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

Antonio Fernández Anta[1(✉)], Chryssis Georgiou[2], Dariusz R. Kowalski[3], and Elli Zavou[1,4]

[1] Institute IMDEA Networks, Madrid, Spain
antonio.fernandez@imdea.org
[2] University of Cyprus, Nicosia, Cyprus
[3] University of Liverpool, Liverpool, UK
[4] Universidad Carlos III de Madrid, Madrid, Spain

Abstract. Reliable task execution on machines that are prone to unpredictable crashes and restarts is both important and challenging, but not much work exists on the analysis of such systems. We consider the online version of the problem, with tasks arriving over time at a single machine under worst-case assumptions. 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). We use three metrics for the evaluation and comparison of their competitive performance, namely, completed load, pending load, and latency. We also investigate the effect of resource augmentation in their performance, by increasing the speed of the machine. Hence, we compare the behavior of the algorithms for different speed intervals and show that there is no clear winner with respect to all the three considered metrics. While SPT is the only algorithm that achieves competitiveness on completed load for small speed, LIS is the only one that achieves competitiveness on latency (for large enough speed).

Keywords: Scheduling · Online algorithms · Task sizes · Failures · Competitive analysis · Resource augmentation

1 Introduction

The demand for processing dynamically introduced jobs that require high computational power 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), which in cloud computing is now the

This research was supported in part by Ministerio de Economía y Competitividad grant TEC2014-55713-R, Regional Government of Madrid (CM) grant Cloud4BigData (S2013/ICE-2894, cofunded by FSE & FEDER), and grant FPU12/00505 from MECD.

F. Pop and M. Potop-Butucaru (Eds.): ARMS-CC 2015, LNCS 9438, pp. 1–16, 2015.
DOI: 10.1007/978-3-319-28448-4_1

norm instead of the exception, things get even worse. In this work, we apply *speed augmentation* [2,15] (i.e., we increase the computational power of the system's machine) in order to overcome such failures, even in the worst possible scenario. 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 (modeling worst-case scenarios), 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 *size* (or processing time). Specifically we assume that each task τ has size $\pi(\tau) \in [\pi_{min}, \pi_{max}]$, where π_{min} and π_{max} are the smallest and largest possible values, respectively, and $\pi(\tau)$ becomes known to the system at the moment of τ's arrival. Since the scheduling decisions must be made continuously and without knowledge of the future (neither of the task injections nor of the machine crashes and restarts), we look at the problem as an *online* scheduling problem [4,5,18,20,23]. The importance of using speedup lies in this online nature of the problem; the future failures, and the instants of arrival of future tasks along with their sizes, are unpredictable. Thus, there is the need to overcome this lack of information. Epstein et al. [8], specifically show the impossibility of competitiveness in a simple non-preemptive scenario (see Example 2 in [8]). We evaluate the performance of the different scheduling policies (*online algorithms*) under *worst-case* scenarios, on a machine with speedup s, which guarantees efficient scheduling even in the worst of cases. For that, we perform competitive analysis [21]. The four scheduling policies we consider are *Longest In System* (LIS), *Shortest In System* (SIS), *Largest Processing Time* (LPT) and *Shortest Processing Time* (SPT). Scheduling policies LIS and SIS are the popular FIFO and LIFO policies respectively. Graham [12] introduced the scheduling policy LPT a long time ago, when analyzing multiprocessor scheduling. Lee et al. [17] studied the offline problem of minimizing the sum of flow times in one machine with a single breakdown, and gave tight worst-case error bounds on the performance of 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 unfold the relationship between the efficiency measures we consider for each scheduling policy, and the amount of speed augmentation 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. The four algorithms we consider are:

(1) *Longest In System* (LIS): the task that has been waiting the longest is scheduled; i.e., it follows the FIFO *(First In First Out)* policy,
(2) *Shortest In System* (SIS): the task that has been injected the latest is scheduled; i.e., it follows the LIFO *(Last In First Out)* policy,

(3) *Largest Processing Time* (LPT): the task with the biggest size is scheduled, and
(4) *Shortest Processing Time* (SPT): the task with the smallest size is scheduled.

