

Noname manuscript No.
(will be inserted by the editor)

Fault Tolerant Scheduling of Tasks of Two Sizes under Resource Augmentation

Dariusz R. Kowalski · Prudence W.H. Wong · Elli Zavou

Abstract Guaranteeing the eventual execution of tasks in machines that are prone to unpredictable crashes and restarts may be challenging, but is also of high importance. Things become even more complicated when tasks arrive dynamically and have different computational demands, i.e., processing time (or sizes). In this paper, we focus on the online task scheduling in such systems, considering one machine and at least two different task sizes. More specifically, algorithms are designed for two different task sizes while the complementary bounds hold for any number of task sizes bigger than one. We look at the *latency* and *1-completed load* competitiveness properties of deterministic scheduling algorithms under worst-case scenarios. For this, we assume an *adversary*, that controls the machine crashes and restarts as well as the task arrivals of the system, including their computational demands.

More precisely, we investigate the effect of *resource augmentation* – in the form of processor speedup – in the machine’s performance, by looking at the two efficiency measures for different speedups. We first identify the threshold of the speedup under which competitiveness cannot be achieved by any deterministic algorithm, and above which there exists some deterministic algorithm that is competitive. We then propose an online algorithm, named γ -Burst, that achieves both latency and 1-completed-load competi-

tiveness when the speedup is over the threshold. This also proves that the threshold identified is also sufficient for competitiveness.

Keywords Scheduling · Online Algorithms · Task Sizes · Adversarial Failures · Resource Augmentation · Competitive Analysis

Acknowledgments This research was partially supported by the Cloud4BigData grant (S2013/ICE-2894) from Madrid Regional Government (CM), the HyperAdapt project (TEC2014-55713-R) from the Spanish Ministry of Economy and Competitiveness (MINECO), the NSFC project 61520106005 from the National Science Foundation of China, the FPU12/00505 grant from the Spanish Ministry of Education, Culture and Sports (MECD) and the Polish National Science Centre grant DEC-2012/06/M/ST6/00459. We would like to thank the anonymous reviewers for their constructive comments and suggestions to improve our work.

1 Introduction

Dealing with computationally intensive jobs is becoming a necessity rather than an additional advantage of new computational systems, i.e., cloud computing [7,20]. Some of the multiple challenges that appear with the complexity of such systems, include the dynamicity of job (or task) arrivals, the diversity of their computational demands (e.g. different processing times), the unpredictability of machine failures, as well as the reduction of power consumption. These characteristics are the norm in cloud computing, not the exception. An in-depth study is thus required in order to have full understanding of the potential performance of such systems.

In this work, we apply *speed augmentation* [3,15], increasing the computational power of the system in order to overcome these unpredictabilities, even under worst-case

Dariusz R. Kowalski · Prudence W.H. Wong
University of Liverpool
Ashton Building, Ashton Street
L69 3BX Liverpool
United Kingdom

Elli Zavou
Universidad Carlos III de Madrid and
IMDEA Networks Institute
Avda. del Mar Mediterraneo 22
28918 Leganés
Madrid, Spain
Tel.: +34 608 435 348
E-mail: elli.zavou@imdea.org

scenarios. We focus on the model of a single machine prone to crashes and restarts, and introduce resource augmentation as an alternative to using more processing entities, e.g., multiprocessor systems. More precisely, we assume that the machine crashes and restarts are controlled by an omniscient adversary \mathcal{A} (worst-case scenario). We also consider a scheduler in the system, that decides the order and assigns the injected tasks to be executed by the machine. Tasks arrive dynamically and have different computational demands, i.e. processing time (or size), also controlled by the adversary \mathcal{A} . We assume that a task i has size $p_i \in [p_{min}, p_{max}]$, where p_{min} and p_{max} represent the smallest and largest possible values respectively. Note that p_i becomes known to the system at the moment of i 's arrival.

Due to the fact that scheduling decisions must be made continuously and without knowledge of the future task arrivals or machine crashes and restarts, we look at the problem as an *online scheduling* problem [16] and perform *competitive analysis* [18, 2] for the performance of the algorithms studied. We focus on two efficiency measures: the *completed load*, which is the aggregated size of all tasks that have been executed completely, and *latency*, which is the longest time a task spends in the system. Latency is also referred to as (maximum) *flowtime* in scheduling [8]. In some sense, the former corresponds to the utilization of the machine, while the latter on the fairness of the scheduling algorithm. In short, an algorithm is considered to be α -*latency competitive*, if under any adversarial pattern, for both task arrivals and machine crashes and restarts, its latency is *at most* α times the latency of the offline optimal algorithm OPT, under the same adversarial pattern. Similarly, it is considered to be α -*completed-load competitive*, if under any adversarial pattern, its completed load is *at least* α times the completed load of the offline optimal algorithm OPT, under the same adversarial pattern. On the other hand, an algorithm is *not competitive* with respect to a measure, when it is not competitive for any bounded α .

In a previous work [6], Fernández Anta et al. have studied the problem of scheduling tasks of different computational demands in an online manner, as in this work, but only considered the number of pending tasks and the total pending load as competitive measures. They have shown that no deterministic algorithm for the problem under study is competitive against the best offline solution, but becomes competitive if *resource augmentation* is applied in the form of machine *speedup* above a certain threshold. In particular, they define parameter $s \geq 1$ to represent the machine speedup, under which the processing time of a task i becomes p_i/s . Nonetheless, its use increases the energy consumption of the machine, and thus, by using it as well, we aim to develop competitive algorithms that require the smallest speedup possible. (It is understood that there is nothing to investigate if the offline solution makes use of

resource augmentation as well). Their work has been a motivation to us, leading to the use of resource augmentation as machine speedup s , in order to overcome the non-competitiveness of the two measures we consider in this paper.

Contributions. Let us now briefly describe our contributions on the problem introduced.

Necessary conditions for competitiveness: In Section 3, we show the necessary conditions (in the form of threshold values) on the value of the speedup s in order to achieve both latency and 1-completed-load competitiveness. Influenced by the work of Fernández et al. [6], we use conditions **C1**: $s < \rho$ and **C2**: $s < 1 + \frac{\gamma}{\rho}$, as defined by them, where ρ is the ratio of the largest over the smallest task size and γ a parameter that will be explained clearly later on. We then show that the threshold for the machine speedup is actually the same as for achieving pending load competitiveness. In particular:

When speedup s satisfies both conditions C1 and C2, no deterministic algorithm ALG is latency competitive or c-completed load competitive, for $c > 1 - \frac{\rho}{(1+\rho+\gamma)(\rho+\gamma)}$, even in a system with a single machine and even when considering only two task sizes available. Note that this result also holds for any number of task sizes.

Algorithm γ -Burst: In Section 4, we propose an online algorithm, named γ -Burst, that considers only two task sizes (p_{min} and p_{max}) and show that it achieves both latency and 1-completed-load competitiveness as soon as one of the conditions does not hold.

In short, algorithm γ -Burst separates the arriving tasks into two queues according to their size. Then, at each decision time it checks: (1) if there are no tasks of one of the sizes, it schedules a task from the available ones, (2) else, if there are at least γ tasks of size p_{min} , it schedules γ of them and then schedules one p_{max} -task, (3) otherwise, it schedules tasks of the two queues alternatively.

We show that as soon as condition **C2** does not hold, algorithm γ -Burst becomes latency-competitive and 1-completed-load competitive. In particular:

Algorithm γ -Burst is both 1-latency and 1-completed-load competitive, when run with speedup $s \in \left[1 + \frac{\gamma}{\rho}, \rho\right)$, for any given p_{min} and p_{max} task sizes. For larger speedup values, Fernández Anta et al. [5] have already shown that other algorithms are optimal. Our goal is to close the gap for the speedup in the given range, showing that the two conditions above are not only necessary but also sufficient to achieve optimal competitiveness for both efficiency measures.

Our results show an interesting dichotomy in utilization of resources (in our case, the speedup): for $s < 1 + \frac{\gamma}{\rho}$ and $s < \rho$ no bounded latency competitiveness and no c -completed load competitiveness, for $c > 1 - \frac{\rho}{(1+\rho+\gamma)(\rho+\gamma)}$,

are possible, while for $s \geq 1 + \frac{\gamma}{\rho}$ (even if $s < \rho$) both 1-latency competitiveness and 1-completed load competitiveness are achievable.

Related Work. Georgiou and Kowalski [12] studied a cooperative computing system of n message-passing processes that are prone to crashes and restarts and collaborate in order to complete the dynamically injected tasks. They performed competitive analysis looking at the number of pending tasks but assumed only unit-length tasks. One of their results showed that if tasks are of different lengths, even under slightly restricted adversarial patterns, competitiveness cannot be achieved.

A previous study [6], by Fernández et al., was inspired by this result, and introduced the term of speedup with notation $s \geq 1$, representing the resource augmentation required to surpass the non-competitiveness. More precisely, in [6] the authors looked at a system of multiple machines and at least two different task sizes, i.e., $p \in [p_{min}, p_{max}]$, and defined parameter $\rho = \frac{p_{max}}{p_{min}}$. They applied distributed scheduling and performed worst-case analysis considering number of pending tasks and pending load competitiveness as their efficiency measure.

They also defined parameter γ representing the minimum number of p_{min} -tasks that an online algorithm running with speedup s can complete in addition to a p_{max} , in an interval of length $(\gamma + 1)p_{min}$. Then, they introduced conditions **C1**: $s < \rho$ and **C2**: $s < 1 + \frac{\gamma}{\rho}$, under which they proved that:

No deterministic algorithm ALG is pending load competitive, when run with speedup s that satisfies both conditions C1 and C2, even in a system with a single machine.

In a recent work of Fernandez et al. [5], the authors study the fault tolerant properties of four fundamental scheduling algorithms - Longest In System (LIS), Shortest In System (SIS), Largest Processing Time (LPT) and Shortest Processing Time (SPT) – in a system with one machine. They focused on three efficiency measures: completed load, pending load, and latency, in order to compare the performance of the four algorithms. However, they did not address the general question of finding optimal solutions for each of the considered measures, which we aim to cover in this work with respect to latency and completed load competitiveness. In fact, the performance of the four algorithms occurred not to be optimal for most of the ranges of speedup, but the results are still non-obvious and interesting due to simplicity and popularity of these algorithms.

In [11], Fernández et al. studied a different setting, considering an unreliable communication link between two nodes, proposing the *completed load* (which was called *asymptotic throughput*) for the efficiency measure of packet scheduling algorithms. They considered only two different packet sizes, p_{min} and p_{max} , and measured the impact of

feedback mechanisms, deferred or immediate, on the competitiveness of scheduling policies under adversarial link jams. They showed that the most intuitive greedy algorithm of scheduling the shortest packet length, is not optimal under adversarial errors, and then proposed an online scheduling policy that matches the upper bound of completed load. However, they did not consider any resource augmentation, which as we will show in this paper improves the completed-load performance of scheduling algorithms. In [14], Jurdzinski et al. extended the work done in [11], designing an algorithm that makes use of additional resources by increasing the transmission rate in the link, and achieves completed load 1. In particular, it achieves at least as high throughput as the best schedule, running however with double speedup, i.e., $s = 2$. In another packet scheduling work, Andrews et al. [4] consider an online packet scheduling problem with a wireless channel whose conditions as well as the packet arrivals are controlled by an adversary. They design scheduling algorithms for the base-station in order to achieve stability in terms of the queue sizes of each user. Our work on the other hand, does not focus on stability; we look at the latency and completed-load instead of the pending-load.

