines.
- **Soft Deadline 1**: The reward is proportional to the number of time slots during which the job data was transferred prior to its deadline, irrespective of whether the all data was fully transferred within deadline. In contrast to hard deadlines, this approach assigns value to partially transferred data within deadline.
- **Soft Deadline 2**: If a job misses its deadline, the delay (i.e., the number of slots by which it exceeds the deadline) is treated as a penalty. The total penalty incurred is used as the evaluation metric.

III. Formulation of Deadline-Aware data transfer Scheduling

A. Target Job data Set for Optimization

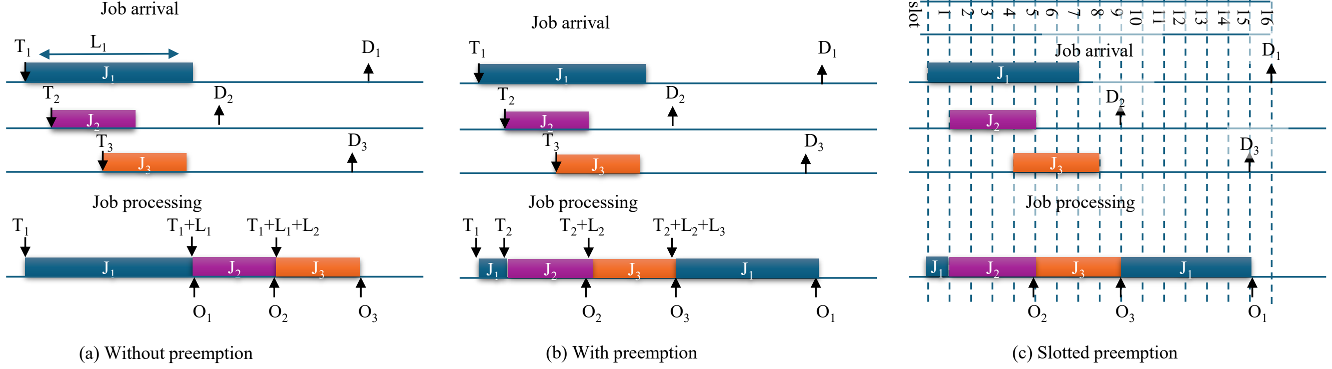
In this study, we focus on optimizing a set of job data that arrive during a *busy period*. The busy period is defined as follows.

Assume that no job data is being processed at time $t = 0$. Let job data J_i $(i = 0, 1, 2, \ldots, N)$ arrive sequentially after $t = 0$, and there are N jobs within the busy period. The following conditions must hold:

$$T_0 + \sum_{i=0}^{k} J_i > T_{k+1} \quad (k = 0, 1, 2, \ldots, N-2) \tag{1}$$

$$T_0 + \sum_{i=0}^{N-1} J_i < T_N \tag{2}$$

Where:



(a) Without preemption (b) With preemption (c) Slotted preemption

Fig. 1: Examples of scheduling models for deadline-aware data transfer: (a) without preemption, (b) with preemption, (c) slotted preemption

- N: Number of jobs in the busy period.
- T_i: Arrival time of job i.
- J_i: Processing time (in slots) required for job i.
- D_i: Deadline for completing the data transfer for job i.

The total duration of the busy period is defined as:

$$busy_period = \sum_{i=0}^{N-1} J_i \tag{3}$$

Each job i has a deadline D_i, which satisfies the following condition:

$$T_i + J_i \leq D_i \tag{4}$$

B. Mathematical Formulation of the Scheduling Model

1) Decision Variables: We define the following binary decision variables:

- $x_{i,j} \in \{0, 1\}$ (Equation 5)

$$x_{i,j} = \begin{cases} 1 & \text{if job } i \text{ data is scheduled at slot } j \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

Here, $i = 0, 1, \ldots, N-1$ and $j = 0, 1, \ldots, busy_period-1$. we call this valuables as job allocation indicator.

- $y_i \in \{0, 1\}$ (Equation 6)

$$y_i = \begin{cases} 0 & \text{if job } i \text{ data completed within its deadline} \\ 1 & \text{otherwise} \end{cases} \tag{6}$$

we call this valuables as deadline satisfaction indicator.

