Competitive Analysis of Task Scheduling Algorithms on a Fault-Prone Machine and the Impact of Resource Augmentation

Antonio Fernández Anta[a], Chryssis Georgiou[b], Dariusz R. Kowalski[c], Elli Zavou[a,d,1]

[a] *Institute IMDEA Networks*
[b] *University of Cyprus*
[c] *University of Liverpool*
[d] *Universidad Carlos III de Madrid*

Abstract

Reliable task execution in machines that are prone to unpredictable crashes and restarts is challenging and of high importance. However, not much work exists on the worst case analysis of such systems. In this paper, we analyze the fault-tolerant properties of four popular scheduling algorithms: Longest In System (LIS), Shortest In System (SIS), Largest Processing Time (LPT) and Shortest Processing Time (SPT), under worst case scenarios on a fault-prone machine. We use three metrics for the evaluation and comparison of their competitive performance, namely, completed time, pending time and latency. We also investigate the effect of resource augmentation in their performance, by increasing the speed of the machine. To do so, we compare the behavior of the algorithms for different speed intervals and show that between LIS, SIS and SPT there is no clear winner with respect to all the three considered metrics, while LPT is not better than SPT.

Keywords: Scheduling, Online Algorithms, Different Task Processing Times, Failures, Competitive Analysis, Resource Augmentation

1. Introduction

Motivation. The demand for processing computationally-intensive jobs has been increasing dramatically during the last decades, and so has the research to face the many challenges it presents. In addition, with the presence of machine failures (and restarts) things get even worse, especially when a malicious entity is causing them. In this work, we apply *speed augmentation* [9, 1] in order to overcome such failures, that is, we increase the computational power of the system's machine. This is an alternative to increasing the number of processing entities, as done in multiprocessor systems. Hence, we consider a speedup $s \geq 1$, under which the machine performs a job s times faster than the baseline execution time.

More precisely, we consider a setting with a single *machine* prone to crashes and restarts that are being controlled by an adversary, and a *scheduler* that assigns injected jobs (or *tasks*) to be executed by the machine. These tasks arrive continuously and have different computational demands and hence *processing time*. Specifically we assume that each task has processing time $\pi \in [\pi_{min}, \pi_{max}]$, where π_{min} and π_{max} are the smallest and largest values, respectively. Since the scheduling decisions must be made continuously and without knowledge of the future, neither the task injections nor the machine crashes and restarts, we look at the problem as an *online* scheduling problem [10, 4, 3, 12, 14]. The importance of using speedup lies on this online nature of the problem; the future failures, and the

[1] Partially supported by FPU Grant from MECD

instant of arrival of future new tasks and their processing time are unpredictable. Thus, the need to overcome this lack of information. We evaluate the performance of the different scheduling policies (*online algorithms*) under *worst-case* scenarios, which guarantee efficient scheduling even in the worst of cases. For that, we perform competitive analysis [13]. The four scheduling policies we consider are *Longest In System* (LIS), *Shortest In System* (SIS), *Largest Processing Time* (LPT) and *Shortest Processing Time* (SPT). Achieving reliable and stable computations in such an environment withholds several challenges. One of our main goals is therefore to confront these challenges considering the use of the smallest possible speedup. However, our primary intention is to discover the dependence of the efficiency measures for each scheduling policy with respect to the speedup used.

Contributions. In this paper we explore the behavior of some of the most widely used algorithms in scheduling, analyzing their fault-tolerant properties under worst-case combination of task injection and crash/restart patterns, as described above. As already mentioned, the four algorithms we consider are:
(1) LIS: the task that has been waiting the longest is scheduled; i.e., it follows the FIFO *(First In First Out)* policy,
(2) SIS: the task that has been injected the latest is scheduled; i.e., it follows the LIFO *(Last In First Out)* policy,
(3) LPT: the task with the longest processing time is scheduled, and
(4) SPT: the task with the shortest processing time is scheduled.

We focus on three *evaluation metrics*, which we regard to embody the most important quality of service parameters: the *completed time,* which is the aggregate processing time of all the tasks that have completed their execution successfully, the *pending time,* which is the aggregate processing time of all the tasks that are in the queue waiting to be completed, and the *latency,* which is the largest time a task spends in the system, from the time of its arrival until it is fully executed. They represent the machine's throughput, queue size and delay respectively, all of which we consider essential. They show how efficient the scheduling algorithms are in a fault-prone setting from different angles: machine utilization (completed time), buffering (pending time) and fairness (latency). The performance of an algorithm ALG is evaluated under these three metrics by means of competitive analysis, in which the value of the metric achieved by ALG when the machine uses speedup $s \geq 1$ is compared with the best value achieved by any algorithm X running without speedup ($s = 1$) under the same pattern of task arrivals and machine failures, at all time instants of an execution.

Table 1 summarizes the results we have obtained for the four algorithms. Although not clearly stated in the table, the first results we show in our work apply to *any work conserving scheduling algorithm* – ones that do not idle as long as there are pending tasks and they do not break the execution of a task unless the machine crashes. These results are: (a) When $s \geq \rho = \frac{\pi_{max}}{\pi_{min}}$, the completed time competitive ratio is lower bounded by $1/\rho$ and the pending time competitive ratio is upper bounded by ρ. (b) When $s \geq 1 + \rho$, the completed time competitive ratio is lower bounded by 1 and the pending time competitive ratio is upper bounded by 1 (i.e., they are 1-competitive). Then, for specific cases of speedup less than $1 + \rho$ we obtain better lower and upper bounds for the different algorithms.

However, it is clear that none of the algorithms is better than the rest. With the exception of SPT, no algorithm is competitive in any of the three metrics considered when $s < \rho$. In particular, algorithm SPT is competitive in terms of completed time, except in the case when tasks may have any arbitrary processing time in the range $[\pi_{min}, \pi_{max}]$ and the machine has no speedup ($s = 1$). In terms of latency, only algorithm LIS is competitive, when $s \geq \rho$, which might not be very surprising since algorithm LIS gives priority to the tasks that have been waiting the longest in the system. Another interesting observation is that algorithms LPT and SPT become 1-competitive as soon as $s \geq \rho$, both in terms of completed and pending time, whereas LIS and SIS require greater speedup to achieve this.

What is more, this is the first thorough and rigorous online analysis of popular scheduling algorithms in a fault-prone setting. In some sense, collectively our results demonstrate in a clear way the differences between two classes of policies: the ones that give priority based on the *arrival time* of the tasks in the system (LIS and SIS) and the ones

	Condition	Completed Time, $\mathcal{C}$	Pending Time, $\mathcal{P}$	Latency, $\mathcal{L}$
LIS	$s < \rho$	0	∞	∞
	$s \in [\rho, 1 + 1/\rho)$	$[\frac{1}{\rho}, \frac{1}{2} + \frac{1}{2\rho}]$	$[\frac{1+\rho}{2}, \rho]$	$(0, 1]$
	$s \in [\max\{\rho, 1 + 1/\rho\}, 2)$	$[\frac{1}{\rho}, \frac{1}{2} + \frac{\pi_{min}}{2\pi}]$	$[\frac{1}{2} + \frac{\pi}{2\pi_{min}}, \rho]$	$(0, 1]$
	$s \geq \max\{\rho, 2\}$	$[1, s]$	1	$(0, 1]$
SIS	$s < \rho$	0	∞	∞
	$s \in [\rho, 1 + \rho)$	$[\frac{1}{\rho}, \frac{\pi_{min}}{\pi'}]$	$[\frac{\pi'/\pi_{min}+\rho}{1+\rho}, \rho]$	∞
	$s \geq 1 + \rho$	$[1, s]$	1	∞
LPT	$s < \rho$	0	∞	∞
	$s \geq \rho$	$[1, s]$	1	∞
SPT	$s < \rho$	$[\frac{1}{2+\rho}, \frac{\hat{\rho}}{\hat{\rho}+\rho}]$ (*)	∞	∞
	$s \geq \rho$	$[1, s]$	1	∞

Table 1: Metrics comparison of the four scheduling algorithms. Recall that s represents the speedup of the system's machine, π_{max} and π_{min} the largest and smallest task processing times respectively. Note that $\rho = \frac{\pi_{max}}{\pi_{min}}$, $\hat{\rho} = \lceil \rho \rceil - 1$ and $\pi, \pi' \in (\pi_{min}, \pi_{max})$ are task processing times such that $\pi < \frac{\pi_{min}}{s-1}$ and $\pi' < \frac{\pi_{min}+\pi_{max}}{s}$. Note also that by definition, 0-completed-time competitiveness ratio equals to non-competitiveness, as opposed to the other two metrics where non-competitiveness corresponds to an ∞ competitiveness ratio. Finally note that the lower bound of result (*) is for injection patterns with tasks of only two processing times π_{min} and π_{max}.

that give priority based on the required *processing time* of the tasks (LPT and SPT). Observe that different algorithms scale differently with respect to the speedup, in the sense that with the increase of the machine speed the competitive performance of each algorithm changes in a different way.

Related Work. We relate our work with the online version of the *bin packing* problem [15], where the objects to be packed are the tasks and the bins are the time periods between two consecutive failures of the machine (i.e., *alive* intervals). Wide research has taken place over the years around this problem, some of which we consider related to ours. For example, Johnson et al. [8] analyzed the worst case performance of two simple algorithms (Best Fit and Next Fit) for the bin packing problem, giving upper bounds on the number of bins needed (corresponding to the completed time in our work). Epstein et al. [6] (see also [15]) considered online bin packing with resource augmentation in the size of the bins (corresponding to the length of alive intervals in our work). Observe that the essential difference of the online bin packing problem with the one that we are looking at in this work, is that in our system the bins and their sizes (corresponding to the machine's alive intervals) are unknown.

On a different tone, Boyar and Ellen [5] have looked into a problem similar to both online bin packing problem and ours, considering job scheduling in the grid. The main difference with our setting is that they consider several machines (or processors), but mainly the fact that the arriving items are processors with limited memory capacities and there is a fixed amount of jobs in the system that must be completed. They also use fixed job sizes and achieve lower and upper bounds that only depend on the fraction of such jobs in the system.

Another related problem is packet scheduling in a link. Andrews and Zhang [2] consider online packet scheduling over a wireless channel whose rate varies dynamically, and perform worst case analysis regarding both the channel conditions and the packet arrivals. A similar work is [3], where packets of two different sizes were scheduled through an unreliable link. In that work, the goodness metric is the long-term competitive ratio, which is called *relative throughput* and online algorithms as well as bounds for any online scheduling protocol are given.