Furthermore, our work is directly related to research done on machine scheduling with availability constraints, e.g., [13,17]. One of the results in the area shows the necessity of online algorithms in case of unexpected machine breakdowns. However, most works allow preemption and prove optimality for nearly online algorithms; ones that need to know the time of the next task arrival or machine availability.

Last but not least, we can relate the problem of our work with the online version of the *bin packing* problem [19], where tasks are the objects to be packed and time periods between two consecutive failures of the machine are the bins. On this problem, thorough research has been done, some of which we consider more related to ours. Epstein et al. [10] is an example of studying online bin packing with resource augmentation in the size of bins (corresponds to the length of alive intervals in our work). The main difference with our work is the fact that we do not know a priori the bins and their sizes (duration of time intervals when the machine is alive and active). Boyar and Ellen [9] have looked into a similar problem, considering job scheduling in the grid. The difference with our setting is that they consider many machines, but also the fact that the arriving items in their model are processors with bounded memory capacities, which must be used to complete a fixed number of jobs. Then using two fixed job sizes, small and large, they show lower and upper bounds that depend only on the fraction of small jobs in the system.

2 Model

Network Setting. We consider a system with one machine prone to unpredictable crashes and restarts, with a *scheduler* that assigns the tasks to the machine following some online algorithm (scheduling policy). The clients of the system submit tasks of different sizes (processing times) to the scheduler, which then decides the order of their execution by the machine.

Tasks. Tasks are injected continuously and dynamically to the scheduler, an operation which we assume to be controlled by an adversary \mathcal{A} . Each task i has a unique *identifier*, an *arrival time* r_i (simultaneous arrivals are totally ordered) and a *size* $p_i \in [p_{min}, p_{max}]$, corresponding to the processing time it requires to be completed by a machine running without additional resource augmentation, i.e., $s = 1$. A task's size becomes known at arrival, and values p_{min} and p_{max} represent the smallest and largest sizes respectively. Throughout the paper we use the term p -task to refer to a task of size p , and use ρ for the ratio of the maximum over the minimum task size, i.e., $\rho = \frac{p_{max}}{p_{min}}$. Whenever we consider only two task sizes, we will refer to them simply as the *small* and *large* tasks. We assume that tasks are *atomic* with respect to their completion: if a machine stops executing a task before it is completed (intentionally or due to a crash), then the machine cannot resume the execution of the task from the point it stopped, i.e., preemption is not allowed. Finally, we assume that tasks are *independent* and *idempotent*, meaning that any execution of the same task produces the same result. The adversary defines the arrival pattern A , which includes triples of the identifiers, the arrival times and the sizes of the tasks injected.

Machine crashes and restarts. Due to the unpredictable nature of the machine, we consider crashes and restarts being defined by error pattern E . We assume pattern E to be coordinated with the arrival pattern A by the adversary \mathcal{A} , in order to create worst-case scenarios for the online algorithms. Since preemption is not allowed, the task being executed at the time of a crash is not completed, and therefore is still considered pending in the scheduler's queue.

Resource Augmentation. As mentioned earlier, we consider a form of resource augmentation by speeding up the machine, but our goal is to keep this speedup $s \geq 1$ as low as possible. For a task i of size p_i and machine with speedup s , its processing time becomes p_i/s .

Notations. Let us now define some more notations that we will extensively use in the rest of the paper. Throughout an execution, it is necessary to keep track of the injected, completed and pending tasks. We therefore introduce corresponding sets $I_t(A)$, $N_t^s(X, A, E)$ and $Q_t^s(X, A, E)$, where

X is a scheduling algorithm, A and E the arrival and error patterns respectively, t the time instant considered and s the speedup of the machine. Let also $T = [0, t]$ be the interval from the beginning of the execution to the current time t . Then, $I_t(A)$ represents the set of injected tasks within time interval T , $N_t^s(X, A, E)$ the set of completed tasks within T and $Q_t^s(X, A, E)$ the set of pending tasks at time instant t . Note that $Q_t^s(X, A, E)$ contains the tasks that were injected by time t inclusively, but not the ones completed before and up to time t . Note also, that $I_t(A) = N_t^s(X, A, E) \cup Q_t^s(X, A, E)$. In further sections of the paper we omit the superscript s , and/or the subscript t , for simplicity. However, the appropriate speedup and/or time instant in each case is clearly stated.

Let us also clarify parameter γ , which was inspired by [6] and is used throughout our work. It is defined as the smallest integer such that an algorithm running with speedup s can complete γ small tasks and a large task in an interval of length $(\gamma + 1)p_{min}$. It hence satisfies these two properties, which will be used later in our analysis:

Property (1) $\frac{\gamma p_{min} + p_{max}}{s} \leq (\gamma + 1)p_{min}$.

Property (2) $\frac{\kappa p_{min} + p_{max}}{s} > (\kappa + 1)p_{min}$, for every non-negative integer $\kappa < \gamma$.

From these properties it is derived that

$$\gamma = \max\left\{\left\lceil \frac{p_{max} - s p_{min}}{(s-1)p_{min}} \right\rceil, 0\right\} = \max\left\{\left\lceil \frac{\rho - s}{s-1} \right\rceil, 0\right\}.$$

Finally, the two conditions for the speedup threshold are **C1:** $s < \rho$ and **C2:** $s < 1 + \frac{\gamma}{\rho}$, under which we base our work.

Efficiency Measures. We focus on two efficiency measures in our work: the *completed load*, which is the aggregate size of all tasks that have been completed successfully, and the *latency*, which is the *longest duration* a task spends in the system. More precisely, when considering an algorithm ALG running with speedup s , under arrival and error patterns A and E respectively, we look at the current time t and calculate the following:

Completed Load:

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

Latency:

$$L_t^s(\text{ALG}, A, E) = \max \left\{ \begin{array}{l} f_i - r_i, \forall i \in N_t^s(\text{ALG}, A, E) \\ t - r_i, \forall i \in Q_t^s(\text{ALG}, A, E) \end{array} \right\},$$

where f_i is the time of completion of task i . Finding the scheduling algorithm that maximizes or minimizes correspondingly the above measures offline (knowing patterns A and E a priori) is an NP-hard problem [6].

As already mentioned, due to the dynamicity of the system, we view this problem as an online scheduling problem. Hence, we pursue *competitive analysis* using these metrics as follows:

Consider any time t in an execution, combinations of arrival and error patterns, A and E respectively, and any algorithm X designed to solve the scheduling problem when run without speedup, i.e., $s = 1$. An online algorithm ALG running with speedup $s \geq 1$, is considered α -completed-load-competitive if $\forall t, X, A, E$, it is true that $C_t^s(\text{ALG}, A, E) \geq \alpha \cdot C_t^1(X, A, E) + \Delta_C$ for some parameter Δ_C that does not depend on t, X, A or E ; α is the completed-load competitive ratio of ALG, which we denote by $\mathcal{C}(\text{ALG})$. Similarly, it is considered α -latency-competitive if $\forall t, X, A, E$, it is true that $L_t^s(\text{ALG}, A, E) \leq \alpha \cdot L_t^1(X, A, E) + \Delta_L$, 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 both metrics. However, along with Δ_C and Δ_L , they may depend on system parameters p_{min}, p_{max} or s , which are not considered inputs of the problem.

3 Non-Competitiveness

In this section, we show that under some threshold on the value of machine speedup, competitiveness cannot be achieved by any deterministic algorithm, neither for 1-completed-load, nor for latency. This, implies the necessary conditions in order to achieve competitiveness. To be exact, we show that this threshold is the same as the one used in [6] for the pending load competitiveness.

3.1 Completed Load.

Let us start by analyzing the completed load measure and the conditions under which 1-completed-load-competitiveness cannot be achieved by any deterministic scheduling algorithm. We will show that it takes place if conditions **C1** and **C2** on speedup s hold. Even more, under these conditions no deterministic algorithm is c -completed load competitive, for $c > 1 - \frac{\rho}{(1+\rho+\gamma)(\rho+\gamma)}$. This result, together with Section 4.1 presenting 1-completed load competitive algorithm in the complementary range of speedup s , shows a dichotomy in utilization of resources (speedup) when optimizing completed load measure.

To prove the negative result, we use the same adversarial strategy used for the pending task analysis done by Fernández et al. in [6]. That is, we consider any deterministic algorithm ALG running with speedup $s \geq 1$ such that conditions **C1** and **C2** hold, and define a universal offline algorithm OFF running with no speedup (i.e., $s = 1$) that

is associated with specific adversarial arrival and error patterns, A and E respectively. The offline algorithm and the adversarial patterns are defined in such a way that its total amount of completed load is always the same as the total size of the injected tasks minus at most $\gamma p_{min} + p_{max}$, while the completed load of ALG can only be a fraction of the total size of the injected tasks. The general idea is to prevent ALG from completing a large task, and to do so we get two types of phases: *short phases*, in which both ALG and OFF perform only small tasks (OFF one more than ALG), and *long phases*, in which ALG performs only small tasks while OFF completes a large one. The details are as follows.

Description of adversarial strategy. At the beginning of the first phase there are γ small and one large tasks injected and the machine is activated. Let us now assume, inductively on the number of phases, that the adversarial arrival and crash patterns are already defined and at the beginning of phase $i \geq 1$ of algorithm ALG there are x small tasks and y large tasks pending. We assume that the adversary does not inject any tasks until the very end of each phase, and thus it can simulate the scheduling choices of ALG during phase i (the algorithm is deterministic and we assumed that the adversary does not inject anything nor cause crashes before the end of the phase). We first define parameter Δ to be the time elapsed from the beginning of the phase until the time at which ALG starts executing a large task (assuming the phase is long enough). Note that, again, since ALG is deterministic, the adversary knows the times at which ALG stops the execution of a task to schedule another (if this is the case), it can therefore adjust the crashes at the time instants it sees fit. There are two scenarios for the phases that may occur:

Short phase. When $\Delta < \frac{\gamma p_{min}}{s}$, ALG schedules a large task sooner than $\gamma p_{min}/s$ time after the beginning of the phase. Let $\kappa = \lfloor \Delta / (p_{min}/s) \rfloor < \gamma$ be the number of small tasks scheduled before the large one in the phase. Then the adversary ends the phase by crashing the machine after $(\kappa + 1)p_{min}$ time from the beginning of the phase. Recall that Property (2) of γ states that $\frac{\kappa p_{min} + p_{max}}{s} > (\kappa + 1)p_{min}$. This means that by the end of the phase, OFF will have completed $\kappa + 1$ small tasks, while ALG will only be able to complete κ of them and not the large task that was scheduled. Also, at the end of the phase, $\kappa + 1$ small tasks are injected, to replace the ones completed by OFF.

Long phase. When $\Delta \geq \frac{\gamma p_{min}}{s}$, ALG schedules a large task no sooner than $\gamma p_{min}/s$ time after the beginning of the phase. In this case, the adversary ends the phase with a crash after p_{max} time, so that OFF is able to complete the large task that is pending. The machine is then crashed, causing ALG to complete at most γ small tasks but no large task. This is because of condition **C2**: $s < 1 + \gamma/\rho$. Also,

at the end of the phase the adversary injects a large task, “replacing” the one that was completed by OFF.

Analyzing now the adversarial behavior described above, we prove the following theorem.