2) Objective Functions: We define four types of objective functions based on the performance metrics introduced in Section II-B

a) Hard Deadline 1:: A unit reward is assigned for each job completed within its deadline. From Equation 6, this corresponds to $1 - y_i = 1$ for completed jobs and 0 otherwise. Therefore, the objective function becomes:

$$\max \sum_{i=0}^{N-1} (1 - y_i) \tag{7}$$

b) Hard Deadline 2:: The reward is the job data length J_i if the job completes within its deadline. This is expressed as:

$$\max \sum_{i=0}^{N-1} (1 - y_i) \cdot J_i \tag{8}$$

c) Soft Deadline 1:: The reward is proportional to the number of slots assigned to the job within its deadline window. Using $x_{i,j}$, the total slot count within deadline is:

$$\max \sum_{i=0}^{N-1} \sum_{j=T_i}^{\min(D_i, busy_period-1)} x_{i,j} \tag{9}$$

d) Soft Deadline 2:: The penalty is the total delay beyond the deadline, i.e., the sum of the delays incurred in slots assigned after the deadline. The total penalty is:

$$\min \sum_{i=0}^{N-1} \sum_{j=D_i+1}^{busy_period-1} x_{i,j} \cdot (j - D_i) \tag{10}$$

3) Constraint Definitions: The scheduling model is subject to the following constraints, denoted as Equations (11)–(14).

a) Allocation Slot Number Constraint: The total number of time slots allocated to job i data transfer must be equal to its processing time J_i:

$$\sum_{j=T_i}^{busy_period-1} x_{i,j} = J_i \quad \forall i \in \{0, \ldots, N-1\} \tag{11}$$

b) Deadline Violated slot number Constraint: If a job data transfer is not fully completed by its deadline D_i, then the number of slots allocated after the deadline must be greater than 0, and cannot exceed the total processing time of the job. while, data transfer is completed within the deadline, the number of slots assigned after the deadline will be equal to 0. These conditions relate the deadline satisfaction indicator y_i to the number of allocated slots beyond the deadline:

$$y_i \leq \sum_{j=D_i+1}^{busy_period-1} x_{i,j} \quad \forall i \in \{0, \ldots, N-1\} \tag{12}$$

$$\sum_{j=D_i+1}^{busy_period-1} x_{i,j} \leq y_i \cdot J_i \quad \forall i \in \{0, \ldots, N-1\} \tag{13}$$

c) Slot Exclusivity Constraint: At most one job can be scheduled in each time slot. That is, each slot j can be assigned to only one job:

$$\sum_{i=0}^{N-1} x_{i,j} = 1 \quad \forall j \in \{0, \ldots, busy_period - 1\} \tag{14}$$

IV. Optimization by Linear Integer Programming

Following the mathematical model formulated in Section III, we perform evaluation using linear programming. In this chapter, we assume that the job arrival pattern within the busy period is known in advance, and optimization calculation is conducted at once referred to as the *optimization-at-once method*. The optimization was implemented using `Python-MIP`, and the corresponding source code is provided in the Appendix.

To validate the correctness of the proposed optimization model, we evaluated it using the following job pattern:

- Job arrival times: $T_i = [0, 1, 4, 6, 13, 17, 19, 24]$
- Processing times: $J_i = [4, 1, 2, 8, 6, 3, 6, 9]$
- Deadlines: $D_i = [5, 2, 8, 16, 20, 21, 27, 36]$

Figure 2 shows the optimization result when using the Hard Deadline 1 objective function. In this example, only one job (Job 4) out of eight fails to meet its deadline. While Job 4 has a deadline of $t = 20$, its data transfer is completed at $t = 38$.

Table I summarizes the evaluation metrics obtained for each of the four objective functions: the number of completed jobs, the total number of completed job slots, the number of in-deadline slots, and the penalty. The numbers shown in parentheses indicate evaluation metrics that were not used as the objective function in the corresponding optimization. From the results, it is evident that each objective function successfully optimizes its respective performance metric.

V. Job Arrival Pattern Is Not Known in Advance

In the previous section, we demonstrated that an optimal solution can be obtained when the job arrival pattern within the busy period is known in advance. This section discusses about more appropriate performance metrics applying for that the arrival pattern is unknown in advance. When the arrival pattern is unknown beforehand, optimization must be recalculated each time a state change occurs. Thus, it is considered the results of optimization recalculated each time when a state change occurs is more appropriate baseline. Here, we refer to this approach as the *sequential-optimization method*.