We can also directly relate our work with research done on machine scheduling with availability constraints (e.g., [11, 7]). One of the most important results in that area is the necessity of online algorithms in case of unex-

pected machine breakdowns. However, in most related works preemptive scheduling is considered and optimality is shown only for nearly online algorithms (need to know the time of the next job or machine availability).

In a previous work [4], we looked at a system of multiple machines and at least two different task costs (i.e., processing times in our work). We applied distributed scheduling and performed worst-case competitive analysis, considering the pending cost competitiveness (corresponding pending time competitiveness in the present work) as our main evaluation metric. We showed the NP-hardness of the offline version of the problem and suggested the use of *speedup* in order to achieve competitiveness. We defined $\rho = \frac{\pi_{max}}{\pi_{min}}$ and proved that if both conditions (a) $s < \rho$ and (b) $s < 1 + \gamma/\rho$ hold for the system's machines (γ is some constant that depends on π_{min} and π_{max}), then *no* deterministic algorithm is competitive with respect to the queue size (pending cost). Additionally, we proposed online algorithms to show that relaxing any of the two conditions is sufficient to achieve competitiveness. In fact, [4] motivated this paper, since it made evident the need of a thorough study of simple algorithms even under the simplest basic model of one machine and scheduler.

2. Model and Definitions

Computing Setting. We consider a system of one machine prone to crashes and restarts with a *Scheduler* responsible for the task assignment to the machine following some algorithm. The clients submit jobs (or *tasks*) of different sizes (processing time) to the scheduler, which in its turn assigns them to be executed by the machine.

Tasks. Tasks are injected to the scheduler by the clients of the system, an operation which is controlled by an arrival pattern A (a sequence of task injections). Each task τ has an *arrival time* $a(\tau)$ and a *processing time* $\pi(\tau)$, being the time it requires to be completed by a machine running with $s = 1$. We use the term π-task to refer to a task of processing time $\pi \in [\pi_{min}, \pi_{max}]$ throughout the paper. We also assume tasks to be *atomic* with respect to their completion; in other words preemption is not allowed (tasks must be fully executed without interruptions).

Machine failures. The crashes and restarts of the machine are controlled by an error pattern E, which we assume to coordinate with the arrival pattern in order to give the worst-case scenarios. We consider that the task being executed at the time of the machine's failure is not completed, and it is therefore still pending in the scheduler until it is eventually re-scheduled (it is not discarded). The machine is *active* in the time interval $[t, t^*]$, if it is executing some task at time t and has not crashed by time t^*. Hence, an error pattern E can be seen as a sequence of active intervals of the machine.

Resource augmentation / Speedup. We also consider a form of resource augmentation by speeding up the machine and the goal is to keep it as low as possible. As mentioned earlier, we denote the speedup with $s \geq 1$.

Notation. Let us denote here some notation that will be extensively used throughout the paper. Because it is essential to clarify the tasks being accounted at each timepoint in an execution, we introduce sets $I_t(A), N_t^s(X, A, E)$ and $Q_t^s(X, A, E)$, where X is an algorithm, A and E the arrival and error patterns respectively, t the time instant we are looking at and s the speedup of the machine. $I_t(A)$ represents the set of injected tasks within the interval $[0, t]$, $N_t^s(X, A, E)$ the set of completed tasks within $[0, t]$ and $Q_t^s(X, A, E)$ the set of pending tasks at time instant t. $Q_t^s(X, A, E)$ contains the tasks that were injected by time t inclusively, but not the ones completed before and up to time t. Observe that $I_t(A) = N_t^s(X, A, E) \cup Q_t^s(X, A, E)$ and note that set I depends only on the arrival pattern A, while sets N and Q also depend on the error pattern E, the algorithm run by the scheduler, X, and the speedup of the machine, s. Note that the superscipt s is omitted in further sections of the paper for simplicity. However, the appropriate speedup in each case is clearly stated.

Efficiency Measures. Considering an algorithm ALG running with speedup s under arrival and error patterns A and E, we look at the current time t and focus on three measures; the *Completed Time*, which is the sum of processing times of the completed tasks

$$C_t^s(\text{ALG}, A, E) = \sum_{\tau \in N_t^s(\text{ALG}, A, E)} \pi(\tau),$$

the *Pending Time*, which is the sum of processing times of the pending tasks

$$P_t^s(\text{ALG}, A, E) = \sum_{\tau \in Q_t^s(\text{ALG}, A, E)} \pi(\tau),$$

and the *Latency*, which is the maximum amount of time a task has spent in the system

$$L_t^s(\text{ALG}, A, E) = \max \left\{ \begin{matrix} f(\tau) - a(\tau), & \forall \tau \in N_t^s(\text{ALG}, A, E) \\ t - a(\tau), & \forall \tau \in Q_t^s(\text{ALG}, A, E) \end{matrix} \right\},$$

where $f(\tau)$ is the time of completion of task τ. Computing the schedule (and hence finding the algorithm) that minimizes or maximizes correspondingly the measures $C_t^s(X, A, E)$, $P_t^s(X, A, E)$, and $L_t^s(X, A, E)$ offline (having the knowledge of the patterns A and E), is an NP-hard problem [4].

Due to the dynamicity of the task arrivals and machine failures, we view the scheduling of tasks as an online problem and pursue *competitive analysis* using the three metrics. Note that for each metric, we consider any time t of an execution, combinations of arrival and error patterns A and E, and any algorithm X designed to solve the scheduling problem:

An algorithm ALG running with speedup s, is considered α-*completed-time-competitive* if $C_t^s(\text{ALG}, A, E) \geq \alpha \cdot C_t^1(X, A, E) + \Delta_C$, for some parameter Δ_C that *does not* depend on t, X, A or E; α is the completed-time competitive ratio of ALG, which we denote by $\mathcal{C}(\text{ALG})$.

Similarly, it is considered α-*pending-time-competitive* if $P_t^s(\text{ALG}, A, E) \leq \alpha \cdot P_t^1(X, A, E) + \Delta_P$, for parameter Δ_P which does not depend on t, X, A or E. In this case, α is the pending-time competitive ratio of ALG, which we denote by $\mathcal{P}(\text{ALG})$.

It is also considered α-*latency-competitive* if $L_t^s(\text{ALG}, A, E) \leq \alpha \cdot L_t^1(X, A, E) + \Delta_L$, where Δ_L is a parameter independent of t, X, A and E. In this case, α is the latency competitive ratio of ALG, which we denote by $\mathcal{L}(\text{ALG})$. Finally, note that α, is independent of t, X, A and E, for the three metrics accordingly.[2]

Both completed and pending time measures are important. Observe that they are not complementary of one another. An algorithm may be completed-time-competitive but not pending-time-competitive, even though the sum of processing times of the successfully completed tasks complements the sum of processing times of the pending ones. For example, think of an online algorithm that manages to complete successfully half of the total injected task processing times up to any point in any execution. This gives a completed time competitiveness ratio $\mathcal{C}(\text{ALG}) = 1/2$. However, it is not necessarily pending-time-competitive since in an execution with infinite task arrivals its total pending time increases unboundedly and there might exist an algorithm X that manages to keep its total pending time constant under the same arrival and error patterns. This is further demonstrated by our results summarized in Table 1.

3. Properties of Work Conserving Algorithms

In this section we present some general properties for all online work-conserving algorithms running with speedup s. They also apply to the four policies we focus on in the rest of the paper.

Observation 1. *Any work-conserving algorithm ALG running with speedup s, has a completed-time competitive ratio $\mathcal{C}(ALG) \leq 1$ when we allow $Q_t(ALG) = \emptyset$ infinitely many times, or $\mathcal{C}(ALG) \leq s$ when the queue never becomes empty after a point in time.*

[2]Parameters $\Delta_C, \Delta_P, \Delta_L$ as well as α may depend on system parameters like π_{min}, π_{max} or s, which are not considered as inputs of the problem.

Proof: Let us first consider the case at which the adversary allows the queue of pending tasks of ALG to become empty infinitely many times in an execution. In particular, let us consider the arrival and error patterns A and E, such that there are time instants $t_k, k = 0, 1, 2 \ldots$ where $t_k = t_{k-1} + \pi$ and $t_0 = 0$. At each t_k there is a machine failure (crash and restart) and exactly one π-task ($\pi \in [\pi_{min}, \pi_{max}]$) injected. We name time interval $T_i = [t_i, t_{i+1}]$. Observe that an algorithm X (running with $s = 1$) completes π-task injected at t_i in interval T_i, while any work-conserving algorithm ALG running with speedup s will complete the same task at time $t_i + \pi/s < t_{i+1}$ resulting in an empty queue. Hence, $\mathcal{C}(\text{ALG}) = 1$ as claimed.

Let us now consider the case at which the adversary never lets the queue of ALG to become empty after some point in time. More precisely we consider the arrival and error patterns A' and E' to be such that ALG always has at least one pending task of any processing time $\pi \in [\pi_{min}, \pi_{max}]$ available to schedule. Therefore, in a period of time π an algorithm $X \in \mathcal{X}$ (running with $s = 1$) is able to complete a π-task while ALG may complete up to πs processing time. This will result in a completed time competitive ratio $\mathcal{C}(\text{ALG}) \leq s$ as claimed. ■

Observation 2. *Any work-conserving algorithm ALG running with speedup s, has a pending-time competitive ratio $\mathcal{P}(ALG) \geq 1$ when its queue of pending tasks never becomes empty after a point in time.*

Proof: Let us consider arrival and error patterns A and E such that algorithm ALG always has at least one pending task of any processing time $\pi \in [\pi_{min}, \pi_{max}]$ available to schedule. We consider phases of arbitrarily chosen lengths π, defined as intervals $T_i = [t_k, t_{k+1}]$ where $t_{k+1} = t_k + \pi$, $t_0 = 0$ and $k = 0, 1, 2 \ldots$ being instants of machine failures. As a result, in a phase of length π an algorithm $X \in \mathcal{X}$ will be able to complete a π-task, while ALG will complete up to πs processing time. Assuming that there are no phases of length less than π_{min}, the complementing pending time at a time t_k will therefore be $P_{t_k}(X) \geq I_{t_k}(A) - t_k$ and $P_{t_k}(\text{ALG}) \geq I_{t_k}(A) - t_k s$. The pending time competitive ratio becomes $\mathcal{P}(\text{ALG}) \geq \frac{I(A)-ts}{I(A)-t}$, which yields to $\mathcal{P}(\text{ALG}) \geq 1$, since we can make $I(A)$ infinitely big. ■

Lemma 1. *No algorithm X (running without speedup) completes more tasks than a work conserving algorithm ALG running with speedup $s \geq \rho$. Formally, $\forall t, A \in \mathcal{A}$ and $E \in \mathcal{E}$, $|N_t(ALG, A, E)| \geq |N_t(X, A, E)|$, and hence $|Q_t(ALG, A, E)| \leq |Q_t(X, A, E)|$.*

Proof: We will prove that $\forall t, A \in \mathcal{A}$ and $E \in \mathcal{E}$, $|Q_t(\text{ALG}, A, E)| \leq |Q_t(X, A, E)|$, which implies that $|N_t(\text{ALG}, A, E)| \geq |N_t(X, A, E)|$. Observe that the claim trivially holds for $t = 0$. We now use induction on t to prove the general case. Consider any time $t > 0$ and the closest time $t' < t$ such that t' is either a failure/restart time point or a point where ALG's pending queue is empty. Let us assume by induction hypothesis that $|Q_{t'}(\text{ALG})| \leq |Q_{t'}(X)|$.

