UNIVERSIDAD CARLOS III DE MADRID

TESIS DOCTORAL

ONLINE SCHEDULING IN FAULT-PRONE SYSTEMS: PERFORMANCE OPTIMIZATION AND ENERGY EFFICIENCY

Autor: Elli Zavou, IMDEA Networks Institute
& Universidad Carlos III de Madrid
Director: Antonio Fernández Anta, IMDEA Networks Institute

DEPARTAMENTO DE INGENIERÍA TELEMÁTICA

Leganés (Madrid), septiembre de 2016

UNIVERSIDAD CARLOS III DE MADRID

PH.D. THESIS

ONLINE SCHEDULING IN FAULT-PRONE SYSTEMS: PERFORMANCE OPTIMIZATION AND ENERGY EFFICIENCY

Author: Elli Zavou, IMDEA Networks Institute
& Universidad Carlos III de Madrid
Director: Antonio Fernández Anta, IMDEA Networks Institute

DEPARTMENT OF TELEMATIC ENGINEERING

Leganés (Madrid), September 2016

Online Scheduling in Fault-prone Systems: Performance Optimization and Energy Efficiency

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Prepared by
Elli Zavou, IMDEA Networks Institute & Universidad Carlos III de Madrid

Under the advice of
Antonio Fernández Anta, IMDEA Networks Institute

Departamento de Ingeniería Telemática, Universidad Carlos III de Madrid

---

Date: septiembre, 2016

Web/contact: http://people.networks.imdea.org/~elli_zavou / elli.zavou@imdea.org

This work has been supported by IMDEA Networks Institute and the FPU12/00505 grant from the Spanish Ministry of Education, Culture and Sports (MECD).

TESIS DOCTORAL

ONLINE SCHEDULING IN FAULT-PRONE SYSTEMS: PERFORMANCE OPTIMIZATION AND ENERGY EFFICIENCY

Autor: Elli Zavou, IMDEA Networks Institute
& Universidad Carlos III de Madrid
Director: Antonio Fernández Anta, IMDEA Networks Institute

Firma del tribunal calificador:

Presidente: Maria José Serna Iglesias

Vocal: Leszek Antoni Gasieniec

Secretario: Vincenzo Mancuso

Calificación:

Leganés, de de

"You don't develop courage by being happy in your relationships everyday.
You develop it by surviving difficult times and challenging adversity."

EPICURUS, 341 – 270 BC

Acknowledgements

Pursuing a Ph.D. has definitely been an unimaginable adventure for me, given that never have I envisioned myself as a Dr. before getting in touch with the *world of research*. With time, it became one of my aspirations in life and the last four years have been at least life changing. I came to realize that as any other big goal it requires two fundamental components; *determination* and *a strong support system*. The first is necessary in order to reach the finish line of a marathon in which you constantly challenge and must outrun yourself. The second is important in order to keep yourself motivated; it is very often that one may find himself losing faith, and having people around you that really care and support you, helps you see with clarity what seems to be miles away. Keeping my determination was a personal issue, but my *support system* was also strong, which made me feel really blessed. There are so many people, without which this thesis would not have been completed today, so let me express my sincere gratitude to each of you.

First of all, I want to thank my supervisor, Dr. Antonio Fernández Anta, for his guidance and countless advice. His faith in me, especially in those low moments, gave me so much courage. His eagerness to keep learning about every subject and his enthusiasm to work on every new idea we had, rightfully places him in the top positions of my rode models. Thank you once more for being the mentor you have been.

I want to give special thanks to my ex-supervisor (at the University of Cyprus) and now collaborator, Dr. Chryssis Georgiou, who has been the first professor I looked up to when I began my studies and has been watching my progress all the way. Thank you for introducing me to the world of research and encouraging me to follow this path. Thank you for being a mentor as well, for believing in me and supporting me in every step.

I cannot overlook another main collaborator, Dr. Dariusz R. Kowalski, who has been willing to help me since the beginning and who has hosted me at the University of Liverpool twice. Thank you very much for the hospitality, the honesty and guidance, but most of all for showing me that one should follow the path he/she really desires.

It has also been a great honor to work with Dr. Joërg Widmer and look at the research problems from a different, more practical angle with you. Along with Dr. Vincenzo Mancuso, you have given me important feedback and advice along the years, for which I am very thankful.

I would also like to thank all the people that have been part of my journey in IMDEA Networks Institute, each in his/her own way. You have all made my life in Spain much easier; the

Management and Administration Team along with the Research Support, have been of great help with the bureaucratic procedures of moving and settling down, the FPU Grant calls, the equipment provided, the intranet procedures and all the event organizations. However, I must say that the greatest merits go to all Ph.D. students, who have been exceptionally willing to help from the first moment and most of them ended up being good friends, something I am very happy about. We have shared so many moments and created many memories which I really cherish; lunch time, coffee breaks, birthday treats, hanging out after work, "a shoulder to cry on" when times were hard for some of us... I am thankful to *all* of you guys.

I would also like to thank some people that I met along this way, that became *friends*, and that have helped me probably without realizing it, with their energy, their comfort and advice: Ioanna Kalograiaki, Dr. Ángela Mendoza, Dr. Christopher Thraves Caro, Sergio Yébenes†, Areli Vázquez, Miguel Ángel López Méndez, Koulla Andreou, Dr. Paolo Casari, Dr. Nikolas Nikolaou and Dr. Andri Ioannou, you have been very special in my path.

To my *best friends*, that have always been there for me, distance or not, each one supporting me in their own way: Olympia, Andria, Klelia, Antje, Lefki, and Katerina, I promise that you will not get rid of me easily!

To my *family*, who I cherish and miss every single day, I want to express my profound gratitude. To my parents, who I love and admire deeply, for they have always supported me and my sister, by being next to us, by encouraging our decisions whether they liked them or not, because they knew it was what we wanted. Thank you Mum and Dad; for your unconditional love, your continuous presence, and most of all for the way you raised both of us. To my dearest sister Christina, thank you, for you have always been there as well, even if you did not realize it. You will always be my little sister, but believe it or not, I think that *you* are the one to have taught me more than *I* have ever taught you; and for that I will always be grateful, and proud. To my forever loved *grandparents*, the ones alive and the ones who passed away†, you are part of who I am. I wish to always make you proud and honor you. Allow me to thank in the same way my "uncle" George Melachrinos, who has been a mentor and a role model to me, for as long as I can remember. You are like a second father, and for that I feel very blessed. With him, my godfather, godmother and my whole *extended family*: uncles, aunts, cousins... with such a big family, one always feels loved. I miss you always.

Last but not least, I want to thank my beloved Dani, for his love and support. Thank you for encouraging me to go after what I want and for challenging me to surpass my limits. You have been my lighthouse and inspiration more than you may imagine, and I am grateful to you too for the person I am today.

Abstract

Everyone is familiar with the problem of *online scheduling* (even if they are not aware of it), from the way we prioritize our everyday decisions to the way a delivery service must decide on the route to follow in order to cover the ongoing requests. In computer science, this is a problem of even greater importance. This thesis considers two main families of online scheduling problems in computer science, and aims to provide an extended clear framework for their analysis, presenting at the same time some common characteristics that connect these problems.

The first and main family of online scheduling problems considered, is *task scheduling in fault-prone computing systems*. As the number of clients and the possibilities offered by the rapid development of computing systems, grow with time, the increase of demands of computationally-intensive tasks is inevitable. Uniprocessors are no longer capable of coping with the escalation of these demands, which among others, has led to the development of multicore-based parallel machines, Internet-based computing platforms and co-operational distributed systems. Nonetheless, the challenges of these systems, even of the simplest ones, are numerous: They have to deal with continuous dynamic requests from the clients, which are probably not of the same nature (require different amount of computational resources). The processing elements (i.e., machines) may suffer from unpredictable failures, either malicious or due to overload. Furthermore, depending on the size of these systems and the exact processing units, their power consumption may be of significant amount; even equal to the electricity needed for a small town. Hence, limiting their power consumption is another challenge.

To analyze such a system one must consider the *online* nature of the problem; the dynamic *task arrivals* (client requests) of different *sizes* (computational demands), and the unpredictable *machine crashes and restarts* (failures). It is important to give guarantees for the performance of the algorithms used in these systems, thus the thesis conducts *worst-case competitive analysis* and covers a significant level of the three dimensions of the problem. More precisely, it studies the effects of the *number of machines*, the *number of different task sizes* and the *speed of the machines* – which as will be explained through the thesis, affects the power consumption of the system – on the efficiency of online scheduling algorithms. As performance measures, this thesis uses the *completed load*, the *pending load* and the *latency* competitiveness of the algorithms. In some cases, it considers the long-term competitiveness versions of these measures as well.

One of the most important results shown, is that *resource augmentation* in the form of in-

creasing the machine speedup, *is necessary* in order to achieve some competitiveness, or to reach optimal competitiveness. The sufficient amount of speedup is found, and online algorithms that achieve the desired competitiveness are proposed and analyzed. Apart from the algorithms designed, some of the most widely used algorithms in scheduling are also analyzed in the model considered for the first time; namely, *Longest In System* (LIS), *Shortest In System* (SIS), *Largest Processing Time* (LPT), and *Smallest Processing Time* (SPT). Nonetheless, deciding on the *best* algorithm between them, is not easy. Each algorithm behaves better with respect to a different evaluation metric and under different model parameters.

