ARTICLE IN PRESS

Future Generation Computer Systems ▮ (▮▮▮▮) ▮▮▮–▮▮▮

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Competitive analysis of fundamental 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,*]

[a] IMDEA Networks Institute, Av. del Mar Mediterráneo 22, 28918 Leganés, Madrid, Spain

[b] University of Cyprus, Department of Computer Science, 75 Kallipoleos Str., P.O. Box 20537, 1678 Nicosia, Cyprus

[c] University of Liverpool, Department of Computer Science, Ashton Building, Ashton Street, Liverpool L69 3BX, United Kingdom

[d] University Carlos III of Madrid, Department of Telematics Engineering, Torres Quevedo Building, Av. Universidad 30, 28911 Leganés, Madrid, Spain

HIGHLIGHTS

- Worst-case analysis of fault-tolerant properties of popular scheduling algorithms.
- Competitive analysis regarding completed/pending load and latency of the algorithms.
- Use of resource augmentation to achieve and/or to improve their performance.
- Differences of scheduling policies based either on *arrival time* or *size* of tasks.
- All deterministic and work-conserving algorithms require speedup to be competitive.

ARTICLE INFO

Article history:
Received 11 November 2015
Received in revised form
27 May 2016
Accepted 29 May 2016
Available online xxxx

Keywords:
Scheduling
Online algorithms
Different task processing times
Failures
Competitive analysis
Resource augmentation

ABSTRACT

Reliable task execution in machines that are prone to unpredictable crashes and restarts is both challenging and of high importance, 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).

1. Introduction

Motivation. 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 norm instead of the exception, things get even worse. In this work, we apply *speed augmentation* [1,2] (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 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 $\pi_{\min}$ and $\pi_{\max}$ are the smallest and largest possible values respectively, and

* Corresponding author at: IMDEA Networks Institute, Av. del Mar Mediterráneo 22, 28918 Leganés, Madrid, Spain.
E-mail addresses: antonio.fernandez@imdea.org (A. Fernández Anta), chryssis@cs.ucy.ac.cy (C. Georgiou), D.Kowalski@liverpool.ac.uk (D.R. Kowalski), elli.zavou@imdea.org (E. Zavou).

http://dx.doi.org/10.1016/j.future.2016.05.042
0167-739X/

$\pi(\tau)$ 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 [3–7]. 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 [9]. 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 [10] introduced the scheduling policy LPT a long time ago, when analyzing multiprocessor scheduling. Lee et al. [11] 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., [12,13]). 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. 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 they 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 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 [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). Over the years, extensive research on this problem has been done, some of which we consider related to ours. For example, Johnson et al. [16] 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. [17] (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. Boyar and Ellen [18] 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 [19] 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., [20,21]). One of the most important results in that area

Please cite this article in press as: A. Fernández Anta, et al., Competitive analysis of fundamental scheduling algorithms on a fault-prone machine and the impact of resource augmentation, Future Generation Computer Systems (2016), http://dx.doi.org/10.1016/j.future.2016.05.042

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. Also, the last column provides the theorem numbers where the results of the corresponding row can be found. 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 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.

Alg.	Condition	Completed load, $\mathcal{C}$	Pending load, $\mathcal{P}$	Latency, $\mathcal{L}$	Thm.
ALG_D	$s = 1$, any task size	0	∞	∞	1, [4,14]
ALG_W	$s = 1$, any task size	0	∞	∞	1, [4,14]
ALG_W	$s \geq \rho$	$[1/\rho, 1]$	$[1, \rho]$	–	2, 3
	$s \geq 1 + \rho$	1	1	–	2, 4
LIS	$s < \rho$, two task sizes	0	∞	∞	5, 12
	$s \in [\rho, 1 + 1/\rho)$	$[1/\rho, \frac{1}{2} + \frac{1}{2\rho}]$	$[\frac{1+\rho}{2}, \rho]$	(0, 1]	8, 13
	$s \in [\max\{\rho, 1 + \frac{1}{\rho}\}, 2)$	$[1/\rho, s/2]$	$[\frac{s}{2(s-1)}, \rho]$	(0, 1]	8, 13
	$s \geq \max\{\rho, 2\}$	1	1	(0, 1]	4, 8, 13
SIS	$s < \rho$, two task sizes	0	∞	∞	5, 11
	$s \in [\rho, 1 + 1/\rho)$	$\frac{1}{\rho}$	ρ	∞	9, 11
	$s \in [1 + 1/\rho, 1 + \rho)$	$[1/\rho, s/(1 + \rho)]$	$[\frac{1}{s} + \frac{\rho}{1+\rho}, \rho]$	∞	9, 11
	$s \geq 1 + \rho$	1	1	∞	4, 9, 11
LPT	$s < \rho$, two task sizes	0	∞	∞	5, 11
	$s \geq \rho$	1	1	∞	10, 11
SPT	$s < \rho$, two task sizes	$[\frac{1}{2+\rho}, \frac{\lfloor(s-1)\rho\rfloor+1}{\lfloor(s-1)\rho\rfloor+1+\rho}]$	∞	∞	6, 7, 11
	$s \geq \rho$	1	1	∞	10, 11

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 [22] 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 [4] we introduced the term of speedup, representing resource augmentation, in order to surpass the NP-hardness shown in [22] 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, [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.

In [5] 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 packets 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 algorithm *Shortest Length*, showing that it is not optimal.