Let i_T be the number of tasks injected in the interval $T = (t', t]$. Since ALG is work conserving, it is continuously executing tasks in the interval T. Also, ALG needs at most $\pi_{max}/s \leq \pi_{min}$ time to execute any task using speedup $s \geq \rho$, regardless of the task being executed. Then it holds that

$$|Q_t(\text{ALG})| \leq |Q_{t'}(\text{ALG})| + i_T - \left\lfloor \frac{t - t'}{\pi_{max}/s} \right\rfloor \leq |Q_{t'}(\text{ALG})| + i_T - \left\lfloor \frac{t - t'}{\pi_{min}} \right\rfloor .$$

On the other hand, X can complete at most one task every π_{min} time. Hence,

$$|Q_t(X)| \quad \geq \quad |Q_{t'}(X)| + i_T - \left\lfloor \frac{t - t'}{\pi_{min}} \right\rfloor .$$

As a result, we have that

$$|Q_t(X)| - |Q_t(\text{ALG})| \quad \geq \quad |Q_{t'}(X)| + i_T - \left\lfloor \frac{t-t'}{\pi_{min}} \right\rfloor - |Q_{t'}(\text{ALG})| - i_T + \left\lfloor \frac{t-t'}{\pi_{min}} \right\rfloor \geq 0.$$

Since this holds for all times t, the claim follows. ■

Theorem 1. *Any work-conserving algorithm ALG running with speedup $s \geq \rho$ has completed-time competitive ratio $\mathcal{C}(ALG) \geq 1/\rho$ and pending-time competitive ratio $\mathcal{P}(ALG) \leq \rho$.*

Proof: It follows directly from Lemma 1, since for any algorithm X the processing time of every task completed by ALG is at least $\frac{\pi_{min}}{\pi_{max}}$ times the processing time of every task completed by X. From Lemma 1, ALG always has at least as many completed tasks as X, and hence its completed-time competitive ratio $\mathcal{C}(\text{ALG})$ is at least $\frac{\pi_{min}}{\pi_{max}} = 1/\rho$. Complementary, since the processing time of any pending task in the queue of ALG is at most $\frac{\pi_{max}}{\pi_{min}}$ times bigger than the processing time of any pending task in X, its pending-time competitive ratio $\mathcal{P}(\text{ALG})$ is at most $\frac{\pi_{max}}{\pi_{min}} = \rho$. ■

Theorem 2. *Any work-conserving algorithm ALG running with speedup $s \geq 1 + \rho$, has completed-time competitive ratio $\mathcal{C}(ALG) \geq 1$ and pending-time competitive ratio $\mathcal{P}(ALG) \leq 1$.*

Proof: Consider an execution of any work-conserving algorithm ALG running with speedup $s \geq 1 + \rho$ under any arrival and error patterns A and E, as well as an algorithm X. Then, looking at any time t of an execution, we define time instant $t' < t$ to be the latest time before t at which one of the following events happens: (1) an *active period* starts (after a machine crash/restart), (2) algorithm X has successfully completed a task, or (3) the queue of pending tasks of ALG is empty, $Q_{t'}(\text{ALG}) = \emptyset$.

It is trivial that $P_0(\text{ALG}, A, E) \leq P_0(X, A, E)$ holds at the beginning of the executions. Now assuming that $P_{t'}(\text{ALG}, A, E) \leq P_{t'}(X, A, E)$ holds at time t', we prove by induction that $P_t(\text{ALG}, A, E) \leq P_t(X, A, E)$ still holds at time t. This also means that the tasks successfully completed by ALG by time t have at least the same total processing time as the ones completed by X.

Considering the interval $T = (t', t]$, there are two cases:

Case 1: X is not able to complete any task in the interval T. Then, it holds that $P_t(X, A, E) = P_{t'}(X, A, E) + i_T$, where i_T denotes the processing time of the tasks injected during the interval T. Similarly, it holds that $P_t(\text{ALG}, A, E) \leq P_{t'}(\text{ALG}, A, E) + i_T$ even if ALG is not able to complete successfully any task in T.

Case 2: X completes successfully a task in the interval T. Note that by definition of time t', during interval T there can only be one task completed by X, and it must be completed at time t. (If that were not the case, t' would not be well defined.) There are two subcases.

(a) First, t' is from case (3) of its definition. Hence, $Q_{t'}(\text{ALG}) = \emptyset$ and $P_t(\text{ALG}, A, E) \leq i_T$. At time t' algorithm X was executing the task that was completed at time t. Hence, the task was injected before t', and X has not completed any of the tasks injected in T. Then, $P_t(X, A, E) \geq i_T \geq P_t(\text{ALG}, A, E)$.

(b) Second, t' is from cases (1) or (2) of its definition. Then, the interval T has length $\pi \geq \pi_{min}$, which is the processing time of the task completed by X. In that interval ALG is continuously executing tasks. Hence, in the interval $(t', t]$ if completes tasks whose aggregate processing time is at least $\pi s - \pi_{max}$. Then, the pending time at time instant t of both algorithms satisfy $P_t(X, A, E) = P_{t'}(X, A, E) + i_T - \pi$ while $P_t(\text{ALG}, A, E) < P_{t'}(\text{ALG}, A, E) + i_T - (\pi s - \pi_{max})$. Observe that $s \geq 1 + \rho$ implies that $\pi \leq \pi s - \pi_{max}$. Hence, from the induction hypothesis, $P_t(\text{ALG}, A, E) \leq P_t(X, A, E)$.

This implies a completed-time competitive ratio $\mathcal{C}(\text{ALG}) \geq 1$ and a pending-time competitive ratio $\mathcal{P}(\text{ALG}) \leq 1$, as claimed. ■

4. Completed and Pending Time Competitiveness

In this section we present a detailed analysis of the four algorithms with respect to the completed and pending time metrics, first for speedup $s < \rho$ and then for speedup $s \geq \rho$.

4.1. Speedup $s < \rho$

Let us start with some negative results, whose proofs involve specifying the combinations of arrival and error patterns that force the claimed *bad* performances of the algorithms. We also give some positive results for SPT, the only algorithm that can achieve a non-zero completed-time competitiveness under some cases.

Lemma 2. *When algorithms LIS and LPT run with speedup $s < \rho$, they both have a completed-time competitive ratio $\mathcal{C}(LIS) = \mathcal{C}(LPT) = 0$ and a pending-time competitive ratio $\mathcal{P}(LIS) = \mathcal{P}(LPT) = \infty$.*

Proof: Let us use the same combination of algorithm X, arrival and error patterns A and E for the proof of non-competitiveness for the two algorithms. We consider an infinite arrival pattern which injects one π_{max}-task at the beginning of the execution, $t = 0$, and after that it keeps injecting one π_{min}-task every π_{min} time. Consider also an infinite error pattern that sets the machine failure points (crash immediately followed by a restart) at time instants $t_i = i \cdot \pi_{min}$, where $i = 1, 2, \ldots$.

It can be easily seen, that an algorithm X running with no speedup ($s = 1$), will be able to complete the π_{min}-tasks injected, while neither LIS nor LPT will manage to complete any task, since they will both insist on scheduling the π_{max}-task injected at the beginning. In an interval of length π_{min}, algorithm X is able to complete a π_{min}-task but neither LIS nor LPT can complete the π_{max}-task since it needs time $\frac{\pi_{max}}{s} > \pi_{min}$. This means that the number of pending tasks in the queues of both LIS and LPT will be continuously increasing with time, while X is able to keep them bounded, with no more than one π_{max} and one π_{min} tasks, and so will their pending processing time. As for the total processing time of completed tasks, $C(\text{LIS}, A, E) = C(\text{LPT}, A, E) = 0$ at all times of the execution, while the one of X grows to infinite as t goes to infinity.

Hence, for speedup $s < \rho$, algorithms LIS and LPT have completed-time competitive ratios $\mathcal{C}(\text{LIS}) = \mathcal{C}(\text{LPT}) = 0$ and pending-time competitive ratios $\mathcal{P}(\text{LIS}) = \mathcal{P}(\text{LPT}) = \infty$ as claimed, which completes the proof. ■

Lemma 3. *When algorithm SIS runs with speedup $s < \rho$, it has a completed-time competitive ratio $\mathcal{C}(SIS) = 0$ and a pending-time competitive ratio $\mathcal{P}(SIS) = \infty$.*

Proof: Let us divide the proof in two parts giving different combinations of arrival and error patterns for the completed time and the pending time respectively.

We first consider a combination of arrival and error patterns A and E that behave as follows. We define time instants t_k where $k = 1, 2, \ldots$ and $t_i = t_{i-1} + \pi_{min}$ with time $t_0 = 0$ being the beginning of the execution. At every such time instants there is a crash and restart at the machine and then an immediate injection of a π_{min}-task followed by a π_{max}-task. This creates *active* intervals $[t_i, t_{i+1})$ of length π_{min}.

It is easy to observe that the patterns described cause algorithm SIS to assign the last π_{max}-task injected, every time it has to make a scheduling decision, since it is the last task injected. Since the *alive* intervals are of length π_{min} and SIS needs $\frac{\pi_{max}}{s} > \pi_{min}$ time to complete the π_{max}-tasks, it is not able to complete any of the tasks it starts executing, giving $C_t(\text{SIS}) = 0$ at all times t (and in particular at t_k time instants). On the same time, an algorithm X is able to schedule and complete all the π_{min}-tasks injected, one in every alive interval, giving a completed time of $C_{t_k}(X) = k \cdot \pi_{min}$ at every t_k time instant.

Now let us consider another combination of arrival and error patterns, A' and E' respectively as well as algorithm X'. We define time instants $t_{k'}$ where $k' = 1, 2, \ldots$ and $t_{k'} = t_{k'-1} + (\pi_{min} + \pi_{max})$ with time $t_0 = 0$ being the beginning of the execution. At every such time instants there is a π_{min}-task injected followed by a π_{max}-task. The crashes of the machine are set every time $t_{k'}$ and $t_{k'} + \pi_{min}$, creating *active* intervals of length π_{min} or π_{max} alternatively.