Theorem 1 *For any given p_{min}, p_{max} and s , if both conditions C1 and C2 are satisfied, no deterministic algorithm ALG is c -completed load competitive, for $c > 1 - \frac{\rho}{(1+\rho+\gamma)(\rho+\gamma)}$, when run with speedup s against an adversary that injects tasks of sizes $c \in [p_{min}, p_{max}]$, even in a system with a single machine.*

Proof Assume that both conditions C1 and C2 are satisfied. Suppose, to the contrary, that there is a deterministic algorithm ALG which is $\left(1 - \frac{\rho}{(1+\rho+\gamma)(\rho+\gamma)} + \epsilon\right)$ -completed load competitive, for some $\epsilon > 0$, when run with on a single machine with speedup s against an adversary that injects tasks of sizes $c \in [p_{min}, p_{max}]$. W.l.o.g. we may consider $0 < \epsilon < \frac{\rho}{(1+\rho+\gamma)(\rho+\gamma)}$, as if the algorithm was competitive for a larger value of ϵ it would still be competitive for $0 < \epsilon < \frac{\rho}{(1+\rho+\gamma)(\rho+\gamma)}$, by definition of competitiveness.

Let us execute algorithm ALG against the adversary defined in the beginning of this section. Let $T = T_1 + T_2$ be the number of phases completed so far in the execution, where T_1 and T_2 are the numbers of short and long phases respectively. Let $\beta = \beta(T) = T_2/T$. Let us also denote the total injected load by $W = W(T)$ (the sum of sizes of all the tasks injected). Note that $W(T)$ grows to infinity with T , by the definition of the adversary and phases.

Observe that

$$T_2 p_{max} \leq W - (\gamma p_{min} + p_{max}) \leq T(\gamma p_{min} + p_{max}).$$

Indeed, the first inequality comes from the properties of injecting new tasks in such a way that the OFF has γ small tasks and one large task after T_2 long phases in which it completed one large task (note that there could be more tasks in the workload W completed by OFF in short phases). The second inequality follows from the fact that at most γ small tasks and one large task are injected per each phase, plus one such combination of tasks in the very beginning of the execution. Consequently,

$$T \geq \frac{W}{\gamma p_{min} + p_{max}} - 1. \quad (1)$$

Consider sufficiently large T such that W is bigger than $2(\gamma p_{min} + p_{max})/\epsilon$; it exists since $W = W(T)$ grows to infinity. We compute upper bounds on the competitive ratio in two ways as follows.

1) The completed load of ALG is $W - T_2 p_{max} = W - \beta T p_{max}$; that is, it is equal to the total processing time of the tasks that have arrived, W , minus the amount of work that OFF was able to complete in the long phases, $T_2 p_{max}$.

We know that ALG is not able to complete any large task, but since there were T_2 long phases, we know that at least that many large tasks were injected in the system and OFF was able to complete them. This is at most

$$W - \beta T p_{max} \leq W - \frac{\beta W p_{max}}{\gamma p_{min} + p_{max}} + \beta p_{max} = W \cdot \left(1 - \frac{\beta p_{max}}{\gamma p_{min} + p_{max}} + \frac{\beta p_{max}}{W}\right),$$

by applying the lower bound on T , c.f., Equation (1). Since we assumed $W > 2(\gamma p_{min} + p_{max})/\epsilon$, and since $\beta \leq 1$, the completed work of ALG is smaller than

$$W \cdot \left(1 - \frac{\beta p_{max}}{\gamma p_{min} + p_{max}} + \epsilon/2\right). \quad (2)$$

2) On the other hand, the completed load of ALG is $W - T_1 p_{min} + T_2 \gamma p_{min} = W - (1 - \beta)T p_{min} + \beta T \gamma p_{min}$. This follows directly from the adversarial strategy and definition of OFF: ALG is not able to complete as many small tasks as OFF in the short phases (it completes one less per phase) but is able to complete γp_{min} in each long phase. Thus, the completed load of ALG is at most

$$W \cdot \left(1 - \frac{p_{min}(1 - \beta - \beta\gamma)}{\gamma p_{min} + p_{max}} + \frac{p_{min}(1 - \beta - \beta\gamma)}{W}\right),$$

by applying the lower bound on T , c.f., Equation (1). This is smaller than

$$W \cdot \left(1 - \frac{p_{min}(1 - \beta - \beta\gamma)}{\gamma p_{min} + p_{max}} + \epsilon/2\right), \quad (3)$$

since we assumed $W > 2(\gamma p_{min} + p_{max})/\epsilon$, which in turn is bigger than $2p_{min}(1 - \beta - \beta\gamma)/\epsilon$ for $\beta \in [0, 1]$.

To summarize the above estimations, the completed load of ALG is smaller or equal to the minimum of the two computed upper bounds stated in Equations (2) and (3). Observe that the first upper bound, $W \cdot \left(1 - \frac{\beta p_{max}}{\gamma p_{min} + p_{max}} + \epsilon/2\right)$, is decreasing with β growing from 0 to 1, while the second upper bound, $W \cdot \left(1 - \frac{p_{min}(1 - \beta - \beta\gamma)}{\gamma p_{min} + p_{max}} + \epsilon/2\right)$, is increasing. Therefore, the minimum of these two upper bounds is not bigger than the values of these formulas for some β^* for which they are equal; for other values of β , at least one of the formulas (and so their minimum) is smaller than the value for β^* . The value of β^* follows from equating the two formulas, Eq. (2) = Eq. (3), and is equal to $\frac{1}{1+\rho+\gamma}$. Plugging it into Eq. (2)

$$\begin{aligned} & W \cdot \left(1 - \frac{\beta^* p_{max}}{\gamma p_{min} + p_{max}} + \epsilon/2\right) \\ &= W \cdot \left(1 - \frac{\rho}{(1 + \rho + \gamma)(\rho + \gamma)} + \epsilon/2\right). \end{aligned}$$

At the same time, the completed load of OFF is at least

$$W - \gamma p_{min} - p_{max} = W \cdot \left(1 - \frac{\gamma p_{min} + p_{max}}{W}\right),$$

by the definition of the adversarial patterns; OFF completed all injected tasks, except the ones that are pending at the time of observation (of length at most $\gamma p_{min} + p_{max}$). Since $W > 2(\gamma p_{min} + p_{max})/\epsilon$, the completed load of OFF is thus bigger than

$$W \cdot (1 - \epsilon/2).$$

The completed load competitiveness is therefore smaller than

$$\frac{1 - \frac{\rho}{(1+\rho+\gamma)(\rho+\gamma)} + \epsilon/2}{1 - \epsilon/2} = 1 - \frac{\rho}{(1+\rho+\gamma)(\rho+\gamma)} + \epsilon'$$

$$< 1 - \frac{\rho}{(1+\rho+\gamma)(\rho+\gamma)} + \epsilon,$$

for some constant $\epsilon' > 0$ that depends only on ϵ, ρ, γ , since $\epsilon < \frac{\rho}{(1+\rho+\gamma)(\rho+\gamma)}$. Recall that this holds for any sufficiently large T , and note that the difference $\epsilon - \epsilon'$ is a positive constant that does not depend on T ; this implies that the load completed by ALG is smaller than the $1 - \frac{\rho}{(1+\rho+\gamma)(\rho+\gamma)} + \epsilon$ fraction of the load completed by OFF by an unbounded value, and thus yields contradiction with the assumed $\left(1 - \frac{\rho}{(1+\rho+\gamma)(\rho+\gamma)} + \epsilon\right)$ -completed load competitiveness (even for the sense of asymptotic competitiveness that allows an additive constant). This completes the proof. \square

3.2 Latency.

One may try to use the same adversarial strategy as in the previous construction for the completed load in order to prove that no deterministic algorithm ALG can reach arbitrarily larger latency than OFF, provided that the speedup s satisfies both conditions C1 and C2. However, that cannot be achieved due to the following algorithm ALG, which guarantees optimal latency competitiveness under the adversary and thus serves as a counter-example: after a crash/restart event schedule a large task. According to the adversarial strategy proposed in Section 3.1, only short phases occur, whereas an offline algorithm OFF schedules exactly one small task and then the machine crashes followed by an injection of a new small task. It is easy to observe, that in such a case both ALG and OFF will freeze the large task in the system, hence leading to infinite latency with time going to infinity. More specifically, the maximum latency of both algorithms will be t at each time t , which means an optimal latency competitiveness.

Therefore, we define the following slightly modified adversarial arrival and error patterns A and E , accompanied

by the adjusted OFF, which, as we show, prohibit any deterministic algorithm of achieving (bounded) latency competitiveness. The idea is to allow ALG to perform a large task from time to time, but with (exponentially) increasing time between two such consecutive performances.

Let us consider any deterministic algorithm ALG running under speedup $s \geq 1$ and define a universal offline algorithm OFF running with no speedup ($s = 1$), associated with the adversarial arrival and error patterns A and E . The offline algorithm OFF will behave in terms of *phases* and *stages* as follows.

Description of adversarial strategy. A *phase* is a closed time interval between a restart (beginning) and a crash (end) point of the system's machine, while it remains continuously alive during the phase. A *stage* consists of consecutive phases during which the adversary allows ALG to complete at most one large task. We number these phases by parameter j ; more specifically, the j -th phase of stage i will be denoted by j_i . At the beginning of the first phase there are γ small and one large tasks injected and the machine is activated.

Let us now assume that the adversarial arrival and error patterns are already defined and at the beginning of stage $i \geq 1$ of algorithm ALG there are x small tasks and y large tasks pending. Let us also assume that the adversary does not inject any tasks until the end of each phase in the stage and simulate the scheduling choices of ALG during stage i . We first define parameter $\Delta(j_i)$ to be the time elapsed from the beginning of phase j in stage i until the time at which ALG starts executing a large task (assuming the phase is long enough). Note that since ALG is deterministic, the adversary knows the times at which ALG stops the execution of a task to schedule another (if such a case), it can therefore adjust the crashes at the time instants it sees fit. First, let $\kappa_{j_i} = \lfloor \Delta(j_i)/(p_{min}/s) \rfloor < \gamma$ be the number of small tasks executed before a large task is scheduled in phase j_i . There are only two types of stages that may occur:

Stage Type 1. When for all phases in stage i , i.e., $\forall j_i$, ALG schedules a large task sooner than $\gamma p_{min}/s$ time after the beginning of the phase, i.e., $\Delta(j_i) < \frac{\gamma p_{min}}{s}$, then the following may occur:

(a) If $p_{max} \leq \frac{\kappa_{j_i} p_{min} + p_{max}}{s}$, then the adversary crashes the machine after p_{max} time and ends phase j_i without injecting any additional task. Then, exactly γ phases of length p_{min} follow, after which the adversary injects a small task. Finally, the stage continues to infinity with phases of length p_{min} where one small task is injected at the end of each phase.

(b) If $p_{max} > \frac{\kappa_{j_i} p_{min} + p_{max}}{s}$ (note that we then have $p_{max} > (\kappa_{j_i} + 1)p_{min}$, by Property (2) of γ), then for phase j_i of the stage i :

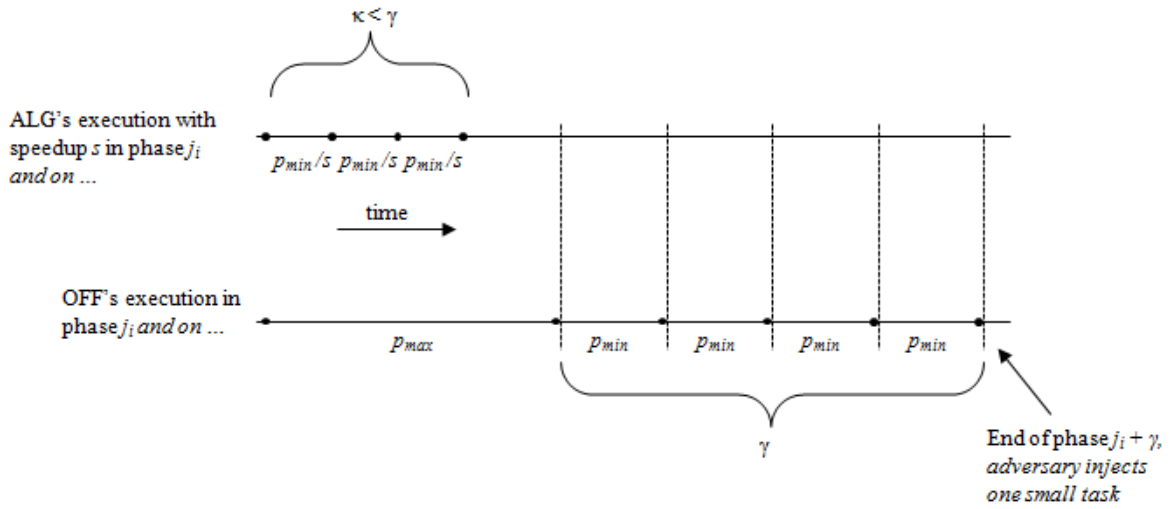


Fig. 1 Illustration of stage of type 1(a).