The second family of problems considered, is *packet scheduling over an unreliable wireless communication link*. As claimed, these problems have a strong connection to the task scheduling problem, especially when considering one machine and no speedup, hence some of the results can be shared. A setting with a single pair of nodes is considered, connected through an unreliable wireless channel. The sending station transmits packets to a receiving station over the channel, which can be jammed and hence corrupt the packet being transmitted. First, worst-case scenarios are assumed for the channel jams, modeled by a malicious adversarial entity. The packet arrivals however, follow a stochastic distribution and competitive analysis of scheduling algorithms is pursued giving matching bounds for the most pessimistic scenarios of channel jams. The aim of the algorithms is to find the schedule (or order or transmission of the arriving packets) in order to maximize the *asymptotic throughout*, which corresponds to the long-term competitive ratio of total length of successfully transmitted packets.

Then, a slightly different problem is considered, assuming infinite amount of data to be transmitted over the same unreliable communication link. This time however, an adversarial entity with constrained power is assumed for the channel jams. The constrained power is modeled by an Adversarial Queueing Theory (AQT) approach, defined with two main parameters; ρ, the error availability rate, and σ, the maximum batch of errors available to the adversary at any time. This is the first time AQT is used to model channel jams; it has been mostly used to model the packet arrivals in networking problems. In this problem, the scheduling algorithms must decide on the length of the packets to be transmitted, with the objective of maximizing the *goodput rate*; the rate of successfully transmitted load. It is seen, that even for the simplest settings, the analysis and results are not trivial.

Table of Contents

Acknowledgements XI

Abstract XIII

Table of Contents XV

List of Tables XIX

List of Figures XXI

1. Introduction 1

1.1. Motivation and purpose 1
1.2. Preliminaries 3
1.3. Contributions 5
1.4. Outline 8

2. Literature Review 11

2.1. Machine scheduling with availability constraints 11
2.2. Parallel online scheduling 13
2.3. Load balancing and distributed online scheduling 13
2.4. Online bin packing 13
2.5. Job scheduling in the grid 14
2.6. Packet scheduling in wireless networks 14
2.6.1. Unconstrained jamming 16
2.6.2. Constrained jamming 17

I Task Scheduling 19

3. Problem Definition and General Model 21

3.1. Computing setting 21
3.1.1. Repository 22
3.2. Machine processing 23

3.3. Adversarial entity 23
3.3.1. Task arrivals 24
3.3.2. Machine crashes and restarts 24
3.4. Global notation 24
3.5. Efficiency measures 26
3.5.1. Competitive analysis 26
3.5.2. Long-term competitiveness 27

4. NP-hardness **29**
4.1. Completed load 29
4.2. Pending load 30
4.3. Latency 31

5. Single Machine **33**
5.1. Properties of *ALL* work-conserving and deterministic algorithms 33
5.1.1. Properties of $M\langle 1,1,1\rangle$ 34
5.1.2. Properties of $M\langle 1,1,\infty\rangle$ 35
5.1.3. Properties of $M\langle 1,1,2\rangle$ 37
5.1.4. Properties of $M\langle 1,s,\infty\rangle$ 40
5.2. Competitiveness without speedup 43
5.2.1. ρ-SPT-LPT: an *optimal* algorithm for $M\langle 1,1,2\rangle$ 44
5.3. Competitiveness with speedup 49
5.3.1. *Necessary* speedup conditions 49
5.3.2. γ-Burst: an *optimal* algorithm for $M\langle 1,\left[1+\frac{\gamma}{\rho},\rho\right),2\rangle$ 60
5.3.3. *Sufficient* speedup conditions 68
5.4. Competitiveness of popular algorithms 70
5.4.1. Completed and pending load competitiveness 71
5.4.2. Latency competitiveness 82

6. Multiple Machines **85**
6.1. Scheduling with *redundancy avoidance* 85
6.2. Properties of *some* work-conserving and deterministic algorithms 87
6.2.1. Properties of $M\langle m,1,1\rangle$ 88
6.2.2. Properties of $M\langle m,s,\infty\rangle$ 88
6.3. Competitiveness without speedup 91
6.3.1. k-Amortized: an *optimal* algorithm for $M\langle m,1,k\rangle$ – pairwise divisible . 91
6.3.2. Gk-Amortized: an *optimal* algorithm for $M\langle m,1,k\rangle$ – general 96
6.4. Competitiveness with speedup 99
6.4.1. Algorithm (m,β)-LIS 99
6.4.2. γm-Burst: an *optimal* algorithm for $M\langle m,\left[1+\frac{\gamma}{\rho},\rho\right),2\rangle$ 103

6.4.3. (m, β)-LAF: an *optimal* algorithm for $M\langle m, 7/2, k\rangle$ 108

II Packet Scheduling 113

7. The Impact of Adversarial Jamming 115
7.1. Problem definition and model specifications 115
7.1.1. Network setting 116
7.1.2. Arrival models 116
7.1.3. Packet bit errors / Channel jams 117
7.1.4. The power of the adversary 117
7.1.5. Efficiency metric: asymptotic throughput 118
7.2. Asymptotic throughput competitiveness 119
7.2.1. Upper bounds 120
7.2.2. Lower bound and algorithm C-ρ-SPT-LPT 126

8. Constrained Jamming 131
8.1. Problem definition and model specifications 131
8.1.1. Dynamic model 132
8.1.2. Static model 134
8.1.3. Moving from the static to the dynamic model 135
8.2. Uniform packets for the static model 135
8.3. Adaptive algorithms for static model with $f = 1$ 137
8.3.1. Algorithm S-DEC 137
8.3.2. Algorithm S-OPT$(T, 1)$: optimal for $f = 1$ 138
8.4. Algorithm S-OPT(T, f): optimal for any $f > 1$ in the static model 140
8.5. Uniform packets for the dynamic model 146
8.6. Discussion 150

9. Conclusions 153
9.1. Task scheduling 153
9.2. Packet scheduling 156

References 159

A. Summary of Publications 169

List of Tables

3.1. Important notation and definitions. 25

5.1. Metric comparison for all work-conserving (ALG_W) or deterministic (ALG_D) scheduling algorithms for different ranges of speedup. The last column provides the theorem numbers where the results of the corresponding row can be found. Recall 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. Note that for the latency there are no general results for speedup at least ρ. . . 34

5.2. Detailed metric comparison of the four widely-used scheduling algorithms, studied in detail for different ranges of speedup. The last column provides the theorem numbers where the results of the corresponding row can be found. Recall 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. 71

6.1. Detailed metric comparison of the different algorithms proposed for the multiple machine setting, as well as negative results from Chapter 5 that still hold, for different ranges of speedup. The last column provides the theorem numbers where the results of the corresponding row can be found. Recall that by definition, 0-completed-load competitiveness ratio equals to non-competitiveness, as opposed to the pending load, where non-competitiveness corresponds to an ∞ competitiveness ratio. Note, that ALG is used for any work-conserving or deterministic algorithm, and GroupLIS is a type of parallel algorithms introduced here. Note also, that parameter β is a constant that characterizes the corresponding algorithms. 86

8.1. Values of packet length p and optimal useful payload $\text{UP}^*_{(T,2)}$ achieved with Algorithm S-OPT$(T, 2)$. 143

List of Figures

1.1. The three directions in analyzing the *online task scheduling* problem. 6

3.1. The computational setting, with the m homogeneous machines, the shared Repository, and the three operations; *inject*, for the dynamic task arrivals from the clients, *get*, for the machines to obtain the set of pending tasks, and *inform* for the repository to update the set of pending tasks. 22

5.1. Illustration of Scenario 1. It uses the fact that $\hat{\rho} < \rho$. 38
5.2. Illustration of Scenario 2. It uses the fact that $\bar{\rho} \leq \rho$. 38
5.3. Upper bound on the long-term completed-load competitiveness under adversarial task arrivals. (a) For any algorithm ALG, $\mathcal{C}(\text{ALG}) \leq \bar{\rho}/(\rho + \bar{\rho})$ (see analysis in Subsection 5.1.3). (b) For algorithm SPT, $\mathcal{C}(\text{SPT}) \leq 1/(\rho + 1)$. Observe that algorithm SPT has a significantly lower bound as ρ increases. 44
5.4. Illustration of Scenario 1. It uses the property $(\kappa\pi_{min} + \pi_{max})/s > (\kappa+1)\pi_{min}$, for any integer $0 \leq \kappa < \gamma$ (Property 2). 50
5.5. Illustration of Scenario 2. It uses the property $(\gamma\pi_{min} + \pi_{max})/s > \pi_{max}$ (condition **C2**). 51
5.6. Illustration of Scenario 1(a). 56
5.7. Illustration of Scenario 1(b)(ii). 56
5.8. Illustration of Scenario 2. 57

7.1. Upper bounds on the asymptotic throughput under stochastic packet arrivals with π_{min}-packet arrival probabilities as follows: (a) and (b) $p = 0.01$, (c) and (d) $p = 0.1$ and (e) and (f) $p = 0.3$. On the left column we give 3D representations for a range of π_{min} and π_{max} values, while on the right we give 2D representations of the same graph, under arbitrarily fixed π_{max}. 121

8.1. The goodput rate of algorithms S-OPT$(T, 1)$, S-DEC and the uniform packet scheduling for both static and dynamic models, with $\sigma = f = 1$ in a time interval $1/\rho = T = 1 \ldots 22$. 150

Chapter 1

Introduction

1.1. Motivation and purpose

Online scheduling problems are problems faced not only by computer scientists, but by every person daily, even if he or she is not familiar with the term. We find them in many forms around us; from the way we give priorities to our everyday decisions – choosing the order of our chores, tasks at work, events we should attend – to the route of a delivery service that must be followed in order to cover the incoming requests, to the way a Central Processing Unit (CPU) scheduler decides the process to be executed next. The main characteristic of these problems is the fact that new information becomes available over time, usually referring to new requests that must be satisfied, and should be taken into consideration for the next decision. This is what makes the scheduling problems *online*.