This behavior causes algorithm SIS to schedule the last π_{max}-task injected, after every $t_{k'}$ time instant, since it is the last task injected. The first alive interval after a $t_{k'}$ time instant is of length π_{min} and thus the π_{max}-task scheduled by SIS cannot be completed. On the same time, an algorithm X is able to complete the last π_{min}-task injected. On the next alive period though, SIS will be able to complete successfully the π_{max}-task since it is of length π_{max}, while X will do the same. As a result, looking at time instants $t_{k'}$, before the injection takes place, $P_{t_{k'}}(\text{SIS}) = k'\pi_{min}$ while $P_{t_{k'}}(X) = 0$, since algorithm SIS will never complete any π_{min}-task injected but X will be able to complete all tasks injected up to that time.

Therefore, for speedup $s < \rho$ algorithm SIS has completed-time competitive ratio $\mathcal{C}(\text{SIS}) = 0$ and pending-time competitive ratio $\mathcal{P}(\text{SIS}) = \infty$ as claimed. ■

Combining now Lemmas 2 and 3 we have the following theorem.

Theorem 3. *NONE of the three algorithms LIS, LPT and SIS is competitive when speedup $s < \rho$, with respect to completed or pending time, even in the case of only two task processing times (i.e., π_{min} and π_{max}).*

Theorem 4. *If tasks can be of only two processing times (π_{min} and π_{max}), algorithm SPT can achieve a completed-time competitive ratio $\mathcal{C}(SPT) \geq \frac{1}{2+\rho}$, for any speedup $s \geq 1$. In particular, $C_t(SPT) \geq \frac{1}{2+\rho} C_t(X) - \pi_{max}$, for any time t.*

Proof: Let us assume fixed arrival and error patterns A and E respectively, as well as an algorithm X, and let us look at any time t in the execution of SPT. Let τ be a task completed by X by time t (i.e., $\tau \in N_t(X)$), where t_τ is the time τ was scheduled and $f(\tau) \leq t$ the time it completed its execution. We associate τ with the following tasks in $N_t(\text{SPT})$:

- The same task τ.
- The task w being executed by SPT at time t_τ, if it was not later interrupted by a crash.

Not every task in $N_t(X)$ is associated to some task in $N_t(\text{SPT})$, but we show now that most tasks are. In fact, we show that the aggregate processing times of the tasks in $N_t(X)$ that are not associated with any task in $N_t(\text{SPT})$ is at most π_{max}. More specifically, there is only one task execution of a π_{max}-task, namely w, by SPT such that the π_{min}-tasks scheduled and completed by X concurrently with the execution of w fall in this class.

Considering the generic task $\tau \in N_t(X)$ from above, we consider the following cases:

- If $\tau \in N_t(\text{SPT})$ then task τ is associated at least with itself in the execution of SPT, regardless of τ's processing time.
- If $\tau \notin N_t(\text{SPT})$, τ is in the queue of SPT at time t_τ. By its greedy nature, SPT is executing some task w at time t_τ.
  - If $\pi(\tau) \geq \pi(w)$, then task w will complete by time $f(\tau)$ and hence it is associated with τ.
  - If $\pi(\tau) < \pi(w)$ (i.e., $\pi(\tau) = \pi_{min}$ and $\pi(w) = \pi_{max}$), then τ was injected after w was scheduled by SPT. If this execution of task w is completed by time t, then task w is associated with τ. Otherwise, task τ is not associated to any task in $N_t(\text{SPT})$ if a crash occurs or the time t is reached before w is completed.

Let t^* the time either event occurs. Since $\tau \notin N_t(\text{SPT})$, τ is not completed by SPT in the interval $[t^*, t]$. Then, SPT never schedules a π_{max}-task in the interval $[t^*, t]$, and the case that a task from $N_t(X)$ is not associated to any task in $N_t(\text{SPT})$ cannot occur again in that interval.

Hence, all the tasks $\tau \in N_t(X)$ that are not associated to tasks in $N_t(\text{SPT})$ are π_{min}-tasks and have been scheduled and completed during the execution of the same π_{max}-task by SPT. Hence, their aggregate processing time is at most π_{max}.

Now let us evaluate the processing times of the tasks in $N_t(X)$ associated to a task in $w \in N_t(\text{SPT})$. Let us consider any task w successfully completed by SPT at a time $f(w) \leq t$. Task w can be associated at most with itself and all the tasks that X scheduled within the interval $T_w = [f(w) - \pi(w), f(w)]$. The latter set can include tasks whose aggregate processing time is at most $\pi(w) + \pi_{max}$, since the first such tasks starts its execution no earlier than $f(w) - \pi(w)$ and in the extreme case a π_{max}-task could have been scheduled af the end of T_w and completed at $t_w + \pi_{max}$. Hence, if task w is a π_{min}-task, it will be associated with tasks completed by X that have a total processing time of at most $2\pi_{min} + \pi_{max}$, and if w is a π_{max}-task, it will be associated with tasks completed by X that have a total processing time of at most $3\pi_{max}$. Observe that $\frac{\pi_{min}}{2\pi_{min}+\pi_{max}} < \frac{\pi_{max}}{3\pi_{max}}$. As a result, we can conclude that

$$C_t(\text{SPT}) \geq \frac{\pi_{min}}{2\pi_{min} + \pi_{max}} C_t(X) - \pi_{max} = \frac{1}{2+\rho} C_t(X) - \pi_{max}.$$ ■

Conjecture 1. *The above lower bound on completed time, still holds in the case of any constant number of task processing times in the range* $[\pi_{min}, \pi_{max}]$.

Theorem 5. *For speedup $s < \rho$, algorithm SPT cannot have a completed-time competitive ratio more than x for $x < \frac{\hat{\rho}}{\hat{\rho}+\rho}$. In other words, $\mathcal{C}(SPT) \leq \frac{\hat{\rho}}{\hat{\rho}+\rho}$. Additionally, it is NOT competitive with respect to the pending time, i.e., $\mathcal{P}(SPT) = \infty$.*

Proof: Assuming speedup $s < \rho$ we consider the following combination of arrival and error patterns A and E respectively: We define time points t_k, where $k = 0, 1, 2 \ldots$, such that t_0 is the beginning of the execution and $t_k = t_{k-1} + \pi_{max} + \hat{\rho}\pi_{min}$ (recall that $\hat{\rho} = \lceil \rho \rceil - 1 < \rho$). At every t_k time instant there are $\hat{\rho}$ tasks of processing time π_{min} injected along with one π_{max}-task. What is more, the crash and restarts of the system's machine are set at times $t_k + \pi_{max}$ and then after every π_{min} time until t_{k+1} is reached.

By the arrival and error patterns described, every *epoch*; time interval $[t_k, t_{k+1}]$, results in the same behavior. Algorithm SPT is able to complete only the $\hat{\rho}$ tasks of processing time π_{min}, while X is able to complete all tasks that have been injected at the beginning of the epoch. From the nature of SPT, it schedules first the smallest tasks, and therefore the π_{max} ones never have the time to be executed; a π_{max}-task is scheduled at the last phase of each epoch which is of size π_{min} (recall $s < \rho \Rightarrow s < \hat{\rho}$). Hence, at time t_k, $C_{t_k}(\text{SPT}, A, E) = k\hat{\rho}\pi_{min}$ and $C_{t_k}(X, A, E) = k\hat{\rho}\pi_{min} + k\pi_{max}$.

Looking at the pending time at such points, we can easily see that SPT's is constantly increasing, while X is able to have pending time zero; $P_{t_k}(\text{SPT}, A, E) = k\pi_{max}$ but $P_{t_k}(X, A, E) = 0$.

As a result, we have a maximum completed-time competitive ratio $\mathcal{C}(\text{SPT}) \leq \frac{k\hat{\rho}\pi_{min}}{k\hat{\rho}\pi_{min}+k\pi_{max}} = \frac{\hat{\rho}}{\hat{\rho}+\rho}$ and a pending time $\mathcal{P}(\text{SPT}) = \infty$. ■

Theorem 6. *If tasks can have any processing time in the range $[\pi_{min}, \pi_{max}]$, and there is no speedup ($s = 1$), Algorithm SPT is NOT competitive with respect to completed time, i.e., $\mathcal{C}(SPT) = 0$.*

Proof: Assuming $s = 1$, we consider the following scenario as a result of adversarial arrival and error patterns A and E respectively. Let us fix some $\epsilon \in (0, 1)$ and use the notation $\Delta(k) = (\pi_{max} - \pi_{min})\epsilon^k$. Then, let w_k be a task with processing time $\pi(w_k) = \pi_{min} + \Delta(k)$, for all $k = 0, 1, 2, 3, \ldots$. Observe that $\forall k, \pi(w_k) \in (\pi_{min}, \pi_{max}]$ and $\pi(w_{k+1}) < \pi(w_k)$. Let us also define time points t_k, such that $t_0 = 0$ (the beginning of the execution) and $t_{k+1} = t_k + \pi_{min} + \frac{1+\epsilon}{2}\Delta(k)$. Let us also define time points $t'_k = t_{k-1} + \frac{1-\epsilon}{2}\Delta(k)$. The arrival pattern A is such that task $w_0 = \pi_{max}$ is injected in the system at time instant t_0. Then, for $k = 1, 2, \ldots$ task w_k is injected at time t'_k. The error pattern E is such that at every time instant t_k there is a crash and restart.

We compare SPT with an algorithm X of our choice. In the execution of SPT, task w_0 is scheduled as soon as it arrives, at time t_0. On the other hand, X waits until time t'_1 for the arrival of w_1 and schedules it immediately. When the processors crashes at time t_1 the task w_0 executed by SPT is interrupted, since $t_1 - t_0 = \pi_{min} + \frac{1+\epsilon}{2}\Delta(0) < \pi(w_0) = \pi_{min} + \Delta(0)$. However, X is able to complete task w_1 because $t'_1 + \pi(w_1) = \frac{1-\epsilon}{2}\Delta(1) + \pi_{min} + \Delta(1) = \pi_{min} + \frac{1+\epsilon}{2}\Delta(0) = t_1$. After the restart at t_1 SPT schedules task w_1, while X waits until t'_2 to schedule w_2.

The general process is as follows. At time instant t_k, SPT schedules task w_k while X waits until task w_{k+1} is injected at time t'_{k+1} and schedules it. When the processors crashes at time t_{k+1} the task w_k of SPT is interrupted, since $t_{k+1} - t_k = \pi_{min} + \frac{1+\epsilon}{2}\Delta(k) < \pi(w_k) = \pi_{min} + \Delta(k)$. However, X is able to complete task w_{k+1} because $t'_{k+1} + \pi(w_{k+1}) = \frac{1-\epsilon}{2}\Delta(k+1) + \pi_{min} + \Delta(k+1) = \pi_{min} + \frac{1+\epsilon}{2}\Delta(k) = t_{k+1}$.