Jurdzinski et al. [23] 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 [5]. 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. [14], inspired by [4] 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

Please cite this article in press as: A. Fernández Anta, et al., Competitive analysis of fundamental scheduling algorithms on a fault-prone machine and the impact of resource augmentation, Future Generation Computer Systems (2016), http://dx.doi.org/10.1016/j.future.2016.05.042

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 superscript 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

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

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-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 $\forall t, X, A, E, P_t^s(\text{ALG}, A, E) \leq \alpha \cdot P_t^1(X, A, E) + \Delta_P$, 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 $\forall t, X, A, E$, $L_t^s(\text{ALG}, A, E) \leq \alpha \cdot L_t^1(X, A, E) + \Delta_L$, 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.[1]

[1] Parameters Δ_C, Δ_P, Δ_L as well as α may depend on system parameters like $\pi_{\min}$, $\pi_{\max}$ or s, which are not considered as inputs of the problem.

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.

3.1. Negative results

We first present some *negative* results, in the sense that they are upper bounds when looking at the completed-load competitiveness and lower bounds when looking at the pending-load and latency competitiveness. The first results show that when there is no speedup these types of algorithms cannot be competitive in any of the goodness metrics we use, which justifies the use of speedup in order to achieve competitiveness (see Theorem 1). We then show that even with the use of speedup, the achievable competitiveness is limited (see Theorem 2).

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 $\mathcal{P}(ALG) = \mathcal{L}(ALG) = \infty$.*

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

The proof of Theorem 1 follows directly from Lemmas 1 and 2, in combination with the non-competitiveness results in [4,14]. The lemmas show the non completed load competitiveness, while the results in [4,14] show the non pending load competitiveness and non latency competitiveness respectively. The proof of Theorem 2 follows directly from Lemmas 3 and 4, showing the two ratio bounds separately.

Lemma 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 ALG is competitive with respect to completed load, i.e. $\mathcal{C}(ALG) = 0$.*

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 size $\pi(w_k) = \pi_{\min} + \Delta(k)$, 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 $t'_k = t_{k-1} + \frac{1-\epsilon}{2}\Delta(k-1)$. 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.

Please cite this article in press as: A. Fernández Anta, et al., Competitive analysis of fundamental scheduling algorithms on a fault-prone machine and the impact of resource augmentation, Future Generation Computer Systems (2016), http://dx.doi.org/10.1016/j.future.2016.05.042

The error pattern E is such that at every time instant t_k there is a crash and restart.

We compare all work-conserving algorithms ALG with an algorithm X of our choice. In the execution of ALG, task w_0 is scheduled as soon as it arrives, at time t_0 (it is the only task pending). On the other hand, X waits until time t'_1 for the arrival of w_1 and schedules it immediately. When the processor crashes at time t_1 the task w_0 executed by ALG 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 $t'_1 + \pi(w_1) = t_0 + \frac{1-\epsilon}{2}\Delta(0) + \pi_{\min} + \Delta(1) = t_0 + \pi_{\min} + \frac{1+\epsilon}{2}\Delta(0) = t_1$. After the restart at t_1, ALG schedules one of the pending tasks $\{w_0, w_1\}$, while X waits until t'_2 to schedule the next task to be injected, w_2.

The general process is as follows. At time instant t_k, ALG schedules one of the pending tasks in $\{w_0, w_1, \ldots, w_k\}$ while X waits until the next task w_{k+1} is injected at time t'_{k+1} and schedules it. When the processor crashes at time t_{k+1} the scheduled by ALG is interrupted, since $t_{k+1} - t_k = \pi_{\min} + \frac{1+\epsilon}{2}\Delta(k) < \pi(w_k) = \pi_{\min} + \Delta(k)$ and all possible tasks scheduled by ALG are at least $\pi(w_k)$ long. However, X is able to complete task w_{k+1} because $t'_{k+1} + \pi(w_{k+1}) = t_k + \frac{1-\epsilon}{2}\Delta(k) + \pi_{\min} + \Delta(k+1) = t_k + \pi_{\min} + \frac{1+\epsilon}{2}\Delta(k) = t_{k+1}$.

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

Lemma 2. *If tasks can have any size in the range $[\pi_{\min}, \pi_{\max}]$ and there is no speedup (i.e. $s = 1$), no deterministic algorithm ALG is competitive with respect to completed load, i.e. $\mathcal{C}(ALG) = 0$.*

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 size $\pi(w_k) = \pi_{\min} + \Delta(k)$, 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 $t'_k = t_{k-1} + \frac{1-\epsilon}{2}\Delta(k-1)$. The arrival pattern A is such that task $w_0 = \pi_{\max}$ is injected in the system at time instant t_0. Then, for $k = 1, 2, \ldots$ two identical tasks w_k are injected at time t'_k.

Now consider current time instant being t_k. Since ALG is deterministic, the adversary knows what decisions the algorithm will take. There are two cases we need to examine:

(a) If ALG schedules a task before t'_{k+1}, then X waits until task w_{k+1} is injected at time t'_{k+1} and schedules it. Then, crashes the machine right after the w_{k+1} is completed. Note that X will complete the task at time $t'_{k+1} + \pi(w_{k+1}) = t_k + \frac{1-\epsilon}{2}\Delta(k) + \pi_{\min} + \Delta(k+1) = t_k + \pi_{\min} + \frac{1+\epsilon}{2}\Delta(k) = t_{k+1}$. On the other hand, ALG will not be able to complete the task that was scheduled before t'_{k+1}. This is because, $t_{k+1} - t_k = \pi_{\min} + \frac{1+\epsilon}{2}\Delta(k) < \pi(w_k)$ and all possible tasks that ALG could have scheduled before t'_{k+1} are of size at least $\pi(w_k)$.