TABLE I: Evaluation metrics for each objective function

Obj. Func.	Comp. Jobs	Comp. Slots	IN-dead. Slots	Penalty
Hard Dead. 1	6	(33)	(36)	(54)
Hard Dead. 2	(6)	35	(36)	(70)
Soft Dead. 1	(6)	(24)	36	(62)
Soft Dead. 2	(4)	(15)	(30)	19

A. Procedure of Sequential Optimization method

The sequential optimization operates according to the following procedure:

- When an event occurs (either the arrival of a new job or the completion of a job transfer), the system triggers an calculation to update the optimization of scheduling order.
- Jobs are selected and processed for data transfer according to the updated schedule obtained from the optimization.

B. Evaluation Results for Known Pattern

Table II presents the results of applying the sequential optimization method to the same job pattern used in Table I (from Section IV). In comparison to the results of optimization-at-once method, performance degradation is observed in certain metrics. For example, in the case of Hard Deadline 2, the total number of completed slots decreases from 35 to 25, and for Soft Deadline 1, the number of in-deadline slots decreases from 36 to 32. In contrast, for Hard Deadline 1 and Soft Deadline 4, the same level of performance is archived.

This degradation is attributed to the fact that future job arrivals are not considered during decision-making, which may lead to suboptimal job selections.

TABLE II: Evaluation metrics for each objective function using sequential optimization

Obj. Func.	Comp. Jobs	Comp. Slots	IN-dead. Slots	Penalty
Hard Dead. 1	6	(25)	(36)	(54)
Hard Dead. 2	(6)	25	(36)	(70)
Soft Dead. 1	(3)	(16)	32	(54)
Soft Dead. 2	(4)	(15)	(29)	19

C. Comparison Across Random Job Patterns

Next, we compare the performance of optimization-at-once method and the sequential-optimization method across multiple randomly generated job patterns. A total of 12 job patterns were generated using the following statistical distributions:

- **Job arrival intervals:** Poisson distribution with a mean of 0.08, rounded to integers.

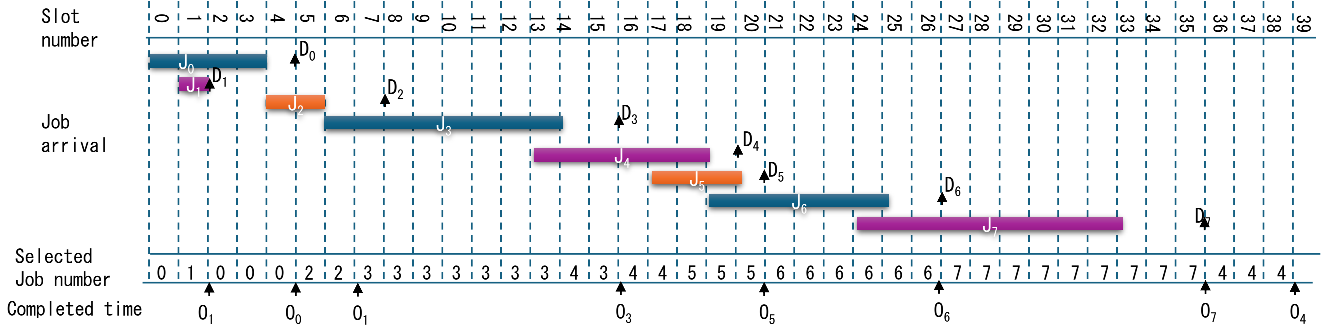


Fig. 2: Optimization result using Hard Deadline 1 as the objective function. Job 4 fails to meet its deadline.