This thesis focuses on two families of online scheduling problems of computer science. The first (and *main*) problem of interest of this thesis is *task (or job) scheduling* in computing systems. During the last decades, there is a dramatic increase on the demand of processing computationally-intensive jobs. Uniprocessors are no longer capable of coping with the high computational demands of such jobs, which has led to the development of multicore-based parallel machines, such as the K-computer [93], and Internet-based supercomputing platforms, such as SETI@home [63] and EGEE Grid [46]. They have all become prominent computing environments that have attracted a lot of interest, and not only from the academic sector.

However, computing in such environments raises several challenges; computational jobs are injected dynamically and continuously, each job may have different computational demands (e.g., CPU usage or processing time) and the processing elements are subject to unpredictable failures. Think of a data center for example; a unified facility of a large number of servers, often tens of thousands, usually hosting petabytes (PB) of data and consuming up to various tens of Mega Watts (MW) to satisfy the needs of its users. Large data centers may use as much electricity as a small town [69]. Preserving power consumption is hence another challenge of rising importance. Therefore, there is a corresponding need for developing algorithmic solutions that would

efficiently cope with such challenges.

As expected, much research has been dedicated to task scheduling problems over the last decades, each work addressing different challenges (e.g., [15,32,33,35,47,49,50,55,61,74,92]). For example, many works address the issue of dynamic task injections, but do not consider failures (e.g., [24,58]). Other works consider scheduling on one machine (e.g., [5,85,91]), with the drawback that the power of parallelism is not exploited (provided that tasks are independent). Some works consider failures, but assume that tasks are known a priori and their number is bounded (e.g., [7, 10, 22, 32, 49, 50, 61]), where others assume that tasks are uniform, that is, they have the same processing times (e.g., [47, 48]). Several works consider power-preserving issues, but do not consider, for example, failures (e.g., [17,24,92]).

However, none of the research done so far has combined all the challenges together, which is something that makes the problem much more difficult. The aim of the first part of the thesis is to combine the aspects of online task scheduling to make a complete worst-case analysis: the dynamicity of task arrivals, the machine crashes and restarts, and the energy efficiency. In my understanding, this is the first extensive and rigorous online analysis, even for some very popular scheduling algorithms, considering a fault-prone setting. The main goal of the thesis is to achieve reliable and stable computations in such environments, surpassing the several challenges they withhold, while keeping the energy consumption to a minimum.

A second area of online scheduling problems considered in the thesis, is *packet scheduling* [68] over an unreliable wireless link. This is one of the fundamental problems in computer networks, and is quite related to the task scheduling problem in computing systems, in the following way. As packets arrive, the sending node (or scheduler) needs to continuously make scheduling decisions, without knowledge of future packet arrivals, and a natural objective is to maximize the throughput of the link or to achieve stability. One should also consider congestion and increased noise levels or transient interference on the wireless communication link, that in the worst case could be caused by a malicious jamming entity. This makes the connection to the task scheduling problem above even stronger. Packet scheduling has been treated as an online scheduling problem before (e.g., [64, 67, 71, 75, 88, 89]) and much work has also been done on overcoming failures in wireless channels (e.g. [11, 19, 30, 31, 72, 80, 82, 87]). However, as in the task scheduling problem, each work has addressed a different challenge. For example, some works consider stability as their efficiency measure [11, 30, 31], which aims to bound the number of packets waiting in the node buffers. Some works consider the design of a robust medium access (MAC) protocol under synchronized discrete time steps [80, 83] or look at randomized algorithms [82], while others consider deadlines on the packet transmissions [64, 67].

As explained, in the volume of research done so far in both task and packet scheduling problems, there has either been some challenging part of the problem missing, or the model considered had a different objective. This thesis, aims to combine all the aspects of online scheduling for these problems – the dynamic task arrivals (resp., packet arrivals), the machine crashes and restarts (resp., channel jams), and the energy efficiency – and complete a thorough, worst-case

analysis, finally giving online algorithms that optimize the system's performance.

1.2. Preliminaries

This section presents some background knowledge, that is important to be understood by the reader before (s)he continues with the details of the work.

Performance measures

The principal method used for the analysis of online algorithms is called *competitive analysis* [4, 20, 76, 86], and was first introduced in the area of memory management algorithms by D. Sleator and R.E. Tarjan in [86]. With competitive analysis, the performance of an online algorithm, that must satisfy a dynamic sequence of requests without any information for the future, is compared to the performance of an optimal offline algorithm that knows the sequence of requests from the beginning of an execution. An algorithm is said to be competitive, with respect to the desired optimization metric, if its *competitiveness ratio* – the ratio between its performance and the performance of the offline algorithm – is bounded. More precisely, when the competitiveness ratio equals α, the algorithm is said to be α-competitive.

Another methodology used in some works for the analysis of online algorithms, as also done in this thesis, is the *long-term competitiveness* (see for example [36, 90]). For an algorithm to be α-competitive in the long run, one must look at the limit of the corresponding (instantaneous) competitive ratio, as time goes to infinity; the ratio between the performance complexity of the online algorithm and the performance complexity of the offline one must converge to α as time grows.

Adversarial models and types of algorithms

In competitive analysis, one usually studies the performance of the online algorithm under some *adversarial model*; that is, a sequence of "difficult" input data (or requests) whose goal is to make the competitiveness ratio as bad as possible (maximize or minimize it, depending on the performance metric). For the problems studied in this thesis, the input consists of the task arrivals (resp., packet arrivals), with their specifications, and the machine crashes and restarts (resp., channel jams). An adversary – the entity responsible for the input sequence – can belong to one of the following three categories:

- *Oblivious:* This adversary has to determine the complete input sequence before the algorithm starts; it cannot adapt to possible random decisions of the algorithm. It is also referred to as the *weak* adversary.

- *Adaptive online:* This adversary knows how the algorithm reacted to earlier inputs; the history of its scheduling decisions, but has no access to the possible randomness used to react to the current input. It is also referred to as the *medium* adversary.

- *Adaptive offline:* This adversary has a complete knowledge of the history of the algorithm's scheduling decisions, as well as the random bits used to serve the current input. It is also referred to as the *strong* adversary.

It is important to note that the distinction of the adversarial power between the different models is mostly meaningful for randomized algorithms. For deterministic algorithms, any of the adversaries can simply compute the state of the algorithm at each time instant of the future and choose the appropriate "difficult" input accordingly from the beginning. In fact, the work done in this thesis is mostly on deterministic algorithms, and unless otherwise stated, it assumes the strongest adversary, the adaptive offline one, for the creation of the input. However, it is important to present the different adversarial models in order to clarify to the reader the reason of the choice.

This thesis focuses mainly on *work-conserving* and *deterministic* online algorithms. By work-conserving, I mean an algorithm that schedules tasks as soon as it has an available one, without any idle periods of time. A deterministic algorithm is one that, given a specific input it will always produce the same output, making the same computational steps; its decisions are based on non-probabilistic rules.

Energy-Efficiency & Resource Augmentation

In Part I of the thesis, when tackling the task scheduling problem, the energy-efficiency of the system under study is considered. More precisely, in the case of speed scaling of the machines, the aim is to minimize the total power consumption. Assuming that each machine has a processing speed $x \geq 1$, where x can be adapted using speed scaling, its power consumption (i.e., the energy consumed per unit of time) grows superlinearly with x. It is typically assumed that the power function is of the form: $P(x) = x^{\alpha}$, for $\alpha > 1$ (see [3, 13, 92]).

Hence, one can consider a form of *resource augmentation* [73], to cope with some limitations of the system's performance. Instead of increasing the number of machines, which is also considered a form of resource augmentation, some static *speed scaling* [6, 8, 24] is introduced, in the form of *speedup* $s \in \mathbb{R}^{+}$, increasing the "natural" processing speed of the machine to $s \geq 1$. This means, that the machine can complete any task s times faster than its baseline system-specified execution time, which will impact the *energy consumption* of the machine. Specifically, it implies an additional factor of $s^{\alpha-1}$ in the power consumed, and hence, the energy consumed is increased accordingly, as well.

Preemption & Migration

When preemption is allowed in scheduling, a task execution may be suspended or interrupted at any time instant and resumed in a future time (after the point of suspension). When migration is also allowed, *that* task can be resumed in the same or a different machine of the system. Note that neither preemption nor migration are easily implemented in real systems.

What is more, considering non-preemptive schedules, creates an important restriction for the

online algorithms. As will be seen in Chapter 2, much of the research done already in the area studies mostly the preemptive model. Despite this restriction, the work presented in this thesis focuses on non-preemptive schedules, which makes a stronger model, and shows how this difficulty can be surpassed.

1.3. Contributions

Task Scheduling

The first part of the thesis considers a computing system in which tasks of *different* execution times arrive *dynamically and continuously*, and must be executed by a set of $m \in \mathbb{N}$ machines that are prone to *crashes and restarts*. Due to the dynamicity involved, this task-executing problem is seen as an *online problem* and *competitive analysis* is pursued as described above.