Letting this adversarial behavior run to infinity we see that at any point in time t, $C_t(\text{SPT}) = 0$, while X will keep completing the injected tasks. This, results to a completed-time competitive ratio $\mathcal{C}(\text{SPT}) = 0$. ■

4.2. Speedup $s \geq \rho$

First, recall that in Theorem 1 we have shown that any work conserving algorithm running with speedup $s \geq \rho$ has pending-time competitive ratio at most ρ and completed-time competitive ratio at least $1/\rho$. So do the four algorithms LIS, LPT, SIS and SPT. A natural question that rises is whether we can improve these ratios. Let us start from some negative results again, focusing at first on the two policies that schedule tasks according to their arrival time, algorithms LIS and SIS.

Lemma 4. *When algorithm LIS runs with speedup $s \in [\rho, 1+1/\rho)$, it has a completed-time competitive ratio $\mathcal{C}(LIS) \leq \frac{1}{2} + \frac{1}{2\rho}$ and a pending-time competitive ratio $\mathcal{P}(LIS) \geq \frac{1+\rho}{2}$.*

Proof: Let speedup $s \in [\rho, 1 + 1/\rho)$. We define a combination of arrival and error patterns A and E, and algorithm X. Patterns A and E behave as follows: Initially, there is a π_{min}-task injected, followed by a π_{max}-task. After every period of π_{max} time the same injection sequence is repeated, when also the machine is crashed and restarted.

This behavior results to the following execution. There are only active phases of size π_{max}, during which an algorithm X can successfully execute the π_{max} task injected, while LIS is forced to schedule the tasks in the order they arrive. Observe that, since $s < 1 + 1/\rho = (\pi_{min} + \pi_{max})/\pi_{max}$, LIS is able to complete only one task in each phase. Observe also, that after k phases, where k is an multiple of 2, there will be exactly k tasks of size π_{min} pending in the queue of X, while LIS will have pending half of the tasks injected, half of which are of processing time π_{min} and the other half π_{max}. Hence, the pending-time competitive ratio of the algorithm becomes $\mathcal{P}(\text{LIS}) = \frac{\pi_{min}+\pi_{max}}{2\pi_{min}} = \frac{1+\rho}{2}$ and the completed-time competitive ratio $\mathcal{C}(\text{LIS}) = \frac{\pi_{min}+\pi_{max}}{2\pi_{max}} = \frac{1}{2} + \frac{1}{2\rho}$, which completes the proof. ■

Lemma 5. *When algorithm LIS runs with speedup $s \in [1+1/\rho, 2)$, it has a completed-time competitive ratio $\mathcal{C}(LIS) \leq \frac{1}{2} + \frac{\pi_{min}}{2\pi}$ and a pending-time competitive ratio $\mathcal{P}(LIS) \geq \frac{1}{2} + \frac{\pi}{2\pi_{min}}$ for any processing time $\pi \in (\pi_{min}, \pi_{max})$ such that $\pi < \frac{\pi_{min}}{s-1}$.*

Proof: Let speedup $s \in [1 + 1/\rho, 2)$. We define a combination of arrival and error patterns A and E, and algorithm X to behave as follows: We define time instants t_k, where $k = 0, 1, 2, \ldots$ and $t_0 = 0$ being the beginning of the execution and $t_k = t_{k-1} + \pi$, where $\pi \in (\pi_{min}, \pi_{max})$ and is such that $\frac{\pi_{min}+\pi}{s} > \pi \Rightarrow \pi < \frac{\pi_{min}}{s-1}$. At each t_k time instant there is a machine crash and restart followed by an injection of a π_{min}-task and then a task of processing time π.

This behavior results to the following execution. All phases are of size π, during which algorithm X completes successfully the π-task injected at the beginning of the phase, while LIS is able to complete either a π_{min}-task or a π-task. Algorithm LIS follows the order of task arrivals to schedule the tasks. However, by the definition of processing time π, in a period of length π LIS cannot complete both a π_{min} and a π-task. Observe that at every time instant t_k where k is a multiple of 2, LIS will be able to complete $k/2$ tasks of processing time π_{min} and $k/2$ tasks of processing time π while X will complete k tasks of processing time π. Respectively, at such time instants $P_{t_k}(\text{LIS}) = \frac{k(\pi_{min}+\pi)}{2}$ while $P_{t_k}(X) = k\pi_{min}$. This leads to a completed-time competitive ratio $\mathcal{C}(\text{LIS}) = \frac{1}{2} + \frac{\pi_{min}}{2\pi}$ while time goes to infinity, and a pending-time competitive ratio $\mathcal{P}(\text{LIS}) = \frac{1}{2} + \frac{\pi}{2\pi_{min}}$ as claimed. ■

Lemma 6. *When algorithm LIS runs with speedup $s \in [2, 1+\rho)$, it has a completed-time competitive ratio $\mathcal{C}(LIS) \geq 1$ and a pending-time competitive ratio $\mathcal{P}(LIS) \leq 1$.*

Proof: Let speedup $s \in [2, 1 + \rho)$ and let us analyze first the completed time metric. Let t^* be the first time in an execution, at which by means of contradiction, $C_{t^*}(\text{LIS}) < C_{t^*}(X) - \frac{3\pi_{max}}{2}$ holds. Also, let time $t' < t^*$ be the latest time instance such that for every $t \in [t', t^*]$, $C_t(\text{LIS}) < C_t(X)$ holds. Note that this implies that the queue of pending tasks of LIS is never empty within the interval $[t', t^*]$. What is more, both instants t' and t^* are times at which algorithm X completes a task. By definition of t', it also holds that $C_{t'}(\text{LIS}) \geq C_{t'}(X) - \pi_{max}$.

We then break the interval $[t', t^*]$ into consecutive periods $[t', t_1]$ and $(t_{i-1}, t_i]$ for $i = 2, 3 \ldots, k$, called *periods* i. Time instance $t_k = t^*$, and the rest of t_i's are the processor crashing points within the interval. Let us denote by $C_i(X)$ and $C_i(\text{LIS})$ the processing time completed in period i by X and LIS respectively. We discard the periods in which $C_i(\text{LIS}) = 0$ since $C_i(X) = 0$ will hold as well. After discarding these periods we renumber the rest in sequence from 1 to k'.

In order to prove the theorem, we need to show that the total completed time by X within the interval $[t', t^*]$ is larger than the total completed time by LIS within the same interval by at least an additive term of $\frac{3\pi_{max}}{2} - \pi_{max}$.

If in a period $j \leq k'$, algorithm LIS completes more task processing time than X, it must be the case that $\sum_{i=1}^{j-1} C_i(X) - \sum_{i=1}^{j-1} C_i(\text{LIS}) > C_j(\text{LIS}) - C_j(X)$, otherwise time t' is not well defined. Else if in a period $j < k'$ algorithm X completes more than LIS, i.e.,

$$C_j(X) > C_j(\text{LIS}), \tag{1}$$

then the following holds,

$$\frac{C_j(\text{LIS}) + \pi(\tau_{j+1})}{s} > C_j(X) \Rightarrow s \cdot C_j(X) - \pi(\tau_{j+1}) < C_j(\text{LIS}), \tag{2}$$

where τ_{j+1} is the last task intended for execution by LIS in period j but is not completed, it ends at the head of the queue of LIS at the end of period j. Hence it will be the first one to be completed in the next period. Therefore $\forall j \in [2, k']$,

$$C_j(\text{LIS}) \geq \pi(\tau_j). \tag{3}$$

From equations 1 and 2, we have that $C_j(X) > C_j(\text{LIS}) > s \cdot C_j(X) - \pi(\tau_{j+1})$. Since $s \geq 2$, it implies

$$(s-1) \cdot C_j(X) < \pi(\tau_{j+1}) \quad \Rightarrow \quad C_j(X) < \pi(\tau_{j+1}).$$

What is more, from equations 1 and 3 we have that $C_j(X) > \pi(\tau_j)$ and hence the following order of relationships holds

$$\pi(\tau_j) \leq C_j(\text{LIS}) < C_j(X) < \pi(\tau_{j+1}) \leq C_{j+1}(\text{LIS}).$$

Combining this with equation 2:

$$\begin{aligned}
s \cdot C_j(X) - C_j(\text{LIS}) &< \pi(\tau_{j+1}) \\
s\sum_{i=1}^{k'} C_i(X) - \sum_{i=1}^{k'} C_i(\text{LIS}) &< \sum_{i=1}^{k'} \pi(\tau_{i+1}) = \sum_{i=2}^{k'+1} \pi(\tau_i) \\
s\sum_{i=1}^{k'} C_i(X) - \sum_{i=1}^{k'} C_i(\text{LIS}) &< \sum_{i=2}^{k'} C_i(\text{LIS}) + \pi(\tau_{k'+1}) \\
s\sum_{i=1}^{k'} C_i(X) &< 2\sum_{i=1}^{k'} C_i(\text{LIS}) - C_1(\text{LIS}) + \pi(\tau_{k'+1}) \\
\sum_{i=1}^{k'} C_i(X) &< \frac{2}{s}\sum_{i=1}^{k'} C_i(\text{LIS}) + \frac{\pi(\tau_{k'+1}) - C_1(\text{LIS})}{s} \\
\sum_{i=1}^{k'} C_i(X) &< \sum_{i=1}^{k'} C_i(\text{LIS}) + \frac{\pi_{max}}{s}.
\end{aligned}$$

Combining this with the fact that $C_{t'}(\text{LIS}) \geq C_{t'}(X) - \pi_{max}$, we have that

$$\begin{aligned}
C_{t^*}(X) &= C_{t'}(X) + \sum_{i=1}^{k'} C_i(X) \\
&< C_{t'}(\text{LIS}) + \pi_{max} + \sum_{i=1}^{k'} C_i(\text{LIS}) + \frac{\pi_{max}}{s} \\
&= C_{t^*}(\text{LIS}) + \pi_{max} + \frac{\pi_{max}}{s} \leq C_{t^*}(\text{LIS}) + \frac{3\pi_{max}}{2},
\end{aligned}$$

which contradicts the initial claim and the definition of time t'. Note that again, the last inequality follows from the fact that speedup $s \geq 2$. Hence, even if algorithm X manages to complete more task processing time in some periods, LIS will eventually surpass its performance.