(b) If ALG does not schedule any task before t'_{k+1}, then X schedules a packet w_k at time t_k and the machine is not crashed until it is completed. Also, a new task w_k is injected only after the completion of the one scheduled. Then, we move to the same state we had at t_k. Observe that X will complete the w_k task at time $t^* = t_k + \pi(w_k) = t_k + \pi_{\min} + \Delta(k)$. On the same time, if ALG schedules any of the available tasks at time t'_{k+1}, say the smallest possible w_k, it will only be able to complete is by $t'_{k+1} + \pi(w_k) = t_k + \frac{1-\epsilon}{2}\Delta(k) + \pi_{\min} + \Delta(k) = t_k + \pi_{\min} + \frac{3-\epsilon}{2}\Delta(k)$, which is bigger than the previously defined t^*.

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

Lemma 3. *Any work-conserving algorithm ALG running with speedup s, has a completed-load competitive ratio $\mathcal{C}(ALG) \leq 1$, more precisely in executions where $Q_t(ALG) = \emptyset$ infinitely many times.*

Proof. Let us consider an adversary that causes 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 $t_k = t_{k-1} + \pi$, where $k = 0, 1, 2 \ldots$ 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 T_i the time interval $[t_i, t_{i+1}]$. 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, $\mathcal{C}(\text{ALG}) \leq 1$ as claimed. ■

Lemma 4. *Any work-conserving algorithm ALG running with speedup s, has a pending-load competitive ratio $\mathcal{P}(ALG) \geq 1$, more precisely in executions where the 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 size $\pi \in [\pi_{\min}, \pi_{\max}]$ 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 will be able to complete a π-task, while ALG will complete up to πs total load. Assuming that there are no phases of length less than $\pi_{\min}$, the complementing pending load 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 load 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. ■

3.2. Positive results

We then present some *positive* results, in the sense that they show that if the speedup is large enough some competitiveness is achieved.

Lemma 5. *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 $|Q_t(ALG, A, E)| \leq |Q_t(X, A, E)|$.*

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

Please cite this article in press as: A. Fernández Anta, et al., Competitive analysis of fundamental scheduling algorithms on a fault-prone machine and the impact of resource augmentation, Future Generation Computer Systems (2016), http://dx.doi.org/10.1016/j.future.2016.05.042

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 $\pi_{\min}$ 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 5.

Theorem 3. *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 4. *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 $\mathcal{P}(ALG) \leq 1$.*

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 size 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 size 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, and therefore, $\mathcal{P}(\text{ALG}, A, E) \leq \mathcal{P}(X, A, E)$.

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 $\pi \in [\pi_{\min}, \pi_{\max}]$, which is the size of the task completed by X. In that interval ALG is continuously executing tasks. Hence, in the interval $(t', t]$ it completes tasks whose aggregate size is at least $\pi s - \pi_{\max}$. Then, the pending load at time instant t of both algorithms satisfy $P_t(X, A, E) = P_{t'}(X, A, E) + i_T - \pi$ while $P_t(\text{ALG}, A, E) \leq P_{t'}(\text{ALG}, A, E) + i_T - (\pi s - \pi_{\max})$. Observe that $s \geq 1 + \rho$ implies that $\pi s - \pi_{\max} \geq \pi$. Hence, from the induction hypothesis, $P_t(\text{ALG}, A, E) \leq P_t(X, A, E)$.

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

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 $s \geq \rho$.

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 circumstances.

Lemma 6. *When algorithms LIS and LPT run with speedup $s < \rho$, they both have a completed-load competitive ratio $\mathcal{C}(LIS) = \mathcal{C}(LPT) = 0$ and a pending-load 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 to prove the non-competitiveness of both algorithms. We consider an infinite arrival pattern which injects one $\pi_{\max}$-task at the beginning of the execution, $t = 0$, and after that it keeps injecting one $\pi_{\min}$-task every $\pi_{\min}$ 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 $\pi_{\min}$-tasks injected, while neither LIS nor LPT will manage to complete any task, running with speedup $s < \rho$, since they will both insist on scheduling the $\pi_{\max}$-task injected at the beginning. In an interval of length $\pi_{\min}$, algorithm X is able to complete a $\pi_{\min}$-task but neither LIS nor LPT can complete the $\pi_{\max}$-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, and so will the total of their pending sizes. At the same time, X is able to keep its pending tasks bounded, with no more than one $\pi_{\max}$ and one $\pi_{\min}$ tasks. As for the total size 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-load competitive ratios $\mathcal{C}(\text{LIS}) = \mathcal{C}(\text{LPT}) = 0$ and pending-load competitive ratios $\mathcal{P}(\text{LIS}) = \mathcal{P}(\text{LPT}) = \infty$ as claimed, which completes the proof. ■

Lemma 7. *When algorithm SIS runs with speedup $s < \rho$, it has a completed-load competitive ratio $\mathcal{C}(SIS) = 0$ and a pending-load 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 load and the pending load 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 of the machine and then an immediate injection

of a $\pi_{\min}$-task followed by a $\pi_{\max}$-task. This creates *active* intervals $[t_i, t_{i+1})$ of length $\pi_{\min}$.

It is easy to observe that the patterns described cause algorithm SIS to assign the last $\pi_{\max}$-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 $\pi_{\min}$ and SIS needs $\frac{\pi_{\max}}{s} > \pi_{\min}$ time to complete the $\pi_{\max}$-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 $\pi_{\min}$-tasks injected, one in every alive interval, giving a completed load 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 an algorithm X'. We define time instants $t_{k'}$, where $k' = 1, 2, \ldots$ as $t_{k'} = t_{k'-1} + \kappa\pi_{\min}$, with time $t_0 = 0$ being the beginning of the execution. At every such time instant there are κ $\pi_{\min}$-tasks injected followed by a $\pi_{\max}$-task. The crashes of the machine are set at time instants $t_{k'}$ as well as $t_{k'} + i\pi_{\min}$ where $i = 1, 2, \ldots, \kappa$. This creates κ *alive* intervals of length $\pi_{\min}$ between $t_{k'}$ and $t_{k'+1}$.