We focus on three *evaluation metrics*, which we regard to embody the most important quality-of-service parameters: the *completed load,* which is the aggregate size of all the tasks that have completed their execution successfully, the *pending load,* which is the aggregate size 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. Latency, is also referred to as *flowtime* in scheduling (e.g., [1,6]). These metrics 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 load), buffering (pending load) 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[1]. The first results we show apply to *all deterministic algorithms* and *all work-conserving algorithms* – algorithms that do not idle while there are pending tasks and do not break the execution of a task unless the machine crashes. We show that, if task sizes are arbitrary, these algorithms cannot be competitive when processors have no resource augmentation ($s = 1$), thus justifying the need of the speedup. Then, for work-conserving algorithms we show the following results: (a) When $s \geq \rho = \frac{\pi_{max}}{\pi_{min}}$, the completed load competitive ratio is lower bounded by $1/\rho$ and the pending load competitive ratio is upper bounded by ρ. (b) When $s \geq 1 + \rho$, the completed load competitive ratio is lower bounded by 1 and the pending load 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 load when tasks have only two possible sizes. 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 load, whereas LIS and SIS require greater speedup to achieve this.

This is the first thorough and rigorous online analysis of these popular scheduling algorithms in a fault-prone setting. In some sense, our results demonstrate in

[1] Most proofs of these results are omitted due to space limit. They will be available in the full version of this paper.

Table 1. General metrics comparison of *ANY* deterministic scheduling algorithm, ALG_D, *ANY* work-conserving one, ALG_W, and detailed metric comparison of the four scheduling algorithms studied in detail. Recall that s represents the speedup of the system's machine, π_{max} and π_{min} the largest and smallest task sizes respectively, and $\rho = \frac{\pi_{max}}{\pi_{min}}$. Note also that, by definition, 0-completed-load competitiveness ratio equals to non-competitiveness, as opposed to the other two metrics, where non-competitiveness corresponds to an ∞ competitiveness ratio.

Algorithm	Condition	Completed Load, $\mathcal{C}$	Pending Load, $\mathcal{P}$	Latency, $\mathcal{L}$
ALG_D	$s = 1$, any task size	0	∞	∞
ALG_W	$s = 1$, any task size	0	∞	∞
ALG_W	$s \geq \rho$	$\geq \frac{1}{\rho}$	$\leq \rho$	–
	$s \geq 1 + \rho$	≥ 1	≤ 1	–
LIS	$s < \rho$, two task sizes	0	∞	∞
	$s \in [\rho, 1 + 1/\rho)$	$[\frac{1}{\rho}, \frac{1}{2} + \frac{1}{2\rho}]$	$[\frac{1+\rho}{2}, \rho]$	$(0, 1]$
	$s \in [\max\{\rho, 1 + \frac{1}{\rho}\}, 2)$	$[\frac{1}{\rho}, \frac{1}{2} + \frac{s}{2}]$	$[\frac{1}{2} + \frac{1}{2(s-1)}, \rho]$	$(0, 1]$
	$s \geq \max\{\rho, 2\}$	$[1, s]$	1	$(0, 1]$
SIS	$s < \rho$, two task sizes	0	∞	∞
	$s \in [\rho, 1 + \frac{1}{\rho})$	$\frac{1}{\rho}$	ρ	∞
	$s \in [1 + \frac{1}{\rho}, 1 + \rho)$	$[\frac{1}{\rho}, \frac{s}{1+\rho}]$	$[\frac{1}{s} + \frac{\rho}{1+\rho}, \rho]$	∞
	$s \geq 1 + \rho$	$[1, s]$	1	∞
LPT	$s < \rho$, two task sizes	0	∞	∞
	$s \geq \rho$	$[1, s]$	1	∞
SPT	$s < \rho$, two task sizes	$[\frac{1}{2+\rho}, \frac{\lfloor (s-1)\rho \rfloor + 1}{\lfloor (s-1)\rho \rfloor + 1 + \rho}]$	∞	∞
	$s \geq \rho$	$[1, s]$	1	∞

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 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 to the online version of the *bin packing* problem [22], 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). Over the years, extensive research on this problem has been done, some of which we consider related to ours. For example, Johnson et al. [13] analyze 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. [9] (see also [22]) 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. Boyar and Ellen [7] have looked into a problem similar to both the 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 [3] 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. We can also directly relate our work to research done on machine scheduling with availability constraints (e.g., [11,19]). One of the most important results in that area is the necessity of online algorithms in case of unexpected 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).