The thesis explores the impact of parallelism, different task execution times and faulty environment, on the competitiveness of the online system considered. The efficiency is measured by three performance metrics; the minimum *completed load*, the maximum *pending load* and the maximum *latency*. Here, *load* is the sum of execution times of the tasks (completed or pending tasks for the corresponding measure), and latency is counted as the maximum time a task remains in the system, from the moment it arrives to its completion. For these, I study competitiveness at any point in the execution as well as in the long run. An algorithm is considered to be α-completed-load competitive, if under any adversarial pattern (for both task arrivals and machine crashes and restarts) its completed load complexity is at least α times larger than the completed load complexity of any offline algorithm X that knows the same adversarial pattern a priori. This holds similarly for α-pending-load and α-latency competitiveness, though the corresponding complexity of the online algorithm must be at most α times the complexity of X. For some parts of the thesis, the three performance measures are also considered in the long run, and the long-term competitiveness is taken into account. This means, that α is taken as the limit of the ratio of the corresponding complexities, as time goes to infinity.

As already mentioned, the impact of *resource augmentation* is also explored in the form of processing speedup, $s \geq 1$, in order to achieve or improve the competitiveness of the three measures. Note, that due to the nature of competitive analysis, there is nothing to investigate if the offline solution makes use of speed-scaling as well. Hence, the online algorithm runs with speedup s, while the offline algorithm X runs with no speedup, i.e., $s = 1$.

After clearly defining the model studied, in Chapter 3, the analysis in divided in two main lines; first focusing a *single machine* system and then an *arbitrary number of machines*. First, in the single machine case, the three performance measures and the conditions under which some competitiveness may be achieved, are studied in depth. The use of *resource augmentation* with speedup $s \geq 1$ is also considered, and investigation for the amount of speedup necessary to achieve or improve and ideally optimize the competitiveness of each measure by different algorithms, is conducted. Then, the case of a fixed arbitrary number of machines is considered,

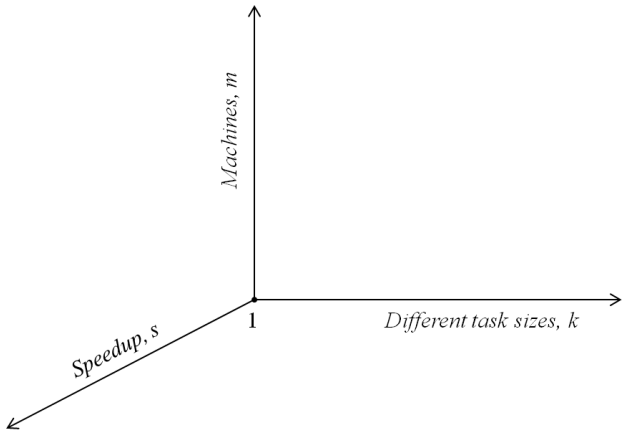

Figure 1.1: The three directions in analyzing the *online task scheduling* problem.

exploiting the parallelism of the system to develop new competitive algorithms. Again, the use of resource augmentation takes place, and this time the analysis is focused mainly on the completed and pending load competitiveness measures.

In summary, the thesis gives *lower and upper bounds* for the three competitiveness measures. For the *negative results*; the upper bounds for the completed-load competitiveness and the lower bounds for the pending-load and latency competitiveness, it presents adversarial strategies for which no algorithm can be competitive, or can only reach a certain *non-optimal* competitiveness. Observe that if these results hold for one machine, they will also hold for multiple machine settings. For the *positive results* on the other hand; lower bounds for the completed-load competitiveness and upper bounds for the pending-load and latency competitiveness, it proposes online scheduling algorithms and gives corresponding proofs that guarantee the claimed competitiveness. However, the results presented for one machine may not hold in the case of multiple machines, hence, different proofs must be provided.

What is more, the analysis includes some of the most widely used scheduling algorithms, in order to show the limits and/or privileges they have in their performance. These are:

- *Longest In System (LIS)*: it schedules the task that has been waiting the longest; i.e., it follows the FIFO (First In First Out) policy,
- *Shortest In System (SIS)*: it schedules the task that has arrived latest; i..e, it follows the LIFO (Last In First Out) policy,
- *Largest Processing Time (LPT)*: schedules the task with the biggest size, and
- *Shortest Processing Time (SPT)*: schedules the task with the smallest size.

As seen in Figure 1.1, there are three directions to be followed in order to cover the whole spectrum of unanswered questions regarding the online task scheduling problem. That is, increasing the number of machines considered, m, the amount of different task sizes, k, and the amount of speedup, s.

As mentioned, the goal is to give an exhaustive analysis of all directions and give a complete framework for the online task scheduling problem. Apart from the analysis of already existing algorithms, the ultimate goal is to develop (optimal) competitive online algorithms that require the smallest possible speedup. Hence, one of the main challenges encountered, is to identify the speedup thresholds under which competitiveness cannot be achieved and over which it is possible. In some sense, this work can be seen as the investigation of the trade-offs between *knowledge* and *energy* in the presence of failures: How much energy (in the form of speedup) does a deterministic online scheduling algorithm need in order to match the efficiency (i.e., to be competitive with) of the optimal off-line algorithm that possesses complete knowledge of failures and task injections? How competitive can some popular scheduling algorithms be in this model without any speedup, and how does the speedup affect their performance? These are only some of the questions that the thesis aims to answer.

Packet Scheduling

The second part of the thesis turns its focus to online packet scheduling over a wireless *unreliable communication link*, where packets of *different lengths* are transmitted over the link. The link may be jammed, thus corrupting the packet that is being transmitted at the moment by some bit errors. Such errors may also be caused by an increased noise level or transient interference on the link, but since the main interest of the work done in the thesis is on worst-case analysis, a malicious entity (or an attacker) is assumed, which is modeled as an adaptive adversary. In the case of an error, the affected packets must be re-transmitted.

A new metric (for this kind of problem) is proposed, called *asymptotic throughput*, used for the analysis of the performance of packet scheduling algorithms under unreliable links. This metric is a variation of the competitive ratio typically considered in online scheduling described earlier. Instead of considering the ratio of the performance of a given algorithm over that of the optimal offline algorithm at each time instant, it considers the limit of this ratio as time goes to infinity. This corresponds to the *long term competitive ratio* of the algorithm with respect to the optimal. Observe the direct relation with the task scheduling performance measures; the asymptotic throughput corresponds to the long-term completed-load competitive ratio, defined for the task scheduling problem.

Within this framework, the *packet arrival* is continuous and can either be controlled by the adversary or be stochastic. It can be easily seen that the case of adversarial packet arrivals is equivalent to the task scheduling problem when considering one single machine and no speedup, i.e., $m = s = 1$. Hence, the second part of the thesis focuses on the differences with the model of *stochastic* arrivals.

Then, the case of *constrained jamming* is studied, prohibiting the adversary from having infinite power. A form of Adversarial Queuing Theory (AQT) [21] is introduced for the channel jams – defining the rate at which the adversary may jam the channel and the length of the largest burst of jams it may cause – and slightly changes the objective to maximizing the amount of data

transmitted with time, called *goodput*. Another difference of this model, is that an infinite amount of data to be sent is assumed, from the beginning of the execution, and the proposed deterministic algorithms decide the lengths of the packets to be sent in order to optimize the goodput.

1.4. Outline

As already mentioned, this thesis is separated in two parts. Part I presents an extensive study of the main problem of interest, Online Task Scheduling, while Part II investigates some aspects of the Online Packet Scheduling problem, also related to task scheduling. Before the two Parts, Chapter 2 summarizes the state-of-the-art research in the related areas.

The aim of the first part, is to complete a rigorous competitive analysis of task scheduling in fault-prone systems of single and multiple machines. Chapter 3 gives the detailed specifications for the general model considered; presenting the model of dynamic task arrivals with different computational demands (sizes) and unexpected machine crashes and restarts. For the realization of the worst case analysis, both task arrivals and machine failures are assumed to be "controlled" by an adversary, and three efficiency measures are presented and explained; completed load, pending load and latency.

Then, Chapter 4 shows the **NP**-hardness of the offline version of the problem for two of the measures considered; completed and pending load. The following Chapters, 5 and 6, separate the analysis of online algorithms in single and multiple machines respectively, aiming to give the reader a good understanding of the analysis approach in the *simplest* model and then continue with the more complicated case. Nonetheless, the single machine case itself is rather challenging. Starting by Section 5.1, the reader can find some facts that apply to *all* work-conserving and deterministic algorithms. Then, in Sections 5.3.1 and 5.3.3, the *necessary and sufficient* conditions for the value of speedup are shown, in order to achieve competitiveness for the pending load and latency measures, and achieve optimal competitiveness for the completed load metric. Section 5.2 shows that without speedup, only *some* competitiveness can be achieved, and only for the completed load metric. The last section of the single machine chapter, Section 5.3, shows good (and improved for the case of completed load) results for all three measures.

In Chapter 6 the analysis for the multiple machines case can be found, again giving some general properties first, followed by the proposal and analysis of optimal algorithms in different versions of the model; in Section 6.3 when run without speedup and in Section 6.4 with speedup.