The arrival pattern A' causes algorithm SIS to schedule the last $\pi_{\max}$-task injected right after time instant $t_{k'}$. However, since all alive intervals are of length $\pi_{\min}$ and $s < \rho$, created by the error pattern E', algorithm SIS can never complete the $\pi_{\max}$-task scheduled, nor any other injected task (does not even get them scheduled). On the same time though, algorithm X' is able to complete the κ $\pi_{\min}$-tasks injected at the last $t_{k'}$ time instant. As a result, looking right before the injection at a time instant t_i in the execution, the pending-load competitive ratio will be $\mathcal{P}_i(\text{SIS}) = \frac{i\kappa\pi_{\min}+i\pi_{\max}}{i\pi_{\max}} = 1 + \frac{\kappa}{\rho}$. Hence, the more $\pi_{\min}$-tasks are injected at every $t_{k'}$ (i.e. the bigger the κ), the bigger the pending-load competitiveness of SIS, growing to infinity.

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

Combining now Lemmas 6 and 7 we have the following theorem.

Theorem 5. *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., $\pi_{\min}$ and $\pi_{\max}$).*

Surprisingly, for algorithm SPT we are not able to prove zero completed-load competitiveness when $s < \rho$. This will be later justified by the fact that for two task sizes SPT achieves a positive completed-load competitiveness, cf., Theorem 7. We can however, prove the following upper bound restriction for the completed-load of algorithm SPT.

Theorem 6. *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 $\pi_{\min}$ injected along with one $\pi_{\max}$-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 $\pi_{\min}$ 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 $\pi_{\min}$, 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 $\pi_{\max}$ ones never have the time to be executed; a $\pi_{\max}$-task is scheduled at the last phase of each epoch which is of size $\pi_{\min}$ (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 $\mathcal{P}(\text{SPT}) = \infty$. ■

Then, restricting the number of different task sizes introduced by the adversary, we can show a positive result for algorithm SPT. More specifically, as shown in the following theorem, non-zero completed-load competitiveness is guaranteed when only two task sizes are introduced.

Theorem 7. *If tasks can be of only two sizes ($\pi_{\min}$ and $\pi_{\max}$), 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 size of the tasks in $N_t(X)$ that are not associated with any task in $N_t(\text{SPT})$ is at most $\pi_{\max}$. More specifically, there is only one task execution of a $\pi_{\max}$-task by SPT, namely w, such that the $\pi_{\min}$-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 $\pi_{\max}$-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 $\pi_{\min}$-tasks and have been scheduled and completed during the execution of the same $\pi_{\max}$-task by SPT. Hence, their aggregate size is at most $\pi_{\max}$.

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

Please cite this article in press as: A. Fernández Anta, et al., Competitive analysis of fundamental scheduling algorithms on a fault-prone machine and the impact of resource augmentation, Future Generation Computer Systems (2016), http://dx.doi.org/10.1016/j.future.2016.05.042

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 task starts its execution no earlier than $f(w) - \pi(w)$ and in the extreme case a $\pi_{\max}$-task could have been scheduled at the end of T_w and completed at $t_w + \pi_{\max}$. Hence, if task w is a $\pi_{\min}$-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 $\pi_{\max}$-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 3 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.

Lemma 8. *When algorithm LIS runs with speedup* $s \in [\rho, 1 + 1/\rho)$, *it has a completed-load competitive ratio* $\mathcal{C}(LIS) \leq \frac{1}{2} + \frac{1}{2\rho}$ *and a pending-load 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 $\pi_{\min}$-task injected, followed by a $\pi_{\max}$-task. After every period of $\pi_{\max}$ 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 $\pi_{\max}$, during which an algorithm X can successfully execute the $\pi_{\max}$ 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, either a $\pi_{\min}$-task or a $\pi_{\max}$-task. Observe also, that after k phases, where k is a multiple of 2, there will be exactly k tasks of size $\pi_{\min}$ pending in the queue of X, while LIS will have pending half of the tasks injected, half of which are of size $\pi_{\min}$ and the other half $\pi_{\max}$. Hence, the pending-load 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-load 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 9. *When algorithm LIS runs with speedup* $s \in [1 + 1/\rho, 2)$, *where* $s \geq \rho$ *as well, it has a completed-load competitive ratio* $\mathcal{C}(LIS) \leq \frac{s}{2}$ *and a pending-load competitive ratio* $\mathcal{P}(LIS) \geq \frac{s}{2(s-1)}$.

Proof. Let speedup $s \in [1 + 1/\rho, 2)$. We define a combination of arrival and error patterns A and E, algorithm X, and consider tasks of sizes $\pi_{\min}$ and π, where $\pi \in (\pi_{\min}, \pi_{\max})$ such that $\frac{\pi_{\min}+\pi}{s} > \pi \Rightarrow \pi < \frac{\pi_{\min}}{s-1}$. Note that such a value π always exists since $s \in [1 + 1/\rho, 2)$. More specifically, let us define $\pi = \varepsilon \frac{\pi_{\min}}{s-1}$, where $\varepsilon \in (0, 1)$.