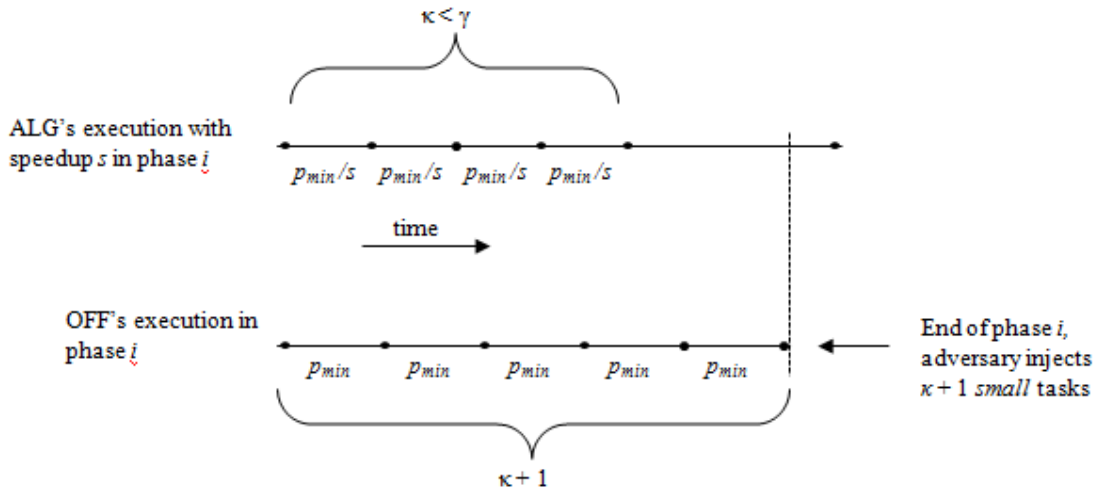


Fig. 2 Illustration of phase j_i for case (i) in stage of type 1(b).

(i) if $(j_i - 1)p_{min} < 2^i p_{max} + \gamma p_{min}$, the adversary crashes the machine after $(\kappa_{j_i} + 1)p_{min}$ time and ends the phase injecting $\kappa_{j_i} + 1$ small tasks (as many as OFF has managed to complete).

(ii) if $(j_i - 1)p_{min} \geq 2^i p_{max} + \gamma p_{min}$, then the adversary crashes the machine after p_{max} time and ends the phase by injecting a large task.

Stage Type 2. When there is a phase j_i in stage i such that, $\Delta(j_i) \geq \frac{\gamma p_{min}}{s}$, it means that ALG schedules a large task no sooner than $\gamma p_{min}/s$ time after the beginning of phase j_i . In such a case, the adversary crashes the machine after p_{max} time, ending phase j_i without injecting any more tasks. The following phases are of length p_{min} . At the end of the γ^{th}

phase and the phases after that, there is exactly one small task injected.

Properties of the Adversarial Strategy. Before continuing with the analysis, let us explain some important properties of the Stage Types described above, which are essential for the proof that follows.

If a stage i is of type 1(a), as also seen in Fig. 1, OFF has time to complete a large task in phase j_i , while ALG is able to complete only up to κ_{j_i} small tasks. With the following phases of length p_{min} , the adversary guarantees an infinite execution of latency 0 for OFF (after phase j_i) while the latency of ALG grows to infinity as it is never able to complete the large tasks pending at the beginning of phase j_i .

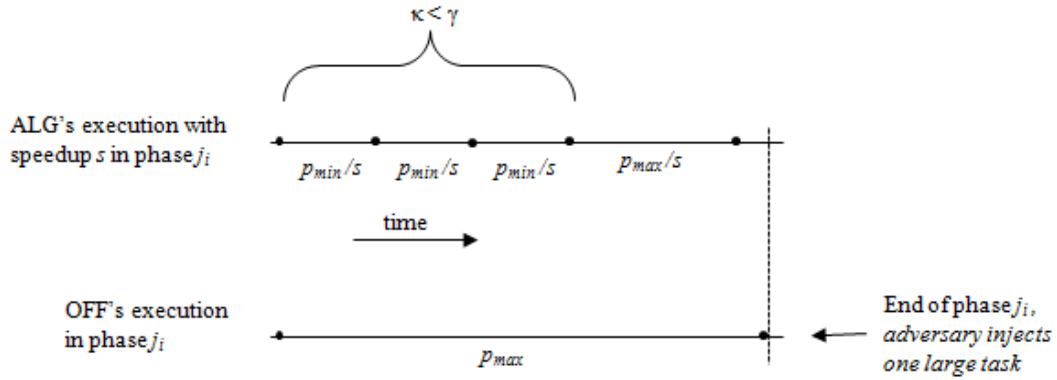


Fig. 3 Illustration of phase j_i for case (ii) in stage of type 1(b).

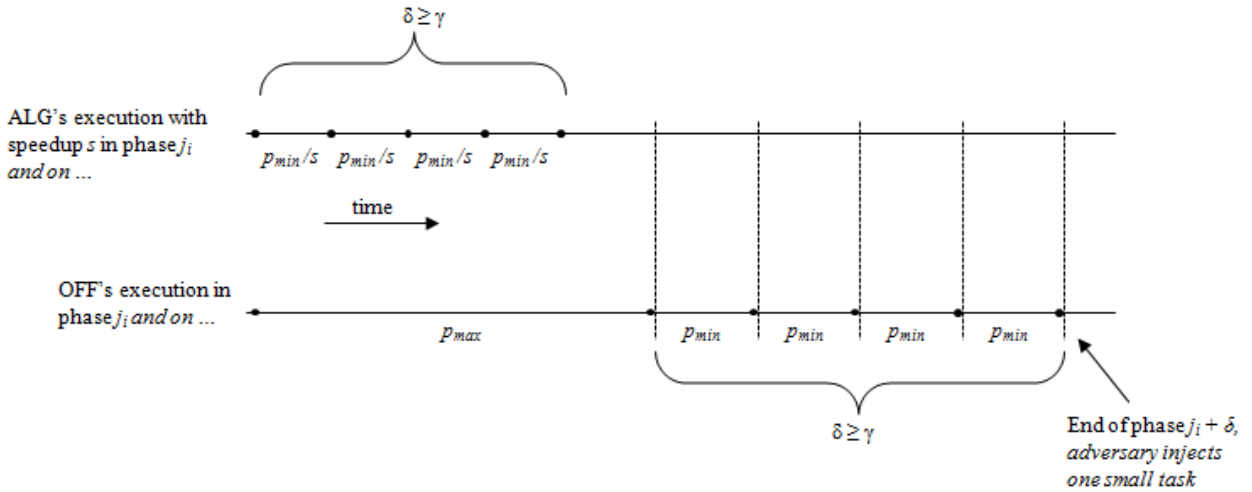


Fig. 4 Illustration of stage of type 2. Note that δ is the integer corresponding to the number of small tasks completed by ALG during $\Delta(j_i)$ and it is at least equal to γ .

If a stage i is of type 1(b) and phase j_i lies in case (i), as also seen in Fig. 2, OFF has the time to complete $\kappa_{j_i} + 1$ small tasks, one small task more than the ones completed by ALG before the machine is crashed. What is more, ALG cannot complete the large task scheduled after the κ_{j_i} small ones, resulting with one small task more than the ones it had at the beginning of the phase. At the end of the phase, the latencies of ALG and OFF increase by $(\kappa_{j_i} + 1)p_{min}$. If a stage i is of type 1(b) and phase j_i lies in case (ii), as also seen in Fig. 3, OFF completes the large pending task, while ALG is able to complete at least κ_{j_i} small tasks in addition to the large one. However, at the end of such a phase the latency of OFF equals the time that the γ pending small tasks have been in the system, while

the latency of ALG is at least equal to $2^i p_{max}$. This, is because the small tasks that are pending in the execution of OFF at the beginning of the phase are of total size at least $2^i p_{max} + \gamma p_{min}$, and each small task is accumulated in intuitively $\approx \gamma p_{min}$ time (equal to the time a phase (b)(i) lasts). Note that, such phases may occur only after intervals of increasing length, more precisely the length of the intervals increases by a factor of 2^i in each stage i . If a stage i is of type 2, as also seen in Fig. 4, OFF is able to complete the large task during phase j_i and then complete the remaining small tasks (γ plus all the injected ones) one at a time, while ALG may also complete some small tasks, but not the large one that is pending. For the previous $(j - 1)$ phases the adversarial behavior followed is the

one described in 1(b)(i); the other kinds of phases assume completing a large task, which would make stage i of type 1 directly.

Analyzing now the adversarial behavior described above, we prove the following lemmas, which lead to the final theorem of the section.

Lemma 1 *The phases, the adversarial pattern and algorithm OFF are well-defined. In particular, at the beginning of each phase in stages of type 1(b) and the first j_i phases in stages of type 1(a) and 2, there are exactly γ small and one large tasks pending in the execution of OFF. For the rest of the phases in stages of type 1(a) and 2, there are less than γ small tasks pending in the execution of OFF.*

Proof We use induction on the number of phases to show that at the beginning of phase k in a stage of type 1(b) and the first j_i phases in a stage of type 1(a) or 2, there are exactly γ small tasks and a large one pending in the execution of OFF; therefore phase k is well defined. The specification of the phase depends on the relation between p_{max} and the number of small tasks scheduled in that phase, as well as the sequence number of the current phase within the stage it is.

The invariant of γ small and one large tasks holds for the first phase of the execution by definition (initial injection). Looking at the definition of the phases in the two types of stages mentioned, we have the following cases:

- (1) Phases of case 1(b)(i) end after OFF completes $(\kappa + 1)$ small tasks, where there are exactly as many injected.
- (2) Phases of case 1(b)(ii) end after OFF completes the large task pending at the beginning, and at the end there is exactly one large task injected.
- (3) The first $j - 1$ phases of stage types 1(a) and 2, satisfy the $\Delta(k) < \frac{\gamma p_{min}}{s}$ and for those ones, subcase 1(b)(i) will be followed (case 1 above), which guarantees that the invariant holds.