The work of Georgiou and Kowalski [10] was the one that initiated our study. They consider a cooperative computing system of n message-passing processes that are prone to crashes and restarts, and have to collaborate to complete the dynamically injected tasks. For the efficiency of the system, they perform competitive analysis looking at the maximum number of pending tasks. One assumption they widely used was the fact that they considered only unit-length tasks. One of their last results, shows that if tasks have different lengths, even under slightly restricted adversarial patterns, competitiveness is not possible. In [5] we introduced the term of speedup, representing resource augmentation, in order to surpass the non competitiveness shown in [10] and achieve competitiveness in terms of pending load. We found the threshold of necessary speedup under which no algorithm can be competitive, and showed that is also sufficient, proposing optimal algorithms that achieve competitiveness. More precisely, we looked at a system of multiple machines and at least two different task costs, i.e., sizes $\pi \in [\pi_{min}, \pi_{max}]$. We applied distributed scheduling and performed worst-case competitive analysis, considering the pending load competitiveness as our main evaluation metric. 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 load). Additionally, we proposed online algorithms to show that relaxing any of the two conditions is sufficient to achieve competitiveness. In fact, [5] 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.

In [4] we looked at a different setting, of an unreliable communication link between two nodes, and proposed the *asymptotic throughput* for the performance evaluation of scheduling algorithms. We showed that immediate feedback is

necessary to achieve competitiveness and we proved upper and lower bounds for both adversarial and stochastic packet arrivals. More precisely, we considered only two packet lengths, π_{min} and π_{max}, and showed that for adversarial arrivals there is a tight asymptotic throughput, giving upper bound with a fixed adversarial strategy and matching lower bound with an online algorithm we proposed. We also gave an upper bound for the algorithm *Shortest Length*, showing that it is not optimal.

Jurdzinski et al. [14] extended our works [4,5] presenting an optimal online algorithm for the case of k fixed packet lengths, achieving the optimal asymptotic throughput shown in [4]. They also showed that considering resource augmentation (specifically doubling the transmission speed) for faster transmission of the packets, the asymptotic throughput scales. Kowalski et al. [16], inspired by [5], proved that for speedup satisfying conditions (a) and (b) as described above ($s < \min\{\rho, 1 + \gamma/\rho\}$), no deterministic algorithm can be latency-competitive or 1-completed-load-competitive, even in the case of one machine and two task sizes. They then proposed an algorithm that achieves 1-latency-competitiveness and 1-completed-load-competitiveness, as soon as speedup $s \geq 1 + \gamma/\rho$.

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* $a(\tau)$ (simultaneous arrivals are totally ordered) and a *size* $\pi(\tau)$, being the processing time it requires to be completed by a machine running with $s = 1$, and is learned at arrival. We use the term π-task to refer to a task of size $\pi \in [\pi_{min}, \pi_{max}]$ 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 is coordinated with the arrival pattern in order to give 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. 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 keep track of injected, completed and pending tasks 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 Load*, which is the sum of sizes 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 Load*, which is the sum of sizes 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

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

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 [5].

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-load-competitive* if $\forall t, X, A, E$, $C_t^s(\text{ALG}, A, E) \geq \alpha \cdot C_t^1(X, A, E) + \Delta_C$ holds for some parameter Δ_C that *does not* depend on t, X, A or E; α is the completed-load competitive ratio of

ALG, which we denote by $\mathcal{C}(\text{ALG})$. Similarly, it is considered *α-pending-load-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-load competitive ratio of ALG, which we denote by $\mathcal{P}(\text{ALG})$. Finally, algorithm ALG is 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 $\mathcal{L}(\text{ALG})$. Note that α, is independent of t, X, A and E, for the three metrics accordingly.[2]

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

3 Properties of Work-Conserving and Deterministic Algorithms

In this section we present some general properties for all online work-conserving and deterministic algorithms. Obviously, these properties apply to the four policies we focus on in the rest of the paper. The first results show that when there is no speedup these types of algorithms can not be competitive in any of the goodness metrics we use, which justifies the use of speedup in order to achieve competitiveness.

Theorem 1. *If tasks can have* any size *in the range $[\pi_{min}, \pi_{max}]$ and there is no speedup (i.e., $s = 1$),* no work-conserving *algorithm and* no deterministic *algorithm is competitive with respect to the three metrics, i.e. $\mathcal{C}(ALG) = 0$ and $\mathcal{P}(ALG) = \mathcal{L}(ALG) = \infty$.*

The rest of results of the section are positive and show that if the speedup is large enough some competitiveness is achieved.