The aim of the second part of the thesis, is to initiate the study of another family of online scheduling problems, the packet scheduling over an unreliable communication, and show some of the common aspects with the task scheduling in fault-prone systems. Considering one wireless channel between a sending and a receiving node, this part aims to investigate the effects of unreliable communication, even considering malicious attackers, to the efficiency of the transmissions. It considers however, two problems related to packet scheduling optimization, separated in Chapters 7 and 8.

More precisely, in Chapter 7 the impact of adversarial (worst-case) jamming is explored. One of the closest relations of this problem to the task scheduling one, is the efficiency measure called *asymptotic throughput*. It corresponds to the long-term competitive ratio of the completed load complexity in task scheduling, something described further in the chapter. Due to this similarity, some of the results presented in Chapter 5 also hold for the packet scheduling problem considered here; the ones assuming adversarial packet arrivals.

After giving the problem definition and a clear model for this study, the chapter focuses on the differences imposed by considering *stochastic packet arrivals* instead of adversarial ones, showing upper and lower bounds of the achievable asymptotic throughput. It also considers two types of error detection and feedback mechanisms for the corrupted packets, showing that *immediate feedback* is necessary.

Then, in Chapter 8 a slightly different problem is presented, along with the specifications of the model. In this case, the jams of the channel are caused by a *constrained jammer*; meaning that the adversarial malicious entity only has limited power to disrupt the communication between the sending and receiving nodes of the network. This is an interesting and challenging research line as well, since the goal is to define a scheduling policy that chooses the optimal packet lengths of the packets sent in order to transmit a maximum amount of data in the most efficient way; i.e., achieve faster and successful data transmissions.

Chapter 2

Literature Review

The two scheduling problems considered in the thesis, along with their results, involve only a small portion of the problems studied in the area of scheduling. Scheduling problems include various *optimization problems* and are studied in different settings and with different objectives. This chapter describes some of the most related topics that have been studied extensively by the computer science community throughout years of research, emphasizing on both *similarities* and *differences* to the work presented in this thesis. For more information on the vast amount of work that exists on online scheduling the reader is adviced to consult these two books [74] and [77].

2.1. Machine scheduling with availability constraints

Probably, the most important research line related to this thesis in the vast area of online scheduling, is the study of *machine scheduling with availability constraints* (e.g., [22,47,51,84]). One of the most important outcomes of this line is the necessity of algorithms that take into account unexpected machine breakdowns. Most works, to cope with this requirement, allow *preemptive scheduling* [51, 55] and optimality is shown only for *nearly online algorithms*; for example, algorithms that need to know the time of the next job arrival or machine availability. Among these works some also consider energy issues, and use speed-scaling to tune the power consumption of the processors (e.g., [6, 17, 23, 24]).

Chlebus et al. [32] consider the DO-ALL problem in a synchronous, message-passing distributed system where processors crash and restart. Unlike the work considered in the thesis, this is an offline problem; all tasks are known from the beginning of an execution, and their objective is to provide deterministic algorithms that complete all the tasks as fast as possible and with the minimum amount of messages among the processors. Their work is the first to consider the cases of allowing restarts of the machines.

Later on, Chlebus et al. [28] considered a similar setting, but using randomized algorithms, almost reaching the upper bound of the worst-case estimate of the deterministic algorithm given in [29]. The latter is another work of Chlebus et al., in which the authors presented an algorithm

against an unbounded adversary; one that can crash all but one processors.

On a related work, Georgiou et al. [48] are the first to show a non-trivial lower bound on the work complexity of any algorithm solving the DO-ALL problem, in terms of total number of tasks, number of processors *and* number of failures (no restarts though). Previous results had failed to accurately reflect the impact of failures to the bounds shown. They consider both shared memory and message-passing models, and improve the analysis of the work and message complexity of the algorithm presented by Chlebus et al. [32].

On a different line, Alistarh et al. [7] consider the DO-ALL problem in an asynchronous model and introduce a concurrent data structure to improve the bounds of complexity known so far. The system has a shared memory in which processors communicate by accessing it using atomic read and write operations. However, the authors do not assume any processor failures, instead they assume an adaptive adversary that controls their speeds.

The whole study of online task scheduling of this thesis, was triggered by the work of Georgiou and Kowalski [47]; the most closely related work to the problem presented in Part I. The authors consider a cooperative computing system of m message-passing processors that are prone to crashes and restarts, and have to collaborate to complete dynamically injected tasks. For the efficiency of the system, they perform competitive analysis looking at the maximum number of pending tasks. Unlike the assumptions made here though, the computation in their work is broken into *synchronous rounds* and the notion of *per-round pending-task competitiveness* is considered. Compared with this work, the thesis looks at the completed- and pending-load competitiveness as well as the latency competitiveness, and assumes continuous and asynchronous task executions. Furthermore, in [47] tasks are assumed to have *unit cost*, i.e., they can be completed in one round. The authors consider at first a setting with central scheduler and then show *how* and under what conditions it can be implemented in a message-passing distributed setting (called local scheduler).

One of their most intriguing results in [47] is that even with a central scheduler, no algorithm can be competitive if tasks have different execution times. *This*, is what has essentially motivated the work presented in Part I of the thesis. It also inspired the use of processor speed-scaling and the study of the conditions on speedup for which competitiveness is possible. As it turns out, extending the problem for tasks with different processing times and considering processor speedup was not trivial; different scheduling policies and techniques had to be devised. For instance, Theorem 5.4. in [47], shows unbounded competitiveness for any upper bound in the length of tasks $d \geq 3$. But could one guarantee bounded competitiveness if (s)he only has tasks of lengths 1 and 2? And is there a correlation between the difference of the task sizes and the amount of competitiveness achieved? These are only some of the questions the thesis hopes to answer in the rest of this Part.

2.2. Parallel online scheduling

The work presented in the thesis is highly related to studies of *parallel online scheduling* using identical machines [74]. Among them, several works consider speed-scaling and speedup issues. Some of them, unlike this work, consider *dynamic scaling* (e.g., [6, 17, 24]), meaning that they adapt the speed of the machines throughout the executions. Usually, in these works *preemption* is allowed. As mentioned before, this is the case that a task execution may be suspended and later restarted from the point of suspension, in the same or a different machine of the system. However, in the work done in this thesis, if a task execution is interrupted then it must be scheduled again at a later time instant and executed from scratch.

Greiner et al. [53] investigate scheduling on m identical speed-scaled processors without *migration*; tasks are not allowed to move among processors. Among other results, they prove that any z-competitive online algorithm for a single processor yields a zB_a-competitive online algorithm for multiple processors, where B_a is the number of partitions of a set of size a. However, unlike this work, they assume an unbounded number of processors available. Anand et al. [8] as well as Albers et al. [6], consider tasks with *deadlines* in real-time computing with migration. Observe that none of these works takes into account processor failures. Considering failures, as done in the thesis, makes parallel scheduling a significantly more challenging problem.

2.3. Load balancing and distributed online scheduling

The work of Awerbuch et al. [15] considered a distributed system of multiple interconnected machines, at which jobs arrived in an online but distributed manner. In other words, the jobs are requested at different machines, and each machine makes decisions on whether to execute one of the jobs it has or send some to other machines. What is more, the system is assumed to be completely decentralized, hence the machines need to communicate, not only to assign the jobs but also to have global knowledge for the state of the system and be able to make decisions. The authors use competitive analysis, considering average response time as their performance measure and show $O(\text{polylog}(n))$ competitiveness, where n is the number of machines.

2.4. Online bin packing

The work of the thesis is also related to the online version of the *bin packing* problem [90], where the objects to be packed are the tasks, and the bins are the time periods between two consecutive failures of the machine considered; i.e., *alive* intervals. Over the years, extensive research on this problem has been done, some of which can be considered related to the one discussed in this thesis. For example, Johnson et al. [59] 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 this work).

Epstein et al. [36] (see also [90]) considered online bin packing with resource augmentation in the size of the bins (corresponding to the length of alive intervals in this work). They also used an *asymptotic performance ratio* for the evaluation of the competitiveness of the algorithm they propose, as done in some cases of the thesis as well (see the definition of *long-term competitiveness* in Subsection 3.5.1 and the definition of *asymptotic throughput* in Subsection 7.1.5). Observe that the essential difference of the online bin packing problem with the one considered here, is that in the latter the bins and their sizes (corresponding to the machine's alive intervals) are unknown.

Chan et al. [25] looked into a more complex version of bin packing, which apart from being online it is also dynamic in the sense that the introduced items may also depart. In this setup, the authors also considered resource augmentation, showing that 1-competitiveness can only be achieved when doubling the size of the bins.

2.5. Job scheduling in the grid

Boyar and Ellen [22] have looked into a closely related problem, both to the online bin packing problem and to the one considered in the thesis. They considered job scheduling in the grid. It can be seen as a bin packing problem for a set of items given from the beginning, and bins of different sizes that arrive dynamically and have to eventually serve all the items in the set. One can see the correlation between their work and the one developed here, if (s)he relates the arrival of processors in the former with the length of periods that machines are alive in the latter. Nonetheless, the main difference with the setting considered here, is 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, whereas the results of the thesis depend only on the ratio of the costs of the largest and smallest tasks.

2.6. Packet scheduling in wireless networks

The second problem considered in this thesis is packet scheduling over an unreliable channel. Several studies have investigated the effect of jamming in wireless channels for throughput maximization. Two exhaustive surveys recommended to the reader include the work of Pelechrinis et al. [72] and the work of Dolev et al., [34]. In the former, the authors present a detailed survey of Denial of Service attacks. They explain the various techniques used to achieve malicious behaviors and describe methodologies for their detection as well as for the network's protection from the jamming attacks. In the latter, the authors present several existing results in adversarial interference environments in the unlicensed bands of the radio spectrum, discussing their vulnerability.