For the rest of the phases in stages of types 1(a) and 2, there are only some small tasks pending in the execution of OFF. Looking first at stage type 1(a), phase j_i will end in p_{max} time, during which OFF will complete the large task pending and after which no task will be injected for the next γ phases of length p_{min} . At the beginning of those phases, there will be exactly $\gamma - i < \gamma$ small tasks pending in the execution of OFF, where $i = 1, 2, \dots, \gamma$. Then, at the end of the last phase a new small task will be injected and every p_{min} time there will be a crash and an injection of a small task, causing the rest of the execution to have phases starting with one small task pending for OFF. Then, in a stage of type 2, after the first j_i phases, the idea that follows is the same as in stage of type 1(a), with the difference that ALG here is able to complete exactly γ small tasks in the j_i^{th} phase, while in scenario 1(a) only κ_{j_i} . The phases that follow will be the same as the ones described above. \square

Lemma 2 *The number of phases is infinite.*

Proof First, by Lemma 1, consecutive phases are well defined. Moreover, they are of finite length, regardless of the stage and scenario type they are in; the alive intervals are always defined by the tasks completed by OFF in each phase, either a large task (in stages of type 1(a), 2 or phases of case 1(b)(ii)), or some small tasks (in stages of type 1(a), 2 or phases of case 1(b)(i)). Therefore in an infinite execution there is an infinite amount of phases. \square

Lemma 3 *Stages of type 1(a) and 2 are terminal (i.e., if such a stage appears, it never stops), have infinite length and cause infinite latency competitiveness for ALG.*

Proof First, note that by the definition of stage types 1(a) and 2, after the completion of a large pending task in phase j_i by OFF, there is no other large task injected during the execution and ALG has not been able to complete it by that time. Then, γ phases of length p_{min} follow, with no task injections except the last one during which a small task arrives. This means that the γ small tasks pending at the beginning of phase j_i have now been completed by OFF and its latency becomes zero. On the other hand, even if ALG has completed some small tasks, it will not be able to complete the large pending task, which will increase its latency. After the γ^{th} phase of length p_{min} , there are infinite phases of length p_{min} that end with a crash and a single p_{min} -task injection. This results in an infinite latency competitiveness for ALG, due to the pending large task in the queue of ALG and the fact that OFF starts performing the arriving load immediately and thus keeping its queue empty. \square

Lemma 4 *A stage i of type 1(b) consists of various phases of case 1(b)(i) and a final phase of case 1(b)(ii). At the end of the phases of case 1(b)(i), the latency of both ALG and OFF is increased by the same value. However, at the end of the last phase of the stage, that is of case 1(b)(ii), the latency of ALG is increased by $2^i p_{max}$, while the latency of OFF is bounded by the time the last injected γ small tasks were waiting in the system.*

Proof When $p_{max} > \frac{\kappa_{j_i} p_{min} + p_{max}}{s}$ (and consequently $p_{max} > (\kappa_{j_i} + 1)p_{min}$) a stage of type 1(b) takes place. While $(j_i - 1)p_{min} < 2^i p_{max} + \gamma p_{min}$, i.e., case 1(b)(i), the adversary ends the phases after $(\kappa_{j_i} + 1)p_{min}$ time. During such phases, OFF completes $(\kappa_{j_i} + 1)$ small tasks, while ALG only κ_{j_i} of them. In both executions there is a large task pending, of which the latency is increased equally.

Then, as soon as $(j_i - 1)p_{min} \geq 2^i p_{max} + \gamma p_{min}$, i.e., case 1(b)(ii), the corresponding phase has a duration of p_{max} time. This allows for both ALG and OFF to complete their large pending task, causing their latency to depend on their small pending tasks. Regarding OFF, we know from Lemma 1 that it has only γ of them, and they cannot have

been injected more than γ phases ago; hence the maximum latency will be γp_{min} . On the other hand, we know that the current phase was preceded by $j_i - 1$ phases of case 1(b)(i), and after each of them ALG has been accumulating one more small task. Therefore, at the beginning of phase j_i it has $j_i - 1$ small tasks more than OFF. Again, from Lemma 1 we know that each phase of case 1(b)(i) that preceded had a duration of maximum γp_{min} time. That, together with the fact that $(j_i - 1)p_{min} \geq 2^i p_{max} + \gamma p_{min}$, and the fact that within the current phase ALG may complete up to γ small tasks, means that its latency will be increased to at least $2^i p_{max}$ (here we use a recursive assumption that at the end of the previous phase of case 1(b)(ii), i.e., at the end of the last stage of type 1(b), it was $2^{i-1} p_{max}$). \square

Combining Lemmas 1 to 4, and the definition of the adversarial arrival and error patterns above, A and E , the following theorem follows.

Theorem 2 *For any given p_{min} , p_{max} and s , if both conditions C1 and C2 are satisfied, no deterministic algorithm ALG is latency competitive when run with speedup s against an adversary that injects tasks of sizes $c \in [p_{min}, p_{max}]$, even in a system with one single machine.*

Proof Lemma 1 states that the defined adversarial strategy and the accompanied OFF result in a valid execution of ALG and OFF. It also upper bounds the amount of pending work for OFF at the beginning of each phase of the execution. Lemma 2 shows that the number of phases is actually infinite, and thus infinite executions are guaranteed.

If an execution enters a stage of type 1(a) or 2, from Lemma 3 we know that in both cases the stage will have infinite length and the latency of ALG will grow to infinity. If, however, an execution stays in stages of type 1(b), from Lemma 4 we know that the latency of ALG will continue to grow exponentially. Hence, regardless of whether an execution eventually enters a stage of type 1(a) or (2), or remains in an infinite execution of continuous stages of type 1(b), the latency of ALG will grow to infinity while the latency of OFF stays bounded, which completes the proof. \square

4 Algorithm γ -Burst

In this section, we propose an online scheduling algorithm, named γ -Burst, which achieves both 1-completed-load and latency competitiveness, as soon as condition C2 does not hold. More precisely, we show that by considering only two task sizes, p_{min} and p_{max} , and running with speedup $s \in [1 + \gamma/\rho, \rho)$, the algorithm achieves optimal competitiveness for both measures. In fact, Fernández Anta et al. [5] have shown that other algorithms are optimal for larger speedup, so we aim to close the gap for the speedup in the range given.

Algorithm Description. Algorithm γ -Burst considers only two task sizes (p_{min} and p_{max}). It separates the pending tasks in two lists (or queues) according to their size, say Q_{max} and Q_{min} , and sorts each of them in ascending order according to their arrival time. This way, the first task in each list is the one to be scheduled next. Algorithm γ -Burst takes its scheduling decisions at the end of each stage, which indicates also the beginning of a new one. A *stage* ends either by being interrupted by a machine crash, or by the completion of the tasks that have been scheduled at the beginning of the stage. The scheduling decisions are then taken (at the beginning of each stage) as follows:

1. If there are no tasks of one of the sizes, i.e., one of the queues of pending tasks is empty, it schedules a task from the available ones.
2. If there are at least γ small tasks, it schedules γ of them and then schedules a large task.
3. Otherwise, it schedules tasks of the two queues alternatively, always starting with a short task after completing one of the previous stages. Each such stage ends after a single task is completed.

These three rules for scheduling automatically partition stages into three types, depending on the rule number applied in the beginning. Note that the above scheduling rules may also influence the lengths of stages. Finally, whenever we decide to schedule a small or large task, we follow the First-In-First-Out order within the selected queue.

The idea behind the algorithm is that, as long as there are no machine crashes, and as long as there are *enough* tasks pending, there will be consecutive executions of batches of tasks consisting of a group of γ small tasks followed by a large task. Therefore, as we will see in the detailed analysis in Sections 4.1 and 4.2, the total size of γ small tasks, γp_{min} , amortizes well the size of the large task that follows, which *may (or not)* be interrupted. What is more, the tasks from both lists are getting executed relatively often, provided the queues are not empty, hence the latency will not suffer from any “frozen” task.

4.1 Completed Load.

Let us focus first on the completed load competitiveness of algorithm γ -Burst, for which we prove the following theorem.

Theorem 3 *For any given p_{min} , p_{max} and speedup satisfying condition C1 \wedge \neg C2, i.e., $s \in [1 + \gamma/\rho, \rho)$, algorithm γ -Burst is 1-completed-load competitive; more specifically, $\forall t, C(t, \gamma\text{-Burst}) \geq C(t, \text{OPT}) - (p_{max} + p_{min})$.*

Proof Let us consider a time instant t' to be the first point in the execution of γ -Burst where our claim for the total completed load does not hold, i.e. $C(t', \gamma\text{-Burst}) < C(t', \text{OPT}) -$

$(p_{max} + p_{min})$. Let us also define time instant $t^* < t'$, being the last time before t' at which algorithm γ -Burst had to make a scheduling decision, i.e., the beginning of the last stage before t' . By definition, $\mathcal{C}(t^*, \gamma\text{-Burst}) \geq \mathcal{C}(t^*, \text{OPT}) - (p_{max} + p_{min})$ holds. However, it could be the case that $\mathcal{C}(t^*, \gamma\text{-Burst}) < \mathcal{C}(t^*, \text{OPT}) - \alpha$, where $\alpha < p_{max} + p_{min}$. Let us denote by $T = (t^*, t']$ the interval between the two time instants. We then look at the different cases for the length of the interval for each type of stage:

(a) *Stage of type 1 or 3 when scheduling a small task.*

This holds when at time t^* , γ -Burst decides to schedule exactly one small task before making its next decision. Observe here, that it cannot be the case that $|T| > p_{min}/s$ since it directly contradicts the definition of time instant t^* . In such case, γ -Burst would have another scheduling decision after the p_{min} -task is completed, marking the beginning of a new *last* stage before t' . We therefore consider the following: $|T| \leq p_{min}/s$.

Algorithm γ -Burst will either be able to complete the small task it scheduled at t^* , or not. Hence, it increases its completed load by at most p_{min} by time t' . At the same time, OPT may complete at most one task (of any size) that was scheduled before t^* (if at t^* there was no machine crash). We therefore define time instant $t^{**} < t^*$ to be the last time before t^* at which OPT scheduled this last completed task. This means that interval $T' = [t^{**}, t')$ will have length $|T'|$ equal to either p_{min} or p_{max} . Considering $|T'| = p_{max}$, the following inequalities hold:

$$\begin{aligned} \mathcal{C}(t', \gamma\text{-Burst}) &\geq \mathcal{C}(t^{**}, \gamma\text{-Burst}) + \gamma p_{min} + p_{max} - \frac{p_{min}}{s} \\ &\geq \mathcal{C}(t^{**}, \text{OPT}) - p_{max} - p_{min} + \gamma p_{min} + p_{max} - \frac{p_{min}}{s} \\ &= \mathcal{C}(t^{**}, \text{OPT}) + (\gamma - 1)p_{min} + \frac{p_{min}}{s} \\ &\geq \mathcal{C}(t', \text{OPT}) - p_{max} + (\gamma - 1)p_{min} - \frac{p_{min}}{s} \\ &\geq \mathcal{C}(t', \text{OPT}) - (p_{max} + p_{min}) \end{aligned}$$

The first inequality is due to $s \geq 1 + \gamma/\rho$ which means that $p_{max} \geq \frac{\gamma p_{min} + p_{max}}{s}$. The second inequality comes from the definition of time t' and the third inequality from the fact that OPT completes a large task. The last inequality holds because $\gamma \geq 1$.

Considering now $|T'| = p_{min}$, the following holds:

$$\begin{aligned} \mathcal{C}(t', \gamma\text{-Burst}) &\geq \mathcal{C}(t^{**}, \gamma\text{-Burst}) + s p_{min} - \frac{p_{min}}{s} \\ &\geq \mathcal{C}(t^{**}, \text{OPT}) - p_{max} - p_{min} + s p_{min} - \frac{p_{min}}{s} \\ &= \mathcal{C}(t^{**}, \text{OPT}) - p_{max} + (s - 1)p_{min} - \frac{p_{min}}{s} \\ &\geq \mathcal{C}(t', \text{OPT}) - p_{min} - p_{max} + (s - 1)p_{min} - \frac{p_{min}}{s} \\ &\geq \mathcal{C}(t', \text{OPT}) - (p_{min} + p_{max}) \end{aligned}$$

The first inequality is due to the fact that $s \geq 1$. The second inequality comes from the definition of t' and the third inequality from the fact that OPT completes a small task. The last inequality holds because $s \geq 1$.