Patterns A and E behave as follows: We define time instants $t_k = t_{k-1} + \pi$, where $k = 0, 1, 2, \ldots$ and $t_0 = 0$ is the beginning of the execution. At each t_k time instant there is a machine crash and restart followed by an injection of a $\pi_{\min}$-task and then a task of size 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 $\pi_{\min}$-task or a π-task. Algorithm LIS schedules the tasks by their arrival times (ascending order). However, by the definition of size π, algorithm LIS cannot complete both a $\pi_{\min}$ and a π-task in a period of length π. 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 size $\pi_{\min}$ and $k/2$ tasks of size π while X will complete k tasks of size π. Hence, the completed-load competitive ratio of LIS becomes $\mathcal{C}(\text{LIS}) = \frac{1}{2} + \frac{\pi_{\min}}{2\pi} = \frac{1}{2} + \pi_{\min}/(2\varepsilon\frac{\pi_{\min}}{s-1}) = \frac{1}{2} + \frac{s-1}{2\varepsilon}$. Respectively, at such time instants $P_{t_k}(\text{LIS}) = \frac{k(\pi_{\min}+\pi)}{2}$ while $P_{t_k}(X) = k\pi_{\min}$. Hence, the pending-load competitive ratio $\mathcal{P}(\text{LIS}) = \frac{1}{2} + \frac{\pi}{2\pi_{\min}} = \frac{1}{2} + (\varepsilon\frac{\pi_{\min}}{s-1})/(2\pi_{\min}) = \frac{1}{2} + \frac{\varepsilon}{2(s-1)}$.

This leads to an upper bound $\mathcal{C}(\text{LIS}) \leq \frac{s}{2}$ and a lower bound $\mathcal{P}(\text{LIS}) \geq \frac{s}{2(s-1)}$ as claimed. Let us assume otherwise, i.e., $\mathcal{C}(\text{LIS}) > \frac{s}{2}$. This means that there exists a parameter $\delta \in (0, 1)$ such that $\mathcal{C}(\text{LIS}) \geq \frac{1}{2} + \frac{s-1}{2\delta}$. Parameter ε mentioned above can be made such that $\varepsilon > \delta$ and $\mathcal{C}_\varepsilon(\text{LIS}) = \frac{1}{2} + \frac{s-1}{2\varepsilon} < \frac{1}{2} + \frac{s-1}{2\delta}$. For the pending-load competitiveness a similar approach can be followed. ■

Lemma 10. *When algorithm LIS runs with speedup* $s \in [2, 1 + \rho)$, *where* $s \geq \rho$ *as well, it has a completed-load competitive ratio* $\mathcal{C}(LIS) \geq 1$ *and a pending-load competitive ratio* $\mathcal{P}(LIS) \leq 1$.

Proof. Let speedup $s \in [2, 1 + \rho)$ and let us analyze first the completed load 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 earliest 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_is are the processor's crashing points within the interval. Let us denote by $C_i(X)$ and $C_i(\text{LIS})$ the load 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 (recall that $s \geq \rho$). 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 load by X within the interval $[t', t^*]$ is larger than the total completed load 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 total load 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 remains 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 $\forall j \in [2, k']$,

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

Please cite this article in press as: A. Fernández Anta, et al., Competitive analysis of fundamental scheduling algorithms on a fault-prone machine and the impact of resource augmentation, Future Generation Computer Systems (2016), http://dx.doi.org/10.1016/j.future.2016.05.042

From Eqs. (1) and (2), we have that $C_j(X) > C_j(\text{LIS}) > s \cdot C_j(X) - \pi(\tau_{j+1})$. Since $s \geq 2$, the following is implied

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

What is more, from Eqs. (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 Eq. (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 total load in some periods, LIS will eventually surpass its performance.

Since the pending load is complementary to the completed load 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-load and pending-load competitive ratios being optimal for algorithm LIS when speedup $s \in [2, 1+\rho)$. ■

Combining Lemmas 8–10 and Theorem 4 we have the following theorem.

Theorem 8. *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* $1 + \rho \geq 2$.

The following lemmas analyze the efficiency of algorithm SIS in a similar way, looking at different speedup intervals for which $s \geq \rho$ always holds.

Lemma 11. *When algorithm SIS runs with speedup* $s \in [\rho, 1+1/\rho)$, *it has a complete-load competitive ratio* $\mathcal{C}(SIS) \leq 1/\rho$ *and a pending-load competitive ratio* $\mathcal{P}(SIS) \geq \rho$.

Proof. Let speedup $s \in [\rho, 1 + 1/\rho)$. We define a combination of arrival and error patterns A and E, and algorithm X as follows: At the beginning of the execution there is a $\pi_{\max}$-task injected, followed by a $\pi_{\min}$-task. After every period of $\pi_{\max}$ time there is a crash and restart of the machine, followed by the same injection sequence (a $\pi_{\max}$-task and then a $\pi_{\min}$-task).

This behavior results to the following execution. There are only active phases of size $\pi_{\max}$, during which an algorithm X can successfully execute the $\pi_{\max}$ tasks injected. At the same time, SIS schedules the task injected the latest. Observe that, since $s < 1 + 1/\rho = (\pi_{\min} + \pi_{\max})/\pi_{\max}$, SIS is able to complete only one task in each phase; only the $\pi_{\min}$-task injected. Observe also, that after k phases, there will be exactly k tasks of size $\pi_{\min}$ pending in the queue of X, while SIS will have pending k tasks of size $\pi_{\max}$. Hence, the completed-load competitive ratio of SIS becomes $\mathcal{C}(\text{SIS}) = \frac{\pi_{\min}}{\pi_{\max}} = 1/\rho$ and its pending-load competitive ratio becomes $\mathcal{P}(\text{SIS}) = \frac{\pi_{\max}}{\pi_{\min}} = \rho$, which completes the proof. ■

Lemma 12. *When algorithm SIS runs with speedup* $s \in [1+1/\rho, 1+\rho)$, *where* $s \geq \rho$ *as well, it has a completed-load competitive ratio* $\mathcal{C}(SIS) \leq \frac{s}{1+\rho}$ *and a pending-load competitive ratio* $\mathcal{P}(SIS) \geq \frac{1}{s} + \frac{\rho}{1+\rho}$.