Furthermore, Gummandi et al. [54], consider 802.11 networks disrupted by radio frequency

interference and show that they are surprisingly vulnerable. In order to cope with these vulnerabilities they propose and analyze a channel hopping design. Thuente et al. [87], studied the effects of different jamming techniques in wireless networks and the trade-off with their energy efficiency. Their study includes from trivial/continuous to periodic and intelligent jamming (taking into consideration the size of packets being transmitted).

Emek et al. [35] consider an online set-packing optimization problem. In particular, they consider a network where large data frames are broken into a few packets and transmitted over the network. At each time-step only one packet can be transmitted, while all the others are lost. However, a data frame is only useful when all its packets are received. The authors do competitive analysis but unlike the thesis, they consider randomized distributed algorithms for their solutions.

In another work, Li et al. [64] considered a switch at which packets arrive and must be forwarded further in the network. Packets have weights and deadlines, and the goal of the online algorithm is to maximize the total weight of the transmitted packets. However, at each time step there can only be one packet sent, and the buffer of the switch has a bounded size. Hence, some packets must be dropped. The authors propose a ϕ-competitive algorithm, for instances that have "agreeable" deadlines.

Mao et al. [67] take a step further, considering multihop communication networks, and consider weights and deadlines. Their aim is to maximize the cumulative weights of the packets that reach their destinations within their deadlines. They first look at the tree topology and show that a well know algorithm (Earliest Deadline First) achieves optimal performance for any feasible arrival pattern. They also look at the problem in the general topology of multiple source-destination pairs and develop an algorithm that is $O(P_M \log P_M)$-competitive, where P_M is the maximum route length among all packets.

On a different flavor, Prabhakar et al. [75] focus on the energy-efficient transmissions over a wireless link, by considering schedules that adapt the transmission rate. Lowering the transmission power and hence transmitting a packet over a longer time interval, the energy to transmit the packet is significantly reduced. However, it is not practical to have arbitrarily long transmission times. Hence, the authors consider schedules that will minimize the energy subject to deadlines or some delay constraint. Among their results, they show an optimal offline schedule for a node with deadline constraints. They also present an online algorithm that varies the transmission times according to backlog, showing that it is more energy-efficient than a deterministic schedule that guarantees stability for the same arrival rates. Abbas El Gamal et al. [71] look at a very similar problem. They show that the offline energy-efficient transmission scheduling problem reduces to a convex optimization problem, but in general it does not have a closed-form solution when multiple users are considered. One of their results consists of an iterative algorithm for a downlink channel with one transmitter and multiple receivers, called MoveRight, that achieves the optimal offline schedule.

Andrews et al. [11] look at packet-switched networks with dynamic packet arrivals, focusing on a model where the average arrival rate of packets requiring the use of any edge is less than 1.

The authors study universal stability of protocols focusing on the maximum queue size necessary and maximum end-to-end delay experienced by any packet. One of their most important results is that there exists a distributed randomized protocol stable for all networks and requires both polynomial queue size and delay.

Note that all these works consider online packet scheduling, however without any jamming. Some examples of specific works more related to the work done in the thesis, are presented next.

2.6.1. Unconstrained jamming

Similar to the work presented in the thesis, Kesselheim [62] considers the packet scheduling problem in wireless networks, looking at both stochastic and adversarial arrivals. Unlike the work developed in the thesis though, it considers only *reliable* links. Its main objective is to achieve maximal throughput while guaranteeing *stability*; meaning bounded time from injection to delivery.

Andrews and Zhang [12] consider online packet scheduling over a wireless channel, where both the channel conditions and the data arrivals are governed by an adversary. Their main objective is to design scheduling algorithms for the base-station to achieve stability in terms of the size of queues of each mobile user. The work presented in the thesis does not focus on stability, but on the transmission of a maximum amount of data instead.

Richa et al. [83] consider the problem of devising local access control protocols for wireless networks with a single channel, that are provably robust against *adaptive adversarial jamming*. At certain time steps, the adversary can jam the communication in the channel in such a way that the wireless nodes do not receive messages. In the work presented in the thesis, this is not the case, since the receiver might receive a message, even if it contains bit errors. Although the model and the objectives of this line of work is different from the one presented here, it shares the same concept of studying the impact of adversarial behavior on network communication.

Another related work is the one of Anantharamu et al. [9], in which the authors explore the effect of adversarial jamming on broadcasting in multiple-access channels under dynamic packet arrivals. They constrain both the arrival and jamming processes and give upper bounds on worst-case latency of widely used protocols.

Tsibonis et al. [88] studied the scenario of scheduling transmissions to multiple users over a wireless channel with time-varying connectivity. Assuming saturated packet queues, they then proposed an algorithm based on the weighted sum of the throughput of the channel.

Finally, the work from Jurdzinski et al. [60] is a follow up work from one of the publications that contribute to this thesis. The authors propose optimal online algorithms for the case of k fixed packet lengths, matching the upper bounds on the asymptotic throughput defined for Part II of the thesis; the online packet scheduling problem. They also suggest a modification to one of the algorithm, in order to adapt it for the case of many independent channels, stating that the analysis is not trivial. However, this modified algorithm cannot be used in the system considered in the thesis – the multiple machines setting for task scheduling – because it uses central scheduling.

The sender in their setting always has updated information and full control of all the channels. In the multiple machines case studied in the thesis though, the algorithms used are parallel and the machines take independent decisions on the tasks they schedule.

2.6.2. Constrained jamming

Regarding the constrained jamming in the second part of the thesis there are some interesting related works as well.

Awerbuch et al. [16] designed a medium access (MAC) protocol for single-hop wireless networks that is robust against adaptive adversarial jamming (the adversary knows the protocol and its history and decides to jam the channel at any time) and requires only limited knowledge about the adversary (an estimate of the number of nodes, n, and an approximation of a time threshold T). One of the differences with the work done in the thesis, is that the adversary they consider is allowed to jam $(1-\varepsilon)$-fraction of the time steps. On a later work [80], Richa et al. explored the design of a robust MAC protocol that takes into consideration the signal to interference plus the noise ratio (SINR) at the receiver end. In [82] they extended their work to the case of multiple co-existing networks, proposing a randomized MAC protocol that guarantees fairness between the different networks and efficient use of the bandwidth. In [81], Richa et al. considered an adaptive adversarial jammer that is also reactive; one that is allowed to make a jamming decision based on the actions of the nodes at the current step. This, is similar to the adversary considered in this work. However, they consider a different constraint on jamming: given a time period of length T, the adversary can jam at most $(1-\varepsilon)T$ of the time steps in that period. In the case considered here, the adversary, within a time period T can cause f channel jams, where f does not correspond to a fraction of time, but on the number of packets it can corrupt. Another difference is that they consider n nodes transmitting over the channel and hence they have to deal with transmission collisions as well. What is more, their objective is to optimize throughput over the non-jammed time periods, whereas this work includes the whole execution.

Gilbert et al. [52] worked on a theoretical analysis of the damage that can be introduced by a tiny malicious entity having limited power in the communication delay between two nodes. In particular, the nodes share a time-slotted single-hop wireless ratio channel and the malicious entity wishes to delay their communication. However, it can only broadcast a message up to β times, which is similar to the restriction imposed in the thesis, but the model can be viewed as a generalization of this restriction by allowing recharging. Nonetheless, the setting and objectives of their work are different. They first show a bound on the number of rounds that the malicious node can delay the communication and then study its implication on an n-node general problem, such as reliable broadcast and leader election.

However, none of the models studied considers an AQT modeling of the power of the adversarial entity. Adversarial queuing has been used in wireless networks as a methodology for studying their stability under worst case scenarios, removing the stochastic assumptions usually made for the generation of traffic. It concerns the arrival process of packets in the system and

it has been introduced by Borodin et al. [21] as a well defined theoretical model since 2001. A variety of works has then followed, using AQT in different network settings, such as on multiple access channels [30, 31] and routing in communication networks [26, 27]. The constrained type of adversarial channel jams considered here, can be associated with the AQT model for the arrival process of packets in the following way. Classical AQT considers a *window adversary* that accounts packets being injected within a time window w in such a way that they give a total load of at most wr at each edge of the paths they need to follow, where $w \geq 1$ and $r \leq 1$. In the channel jams considered, for every window of duration $1/\rho$, there is exactly one new error token available for the adversary to use. In a long execution, considering for example a time interval $T > 1/\rho$, there will be up to $T\rho$ new error tokens available to the adversary.

Part I

Task Scheduling

Chapter 3

Problem Definition and General Model

As discussed in the Introduction, the main interest of the first part of the thesis lies on *online task scheduling* in multiprocessor computing systems prone to failures. Along with the increase in the number of demands for processing jobs of high computational power in such systems, the presence of machine failures is now becoming a norm instead of the exception.

The dynamic nature of the environment makes it an online one with several challenges. As also explained in Chapter 1, preserving the power consumption of the system is another challenge of rising importance. Hence, the main goal of this part of the thesis is to achieve reliable and stable computations in such environments, surpassing the several challenges they withhold and keeping the energy consumption to a minimum.