(b) *Stage of type 1 or 3 when scheduling a large task.*

This holds when at time t^* , γ -Burst decides to schedule exactly one large task before making its next decision. Similar to (a), it cannot be the case that $|T| > p_{max}/s$, since again it contradicts directly the definition of time t^* . We therefore consider the following: $|T| \leq p_{max}/s$.

Algorithm γ -Burst is either able to complete the large task scheduled or not; hence increasing its completed load by at most p_{max} by time t' . At the same time, OPT may complete at most one task (of any size) that was scheduled before t^* , plus a small task within the interval T (if there is enough time left), resulting to a maximum completion time of $p_{min} + p_{max}$. Let us then define time instant $t^{**} < t^*$, being the last time before t^* at which OPT scheduled this second-to-last completed task. This means that the interval $T' = [t^{**}, t')$ will have length $|T'|$ equal to either $p_{max} + p_{min}$ or $2p_{min}$. Hence, considering first $|T'| \leq p_{max} + p_{min}$, the following holds:

$$\begin{aligned} \mathcal{C}(t', \gamma\text{-Burst}) &\geq \mathcal{C}(t^{**}, \gamma\text{-Burst}) + s p_{min} + s p_{max} - \frac{p_{max}}{s} \\ &\geq \mathcal{C}(t^{**}, \text{OPT}) - p_{max} - p_{min} + s p_{min} + s p_{max} - \frac{p_{max}}{s} \\ &\geq \mathcal{C}(t', \text{OPT}) - p_{max} - p_{min} + (s - 1)(p_{max} + p_{min}) \\ &\quad - \frac{p_{max}}{s} \\ &\geq \mathcal{C}(t', \text{OPT}) - (p_{max} + p_{min}) \end{aligned}$$

The first inequality is due to the fact that $s \geq 1$. The second inequality comes from the definition of t' and the third from the fact that OPT completes at most $p_{max} + p_{min}$. The last inequality is again true because of $s \geq 1$.

Considering now $|T'| \leq 2p_{min}$, the following inequalities hold:

$$\begin{aligned}
\mathcal{C}(t', \gamma\text{-Burst}) &\geq \mathcal{C}(t^{**}, \gamma\text{-Burst}) + 2sp_{min} - \frac{p_{max}}{s} \\
&\geq \mathcal{C}(t^{**}, \text{OPT}) - p_{max} - p_{min} + 2sp_{min} - \frac{p_{max}}{s} \\
&\geq \mathcal{C}(t', \text{OPT}) - 2p_{min} - p_{max} - p_{min} + 2sp_{min} - \frac{p_{max}}{s} \\
&\geq \mathcal{C}(t', \text{OPT}) - (p_{min} + p_{max}) + 2(s-1)p_{min} - \frac{p_{max}}{s} \\
&\geq \mathcal{C}(t', \text{OPT}) - (p_{min} + p_{max})
\end{aligned}$$

The first inequality is due to $s \geq 1$. The second inequality comes from the definition of t' , while the third from the fact that OPT completes at most $2p_{min}$. The last inequality is again true from $s \geq 1$.

(c) *Stage of type 2.*

This holds, when at time t^* , γ -Burst has enough tasks and decides to schedule γ small tasks followed by a large one. Similar to (a) and (b) before, it cannot be the case that $|T| > \frac{\gamma p_{min} + p_{max}}{s}$, since it contradicts directly the definition of time t^* . In this case there are four cases to examine for the period T :

$$1) |T| \leq \frac{p_{min}}{s}.$$

In this case, γ -Burst is able to complete up to one small task, while OPT may complete up to one task (of any size) that was scheduled before time t^* . Following the analysis of case (a) above, $\mathcal{C}(t', \gamma\text{-Burst}) \geq \mathcal{C}(t', \text{OPT}) - (p_{max} + p_{min})$.

$$2) \frac{p_{min}}{s} < |T| \leq \frac{p_{max}}{s}.$$

In this case, γ -Burst is able to complete at most κ tasks of size p_{min} , where $\kappa = \lfloor \frac{p_{max}/s}{p_{min}} \rfloor = \lfloor \frac{p}{s} \rfloor$ and $\kappa \geq 1$. At the same time, OPT is able to complete tasks of total size up to $p_{max} + p_{min}$ for the same reasons as the ones in case (b) above. Hence, following the definition of time instant t^{**} used in case (b) above and the fact that $\kappa p_{min} \leq \frac{p_{max}}{s}$, $\mathcal{C}(t', \gamma\text{-Burst}) \geq \mathcal{C}(t', \text{OPT}) - (p_{max} + p_{min})$.

$$3) \frac{p_{max}}{s} < |T| \leq \frac{\gamma p_{min}}{s}.$$

In this case, γ -Burst can only complete up to γ scheduled small tasks; more precisely, $\kappa \leq \gamma$ of them, where $\kappa > 1$. At the same time, OPT is able to complete tasks of total size up to $p_{max} + \frac{\gamma p_{min}}{s}$; that is, one task of any size that was scheduled before t^* , and then up to total time equal to the length of the interval. Let us define time instant $t^{**} < t^*$, being the last time before t^* at which OPT scheduled that extra task. This means, that the new interval $T' = [t^{**}, t')$ will have length either $|T'| \leq p_{max} + \frac{\gamma p_{min}}{s}$ or $|T'| \leq p_{min} + \frac{\gamma p_{min}}{s}$. Considering first the case of $|T'| \leq p_{max} + \frac{\gamma p_{min}}{s}$, the fol-

lowing holds:

$$\begin{aligned}
\mathcal{C}(t', \gamma\text{-Burst}) &\geq \mathcal{C}(t^{**}, \gamma\text{-Burst}) + sp_{max} + \gamma p_{min} - \frac{p_{min}}{s} \\
&\geq \mathcal{C}(t^{**}, \text{OPT}) - p_{max} - p_{min} + sp_{max} + \gamma p_{min} \\
&\quad - \frac{p_{min}}{s} \\
&= \mathcal{C}(t^{**}, \text{OPT}) + (s-1)p_{max} + (\gamma-1)p_{min} - \frac{p_{min}}{s} \\
&\geq \mathcal{C}(t', \text{OPT}) - p_{max} - \frac{\gamma p_{min}}{s} + (s-1)p_{max} \\
&\quad + (\gamma-1)p_{min} - \frac{p_{min}}{s} \\
&= \mathcal{C}(t', \text{OPT}) - (p_{max} + p_{min}) + (s-1)p_{max} \\
&\quad + \gamma p_{min} - \frac{(\gamma+1)p_{min}}{s} \\
&\geq \mathcal{C}(t', \text{OPT}) - (p_{max} + p_{min})
\end{aligned}$$

The argumentation is similar to the previous cases. The first and last inequalities are due to the fact that $s \geq 1$. The second inequality comes from the definition of t' and the third inequality from the fact that OPT completes at most $p_{max} + \frac{\gamma p_{min}}{s}$.

Considering now $|T'| \leq p_{min} + \frac{\gamma p_{min}}{s}$, the following holds:

$$\begin{aligned}
\mathcal{C}(t', \gamma\text{-Burst}) &\geq \mathcal{C}(t^{**}, \gamma\text{-Burst}) + sp_{min} + \gamma p_{min} - \frac{p_{min}}{s} \\
&\geq \mathcal{C}(t^{**}, \text{OPT}) - p_{max} - p_{min} + sp_{min} + \gamma p_{min} \\
&\quad - \frac{p_{min}}{s} \\
&= \mathcal{C}(t^{**}, \text{OPT}) - p_{max} + (s+\gamma-1)p_{min} - \frac{p_{min}}{s} \\
&\geq \mathcal{C}(t', \text{OPT}) - p_{min} - \frac{\gamma p_{min}}{s} - p_{max} \\
&\quad + (s+\gamma-1)p_{min} - \frac{p_{min}}{s} \\
&= \mathcal{C}(t', \text{OPT}) - (p_{min} + p_{max}) + (s+\gamma-1)p_{min} \\
&\quad - \frac{(\gamma-1)p_{min}}{s} \\
&\geq \mathcal{C}(t', \text{OPT}) - (p_{min} + p_{max})
\end{aligned}$$

Again the first and last inequalities are due to the fact that $s \geq 1$. The second inequality comes from the definition of t' and the third inequality from the fact that OPT completes at most $p_{min} + \frac{\gamma p_{min}}{s}$.

$$4) \frac{\gamma p_{min}}{s} < |T| \leq \frac{\gamma p_{min} + p_{max}}{s}.$$

In this case, γ -Burst completes the γ small tasks and it is either able to complete the last large task scheduled, or not. At the same time, OPT is able to complete tasks of total size up to $p_{max} + \frac{\gamma p_{min} + p_{max}}{s}$ (one task of any size that was scheduled before t^* and then up to a total size equal to the

length of the interval). Let us define time instant $t^{**} < t^*$ as the last time before t^* that OPT scheduled that extra task, and interval $T' = [t^{**}, t')$ with length either $|T'| \leq p_{max} + \frac{\gamma p_{min} + p_{max}}{s}$ or $|T'| \leq p_{min} + \frac{\gamma p_{min} + p_{max}}{s}$. Considering first $|T'| \leq p_{max} + \frac{\gamma p_{min} + p_{max}}{s}$, the following holds:

$$\begin{aligned}
\mathcal{C}(t', \gamma\text{-Burst}) &\geq \mathcal{C}(t^{**}, \gamma\text{-Burst}) + s p_{max} + \gamma p_{min} + p_{max} - p_{max} \\
&\geq \mathcal{C}(t^{**}, \text{OPT}) - p_{max} - p_{min} + s p_{max} + \gamma p_{min} \\
&\geq \mathcal{C}(t', \text{OPT}) - p_{max} - \frac{\gamma p_{min} + p_{max}}{s} + (s-1)p_{max} \\
&\quad + (\gamma-1)p_{min} \\
&= \mathcal{C}(t', \text{OPT}) - (p_{max} + p_{min}) + (s-1)p_{max} + \gamma p_{min} \\
&\quad - \frac{\gamma p_{min} + p_{max}}{s} \\
&\geq \mathcal{C}(t', \text{OPT}) - (p_{max} + p_{min})
\end{aligned}$$

Again, the arguments are similar to the previous cases. The first and last inequalities are due to the fact that $s \geq 1$. The second inequality follows from the definition of t' and the third comes from the fact that OPT completes at most $p_{max} + \frac{\gamma p_{min} + p_{max}}{s}$.

Considering now $|T'| \leq p_{min} + \frac{\gamma p_{min} + p_{max}}{s}$, the following holds:

$$\begin{aligned}
\mathcal{C}(t', \gamma\text{-Burst}) &\geq \mathcal{C}(t^{**}, \gamma\text{-Burst}) + s p_{min} + \gamma p_{min} + p_{max} - p_{max} \\
&\geq \mathcal{C}(t^{**}, \text{OPT}) - p_{max} - p_{min} + (s + \gamma)p_{min} \\
&\geq \mathcal{C}(t', \text{OPT}) - p_{min} - \frac{\gamma p_{min} + p_{max}}{s} - p_{max} \\
&\quad + (s + \gamma - 1)p_{min} \\
&\geq \mathcal{C}(t', \text{OPT}) - (p_{min} + p_{max})
\end{aligned}$$

The first and last inequalities are due to $s \geq 1$. The second inequality follows from the definition of t' and the third from the fact that OPT completes at most $p_{min} + \frac{\gamma p_{min} + p_{max}}{s}$.