Since the pending time is complementary to the completed time we can claim the following:

$$\begin{aligned}
C_t(\text{LIS}) &\geq C_t(X) - \frac{3\pi_{max}}{2} \\
I_t - C_t(\text{LIS}) &\leq I_t - C_t(X) + \frac{3\pi_{max}}{2} \\
P_t(\text{LIS}) &\leq P_t(X) + \frac{3\pi_{max}}{2}.
\end{aligned}$$

which completes the proof for both completed-time and pending-time competitive ratios being optimal for algorithm LIS when speedup $s \in [2, 1 + \rho)$. ■

Combining Lemmas 4, 5, 6 and Theorem 2 we have the following theorem.

Theorem 7. *Algorithm LIS has a completed-time competitive ratio* $\mathcal{C}(LIS) \leq \begin{cases} \frac{1}{2} + \frac{1}{2\rho} & s \in [\rho, 1 + 1/\rho) \\ \frac{1}{2} + \frac{\pi_{min}}{2\pi} & s \in [1 + 1/\rho, 2) \end{cases}$, *and* $\mathcal{C}(LIS) \geq 1$ *when* $s \geq \max\{\rho, 2\}$. *It also has a pending-time competitive ratio* $\mathcal{P}(LIS) \geq \begin{cases} \frac{1+\rho}{2} & s \in [\rho, 1 + 1/\rho) \\ \frac{1}{2} + \frac{\pi}{2\pi_{min}} & s \in [1 + 1/\rho, 2) \end{cases}$, *and* $\mathcal{P}(LIS) \leq 1$ *when* $s \geq \max\{\rho, 2\}$. *Processing time* $\pi \in (\pi_{min}, \pi_{max})$ *is such that* $\pi < \frac{\pi_{min}}{s-1}$.

Also, combining Lemma 6 and Theorem 2, yields a completed-time competitive ratio $\mathcal{C}(\text{LIS}) \leq 1$ and a pending-time competitive ratio $\mathcal{P}(\text{LIS}) \geq 1$, when speedup $s \geq \max\{\rho, 2\}$. Recall that $\rho \geq 1$ which means that $1 + \rho \geq 2$.

Theorem 8. *When algorithm SIS runs with speedup* $s \in [\rho, 1+\rho)$*, it has a completed-time competitive ratio* $\mathcal{C}(SIS) \leq \frac{\pi_{min}}{\pi}$ *and a pending-time competitive ratio* $\mathcal{P}(SIS) \geq \frac{\pi+\pi_{max}}{\pi_{min}+\pi_{max}}$*, where* $\pi < \frac{\pi_{min}+\pi_{max}}{s}$ *and* $\pi \in (\pi_{min}, \pi_{max})$.

Proof: Let speedup $s \in [\rho, 1 + \rho)$. We define a combination of arrival and error patterns A and E, algorithm X and consider tasks of processing time π_{max}, π_{min} and π, where $\pi \in (\pi_{min}, \pi_{max})$, such that $\pi < \frac{\pi_{min}+\pi_{max}}{s}$. Note that, such a value π always exists since $s < 1 + \rho$.

Patterns A and E behave as follows: We define time instants t_k, where k is an increasing positive integer ($k = 0, 1, 2, \dots$), with $t = 0$ being the beginning of the execution and $t_k = t_{k-1} + \pi$. At time t_k there is exactly one π-task injected, followed by one π_{max}-task, followed by one π_{min}-task. Crashes and restarts are also set at times t_k, causing *active* intervals of π duration.

This behavior results to executions where an algorithm X is able to complete the last π-task injected, while SIS is forced to schedule the latest π_{min}-task followed by the latest π_{max}-task, being able to complete only the π_{min}-task. Therefore, at the end of each alive interval, $C_{t_k}(\text{SIS}) = k\pi_{min}$, $C_{t_k}(X) = k\pi$, $P_{t_k}(\text{SIS}) = k(\pi + \pi_{max})$ and $P_{t_k}(X) = k(\pi_{min} + \pi_{max})$. Hence, the completed-time competitive ratio of algorithm SIS becomes $\mathcal{C}(\text{SIS}) = \frac{\pi_{min}}{\pi}$ and its pending-time competitive ratio $\mathcal{P}(\text{SIS}) = \frac{\pi+\pi_{max}}{\pi_{min}+\pi_{max}}$. ■

The nature of algorithms LPT and SPT however (scheduling according to the processing time of tasks rather than their arrival time), gives better results for both the completed and pending time measures.

Lemma 7. *When algorithm LPT runs under speedup* $s \geq \rho$*, it has a completed-time competitive ratio* $\mathcal{C}(LPT) \geq 1$.

Proof: As proven in Lemma 1, the number of completed tasks of any work conserving algorithm under any combination of arrival and error patterns A and E, and speedup $s \geq \rho$, is never smaller than the number of completed tasks of X. The same holds for algorithm LPT, $|N_t(\text{LPT})| \geq |N_t(X)|$.

Since the policy of LPT is to schedule first the tasks with the biggest processing time, the ones completed will be of the maximum size available at all times, which trivially results to a total completed processing time at least as much as the one of X, $C_t(\text{LPT}) \geq C_t(X)$ at any time t. This gives a completed-time competitive ratio of at least 1 as claimed. ■

Theorem 9. *When algorithm LPT runs with speedup* $s \geq \rho$*, it has a completed-time competitive ratio* $\mathcal{C}(LPT) \geq 1$ *and a pending-time competitive ratio* $\mathcal{P}(LPT) \leq 1$.

Proof: The completed-time competitiveness of algorithm LPT follows from Lemma 7 above. Here we show its pending-time complexity; the proof is much more challenging.

We consider a time t and define the ordered set of pending tasks at the scheduler as $Q_t(\text{LPT}) = \langle v_1, v_2, \dots, v_z \rangle$ in an execution of algorithm LPT, and as $Q_t(X) = \langle u_1, u_2, \dots, u_{z'} \rangle$ in an execution of an algorithm X. We sort the

tasks in both sets in descending lexicographic order according to their processing time and IDs, so $v_1 \geq v_2 \geq \cdots \geq v_z$ and $u_1 \geq u_2 \geq \cdots \geq u_{z'}$. By Lemma 1, we know that $|Q_t(\text{LPT})| \leq |Q_t(X)|$ at any time t, which means that $z \leq z'$ and also $|N_t(\text{LPT})| \geq |N_t(X)|$.

We consider any execution and focus on the time instances where algorithm LPT has to make a scheduling decision, denoting them as t_k where $k = 0, 1, 2, \ldots$. We then define the following property:

> (*) At time t, for any $1 < i \leq z$ and $j = |N_t(\text{LPT})| - |N_t(X)|$, the processing time of the i^{th} task in the set $Q_t(\text{LPT})$ is not bigger than the processing time of $(i + j - 1)^{st}$ task in the set $Q_t(X)$; in other words $v_i \leq u_{i+j-1}$.

We now make three claims regarding property (*). First, Claim 1 argues that the property holds at all time instants t_k. In Claim 2 we consider the property at times of injections and show that it is still true. Finally, Claim 3 argues that the property holds at all times between time instants t_k. Combining the results of the three claims yields that property (*) holds at all times in any execution. Hence, the total pending processing time of algorithm LPT is never bigger than that of X plus $\pi_{max} - \pi_{min}$ which might be the difference in their sets of pending tasks. This results to a pending-time competitive ratio $\mathcal{P}(\text{LPT}) = 1$, as claimed. So the proof of the theorem concludes by stating and proving the three claims.

Claim 1: For all times t_k where $k = 0, 1, 2, \ldots$ property (*) holds.

We now prove Claim 1. Taking the beginning of any execution, t_0, it is clear that the property (*) holds, as none of the two algorithms has computed any task yet and the pending sets are exactly the same, $\forall i, v_i = u_i$. To be more precise, since $|N_0(\text{LPT})| = |N_0(X)| = 0$, $j = 0$ and for any i where $1 < i \leq z, v_i \leq u_{i-1}$ also holds.

Now consider by contradiction time t_k, the smallest time in an execution, where (*) does not hold. It is clear that it must be a point at which the machine completes a task in the execution of LPT. We know that right before such points $|N_{t_k^-}(\text{LPT})| \geq |N_{t_k^-}(X)|$. So there are two cases:
(1) If the two algorithms had the same amount of completed tasks before t_k (and hence $j^- = 0$), then for $1 < i^- \leq z, v_{i^-} \leq u_{i^-+j^--1} = u_{i^--1}$, where i^- and j^- are the corresponding parameters right before time t_k. At time t_k, the first task in $Q_{t_k}(\text{LPT}), v_1$, has been completed and hence removed from the set, causing the rest of the tasks to move up in the ordered set by one position. Then index $i = i^- - 1$ and thus $v_i = v_{i^--1} \leq u_{i^--1}$ still holds. Also, since $|N_{t_k}(\text{LPT})| = |N_{t_k}(X)| + 1$, index $j = j^- + 1 = 1$ and $u_{i+j-1} = u_{i^--1+j^-+1-1} = u_{i^--1}$. Therefore, $v_i \leq u_{i+j-1}$ and the property (*) still holds.
(2) If $|N_{t_k^-}(\text{LPT})| > |N_{t_k^-}(X)|$, then $v_{i^-} \leq u_{i^-+j^--1}$ right before t_k. At time t_k all the tasks in the set of LPT will move up the list by one position; $i = i^- - 1$ and thus $v_i = v_{i^--1}$. Also, the difference between the completed tasks of LPT and X increases by 1; $j = j^- + 1$ and thus $u_{i^-+j^--1} = u_{i+j^-} = u_{i+j-1}$ (the tasks remain at the same position for the set of X). Since $v_{i^-} \leq u_{i^-+j^--1} = u_{i+j-1} \Rightarrow v_i = v_{i^--1} \leq u_{i+j-1}$ holds and so does the property (*).
In both cases we have come to contradiction, so at time t_k the property (*) still holds. This completes the proof of Claim 1.

Claim 2: When a task τ is injected, property (*) still holds.

We now prove Claim 2. Assume a time t when a task τ arrives in the system, so it is added to both $Q_t(\text{LPT})$ and $Q_t(X)$. For the purpose of analysis we will look at times t_k where $k = 0, 1, 2, \ldots$, since during a time interval (t_{k-1}, t_k) LPT completes exactly one task and even if X has also executed a task and scheduled the next one (even one that has just been injected), it won't be able to compute it successfully within the interval. Thus, since it was not in any of the sets at time t_{k-1} but is in both of them at time t_k, we consider the time t_k to show that an injection of a task does not affect the property (*).