Proof. Let speedup $s \in [1 + 1/\rho, 1 + \rho)$. We define a combination of arrival and error patterns A and E, algorithm X and consider tasks of sizes $\pi_{\max}$, $\pi_{\min}$ 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 $s \in [1+1/\rho, 1+\rho)$. More specifically, let us define $\pi = \varepsilon \frac{\pi_{\min}+\pi_{\max}}{s}$, where $\varepsilon \in (0, 1)$.

Patterns A and E behave as follows: We define time instants $t_k = t_{k-1} + \pi$, where k is an increasing positive integer ($k = 0, 1, 2, \ldots$), with $t_0 = 0$ being the beginning of the execution. At each time t_k there is exactly one π-task injected, followed by one $\pi_{\max}$-task, followed by one $\pi_{\min}$-task. Crashes and restarts are also set at times t_k, before the new injection, causing *active* intervals of duration π.

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 $\pi_{\min}$-task followed by the latest $\pi_{\max}$-task, and hence being able to complete only the $\pi_{\min}$-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-load competitive ratio of algorithm SIS becomes

$$\mathcal{C}(\text{SIS}) = \frac{\pi_{\min}}{\pi} = \frac{\pi_{\min}}{\varepsilon \frac{\pi_{\min}+\pi_{\max}}{s}} = \frac{s}{\varepsilon(1+\rho)}$$

and its pending-load competitive ratio

$$\begin{aligned}
\mathcal{P}(\text{SIS}) &= \frac{\pi + \pi_{\max}}{\pi_{\min} + \pi_{\max}} = \frac{\varepsilon \frac{\pi_{\min}+\pi_{\max}}{s} + \pi_{\max}}{\pi_{\min} + \pi_{\max}} \\
&= \frac{\varepsilon(1+\rho) + s\rho}{s(1+\rho)} = \frac{\epsilon}{s} + \frac{\rho}{1+\rho}.
\end{aligned}$$

Please cite this article in press as: A. Fernández Anta, et al., Competitive analysis of fundamental scheduling algorithms on a fault-prone machine and the impact of resource augmentation, Future Generation Computer Systems (2016), http://dx.doi.org/10.1016/j.future.2016.05.042

This leads to the upper and lower bounds claimed, i.e., $\mathcal{C}(\text{SIS}) \leq \frac{s}{1+\rho}$ and $\mathcal{P}(\text{SIS}) \geq \frac{1}{s} + \frac{\rho}{1+\rho}$.

Let us assume otherwise, i.e., $\mathcal{C}(\text{SIS}) > \frac{s}{1+\rho}$. This means that there exists a parameter $\delta \in (0, 1)$ such that $\mathcal{C}(\text{SIS}) \geq \frac{s}{\delta(1+\rho)}$. Parameter ε mentioned above can be made such that $\varepsilon > \delta$ and $\mathcal{C}_\varepsilon(\text{SIS}) = \frac{s}{\varepsilon(1+\rho)} < \frac{s}{\delta(1+\rho)}$. For the pending-load competitiveness a similar approach can be followed. ■

Combining Lemmas 11, 12 and Theorem 4, we have the following theorem.

Theorem 9. *Algorithm SIS has a completed-load competitive ratio*

$$\mathcal{C}(SIS) \leq \begin{cases} 1/\rho & s \in [\rho, 1 + 1/\rho) \\ \dfrac{s}{1+\rho} & s \in [1 + 1/\rho, 1 + \rho), \end{cases}$$

and $\mathcal{C}(SIS) \geq 1$ *when* $s \geq 1 + \rho$.

It also has a pending-load competitive ratio

$$\mathcal{P}(SIS) \geq \begin{cases} \rho & s \in [\rho, 1 + 1/\rho) \\ \dfrac{1}{s} + \dfrac{\rho}{1+\rho} & s \in [1 + 1/\rho, 1 + \rho), \end{cases}$$

and $\mathcal{P}(SIS) \leq 1$ *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 algorithms (scheduling according to the sizes of tasks rather than their arrival time), gives better results for both the completed and pending load measures.

Lemma 13. *When algorithm LPT runs with speedup* $s \geq \rho$, *it has completed-load competitive ratio* $\mathcal{C}(LPT) \geq 1$ *and pending-load competitive ratio* $\mathcal{P}(LPT) \leq 1$.

Proof. As proven in Lemma 5, 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 size, the ones completed will be of the maximum size available at all times, which trivially results to a total completed load 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-load competitive ratio of $\mathcal{C}(\text{LPT}) \geq 1$, as claimed.

For the pending-load competitiveness let us use the fact that at any time of any execution the sum of completed and pending task load sums up to the same total load independent of the algorithm; i.e., $\forall t, A, E, X, C_t(\text{ALG}) + P_t(\text{ALG}) = C_t(X) + P_t(X)$. Let us denote it by K. This holds for LPT as well, hence $C_t(\text{LPT}) + P_t(\text{LPT}) = K$. We have already shown that $C_t(\text{LPT}) \geq C_t(X)$. Hence replacing with the corresponding expressions for the pending load, $K - P_t(\text{LPT}) \geq K - P_t(X)$ which leads to $P_t(\text{LPT}) \leq P_t(X)$ and $\mathcal{P}_t(\text{LPT}) \leq 1$ as claimed. ■

Lemma 14. *When algorithm SPT runs with speedup* $s \geq \rho$, *it has completed-load competitive ratio* $\mathcal{C}(SPT) \geq 1$ *and pending-load 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 load of SPT is more than that of an algorithm X, i.e., $C(\text{SPT}) \geq C(X)$.

By contradiction, we assume a point in time t to be 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^-, $C_{t^-}(\text{SPT}) \geq C_{t^-}(X)$.