This chapter presents in detail the general model for the Online Task Scheduling problem. As already mentioned in the Introduction, it considers a fault-prone distributed system that has to cope with tasks of different computational demands (or sizes). What is more, the use of resource augmentation is suggested, in the form of processor speedup, in order to achieve efficiency. Hence, the model is characterized by these three parameters: the *number of machines*, m, the *amount of speedup*, s, and the *number of different task sizes*, k. The model is denoted by $M\langle m, s, k\rangle$.

3.1. Computing setting

Consider a system of m homogeneous, fault-prone machines, with unique ids from the set $[m] = \{1, 2, \ldots, m\}$, and assume that they have access to a shared object, called *Shared Repository* (or *Repository* for short). Assume also, that the machines are *identical* (homogeneous), meaning that a task computation on any of them consumes equal or comparable local resources.

A form of *resource augmentation* is also considered, by speeding up the machines; denoting the *speedup* by $s \geq 1$. A *global, static* speedup is assumed, meaning that it is the same for every machine in the system, and its value is either set at the beginning of an execution or is simply given by the system, i.e., it is known from the machine's specifications.

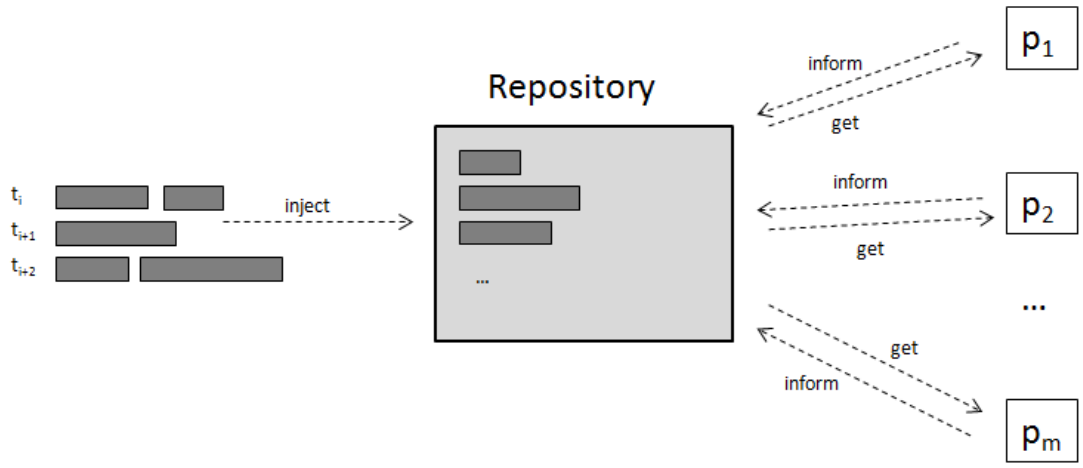

Figure 3.1: The computational setting, with the m homogeneous machines, the shared Repository, and the three operations; *inject*, for the dynamic task arrivals from the clients, *get*, for the machines to obtain the set of pending tasks, and *inform* for the repository to update the set of pending tasks.

3.1.1. Repository

The Repository is a shared object that represents the interface of the system between the clients and the machines[1]. It is used by the clients when submitting jobs (or *computational tasks*) and when receiving the notifications about the ones that have been completed. The machines use it when reading the pending tasks and when announcing the one they have just executed. Its data type is hence a *set of tasks* (full details are given below) and supports three operations: *inject, get,* and *inform*.

- The *inject* operation is executed by the clients of the system, when adding computational tasks to the current set in the Repository. This operation is controlled by an *adversary* (full details are discussed below), whereas the other two operations are executed by the system's machines.
- By executing a *get* operation, a machine obtains the set of *pending tasks*, $\mathbf{P}$, from the Repository; it includes the tasks that have been injected into the system, but the Repository has not been notified of their completion yet. To simplify the model, if there are no pending tasks when the *get* operation is executed, i.e., $\mathbf{P} = \emptyset$. The operation blocks until some new task is injected, and then it immediately returns the set of new tasks.
- Upon computing a task, say τ, a machine executes an *inform* operation, which notifies the Repository about the task completion. Then the Repository removes this task from the set of pending tasks, i.e., $\mathbf{P} = \mathbf{P} \backslash \{\tau\}$.

[1] One can also relate the repository with the shared memory in parallel systems [14, 66, 79].

It is important to note, that the Repository (unlike the machines) is *reliable*; it does not fail. The computational setting, along with the three operations for the Repository are shown in Figure 3.1. Note that each operation performed by a machine is associated with a point in time (with the exception of the *get* operations that block) and the outcome of the operation is *instantaneous*. Note also, that due to the machine crashes, it would not be helpful for a machine to notify the repository when scheduling a task before it has actually executed it completely. This will be made more clear by the end of the chapter.

3.2. Machine processing

The system's machines run in *real-time cycles*, following an algorithm. Each cycle consists of a *get* operation, a computation of a task, and an *inform* operation (if the task is completed). Between two consecutive cycles an algorithm may choose to have a machine idling for a period of predefined length. It is assumed that the *get* and *inform* operations consume negligible time, unless *get* finds no pending tasks, in which case it blocks but returns immediately as soon as a new task is injected. The computation part of the cycle, which involves executing a task, consumes the time needed for the specific task to be computed, divided by the *speedup* $s \geq 1$. It is thus the part that defines the processing time of a task. What is more, processing cycles may not always complete; an algorithm may decide to break the current cycle of a machine at any moment and start a new one. In a similar way, a machine crash breaks forcefully and instantaneously the current processing cycle of a machine and when the machine restarts a new cycle begins. Note that when a processing cycle is interrupted, either by the algorithm's decision or by a machine crash, the progress in the task execution being processed is lost.

Due to the concurrent nature of the computing system considered, machine's processing cycles may be asynchronous with respect to the ones of the other machines, thus may overlap not only between themselves but also with the clients' inject operations. The following *event ordering* is therefore defined for the repository at a time instant t: first, the *inform* operations executed by machines are processed, then the *inject* operations, and last the *get* operations of machines. This, implies that the set of pending tasks returned by a *get* operation executed at time t, includes not only the older uncompleted tasks but also the tasks injected at time t, and excludes the tasks reported as completed at time t. Note that this event ordering is done only for the ease of presentation and reasoning; it does not affect the generality of results.

3.3. Adversarial entity

An *omniscient* and *adaptive* adversary is assumed, that controls both the task injections and the machine failures (crashes and restarts). In particular, it defines two event sequences in order to create worst-case scenarios; an *arrival pattern* A, and an *error pattern* E.

3.3.1. Task arrivals

Tasks are injected to the system by the clients as established by the adversarial arrival pattern A. Each task τ is associated with an *arrival time*, $a(\tau)$ (the time it was injected in the system based on the repository's clock) and a *size*, $\pi(\tau)$, being the processing time required for its completion by a machine running without speedup, i.e., $s = 1$. Note, that simultaneous task arrivals are totally ordered, and that both the arrival time and the size of each task get to be known at arrival. It is important to note here, that unless otherwise stated, tasks are considered to be ordered in an ascending order in the repository, according to their arrival time and size, to break ties. However, algorithms may decide to sort them differently for their purposes. The term π-task is used to refer to a task of size $\pi \in [\pi_{min}, \pi_{max}]$, where π_{min} and π_{max} denote the smallest and largest task size. Unless otherwise stated, the values π_{min} and π_{max} are known to the machines.

What is more, it is assumed that tasks are *atomic* with respect to their completion; in other words, preemption is not allowed. Tasks must be fully executed without interruptions, otherwise, if a machine stops the computation cycle (intentionally or due to a crash), neither any partial information can be shared with the repository, nor the machine may resume the execution of the task from the point it stopped. Note also, that if a machine executes a task but crashes before the *inform* operation, then this task is not considered completed. Moreover, tasks are assumed to be *independent* and *idempotent*, meaning that their completion will not affect any other task and that multiple executions of the same task will produce the same final result, respectively. Several applications involving tasks with such properties are discussed in [49].

3.3.2. Machine crashes and restarts

The machines of the system are prone to failures, that are defined by the adversarial error pattern E, as a collection of crash and restart events, each being associated with the time it occurs; i.e., $crash(t, i)$ specifies that machine i is crashed at time t, and $restart(t, i)$ that machine i is restarted at time t. A machine i is also referred to being *active* within time interval $T = [t, t']$, if it is operational at time t and does not crash by time t'. Hence, an error pattern can also be defined as a sequence of active intervals for all the machines of the system. Upon a restart, it is assumed that the machine has knowledge only of the algorithm being executed, its id i and its speedup s, as well as parameter m (the number of machines). Thus, it simply starts a new processing cycle.

3.4. Global notation

The notation already defined is summarized here, along with some new notation that will be extensively used throughout the Task Scheduling part of the thesis. Table 3.1 is provided for the help of the reader; in case (s)he needs a fast reminder of some notation or terminology.