Hence, consider by contradiction a time t_k, such that it is the smallest time in an execution where there is an injection of task τ and the property (*) does not hold. We know that $|N_{t_k^-}(\text{LPT})| \geq |N_{t_k^-}(X)|$, so before time t_k, $v_{i^-} \leq u_{i^-+j^--1}$. After the injection, $j = j^-$ (remains the same), as τ is added in both sets of pending tasks. Assume that in $Q_{t_k}(\text{LPT})$ task τ is placed between v_{i^-} and v_{i^-+1}. This means that $v_{i^-} \geq \tau \geq v_{i^-+1}$. By property (*) it is trivial that $v_{i^-} \leq u_{i^-+j^--1}$, thus $\tau \leq u_{i^-+j^--1}$ as well. So, we know that τ will be placed after $u_{i^-+j^--1}$ in $Q_{t_k}(X)$. It is also clear that v_{i^-} and $u_{i^-+j^--1}$ do not change position in the sets and thus at time t_k we can say that v_{i^-} becomes v_i, still having processing time less than or equal to $u_{i^-+j^--1}$ that becomes u_{i+j-1}.

From property (*) we also know that $v_{i^-+1} \leq u_{i^-+j^-}$. If $\tau \geq u_{i^-+j^-}$, then it will also be placed right between $u_{i^-+j^--1}$ and $u_{i^-+j^-}$ in the ordered set of X, $Q_{t_k}(X)$. They will be renamed to u_{i+j-1} and u_{i+j} respectively and $u_{i+j-1} \geq \tau \geq u_{i+j}$ will hold as well as the property (*), since the mapping between tasks did not change and the new task in $Q_{t_k}(\text{LPT})$ is mapped to itself in $Q_{t_k}(X)$, $\tau \leq \tau$. If $\tau \leq u_{i^-+j^-}$, then it will be added somewhere after $u_{i^-+j^-}$ in the set $Q_{t_k}(X)$. Task $u_{i^-+j^-}$ does not change position in the set and thus at time t_k we can say it becomes u_{i+j} and is mapped with task τ from the set of LPT, $\tau \leq u_{i+j}$. Now let us assume that τ is placed after task u_{i+j+m} in $Q_{t_k}(X)$. So for all the tasks v_{i+c} in $Q_{t_k}(\text{LPT})$ and all the tasks u_{i+j+c} in $Q_{t_k}(X)$ where $1 \leq c \leq m$, it is true that, since $\tau \geq v_{i+c}$ and $\tau \leq u_{i+j+c}$, $v_{i+c} \leq u_{i+j+c}$. For the next task in the set of LPT, v_{i+m+1} we can say for sure that $v_{i+m+1} \leq \tau$ since τ is inserted in a higher place in $Q_{t_k}(\text{LPT})$. For the rest of tasks in both ordered sets we can safely say that, since τ has been added to both in higher positions, they have the mapping they had before the injection of τ. As a result, property (*) still holds after a task injection. This completes the proof of Claim 2.

Claim 3: For every time t in the interval (t_{i-1}, t_i), property (*) still holds.

Finally, we prove Claim 3. From Claim 1 we have that at time t_{i-1}, property (*) holds, so for any $1 < m \leq z$ and $j = |N_{t_i}(\text{LPT})| - |N_{t_i}(X)|$, it holds $v_m \leq u_{m+j-1}$. We want to prove that this is true for all times t in the interval (t_{i-1}, t_i). Assume by contradiction, that there is a time t^* in that interval, the first instance where the property does not hold. Note that for this to occur, X must complete a task at time t^*. We know by definition, that during the interval (t_{i-1}, t_i) LPT can not fully execute any task, as the points where tasks are completed are times t_k where $k = 1, 2, 3, \ldots$. There are therefore, two cases to consider:
(1) During the interval neither X completes any task. This means that there is no change in the sets of pending tasks of the two algorithms and thus at any point in the interval (t_{i-1}, t_i), for all $1 < m \leq z$ and $j = |N_{t_i}(\text{LPT})| - |N_{t_i}(X)|, v_m \leq u_{m+j-1}$.
(2) During the interval, X successfully completes a task. Note that it can complete only one task (since $s \geq \rho$), and that it finishes at time t^*. Also note that before time t^*, $|N_{t^*-}(\text{LPT})| - |N_{t^*-}(X)| = j' > 0$ must hold because if the amount of completed tasks were equal, after the removal of the just fully computed task of X, LPT would have more pending tasks than X, which by Lemma 1 is not possible. So, we know that the set $Q_{t^*}(\text{LPT})$ remains the same, whereas in the $Q_{t^*}(X)$ the task that was just completed is removed. Recall that, before t^*, $v_{i^-} \leq u_{i^-+j^--1}$. Using similar arguments as in case (2) of the proof of Claim 1, we may conclude the proof: By X successfully completing a task, say the k^{th} task u_k, where $1 < k \leq |N_{t^*}(X)|$, all the tasks with $i > k$ will move up in the list of X by one position, to form the new sequence $\langle u_1, u_2, \ldots \rangle$. Hence $|N_{t^*}(\text{LPT})| \geq |N_{t^*}(X)|$ remains true and $|N_{t^*}(\text{LPT})| - |N_{t^*}(X)| = j = j^- - 1$. We want to prove that $v_i \leq u_{i+j-1}$. For all the tasks v_i such that $i + j - 1 < k$, the task $u_{i^-+j^--1}$ did not change its position in $Q(X)$. Hence, $v_i \leq u_{i^-+j^--1} = u_{i+j} \leq u_{i+j-1}$ ($i = i^-$). However, for all the tasks v_i such that $i + j - 1 \geq k$, the task $u_{i^-+j^--1}$ has moved up one position in $Q(X)$. Hence, $v_i \leq u_{i^-+j^--1} = u^-_{i+j-2} = u_{i+j-1}$ ($i =$). This completes the proof of Claim 3 and of the theorem. ■

Theorem 10. *When algorithm SPT runs with speedup $s \geq \rho$, it has a completed-time competitive ratio $\mathcal{C}(SPT) \geq 1$*

and a pending-time competitive ratio $\mathcal{P}(SPT) \leq 1$.

Proof: Let us consider any execution of algorithm SPT running speedup $s \geq \rho$ under any arrival and error patterns A and E respectively. We will prove that at all times in the execution, the completed processing time of SPT is more than that of an algorithm X; $C(\text{SPT}) \geq C(X)$.

By way of contradiction, we assume a point in time t such that, it is the first time in the execution where $C_t(\text{SPT}) < C_t(X)$. It must be the case that X has just completed a task, since at all earlier times, up to t^-, $C_{t^-}(\text{SPT}) \geq C_{t^-}(X)$.

Now let us consider that X has completed a π_{min}-task. This means that during the interval $(t - \pi_{min}, t)$ no machine failure has occurred and hence algorithm SPT was also able to complete some tasks. Let t^* be the last time in $(t - \pi_{min}, t)$ that SPT completes a task. It holds that $C_{t^*}(\text{SPT}) = C_{t-\pi_{min}}(\text{SPT}) + \lfloor s \cdot \pi_{min} \rfloor \geq C_{t-\pi_{min}}(\text{SPT}) + s \cdot \pi_{min} + 1$, while $C_{t^*}(X) = C_{t-\pi_{min}}(X)$. At time t, algorithm SPT has the same completed processing time as at time t^*, whereas X's increases by π_{min}. Hence

$$C_t(\text{SPT}) = C_{t^*}(\text{SPT}) \geq C_{t-\pi_{min}}(\text{SPT}) + s \cdot \pi_{min} + 1 \geq C_t(X) + (s-1)\pi_{min} + 1,$$

which contradicts the initial assumption. We have therefore shown that $C(\text{SPT}) \geq C(X)$ at all times, which results to a completed-time competitive ratio $\mathcal{C}(\text{SPT}) \geq 1$.

Complementary to the completed time shown above, observe that for the pending time it will be the case that $P_t(\text{SPT}) \leq P_t(X) - (s-1)\pi_{min} - 1$ which gives a pending-time competitive ratio $\mathcal{P}(\text{SPT}) \leq 1$. ■

5. Latency Competitiveness

In the case of latency, things are clearer between the competitiveness ratio and the speedup bounds for the four scheduling policies.

Theorem 11. *NONE of the algorithms LPT, SIS nor SPT can be competitive with respect to the latency for any speedup $s \geq 1$. That is, $\mathcal{L}(LPT) = \mathcal{L}(SIS) = \mathcal{L}(SPT) = \infty$.*

Proof: We consider one of the three algorithms ALG $\in \{\text{LPT}, \text{SIS}, \text{SPT}\}$, and assume ALG is competitive with respect to the latency metric, say there is a bound $\mathcal{L}(\text{ALG}) \leq B$ on its latency competitive ratio. Then, we define a combination of arrival and error patterns, A and E, under which this bound is violated. More precisely, we show a latency bound larger than B, which contradicts the initial assumption and proves the claim.

Let R be a large enough integer that satisfies $R > B + 2$ and x be an integer larger than $s\rho$ (recall that $s \geq 1$ and $\rho > 1$, so $x \geq 2$). Let also $\pi = \pi_{min}$ if ALG = SPT and $\pi = \pi_{max}$ otherwise. We now define time instants t_k for $k = 0, 1, 2, \ldots, R$ as follows: time $t_0 = 0$ (the beginning of the execution), $t_1 = \pi(x^{R-1} + x^R) - \pi(w)$ (observe that $x \geq 2$ and we set R large so t_1 is not negative), and $t_k = t_{k-1} + \pi(x^{R-1} + x^R) - \pi x^{k-1}$, for $k = 2, \ldots, R$. Finally, let us define the time instants t'_k for $k = 0, 1, 2, \ldots, R$ as follows: time $t'_0 = t_0$, $t'_1 = t_1 + \pi(w)$, and $t'_k = t_k + \pi x^{k-1}$, for $k > 1$.

We define arrival pattern A as follows. At time t_0 there is a task w injected by A, where $\pi(w) = \pi_{max}$ if ALG = SPT and $\pi(w) = \pi_{min}$ otherwise and at every time instant t_k, for $k \geq 1$, there are x^k tasks of processing time π injected. Observe that π-tasks are such, that ALG always gives priority to them over task w. The error pattern E is as follows. The machine runs continuously without crashes in every interval $[t_k, t'_k]$, where $k = 0, 1, \ldots, R$. It then crashes at t'_k and does not recover until t_{k+1}.

We now define the behavior of a given algorithm X that runs without speedup. In the first alive interval, $[t_1, t'_1]$, algorithm X completes task w. In general, in each interval $[t_k, t'_k]$ for every $k = 2, \ldots, R$, it completes the x^{k-1} tasks of processing time π injected at time t_{k-1}.