Since we have shown contradiction of the initial claim for all possible stages, γ -Burst is 1-completed-load competitive as claimed, with exact completed load competitiveness $\mathcal{C}(t, \gamma\text{-Burst}) \geq \mathcal{C}(t, \text{OPT}) - (p_{max} + p_{min})$ at all time instants t . \square

4.2 Latency.

Let us now analyze the latency of algorithm γ -Burst under the same speedup condition, $s \in [1 + \gamma/\rho, \rho)$.

We first define a basic invariant I_0 , defining the 1-1 relationship of the latency of large tasks between the queues of γ -Burst and OPT, at any time in an execution. We then define invariants I_1 and I_2 , that characterize the general p_{max} -latency of γ -Burst with respect to the latency of OPT and

its corresponding p_{min} -latency. (The latter concerns the latency of the algorithm's small tasks, while the former concerns the latency of its large tasks.)

- I_0 : For any $1 < i \leq z$, where $z = \min\{|Q(p_{max}, \gamma\text{-Burst})|, |Q(p_{max}, \text{OPT})|\}$, the latency of the i^{th} large task in the queue of γ -Burst is not bigger than the latency of the i^{th} large task in the queue of OPT.
- I_1 : $\mathcal{L}(t, p_{max}, \gamma\text{-Burst}) \leq \mathcal{L}(t, p_{max}, \text{OPT})$: This means that at time t of the execution, the maximum latency of large tasks that are pending in γ -Burst is at most equal to the maximum latency of large tasks that are pending in OPT.
- I_2 : $\mathcal{L}(t, p_{min}, \gamma\text{-Burst}) \leq \max\{\mathcal{L}(t, p_{min}, \text{OPT}), \mathcal{L}(t, p_{max}, \text{OPT})\}$: This means that the maximum latency of the pending small tasks in the execution of γ -Burst is at most equal to the maximum of the latencies of the small and large tasks that are pending in the execution of OPT at time t .

Before proving the necessary lemmas for the above invariants, let us specify the construction of the two queues $Q(p_{max}, \gamma\text{-Burst})$ and $Q(p_{max}, \text{OPT})$. When a new large task arrives in the system, it is put at the end of the queue, indexed as τ_{z+1} , where $z = \min\{|Q(p_{max}, \gamma\text{-Burst})|, |Q(p_{max}, \text{OPT})|\}$ (as defined in the invariant I_0). At any time instant t , the large task indexed as τ_1 is the first task in the queue; the one injected the earliest and is pending the longest; hence has the largest latency among the rest pending large tasks. It is also the one to be scheduled at the next time that γ -Burst, or OPT respectively, decides to schedule a large task.

Lemma 5 *When γ -Burst runs with speedup $s \in [1 + \gamma/\rho, \rho)$, the invariant I_0 holds at all times of the execution, i.e. $\forall t, \mathcal{L}(t, \tau_i, \gamma\text{-Burst}) \leq \mathcal{L}(t, \tau_i, \text{OPT})$, where τ_i the i^{th} large task in the corresponding queue.*

Proof Let us define time instants t_k of an execution of γ -Burst, where $k = 0, 1, 2, \dots$, being the times at which algorithm γ -Burst completed the k^{th} large task. Let us now focus on all t_k time instants and prove by induction that invariant I_0 holds.

Base case. At time $t_0 = 0$, the beginning of the execution, no algorithm has yet completed any task, and hence the invariant holds, $\mathcal{L}(t_0, \tau_i, \gamma\text{-Burst}) \leq \mathcal{L}(t_0, \tau_i, \text{OPT})$.

Inductive Hypothesis. We assume that at time instant t_{k-1} the invariant holds, i.e., $\mathcal{L}(t_{k-1}, \tau_i, \gamma\text{-Burst}) \leq \mathcal{L}(t_{k-1}, \tau_i, \text{OPT})$.

Inductive Step. We now look at time instant t_k , where one of the following may occur:

- (a) $t_k = t_{k-1} + \frac{p_{max}}{s}$. In this case, we know that a stage of type 2 or 4 has occurred between the two time instants, during which γ -Burst completed exactly one large task. During interval $(t_{k-1}, t_k]$ algorithm OPT could have only completed one large task, which was already scheduled before

t_{k-1} . Hence, every large task with index i at time t_{k-1} in the queue of γ -Burst, now has index $i - 1$ ($\tau_i \rightarrow \tau_{i-1}$). In the queue of OPT, they either remain with the same index (if no large task is completed) or they move one position on the same way as well. From the induction hypothesis, it follows that $\mathcal{L}(t_k, \tau_i, \gamma\text{-Burst}) \leq \mathcal{L}(t_k, \tau_i, \text{OPT})$.

(b) $t_k = t_{k-1} + \frac{p_{max} + \kappa p_{min}}{s}$, where:

1) $\kappa = \gamma$. This is the case that in the interval $(t_{k-1}, t_k]$ a stage of type 3 was executed by γ -Burst, during which OPT was able to complete at most one large task as well, that was however scheduled before t_{k-1} . Recall that $s \geq \frac{\gamma p_{min} + p_{max}}{p_{max}}$. This means that $\mathcal{L}(t_k, \tau_i, \gamma\text{-Burst}) \leq \mathcal{L}(t_k, \tau_i, \text{OPT})$, for τ_i begin the i^{th} large task in the corresponding queue.

2) $\kappa = 1$. This is the case that in the interval $(t_{k-1}, t_k]$ two consecutive stages of type 4 were executed by γ -Burst, and hence a p_{min} followed by a large task were completed. On the same time OPT could only complete at most one large task, that was however scheduled before t_{k-1} . Hence again $\mathcal{L}(t_k, \tau_i, \gamma\text{-Burst}) \leq \mathcal{L}(t_k, \tau_i, \text{OPT})$ as claimed.

3) Otherwise, it means that γ -Burst has been scheduling some small tasks before scheduling the next large task, but does not fall in cases 1) or 2). This could only happen if the interval $(t_{k-1}, t_k]$ starts with γ -Burst having no large tasks pending and hence it was choosing stages of type 1, until time instant $t^* \in (t_{k-1}, t_k)$ where some large tasks were injected. Hence, even if OPT could have completed some large tasks within $(t_{k-1}, t_k]$, due to the queue construction policy, it will not be able to complete any of the newly injected large tasks within interval $(t^*, t_k]$. Observe that $t_k - t^* = \frac{p_{max}}{s} < p_{max}$. Hence the invariant holds at time t_k as well, even if OPT completes some large tasks. Observe that as long as γ -Burst has no large tasks pending, $\mathcal{L}(t, p_{max}, \gamma\text{-Burst}) = 0$.

This completes the inductive step, and hence at all time instants t_k , the invariant I_0 holds, i.e., $\mathcal{L}(t_k, \tau_i, \gamma\text{-Burst}) \leq \mathcal{L}(t_k, \tau_i, \text{OPT})$ where τ_i the i^{th} large task in the corresponding queue.

Let us now look closer at the time when a new task π is injected. Assume this happens at a time instant t^* . The task will be injected at the end of both queues $Q(p_{max}, \gamma\text{-Burst})$ and $Q(p_{max}, \text{OPT})$, and hence all tasks already in the queues will keep their indexes i . Invariant I_0 will continue to hold; $\mathcal{L}(t^*, p_{max}, \gamma\text{-Burst}) \leq \mathcal{L}(t^*, p_{max}, \text{OPT})$. \square

Corollary 1 *When γ -Burst runs with speedup $s \in [1 + \gamma/\rho, \rho)$, the invariant I_1 holds at all times of the execution, i.e. $\forall t, \mathcal{L}(t, p_{max}, \gamma\text{-Burst}) \leq \mathcal{L}(t, p_{max}, \text{OPT})$.*

Proof Using Lemma 5, invariant I_0 holds. This means that for $z = \min\{|Q(p_{max}, \gamma\text{-Burst})|, |Q(p_{max}, \text{OPT})|\}$ and $1 < i \leq z$, the latency of the i^{th} large task, denoted as τ_i , in the queue of γ -Burst is not bigger than the latency of the corresponding large task in the queue of OPT. Recall that

the p_{max} -latency of an algorithm is defined as the maximum latency of all large tasks pending in its queue. Hence, at all time instants of an execution,

$$\begin{aligned} \mathcal{L}(t, p_{max}, \gamma\text{-Burst}) &= \mathcal{L}(t, \tau_1, \gamma\text{-Burst}) \\ &\leq \mathcal{L}(t, \tau_1, \text{OPT}) = \mathcal{L}(t, p_{max}, \text{OPT}) \end{aligned}$$

as claimed. \square

Lemma 6 *When γ -Burst runs with speedup $s \in [1 + \gamma/\rho, \rho)$, the invariant I_2 holds at all times of the execution, i.e. $\forall t, \mathcal{L}(t, p_{min}, \gamma\text{-Burst}) \leq \max\{\mathcal{L}(t, p_{min}, \text{OPT}), \mathcal{L}(t, p_{max}, \text{OPT})\}$.*

Proof Let us define time instants t_i of an execution of γ -Burst, where $i = 0, 1, 2, 3, \dots$, representing the end of the i^{th} stage. Let us now focus on these t_i time instants and prove by induction that the invariant I_2 holds.

Base case: Observe that for the first time instant $t_0 = 0$, since it is the beginning of the execution, no algorithm may complete any task yet, so the invariant holds; $\mathcal{L}(t_0, p_{min}, \gamma\text{-Burst}) \leq \max\{\mathcal{L}(t_0, p_{min}, \text{OPT}), \mathcal{L}(t_0, p_{max}, \text{OPT})\}$.

Inductive Hypothesis: We assume that the invariant holds at a time instant t_{i-1} , i.e., $\mathcal{L}(t_{i-1}, p_{min}, \gamma\text{-Burst}) \leq \max\{\mathcal{L}(t_{i-1}, p_{min}, \text{OPT}), \mathcal{L}(t_{i-1}, p_{max}, \text{OPT})\}$.