We first consider the case where X has completed a $\pi_{\min}$-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. Since $s \geq \rho > 1$, it holds that $C_{t^*}(\text{SPT}) \geq C_{t-\pi_{\min}}(\text{SPT}) + \pi_{\min}$. At the same time, $C_{t^*}(X) = C_{t-\pi_{\min}}(X)$. At time t, algorithm SPT has the same completed load as at time t^*, whereas X's completed load increases by $\pi_{\min}$. Hence

$$\begin{aligned} C_t(\text{SPT}) &= C_{t^*}(\text{SPT}) \geq C_{t-\pi_{\min}}(\text{SPT}) + \pi_{\min} \\ &\geq C_{t-\pi_{\min}}(X) + \pi_{\min} = C_t(X), \end{aligned}$$

which contradicts the initial assumption.

We then consider the case where X has completed a $\pi_{\max}$-task. This means that during the interval $(t - \pi_{\max}, t)$ no machine failure has occurred and hence algorithm SPT was also able to complete some tasks. Let t^* be the last time in $(t - \pi_{\max}, t)$ that SPT completes a task. Since $s \geq \rho > 1$, it holds that $C_{t^*}(\text{SPT}) \geq C_{t-\pi_{\max}}(\text{SPT}) + \pi_{\max} = C_{t-\pi_{\max}}(\text{SPT}) + \rho\pi_{\min}$. At the same time, $C_{t^*}(X) = C_{t-\pi_{\max}}(X)$. At time t, algorithm SPT has the same completed load as at time t^*, whereas X's completed load increases by $\pi_{\max}$. Hence

$$\begin{aligned} C_t(\text{SPT}) &= C_{t^*}(\text{SPT}) \geq C_{t-\pi_{\max}}(\text{SPT}) + \pi_{\max} \\ &\geq C_{t-\pi_{\max}}(X) + \pi_{\max} = C_t(X), \end{aligned}$$

which again contradicts the initial assumption.

We have therefore shown that $C(\text{SPT}) \geq C(X)$ at all times, which results to a completed-load competitive ratio $\mathcal{C}(\text{SPT}) \geq 1$. Observe that with the same scenarios, for the pending load it will be the case that $P_t(\text{SPT}) \leq P_t(X)$ which gives a pending-load competitive ratio $\mathcal{P}(\text{SPT}) \leq 1$. ■

Combining Lemmas 13 and 14 we have the following theorem.

Theorem 10. *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 11. *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 $\in$ {LPT, SIS, 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

Please cite this article in press as: A. Fernández Anta, et al., Competitive analysis of fundamental scheduling algorithms on a fault-prone machine and the impact of resource augmentation, Future Generation Computer Systems (2016), http://dx.doi.org/10.1016/j.future.2016.05.042

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, $L_{t'_R}(\text{ALG}) = t'_R$. Then,

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

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

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

The three fractions in the second 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 on 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 12. *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 $\pi_{\min}$-task at the beginning of the execution, followed by a $\pi_{\max}$-task (after an infinitesimally small time ε). After that, it injects only $\pi_{\min}$-tasks, one every $\pi_{\min}$ time. Pattern E sets the first crash/restart instant at $\pi_{\max} + \varepsilon$ time from the beginning and then every $\pi_{\min}$ period of time, creating a *phase* (time period between a restart and the next crash) of length $\pi_{\max}$ followed by infinite phases of length $\pi_{\min}$. These patterns allow an algorithm X to execute successfully the $\pi_{\max}$-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 $\pi_{\min}$-task. Even though it will also be executed, the $\pi_{\max}$-task scheduled next will never be completed in any of the following phases since they are all of size $\pi_{\min}$ 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. ■

Theorem 13. *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 tasks that were pending in the 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 5, 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. A challenging future work, apart from enhancing the analysis of these four popular algorithms, is designing new ones in order to overcome the limitations these present. Some other natural next steps are to extend our investigation to the setting with multiple machines, or to consider preemptive scheduling.

Please cite this article in press as: A. Fernández Anta, et al., Competitive analysis of fundamental scheduling algorithms on a fault-prone machine and the impact of resource augmentation, Future Generation Computer Systems (2016), http://dx.doi.org/10.1016/j.future.2016.05.042

Acknowledgments

This research was partially supported by the grant TEC2014-55713-R from the Spanish Ministry of Economy and Competitiveness (MINECO), the Cloud4BigData grant (S2013/ICE-2894) from Madrid Regional Government (CM), co-financed by Structural Funds of the European Union (FSE & FEDER), and the FPU12/00505 grant from the Spanish Ministry of Education, Culture and Sports (MECD). In addition, the second author was supported by the grant UCY/ED2015 of the University of Cyprus and the third author by the grant 2015/17/B/ST6/01897 of the National Science Centre, Poland.