- **Job durations:** Exponential distribution with a mean of 10, rounded to integers.
- **Deadline:** Arrival time + Job duration + 50 (fixed). The additional 50 time units represent a constant deadline margin for all jobs

The computational environment used in this study is summarized below:

- **Operating System:** Ubuntu 20.04.6 LTS
- **Kernel Version:** 5.4.0-216-generic
- **CPU:** Intel(R) Xeon(R) CPU E5-2640 v3 @ 2.60GHz
- **Python-MIP Version:** 1.15.0

D. Performance Comparison Based on Hard Deadline 1

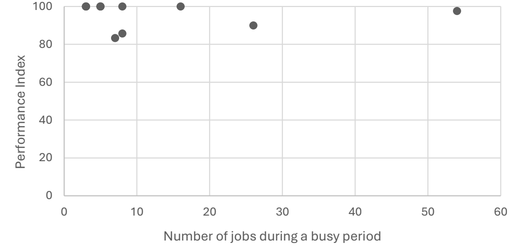
To evaluate the effectiveness of the scheduling strategies, we use **Hard Deadline 1** as the objective function and focus on the number of successfully completed jobs. The following metric (Equation 15) is used to compare the performance of the *sequential-optimization* scheduling method against the *optimization-at-once* scheduling method:

$$\text{Index} = \frac{\text{Metric Value under Sequential optimization}}{\text{Metric Value under optimization at once}} \times 100 \tag{15}$$

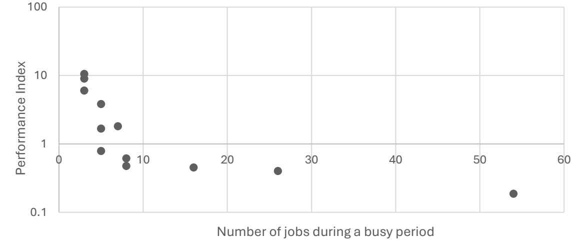
Figure 3 (a) illustrates the trend in the number of completed job number as the number of jobs within a busy period increases. It is observed that the number of completed jobs in the sequential-optimization method is slightly lower compared to the optimization-at-once method. This difference arises due to the lack of foresight in the sequential-optimization method, which cannot account for jobs that will arrive in the future. Nevertheless, the performance degradation remains within approximately 20%, indicating that the sequential-optimization approach is still reasonably effective.

E. Computing Time Comparison

Figure 3 (b) shows the computing time required for both scheduling methods. When the number of jobs within a busy period is small, the sequential-optimization method requires more total computing time. This is primarily due to the higher frequency of optimization calculation executions. On the other hand, when the number of jobs increases, the computing time for the sequential-optimization method decreases. This is because each optimization calculation includes fewer jobs, reducing the complexity of each optimization calculation.



(a) Comparison of Completed Jobs (Hard Deadline 1)



(b) Computation Time Comparison

Fig. 3: Performance comparison between optimization-at-once and sequential-optimization methods

VI. Performance Evaluation of Online Scheduling Algorithms

To clarify the potential improvements in online scheduling algorithm, we compare the performance of the algorithms against the *sequential-optimization method* described in Section V, which serves as our baseline. In this chapter, we focus on the most widely used heuristic—Earliest Deadline First (EDF)—as a benchmark for comparison.

A. Evaluation on a Reference Job Pattern

The performance metrics obtained using EDF for the same job pattern are 6, 15, 29, and 19 for Completed Jobs, Completed Slots, In-deadline Slots, and Penalty, respectively. The

results indicate that EDF achieves comparable performance to the sequential-optimization method in terms of the number of completed jobs and penalty values. However, EDF performs worse with respect to the total slots of completed jobs and in-deadline slots, suggesting that EDF is less effective when evaluated by these metrics.

B. Performance Comparison over Multiple Job Patterns

Next, we evaluate the performance of EDF across multiple randomly generated 12 job patterns (same pattern in Figure 3). The performance is assessed based on two primary metrics: the **number of completed jobs** and the **total penalty value**. The results are summarized in Figure 4.

In terms of the number of completed jobs, EDF performs reasonably well in fewer job number pattern (when the number of jobs within a busy period is fewer than 20), achieving between 40% and 120% of the performance of sequential scheduling. Note that in some cases, EDF surpasses the sequential method (i.e., performance index $>$ 100%) due to the non-optimal job selection in the sequential method when future job arrivals are unknown.

However, as the number of jobs increases beyond 20, EDF's performance degrades significantly, dropping to 20% or even just a few percent. This suggests that EDF is sensitive to job density within the busy period. One possible enhancement is to modify EDF to avoid selecting jobs whose deadlines have already passed, which may help improve its effectiveness in high-load conditions.