Lemma 1. *No algorithm X (running without speedup) completes more tasks than a work-conserving algorithm ALG running with speedup $s \geq \rho$. Formally, for any arrival and error patterns A and E, $|N_t(ALG, A, E)| \geq |N_t(X, A, E)|$ and hence $|Q_t(ALG, A, E)| \leq |Q_t(X, A, E)|$.*

[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. 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 corresponding time $t' < t$ such that t' is the latest time instant before t that is either a failure/restart time point or a point where ALG's pending queue is empty. Observe here, that by the definition of t', the queue is never empty within interval $T = (t', t]$. By the induction hypothesis, $|Q_{t'}(\text{ALG})| \leq |Q_{t'}(X)|$.

Let i_T be the number of tasks injected in the interval 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, $|Q_t(X)| \geq |Q_{t'}(X)| + i_T - \left\lfloor \frac{t-t'}{\pi_{min}} \right\rfloor$. As a result, we have that

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

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

The following theorem now follows directly from Lemma 1.

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

Finally, increasing even more the speedup we can show that both competitiveness ratios improve.

Theorem 3. *Any work-conserving algorithm ALG running with speedup $s \geq 1+\rho$, has completed-load competitive ratio $\mathcal{C}(ALG) \geq 1$ and pending-load competitive ratio $\mathcal{P}(ALG) \leq 1$.*

4 Completed and Pending Load Competitiveness

In this section we present a detailed analysis of the four algorithms with respect to the completed and pending load 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-load competitiveness under some cases.

Theorem 4. *NONE of the three algorithms LIS, LPT and SIS is competitive when speedup $s < \rho$, with respect to completed or pending load, even in the case of only two task sizes (i.e., π_{min} and π_{max}).*

Theorem 5. *For speedup $s < \rho$, algorithm SPT cannot have a completed-load competitive ratio more than $\mathcal{C}(SPT) \leq \frac{\lfloor(s-1)\rho\rfloor+1}{\lfloor(s-1)\rho\rfloor+1+\rho}$. Additionally, it is NOT competitive with respect to the pending load, i.e., $\mathcal{P}(SPT) = \infty$.*

Proof. For all speedup $s < \rho$, let us define parameter γ to be the smallest integer such that $\frac{\gamma\pi_{min}+\pi_{max}}{s} > \pi_{max}$ holds. This leads to $\gamma > (s-1)\rho$ and hence we can fix $\gamma = \lfloor(s-1)\rho\rfloor + 1$. 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 $t_k = t_{k-1} + \pi_{max} + \gamma\pi_{min}$. At every t_k time instant there are γ tasks of size 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 γ tasks of size 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 $s < \rho \Rightarrow \pi_{min} < \pi_{max}/s$). Hence, at time t_k, $C_{t_k}(\text{SPT}, A, E) = k\gamma\pi_{min}$ and $C_{t_k}(X, A, E) = k\gamma\pi_{min} + k\pi_{max}$.

Looking at the pending load at such points, we can easily see that SPT's is constantly increasing, while X is able to have pending load 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-load competitive ratio $\mathcal{C}(\text{SPT}) \leq \frac{\gamma}{\gamma+\rho} = \frac{\lfloor(s-1)\rho\rfloor+1}{\lfloor(s-1)\rho\rfloor+1+\rho}$ and a pending load $\mathcal{P}(\text{SPT}) = \infty$. □

We now have a positive result but only for the special case of two task sizes.

Theorem 6. *If tasks can be of only two sizes (π_{min} and π_{max}), algorithm SPT can achieve a completed-load 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 $N_t(\text{SPT})$: (i) The same task τ. (ii) 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 sizes 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 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 size.
- 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, if a crash occurs or the time t is reached before w is completed, task τ is not associated to any task in $N_t(\text{SPT})$. Let t^* be the time one of the two events occurs (a crash occurs or $t^* = t$). Hence SPT is not able to complete task w. Also, since $\tau \notin N_t(\text{SPT})$, it means that τ is not completed by SPT in the interval $[t^*, t]$ either. Hence, 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 size is at most π_{max}.

Now let us evaluate the sizes 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 size 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 at 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 total size 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 size 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 load, still holds in the case of any bounded number of task sizes in the range $[\pi_{min}, \pi_{max}]$.

4.2 Speedup $s \geq \rho$

First, recall that in Theorem 2 we have shown that any work conserving algorithm running with speedup $s \geq \rho$ has pending-load competitive ratio at most ρ and completed-load 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, focusing at first on the

two policies that schedule tasks according to their arrival time, algorithms LIS and SIS.