References

[1] B. Kalyanasundaram, K. Pruhs, Speed is as powerful as clairvoyance [scheduling problems], in: 36th Annual Symposium on Foundations of Computer Science, 1995. Proceedings., Oct 1995, pp. 214–221.
[2] S. Anand, Naveen Garg, Nicole Megow, Meeting deadlines: How much speed suffices? in: Luca Aceto, Monika Henzinger, Ji Sgall (Eds.), Automata, Languages and Programming, in: Lecture Notes in Computer Science, vol. 6755, Springer, Berlin, Heidelberg, 2011, pp. 232–243.
[3] Kirk Pruhs, Jiri Sgall, Eric Torng, Online scheduling, in: Handbook of Scheduling: Algorithms, Models, and Performance Analysis, 2004, pp. 1–15.
[4] Antonio Fernández Anta, Chryssis Georgiou, Dariusz R. Kowalski, Elli Zavou, Online parallel scheduling of non-uniform tasks, Theoret. Comput. Sci. 590 (C) (2015) 129–146.
[5] Antonio Fernández Anta, Chryssis Georgiou, Dariusz R. Kowalski, Joerg Widmer, Elli Zavou, Measuring the impact of adversarial errors on packet scheduling strategies, J. Sched. (2015) 1–18.
[6] K. Schwan, H. Zhou, Dynamic scheduling of hard real-time tasks and real-time threads, IEEE Trans. Softw. Eng. 18 (8) (1992) 736–748.
[7] F. Yao, A. Demers, S. Shenker, A scheduling model for reduced cpu energy, in: 36th Annual Symposium on Foundations of Computer Science, 1995. Proceedings, Oct 1995, pp. 374–382.
[8] Leah Epstein, Asaf Levin, Alberto Marchetti-Spaccamela, Nicole Megow, Julián Mestre, Martin Skutella, Leen Stougie, Universal sequencing on an unreliable machine, SIAM J. Comput. 41 (3) (2012) 565–586.
[9] Daniel D. Sleator, Robert E. Tarjan, Amortized efficiency of list update and paging rules, Commun. ACM 28 (2) (1985) 202–208.
[10] Ronald L. Graham, Bounds on multiprocessing timing anomalies, SIAM J. Appl. Math. 17 (2) (1969) 416–429.
[11] Chung-Yee Lee, Surya Danusaputro Liman, Single machine flow-time scheduling with scheduled maintenance, Acta Inform. 29 (4) (1992) 375–382.
[12] Nikhil Bansal, Algorithms for flow time scheduling (Ph.D. thesis), IBM, 2003.
[13] Igal Adiri, John Bruno, Esther Frostig, A.H.G. Rinnooy Kan, Single machine flow-time scheduling with a single breakdown, Acta Inform. 26 (7) (1989) 679–696.
[14] Dariusz R. Kowalski, Prudence W.H. Wong, Elli Zavou, 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, 2015, pp. 244–246.
[15] Rob van Stee, Online scheduling and bin packing (Ph.D. thesis), 2002.
[16] D. Johnson, A. Demers, J. Ullman, M. Garey, R. Graham, Worst-case performance bounds for simple one-dimensional packing algorithms, SIAM J. Comput. 3 (4) (1974) 299–325.
[17] Leah Epstein, Rob van Stee, Online bin packing with resource augmentation, Discrete Optim. 4 (34) (2007) 322–333.
[18] Joan Boyar, Faith Ellen, Bounds for scheduling jobs on grid processors, in: Andrej Brodnik, Alejandro Lpez-Ortiz, Venkatesh Raman, Alfredo Viola (Eds.), Space-Efficient Data Structures, Streams, and Algorithms, in: Lecture Notes in Computer Science, vol. 8066, Springer, Berlin, Heidelberg, 2013, pp. 12–26.
[19] Matthew Andrews, Lisa Zhang, Scheduling over a time-varying user-dependent channel with applications to high-speed wireless data, J. ACM 52 (5) (2005) 809–834.
[20] Eric Sanlaville, Gnter Schmidt, Machine scheduling with availability constraints, Acta Inform. 35 (9) (1998) 795–811.
[21] Anis Gharbi, Mohamed Haouari, Optimal parallel machines scheduling with availability constraints, Discrete Appl. Math. 148 (1) (2005) 63–87.
[22] Chryssis Georgiou, Dariusz R. Kowalski, On the competitiveness of scheduling dynamically injected tasks on processes prone to crashes and restarts, J. Parallel Distrib. Comput. 84 (2015) 94–107.
[23] Tomasz Jurdzinski, Dariusz R. Kowalski, Krzysztof Lorys, Online packet scheduling under adversarial jamming, in: Approximation and Online Algorithms, Springer, 2014, pp. 193–206.

Antonio Fernández Anta is a Research Professor at IMDEA Networks. Previously he was a Full Professor at the Universidad Rey Juan Carlos and was on the Faculty of the Universidad Politécnica de Madrid, where he received an award for his research productivity. He was a postdoc at MIT from 1995 to 1997. He has more than 20 years of research experience, with a productivity of more than 5 papers per year on average. He is Chair of the Steering Committee of DISC and has served in the TPC of numerous conferences and workshops. He is Senior Member of ACM and IEEE.

Chryssis Georgiou is an Associate Professor in the Department of Computer Science at the University of Cyprus. He holds a Ph.D. (2003) in Computer Science & Engineering from the University of Connecticut, USA. His research interests span the Theory and Practice of Fault-tolerant Distributed and Parallel Computing with a focus on Algorithms and Complexity. He has published more than 70 articles in journals and conference proceedings and co-authored two books on Robust Distributed Cooperative Computing. He served on the Program Committees of many conferences related to Distributed Computing. In 2015 he was the General Chair of the ACM Symposium PODC.

Dariusz R. Kowalski received his Ph.D. degree in Computer Science in 2001 and M.Sc. degree in Mathematics in 1996, both from the University of Warsaw, Poland. He is currently a Professor at the University of Liverpool, UK. He published over 115 peer reviewed research papers, mainly on distributed and parallel computing, network protocols and fault-tolerance.

Elli Zavou received her B.Sc. in Computer Science from the University of Cyprus, in 2011, having a Scholarship from the Cyprus State Scholarship Foundation. She then joined IMDEA Networks Institute in Spain, and received her M.Sc. in Telematics Engineering from the Universidad Carlos III de Madrid, in 2012. She is currently working towards her Ph.D. in Telematics Engineering under the supervision of Prof. Antonio Fernández Anta, and is supported by an FPU Grant from MECD. Her research focuses on energy efficient fault-tolerant dynamic/online scheduling, both of non-uniform tasks in unreliable machine systems and of packets in unreliable communication networks.

Please cite this article in press as: A. Fernández Anta, et al., Competitive analysis of fundamental scheduling algorithms on a fault-prone machine and the impact of resource augmentation, Future Generation Computer Systems (2016), http://dx.doi.org/10.1016/j.future.2016.05.042