When the **penalty** is used as the performance metric, both EDF and sequential scheduling yield nearly equivalent results. Since EDF always prioritizes the job with the earliest deadline, it tends to minimize lateness effectively, achieving close to optimal penalty values.

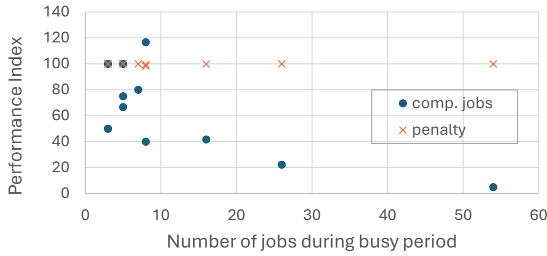


Fig. 4: perfomance evalustion of EDF

VII. Closing Remarks

This paper formulates the deadline-aware job scheduling problem as an integer linear programming (ILP) model aimed at minimizing deadline violations. Assuming full knowledge of job arrivals, time-slot discretization, and preemption support, we derive an ideal solution that serves as a performance upper bound. The efficacy of this approach is benchmarked against the standard Earliest Deadline First (EDF) algorithm, revealing its advantages and limitations under various load conditions.

Acknowledgement

This work was partially supported by JST, CRONOS, Japan Grant Number JPMJCS24N9.

References

[1] Eric C Bellm et.al. The zwicky transient facility: System overview, performance, and first results. *Publications of the Astronomical Society of the Pacific*, 131(995):018002, December 2018.

[2] Maen Saleh and Liang Dong. Comparing fcfs edf scheduling algorithms for real-time packet switching networks. In *2010 International Conference on Networking, Sensing and Control (ICNSC)*, pages 698–703, 2010.

[3] D. Liu and Y.-H. Lee. An efficient scheduling discipline for packet switching networks using earliest deadline first round robin*. In *Proceedings. 12th International Conference on Computer Communications and Networks (IEEE Cat. No.03EX712)*, pages 5–10, 2003.

[4] Zhoujia Mao, Can Emre Koksal, and Ness B. Shroff. Optimal online scheduling with arbitrary hard deadlines in multihop communication networks. *IEEE/ACM Transactions on Networking*, 24(1):177–189, 2016.

[5] Arezou Mohammadi, Selim G. Akl, and Firouz Behnamfar. Optimal linear-time algorithm for uplink scheduling of packets with hard or soft deadlines in wimax. In *2008 IEEE 68th Vehicular Technology Conference*, pages 1–5, 2008.

Appendix

Listing 1: Python-MIP Model Code

```
from mip import Model, xsum, BINARY, maximize, minimize
# Define parameters
n = len(job_len)
end_time = sum(job_len[i] for i in range(n))
# Create model
model = Model()
# Define decision variables
x = [[model.add_var(var_type=BINARY) for j in range(
   end_time)] for i in range(n)]
y = [model.add_var(var_type=BINARY) for i in range(n)]
# Objective Function (Select one of the following)
# 1. Hard Deadline 1: Maximize number of completed jobs
model.objective = maximize(xsum((1 - y[i]) for i in range(n
   )))
# 2. Hard Deadline 2: Maximize total slot time of completed
    jobs
# model.objective = maximize(xsum((1 - y[i]) * job_len[i]
   for i in range(n)))
# 3. Soft Deadline 1: Maximize slots used before deadline
# model.objective = maximize(xsum(x[i][j]
#      for i in range(n)
#      for j in range(0, min(dead_line[i] + 1, end_time))))
# 4. Soft Deadline 2: Minimize late penalties (slots used
   after deadline)
# model.objective = minimize(xsum(x[i][j] * (j - dead_line[
   i])
#      for i in range(n)
#      for j in range(dead_line[i] + 1, end_time)))
# Constraints
for i in range(n):
    # Total slots assigned to job i must equal its length
    model += xsum(x[i][j] for j in range(arr_time[i],
       end_time)) == job_len[i]
    # Define deadline miss flag y[i]
    model += y[i] <= xsum(x[i][j] for j in range(dead_line[
       i], end_time))
    model += xsum(x[i][j] for j in range(dead_line[i],
       end_time)) <= y[i] * job_len[i]
for j in range(end_time):
    # Only one job can be processed in each slot
    model += xsum(x[i][j] for i in range(n)) == 1
# Run optimization
model.optimize()
```