Theorem 7. *Algorithm LIS has a completed-load competitive ratio* $\mathcal{C}(LIS) \leq \begin{cases} \frac{1}{2} + \frac{1}{2\rho} & s \in [\rho, 1 + 1/\rho) \\ \frac{s}{2} & s \in [1 + 1/\rho, 2) \end{cases}$, *and* $\mathcal{C}(LIS) \geq 1$ *when* $s \geq \max\{\rho, 2\}$. *It also has a pending-load competitive ratio* $\mathcal{P}(LIS) \geq \begin{cases} \frac{1+\rho}{2} & s \in [\rho, 1 + 1/\rho) \\ \frac{s}{2(s-1)} & s \in [1 + 1/\rho, 2) \end{cases}$, *and* $\mathcal{P}(LIS) \leq 1$ *when* $s \geq \max\{\rho, 2\}$. *Recall that* $\rho \geq 1$, *which means that* $1 + \rho \geq 2$.

Theorem 8. *Algorithm SIS has a completed-load competitive ratio* $\mathcal{C}(LIS) \leq \begin{cases} \frac{1}{\rho} & s \in [\rho, 1 + 1/\rho) \\ \frac{s}{1+\rho} & s \in [1 + 1/\rho, 1 + \rho) \end{cases}$, *and* $\mathcal{C}(LIS) \geq 1$ *when* $s \geq 1 + \rho$. *It also has a pending-load competitive ratio* $\mathcal{P}(LIS) \geq \begin{cases} \rho & s \in [\rho, 1 + 1/\rho) \\ \frac{1}{s} + \frac{\rho}{1+\rho} & s \in [1 + 1/\rho, 1 + \rho) \end{cases}$, *and* $\mathcal{P}(LIS) \leq 1$ *when* $s \geq 1 + \rho$.

In contrast with these negative results, we present positive results for algorithms LPT and SPT. It seems then that the nature of these two algorithms (scheduling according to the sizes of tasks rather than their arrival time), gives better results for both the completed and pending load measures.

Theorem 9. *When algorithms LPT and SPT run with speedup* $s \geq \rho$, *they have completed-load competitive ratios* $\mathcal{C}(LPT) \geq 1$ *and* $\mathcal{C}(SPT) \geq 1$ *and pending-load competitive ratios* $\mathcal{P}(LPT) \leq 1$ *and* $\mathcal{P}(SPT) \leq 1$.

5 Latency Competitiveness

In the case of latency, the relationship between the competitiveness ratio and the amount of speed augmentation is more neat for the four scheduling policies.

Theorem 10. *NONE of the algorithms LPT, SIS or 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 a task w be the first task injected by the adversary. Its size is $\pi(w) = \pi_{min}$ if ALG = SPT and $\pi(w) = \pi_{max}$ 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 $k > 1$.

The arrival and error patterns A and E are as follows. At time t_0 task w is injected (with $\pi(w) = \pi_{max}$ 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 size π injected. Observe that π-tasks are such that ALG always gives priority to them over task w. Also, 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 size π injected at time t_{k-1}.

On the other 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, $L_{t'_R}(\mathrm{ALG}) = t'_R$. Then,

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

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

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

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 load for $s < \rho$, neither should it be in terms of latency. Apart from that, the positive result verifies the intuition for competitiveness, since for $s \geq \rho$ algorithm LIS is pending-load competitive **and** it gives priority to the tasks that have been waiting the longest in the system.

Theorem 11. *For speedup $s < \rho$, algorithm LIS is not competitive in terms of latency, i.e., $\mathcal{L}(LIS) = \infty$.*

The proof of the above claim uses the fact that one can force a scenario where LIS attempts to execute the same π_{max}-task forever while a different algorithm can complete infinite π_{min}-tasks.

Theorem 12. *For speedup $s \geq \rho$, algorithm LIS has 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. □

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 load, pending load 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-load and lower bounds for the pending-load and latency competitive ratios. Also, apart from completing the analysis of these four popular algorithms, designing new ones in order to overcome the limitations these present, is another challenging future work. Some other natural next steps are to extend our investigation to the setting with multiple machines, or to consider preemptive scheduling.