Inductive Step: We will show that it will still hold at time t_i . For that we need to consider a few cases regarding the i^{th} stage:

(a) *Its length is equal to p_{min}/s .*

In this case, the stage belongs in type 1 or 4 from the algorithm description. Observe that during this stage, γ -Burst executes exactly one small task while OPT is not able to complete any task scheduled within the stage. By the definition of speedup, $s > 1$, and hence OPT does not have time to complete even a small task within at most p_{min}/s time. However, it may complete a task that was scheduled before the beginning of the stage, either a small or a large task. At the beginning of the stage, the p_{min} -latency of γ -Burst was $\mathcal{L}(t_{i-1}, p_{min}, \gamma\text{-Burst}) \leq \max\{\mathcal{L}(t_{i-1}, p_{min}, \text{OPT}), \mathcal{L}(t_{i-1}, p_{max}, \text{OPT})\}$, and by time t_i OPT may only complete up to one task. Hence, at least one of its partial latencies (p_{min} -latency or p_{max} -latency) will be increased by p_{min}/s and thus the p_{min} -latency of γ -Burst will still be at most equal to the maximum of the two latencies. Note that since γ -Burst completes a small task, its p_{min} -latency might decrease or stay the same as at time t_{i-1} depending on the arrival time of the first small task that is still pending at time t_i . Therefore, $\mathcal{L}(t_i, p_{min}, \gamma\text{-Burst}) \leq \mathcal{L}(t_{i-1}, p_{min}, \gamma\text{-Burst})$. Combining it with the inductive assumption, the following holds:

$$\mathcal{L}(t_i, p_{min}, \gamma\text{-Burst}) \leq \max \left\{ \mathcal{L}(t_{i-1}, p_{min}, \text{OPT}), \mathcal{L}(t_{i-1}, p_{max}, \text{OPT}) \right\}$$

$$\begin{aligned}
&< \max \left\{ \mathcal{L}(t_{i-1}, p_{min}, \text{OPT}) + p_{min}/s \right. \\
&\quad \left. \mathcal{L}(t_{i-1}, p_{max}, \text{OPT}) + p_{min}/s \right\} \\
&\leq \max \{ \mathcal{L}(t_i, p_{min}, \text{OPT}), \mathcal{L}(t_i, p_{max}, \text{OPT}) \}.
\end{aligned}$$

(b) *Its length is equal to p_{max}/s .*

In this case, the stage belongs in type 2 or 4 from the algorithm description. During this stage γ -Burst executes a large task, while OPT may complete some small tasks and no large ones. In particular, it may complete up to $\left\lfloor \frac{p_{max}/s}{p_{min}} \right\rfloor = \left\lfloor \frac{\rho}{s} \right\rfloor$ small tasks. Since γ -Burst does not complete any small tasks during this stage, its p_{min} -latency increases by the length of the period, i.e., $\mathcal{L}(t_i, p_{min}, \gamma\text{-Burst}) = \mathcal{L}(t_{i-1}, p_{min}, \gamma\text{-Burst}) + p_{max}/s$. On the other hand, OPT's p_{min} -latency may decrease with the completion of some of the small tasks. Nonetheless, since the p_{max} -latency of OPT increases by p_{max}/s , the p_{min} -latency of γ -Burst will still be at most equal to the maximum of the two latencies:

$$\begin{aligned}
\mathcal{L}(t_i, p_{min}, \gamma\text{-Burst}) &= \mathcal{L}(t_{i-1}, p_{min}, \gamma\text{-Burst}) + p_{max}/s \\
&\leq \mathcal{L}(t_{i-1}, p_{max}, \text{OPT}) + p_{max}/s = \mathcal{L}(t_i, p_{max}, \text{OPT}) \\
&\leq \max \{ \mathcal{L}(t_i, p_{min}, \text{OPT}), \mathcal{L}(t_i, p_{max}, \text{OPT}) \}.
\end{aligned}$$

(c) *Its length is equal to $\frac{\gamma p_{min} + p_{max}}{s}$.*

In this case the stage is of type 3 from the algorithm description. During this stage γ -Burst executes γ small tasks followed by a large one. Since condition C2 is not satisfied, i.e. $s \geq \frac{\gamma}{\rho} + 1$, and since $s \geq \frac{\gamma + \rho}{\gamma + 1}$ (follows from the definition of parameter γ), OPT is able to complete only up to γ small tasks and no large ones within the stage. The p_{min} -latency of γ -Burst may decrease due to the γ small tasks that were completed. However, OPT's p_{min} -latency may also decrease, since up to the same number of small tasks may be completed. Note that even in the case that it does not decrease, OPT's p_{max} -latency will increase by the length of the period, i.e. $\frac{\gamma p_{min} + p_{max}}{s}$. This increases the maximum of the two partial latencies of OPT and hence the following holds:

$$\begin{aligned}
\mathcal{L}(t_i, p_{min}, \gamma\text{-Burst}) &\leq \mathcal{L}(t_{i-1}, p_{min}, \gamma\text{-Burst}) \\
&\leq \mathcal{L}(t_{i-1}, p_{max}, \text{OPT}) \\
&< \mathcal{L}(t_{i-1}, p_{max}, \text{OPT}) + \frac{\gamma p_{min} + p_{max}}{s} = \mathcal{L}(t_i, p_{max}, \text{OPT}) \\
&\leq \max \{ \mathcal{L}(t_i, p_{min}, \text{OPT}), \mathcal{L}(t_i, p_{max}, \text{OPT}) \}.
\end{aligned}$$

(d) *The stage has ended due to a machine crash.*

This case leaves at least one task from the scheduled ones incomplete. If the stage was of type 1,2 or 4, then γ -Burst was not able to complete any task and hence both its partial latencies will be increased by the length of the stage. On the same time, only in the case of γ -Burst executing a large task,

OPT would have been able to complete at most $\left\lfloor \frac{\rho}{s} \right\rfloor$ tasks of processing time p_{min} . As shown above, the invariant I_2 will be preserved true at the end of the stage. If the stage was of type 3, then regardless of the time of the machine crash and the number of small tasks that γ -Burst was able to complete, OPT's p_{max} -latency will increase by the actual length of the stage. Hence, even if the p_{min} -latency of both γ -Burst and OPT decreases, the following will hold:

$$\begin{aligned}
\mathcal{L}(t_i, p_{min}, \gamma\text{-Burst}) &\leq \\
&\quad \max \{ \mathcal{L}(t_i, p_{min}, \text{OPT}), \mathcal{L}(t_i, p_{max}, \text{OPT}) \}.
\end{aligned}$$

As claimed, all cases for the i^{th} stage guarantee the invariant I_2 , which completes the proof. \square

Theorem 4 *For any given p_{min} , p_{max} and speedup satisfying condition C1 \wedge \neg C2, i.e. $s \in [1 + \gamma/\rho, \rho)$, algorithm γ -Burst is 1-latency competitive, i.e. $\mathcal{L}(t, \gamma\text{-Burst}) \leq \mathcal{L}(t, \text{OPT})$.*

Proof From Lemmas 1 and 6, the latency of γ -Burst is

$$\begin{aligned}
\mathcal{L}(t, \gamma\text{-Burst}) &= \max \{ \mathcal{L}(t, p_{max}, \gamma\text{-Burst}), \mathcal{L}(t, p_{min}, \gamma\text{-Burst}) \} \\
&\leq \max \{ \mathcal{L}(t, p_{max}, \text{OPT}), \mathcal{L}(t, p_{min}, \text{OPT}) \} \\
&= \mathcal{L}(t, \text{OPT})
\end{aligned}$$

as claimed. \square

5 Conclusions

To conclude, in this paper we develop a detailed study on the latency and completed load competitiveness of deterministic online scheduling algorithms in a system with a machine of unpredictable crashes and restarts. More precisely, we looked at the worst-case combinations of adversarial task arrivals and machine crashes and restarts, and used resource augmentation – in the form of machine speedup – in order to guarantee latency and 1-completed-load competitiveness.

Our major contribution is showing that a specific amount of resource augmentation is *necessary* in order to achieve both latency competitiveness and 1-completed-load competitiveness. To be exact, we showed that if $s < \min\{\rho, 1 + \gamma/\rho\}$, no deterministic algorithm can be latency competitive or 1-completed-load competitive. Even more, for this range of speedup s no deterministic algorithm is c -completed load competitive, for $c > 1 - \frac{\rho}{(1+\rho+\gamma)(\rho+\gamma)}$. We then proposed an online algorithm, named γ -Burst, for the case of two task sizes, p_{min} and p_{max} . We showed, that as soon as $s \geq 1 + \gamma/\rho$ (even if $s < \rho$), our algorithm guarantees 1-latency and 1-completed-load competitiveness. These results, together with the corresponding bounds on competitiveness, show an interesting dichotomy in utilization of

resources (in this case, the speedup) when optimizing latency and completed load measures in dynamic fault-prone scheduling.

A non-trivial future line that we believe worth of further investigation, is to study systems with more than one machines prone to crashes and restarts, and applying resource augmentation to achieve competitiveness. It would be interesting to find a clear relationship between the necessary speedup and the amount of reachable competitiveness in such systems. Another future line could be to restrict the power of the adversary (e.g., bounded number of simultaneously crashes machines) and see whether with smaller speedup we can achieve better competitiveness. Accommodating dependent tasks in the considered setting could give another challenging twist to the problem. Here, by dependent task we should understand all kinds of dependencies, even having deadlines or other restrictions not necessarily with respect to other tasks, but also to various system parameters. Finally, we believe that exploring total flowtime as presented in other scheduling works, i.e., [1], as a future measure would give a nice comparison with our current results on latency competitiveness.

References

1. Igal Adiri, John Bruno, Esther Frostig, and AHG Rinnooy Kan. Single machine flow-time scheduling with a single breakdown. *Acta Informatica*, 26(7):679–696, 1989.
2. Miklos Ajtai, James Aspnes, Cynthia Dwork, and Orli Waarts. A theory of competitive analysis for distributed algorithms. In *Foundations of Computer Science, 1994 Proceedings., 35th Annual Symposium on*, pages 401–411. IEEE, 1994.
3. S Anand, Naveen Garg, and Nicole Megow. Meeting deadlines: How much speed suffices? In *Automata, Languages and Programming*, pages 232–243. Springer, 2011.
4. Matthew Andrews and Lisa Zhang. Scheduling over a time-varying user-dependent channel with applications to high-speed wireless data. *Journal of the ACM (JACM)*, 52(5):809–834, 2005.
5. Antonio Fernández Anta, Chryssis Georgiou, Dariusz R. Kowalski, and Elli Zavou. Competitive analysis of task scheduling algorithms on a fault-prone machine and the impact of resource augmentation. In *Adaptive Resource Management and Scheduling for Cloud Computing*, volume 9438. Springer, 2015.
6. Antonio Fernández Anta, Chryssis Georgiou, Dariusz R. Kowalski, and Elli Zavou. Online parallel scheduling of non-uniform tasks: Trading failures for energy. *Theor. Comput. Sci.*, 590:129–146, 2015.
7. Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, April 2010.
8. Nikhil Bansal. *Algorithms for flow time scheduling*. PhD thesis, IBM, 2003.
9. Joan Boyar and Faith Ellen. Bounds for scheduling jobs on grid processors. In *Space-Efficient Data Structures, Streams, and Algorithms*, pages 12–26. Springer, 2013.
10. Leah Epstein and Rob van Stee. Online bin packing with resource augmentation. *Discrete Optimization*, 4(34):322 – 333, 2007.
11. Antonio Fernández Anta, Chryssis Georgiou, Dariusz R. Kowalski, Joerg Widmer, and Elli Zavou. Measuring the impact of adversarial errors on packet scheduling strategies. *Journal of Scheduling*, pages 1–18, 2015.
12. Chryssis Georgiou and Dariusz R Kowalski. Performing dynamically injected tasks on processes prone to crashes and restarts. In *Distributed Computing*, pages 165–180. Springer, 2011.
13. Anis Gharbi and Mohamed Haouari. Optimal parallel machines scheduling with availability constraints. *Discrete Applied Mathematics*, 148(1):63–87, 2005.
14. Tomasz Jurdzinski, Dariusz R Kowalski, and Krzysztof Lorys. Online packet scheduling under adversarial jamming. In *Approximation and Online Algorithms*, pages 193–206. Springer, 2014.
15. Bala Kalyanasundaram and Kirk Pruhs. Speed is as powerful as clairvoyance [scheduling problems]. In *Foundations of Computer Science, 1995. Proceedings., 36th Annual Symposium on*, pages 214–221. IEEE, 1995.
16. Kirk Pruhs, Jiri Sgall, and Eric Torng. Online scheduling. *Handbook of scheduling: algorithms, models, and performance analysis*, pages 1–15, 2004.
17. Eric Sanlaville and Günter Schmidt. Machine scheduling with availability constraints. *Acta Informatica*, 35(9):795–811, 1998.
18. Daniel D Sleator and Robert E Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.
19. Rob van Stee. *Online Scheduling and Bin Packing*. PhD thesis, 2002.
20. Qi Zhang, Lu Cheng, and Raouf Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of Internet Services and Applications*, 1(1):7–18, 2010.