On its hand, ALG always gives priority to the x π-tasks over w. Hence, in the interval $[t_1, t'_1]$ it will start executing the π-tasks injected at time t_1. The length of the interval is $\pi(w)$. Since $x > s\rho$, then $x > (s-1)\pi(w)/\pi$ and hence $\frac{\pi x+\pi(w)}{s} > \pi(w)$. This implies that ALG is not able to complete w in the interval $[t_1, t'_1]$. Regarding any other interval $[t_k, t'_k]$, whose length is πx^{k-1}, the x^k π-tasks injected at time t_k have priority over w. Observe then, that since $x > s\rho$, then $\pi x^k + \pi(w) > s\pi x^{k-1}$ and hence $\frac{\pi x^k+\pi(w)}{s} > \pi x^{k-1}$. Then, ALG again will not be able to complete w in the interval.

As a result, the latency of X at time t'_R is $L_{t'_R}(X) = \pi(x^{R-1} + x^R)$. This follows since, on the one hand, w is completed at time $t'_1 = \pi(x^{R-1} + x^R)$. On the other hand, for $k = 2, \ldots, R$, the tasks injected at time t_{k-1} are completed by time t'_k, and $t'_k - t_{k-1} = t_k + \pi x^{k-1} - t_{k-1} = t_{k-1} + \pi(x^{R-1} + x^R) - \pi x^{k-1} + \pi x^{k-1} - t_{k-1} = \pi(x^{R-1} + x^R)$. At the same time t'_R, the latency of ALG is determined by w since it is still not completed, $L_{t'_R}(\text{ALG}) = t'_R$. Then,

$$\begin{aligned}
L_{t'_R}(\text{ALG}) &= t_R + \pi x^{R-1} = t_{R-1} + \pi(x^{R-1}+x^R) - \pi x^{R-1} + \pi x^{R-1} \\
&= t_{R-2} + 2\pi(x^{R-1}+x^R) - \pi x^{R-2} = t_{R-3} + 3\pi(x^{R-1}+x^R) - \pi x^{R-2} - \pi x^{R-3} \\
&= \cdots = t_1 + (R-1)\pi(x^{R-1}+x^R) - \pi \sum_{i=1}^{R-2} x^i \\
&= R\pi(x^{R-1}+x^R) - \pi(w) - \pi\frac{x^{R-1}-x}{x-1}.
\end{aligned}$$

Hence, the latency competitive ratio of ALG is no smaller than

$$\begin{aligned}
\frac{L_{t'_R}(\text{ALG})}{L_{t'_R}(X)} &= \frac{R\pi(x^{R-1}+x^R) - \pi(w) - \pi\frac{x^{R-1}-x}{x-1}}{\pi(x^{R-1}+x^R)} \\
&= R - \frac{\pi(w)}{\pi(x^{R-1}+x^R)} - \frac{x^{R-1}-x}{(x-1)(x^{R-1}+x^R)} \\
&= R - \frac{\pi(w)}{\pi(x^{R-1}+x^R)} - \frac{1}{x^2-1} + \frac{1}{x^R - x^{R-2}} \geq R - 2 > B.
\end{aligned}$$

The three fractions in the third line are no larger than 1 since $x \geq 2$, and R is large enough so that $t_1 \geq 0$ and hence $\pi(x^{R-1}+x^R) \geq \pi(w)$. ■

For algorithm LIS one the other hand, we show that even though latency competitiveness cannot be achieved for $s < \rho$, as soon as $s \geq \rho$ LIS becomes competitive. The negative result verifies the intuition that since the algorithm is not competitive in terms of pending time for $s < \rho$, neither should it be in terms of latency. Nonetheless, the positive result verifies the intuition for competitiveness, since for $s \geq \rho$ algorithm LIS is pending-time competitive **and** it gives priority to the tasks that have been waiting the longest in the system.

Lemma 8. *For speedup $s < \rho$, algorithm LIS is not competitive in terms of latency, i.e., $\mathcal{L}(LIS) = \infty$.*

Proof: Let us consider a combination of arrival and error patterns A and E, and algorithm X. Pattern A is an infinite arrival pattern that injects a π_{min}-task at the beginning of the execution, followed by a π_{max}-task (after infinitesimally small time ε). After that, it injects only π_{min}-tasks, one every π_{min} time. Pattern E sets the first crash/restart instant at $\pi_{max} + \varepsilon$ time from the beginning and then every π_{min} period of time, creating a *phase* (time period between a restart and the next crash) of length π_{max} followed by infinite phases of length π_{min}. These patterns allow an algorithm X to execute successfully the π_{max}-task injected at the beginning on the first phase, while algorithm LIS's policy to schedule the one that was injected earlier in the system forces it to schedule the π_{min}-task. Even though it will also

be executed, the π_{max}-task scheduled next will never be completed in any of the following phases since they are all of size π_{min} and $\frac{\pi_{max}}{s} > \pi_{min}$. This means that algorithm's LIS latency will increase to infinity with time, while X's latency will remain bounded (each task is completed at most $\pi_{max} + \pi_{min}$ time after its injection).

Hence, completing the theorem, for speedup $s < \rho$ algorithm LIS is not competitive in terms of latency, $\mathcal{L}(\text{LIS}) = \infty$, as claimed. ■

Lemma 9. *For speedup $s \geq \rho$, algorithm LIS has a latency competitive ratio $\mathcal{L}(LIS) \leq 1$.*

Proof: Consider an execution of algorithm LIS running with speedup $s \geq \rho$ under any arrival and error patterns $A \in \mathcal{A}$ and $E \in \mathcal{E}$. Assume interval $T = [t_0, t_1)$ where time t_0 is the instant at which a task w arrived and t_1 the time at which it was completed in the execution of algorithm LIS. Also, assume by contradiction, that task w is such that $L_{t_1}(\text{LIS}, w) > \max\{L_{t_1}(X, \tau)\}$, where τ is some task that arrived before time t_1. We will show that this cannot be the case, which proves latency competitiveness with ratio $\mathcal{L}(\text{LIS}) \leq 1$.

Consider any time $t \in T$, such that task w is being executed in the execution of LIS. Since its policy is to schedule tasks in the order of their arrival, it means that it has already completed successfully all task that were pending in the central scheduler at time t_0 before scheduling task w. Hence, at time t, algorithm LIS's queue of pending tasks has all the tasks injected after time t_0 (say x), plus task w, which is still not completed. By Lemma 1, we know that the there are never more pending tasks in the queue of LIS than that of X and hence $|Q_t(\text{LIS})| = x + 1 \leq |Q_t(X)|$. This means that there is at least one task pending for X which was injected up to time t_0. This contradicts our initial assumption of the latency of task w being bigger than the latency of any task pending in the execution of X at time t_1. Therefore LIS's latency competitive ratio when speedup $s \geq \rho$, is $\mathcal{L}(\text{LIS}) \leq 1$, as claimed. ■

The next theorem follows by combining Lemmas 8 and 9.

Theorem 12. *Algorithm LIS has latency competitive ratio $\mathcal{L}(LIS) = \infty$ when speedup $s < \rho$, and $\mathcal{L}(LIS) \leq 1$ otherwise.*

6. Conclusions

In this paper we performed a thorough study on the competitiveness of four popular online scheduling algorithms (LIS, SIS, LPT and SPT) under dynamic task arrivals and machine failures. More precisely, we looked at worst-case (adversarial) task arrivals and machine crashes and restarts and compared the behavior of the algorithms under various speedup intervals. Even though our study focused on the simple setting of one machine, interesting conclusions have been derived with respect to the efficiency of these algorithms under the three different metrics – completed time, pending time and latency – and under different speedup values. An interesting open question is whether one can obtain efficiency bounds as functions of speedup s, upper bounds for the completed-time and lower bounds for the pending-time and latency competitive ratios. Another natural next step is to extend our investigation to the setting with multiple machines.

References

[1] S. Anand, Naveen Garg, and Nicole Megow. Meeting deadlines: How much speed suffices? In Luca Aceto, Monika Henzinger, and Ji Sgall, editors, *Automata, Languages and Programming*, volume 6755 of *Lecture Notes in Computer Science*, pages 232–243. Springer Berlin Heidelberg, 2011.

[2] Matthew Andrews and Lisa Zhang. Scheduling over a time-varying user-dependent channel with applications to high-speed wireless data. *J. ACM*, 52(5):809–834, September 2005.

[3] Antonio Fernández Anta, Chryssis Georgiou, Dariusz R. Kowalski, Joerg Widmer, and Elli Zavou. Measuring the impact of adversarial errors on packet scheduling strategies. In *SIROCCO*, pages 261–273, 2013.

[4] Antonio Fernández Anta, Chryssis Georgiou, Dariusz R. Kowalski, and Elli Zavou. Online parallel scheduling of non-uniform tasks: Trading failures for energy. In *FCT*, pages 145–158, 2013.

[5] Joan Boyar and Faith Ellen. Bounds for scheduling jobs on grid processors. In Andrej Brodnik, Alejandro Lpez-Ortiz, Venkatesh Raman, and Alfredo Viola, editors, *Space-Efficient Data Structures, Streams, and Algorithms*, volume 8066 of *Lecture Notes in Computer Science*, pages 12–26. Springer Berlin Heidelberg, 2013.

[6] Leah Epstein and Rob van Stee. Online bin packing with resource augmentation. *Discrete Optimization*, 4(34):322 – 333, 2007.

[7] Anis Gharbi and Mohamed Haouari. Optimal parallel machines scheduling with availability constraints. *Discrete Applied Mathematics*, 148(1):63 – 87, 2005.

[8] D. Johnson, A. Demers, J. Ullman, M. Garey, and R. Graham. Worst-case performance bounds for simple one-dimensional packing algorithms. *SIAM Journal on Computing*, 3(4):299–325, 1974.

[9] B. Kalyanasundaram and K. Pruhs. Speed is as powerful as clairvoyance [scheduling problems]. In *Foundations of Computer Science, 1995. Proceedings., 36th Annual Symposium on*, pages 214–221, Oct 1995.

[10] Kirk Pruhs, Jiri Sgall, and Eric Torng. Online scheduling. *Handbook of scheduling: algorithms, models, and performance analysis*, pages 15–1, 2004.

[11] Eric Sanlaville and Gnter Schmidt. Machine scheduling with availability constraints. *Acta Informatica*, 35(9):795–811, 1998.

[12] K. Schwan and H. Zhou. Dynamic scheduling of hard real-time tasks and real-time threads. *Software Engineering, IEEE Transactions on*, 18(8):736–748, Aug 1992.

[13] Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM*, 28(2):202–208, February 1985.

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

[15] Rob van Stee. *Online Scheduling and Bin Packing.* PhD Thesis, Leiden University and Centre for Mathematics and Computer Science, Amsterdam, The Netherlands, 2002.