References

1. Adiri, I., Bruno, J., Frostig, E., Rinnooy Kan, A.H.G.: Single machine flow-time scheduling with a single breakdown. Acta Informatica **26**(7), 679–696 (1989)
2. Anand, S., Garg, N., Megow, N.: Meeting deadlines: how much speed suffices? In: Aceto, L., Henzinger, M., Sgall, J. (eds.) ICALP 2011, Part I. LNCS, vol. 6755, pp. 232–243. Springer, Heidelberg (2011)
3. Andrews, M., Zhang, L.: Scheduling over a time-varying user-dependent channel with applications to high-speed wireless data. J. ACM **52**(5), 809–834 (2005)
4. Fernández Anta, A., Georgiou, C., Kowalski, D.R., Widmer, J., Zavou, E.: Measuring the impact of adversarial errors on packet scheduling strategies. J. Sched. **18**, 1–18 (2015)
5. Fernández Anta, A., Georgiou, C., Kowalski, D.R., Zavou, E.: Online parallel scheduling of non-uniform tasks. Theor. Comput. Sci. **590**(C), 129–146 (2015)
6. Bansal, N.: Algorithms for flow time scheduling. Ph.D. thesis, IBM (2003)
7. Boyar, J., Ellen, F.: Bounds for scheduling jobs on grid processors. In: Brodnik, A., López-Ortiz, A., Raman, V., Viola, A. (eds.) Space-Efficient Data Structures, Streams, and Algorithms. LNCS, vol. 8066, pp. 12–26. Springer, Heidelberg (2013)
8. Epstein, L., Levin, A., Marchetti-Spaccamela, A., Megow, N., Mestre, J., Skutella, M., Stougie, L.: Universal sequencing on an unreliable machine. SIAM J. Comput. **41**(3), 565–586 (2012)
9. Epstein, L., van Stee, R.: Online bin packing with resource augmentation. Discrete Optim. **4**(34), 322–333 (2007)
10. Georgiou, C., Kowalski, D.R.: On the competitiveness of scheduling dynamically injected tasks on processes prone to crashes and restarts. J. Parallel Distrib. Comput. **84**(C), 94–107 (2015)
11. Gharbi, A., Haouari, M.: Optimal parallel machines scheduling with availability constraints. Discrete Appl. Mathe. **148**(1), 63–87 (2005)
12. Graham, R.L.: Bounds on multiprocessing timing anomalies. SIAM J. Appl. Math. **17**(2), 416–429 (1969)
13. Johnson, D.S., Demers, A., Ullman, J.D., Garey, M.R., Graham, R.L.: Worst-case performance bounds for simple one-dimensional packing algorithms. SIAM J. Comput. **3**(4), 299–325 (1974)
14. Jurdzinski, T., Kowalski, D.R., Lorys, K.: Online packet scheduling under adversarial jamming. In: Bampis, E., Svensson, O. (eds.) WAOA 2014. LNCS, vol. 8952, pp. 193–206. Springer, Heidelberg (2015)
15. Kalyanasundaram, B., Pruhs, K.: Speed is as powerful as clairvoyance [scheduling problems]. In: 1995 Proceedings of the 36th Annual Symposium on Foundations of Computer Science, pp. 214–221, October 1995

16. Kowalski, D.R., Wong, P.W.H., Zavou, E.: Fault tolerant scheduling of non-uniform tasks under resource augmentation. In: Proceedings of the 12th Workshop on Models and Algorithms for Planning and Scheduling Problems, pp. 244–246 (2015)
17. Lee, C.-Y., Liman, S.D.: Single machine flow-time scheduling with scheduled maintenance. Acta Informatica **29**(4), 375–382 (1992)
18. Pruhs, K., Sgall, J., Torng, E.: Online scheduling. In: Leung, J. (ed.) Handbook of Scheduling: Algorithms, Models, and Performance Analysis, pp. 15-1–15-14. CRC Press, Boca Raton (2004)
19. Sanlaville, E., Schmidt, G.: Machine scheduling with availability constraints. Acta Informatica **35**(9), 795–811 (1998)
20. Schwan, K., Zhou, H.: Dynamic scheduling of hard real-time tasks and real-time threads. IEEE Trans. Softw. Eng. **18**(8), 736–748 (1992)
21. Sleator, D.D., Tarjan, R.E.: Amortized efficiency of list update and paging rules. Commun. ACM **28**(2), 202–208 (1985)
22. van Stee, R.: Online scheduling and bin packing. Ph.D. thesis (2002)
23. Yao, F., Demers, A., Shenker, S.: A scheduling model for reduced cpu energy. In: 1995 Proceedings of the 36th Annual Symposium on Foundations of Computer Science, pp. 374–382, October 1995