First, recall the three basic model parameters, m, s and k, being the number of machines, the amount of processing speedup, and the number of different task sizes considered. Recall that a

Term	Description
$m \in \mathbb{N}^+$	Number of machines in the system
$s \geq 1$	Speedup of machines
$k \in \mathbb{N}^+$	Number of different task sizes considered
π_{min}, π_{max}	Smallest & largest task sizes
π-task	Task of size $\pi \in [\pi_{min}, \pi_{max}]$
$\rho = \pi_{max}/\pi_{min}$	Ratio of task sizes
$\bar{\rho}$	$\lfloor \rho \rfloor$
$\hat{\rho}$	$\lceil \rho \rceil - 1$
A, E	Adversarial arrival and error patterns
$\mathbf{I}_t, \mathbf{P}_t, \mathbf{C}_t$	Sets of injected, pending and completed tasks at time t
$\#I_t, \#P_t, \#C_t$	Cardinality of corresponding sets
I_t, P_t, C_t	Total load of corresponding sets; sum of sizes of all elements
L_t	Latency; maximum time a task has spent in $\mathbf{P}$ up to time t
$\mathcal{C}, \mathcal{P}, \mathcal{L}$	Competitiveness rations: Completed-load, Pending-load and Latency
$\gamma = \max\{\lceil \frac{\rho - s}{s-1} \rceil, 0\}$	Parameter γ used to define competitiveness thresholds
Condition **C1**	$s < \rho$
Condition **C2**	$s < 1 + \gamma/\rho$

Table 3.1: Important notation and definitions.

task may have size $\pi \in [\pi_{min}, \pi_{max}]$, where π_{min} and π_{max} denote the smallest and largest task sizes respectively. Let parameter $\rho = \frac{\pi_{max}}{\pi_{min}}$ be defined as the ratio between these task sizes. Let also, $\bar{\rho}$ and $\hat{\rho}$ be defined as $\bar{\rho} = \lfloor \rho \rfloor$ and $\hat{\rho} = \lceil \rho \rceil - 1$; these parameters are extensively used throughout the thesis.

Because it is essential to keep track of injected, completed and pending tasks at each time instant in an execution, let sets $\mathbf{I}_t(\mathsf{A})$, $\mathbf{C}_t^s(X, \mathsf{A}, \mathsf{E})$ and $\mathbf{P}_t^s(X, \mathsf{A}, \mathsf{E})$ be defined, where X is an algorithm, A and E the arrival and error patterns respectively (as defined above), t the time instant considered and s the speedup of the machines. Set $\mathbf{I}_t(\mathsf{A})$ represents the set of injected tasks from the start of the execution up to time t inclusively, set $\mathbf{C}_t^s(X, \mathsf{A}, \mathsf{E})$ is the set of completed tasks up to time t inclusively, and set $\mathbf{P}_t^s(X, \mathsf{A}, \mathsf{E})$ the set of pending tasks in the Repository *at* time instant t. Interval notation is also used, for example time interval T instead of time instant t, as a subscript, in the cases where a specific time interval is considered, i.e., $\mathbf{I}_T(\mathsf{A})$, $\mathbf{C}_T^s(X, \mathsf{A}, \mathsf{E})$ and $\mathbf{P}_T^s(X, \mathsf{A}, \mathsf{E})$. Observe that $\mathbf{I}_t(\mathsf{A}) = \mathbf{C}_t^s(X, \mathsf{A}, \mathsf{E}) \cup \mathbf{P}_t^s(X, \mathsf{A}, \mathsf{E})$. Hence, set $\mathbf{P}_t^s(X, \mathsf{A}, \mathsf{E})$ contains the tasks that were injected by time t inclusively, but not the ones completed before and up to time t. Note also, that set $\mathbf{I}$ depends only on the arrival pattern A, while sets $\mathbf{C}$ and $\mathbf{P}$ also depend on the error pattern E, the algorithm run by the machines, X, and the speedup of the machines, s.

What is more, in order to refer directly to the number of tasks in each set, notation $\#I_t(\mathsf{A})$, $\#P_t^s(X, \mathsf{A}, \mathsf{E})$, and $\#C_t^s(X, \mathsf{A}, \mathsf{E})$ is used, for the total amount of injected, pending and completed tasks respectively. By $I_t(\mathsf{A})$, the sum of sizes of all tasks in $\mathbf{I}_t(\mathsf{A})$ is denoted, corresponding to the *total injected load* by time t. The notation for the *completed and pending load*, along

with *latency* is given in the next Section, where all the efficiency measures are clearly defined. Also, parameter γ along with conditions **C1** and **C2** will be defined clearly in Chapter 5. They are shown here, so that the reader can find everything together, but there is no need to get into further details in this stage.

Finally, for the simplicity of presentation, the superscript s as well as the arrival and error patterns A and E, are sometimes omitted in later Chapters of the thesis. However, the appropriate speedup, as well as the adversarial behavior, is clearly stated in each case.

3.5. Efficiency measures

Consider an online algorithm ALG running with speedup s under arrival and error patterns A and E respectively. Then look at a time instant t of an execution and focus on three measures: the *Completed Load*, which is the sum of sizes of the completed tasks

$$C_t^s(\text{ALG}, \mathsf{A}, \mathsf{E}) = \sum_{\tau \in \mathbf{C}_t^s(\text{ALG},\mathsf{A},\mathsf{E})} \pi(\tau),$$

the *Pending Load*, which is the sum of sizes of the pending tasks

$$P_t^s(\text{ALG}, \mathsf{A}, \mathsf{E}) = \sum_{\tau \in \mathbf{P}_t^s(\text{ALG},\mathsf{A},\mathsf{E})} \pi(\tau)$$

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

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

where $f(\tau)$ is the time of completion of task τ.

Observe that $C_t^s(\text{ALG}, \mathsf{A}, \mathsf{E}) + P_t^s(\text{ALG}, \mathsf{A}, \mathsf{E}) = I_t(\mathsf{A})$. Nonetheless, computing the schedule – and hence finding the algorithm – that maximizes or minimizes accordingly the measures $C_t^s(X, \mathsf{A}, \mathsf{E})$ and $P_t^s(X, \mathsf{A}, \mathsf{E})$ offline, (having the knowledge of the patterns A and E), is an NP-hard problem. For the details, see Chapter 4.

3.5.1. Competitive analysis

Due to the dynamicity of the task arrivals and machine failures, the scheduling of tasks is viewed as an online problem and *competitive analysis* is pursued using the three metrics defined above. For each metric, 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

$$C_t^s(\text{ALG}, \mathsf{A}, \mathsf{E}) \geq \alpha \cdot C_t^1(X, \mathsf{A}, \mathsf{E}) + \Delta_C$$

holds $\forall t, X, \mathsf{A}, \mathsf{E}$, and for some parameter Δ_C that *does not* depend on t, X, A or E. The *completed-load competitive ratio* of ALG is denoted by $\mathcal{C}(\text{ALG}) = \alpha$.

Similarly, it is considered α-*pending-load-competitive* if

$$P_t^s(\text{ALG}, \mathsf{A}, \mathsf{E}) \leq \alpha \cdot P_t^1(X, \mathsf{A}, \mathsf{E}) + \Delta_P$$

holds $\forall t, X, \mathsf{A}, \mathsf{E}$, and for parameter Δ_P which does not depend on t, X, A or E. In this case, the *pending-load competitive ratio* of ALG is denoted by $\mathcal{P}(\text{ALG}) = \alpha$.

Finally, algorithm ALG is considered α-*latency-competitive* if

$$L_t^s(\text{ALG}, \mathsf{A}, \mathsf{E}) \leq \alpha \cdot L_t^1(X, \mathsf{A}, \mathsf{E}) + \Delta_L$$

holds $\forall t, X, \mathsf{A}, \mathsf{E}$, where Δ_L is a parameter independent of t, X, A and E. In this case, the *latency competitive ratio* of ALG is denoted by $\mathcal{L}(\text{ALG}) = \alpha$.

It is important to note, that α is independent of t, X, A and E, for the three metrics accordingly. Nonetheless, parameters $\Delta_C, \Delta_P, \Delta_L$ as well as α may depend on system parameters like π_{min}, π_{max}, m or s, which are **not** considered inputs of the problem; they are all fixed and given upfront. Note, that the number of machines m is *fixed* for a given execution, and that the algorithm used may take it into consideration, hence different m may result to a different performance of the same algorithm; this however affects only the additive term of the competitiveness.

Observation 3.1. *Completed and Pending Load competitiveness measures are* not complementary *to one another.*

An algorithm may be completed-load competitive but not pending-load competitive, even though the total sum of sizes of the successfully completed tasks complements the sum of sizes of the pending ones (total load). It is essential to understand the importance of both measures, and in order to make the observation clear, the following example is given.

EXAMPLE 3.1. *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 $\mathcal{C}(ALG) = 1/2$. 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.*

3.5.2. Long-term competitiveness

Finally, in some cases, competitiveness is allowed to hold beyond a certain time instant t in the execution. In particular, when instantaneous competitiveness, as presented before in its classical use, is difficult to be achieved, the analysis also looks at the *long-term competitiveness* of the algorithms, considering their competitive ratio as time goes to infinity. The long-term competitive ratios of the three measures are then defined as follows:

- Completed load: $\mathcal{C}(\text{ALG}) = \lim_{t\to\infty} \inf_{\forall \mathsf{A},\forall \mathsf{E}} \frac{C_t^s(\text{ALG},\mathsf{A},\mathsf{E})}{C_t^1(X,\mathsf{A},\mathsf{E})}$,
- Pending load: $\mathcal{P}(\text{ALG}) = \lim_{t\to\infty} \sup_{\forall \mathsf{A},\forall \mathsf{E}} \frac{P_t^s(\text{ALG},\mathsf{A},\mathsf{E})}{P_t^1(X,\mathsf{A},\mathsf{E})}$, and
- Latency: $\mathcal{L}(\text{ALG}) = \lim_{t\to\infty} \sup_{\forall \mathsf{A},\forall \mathsf{E}} \frac{L_t^s(\text{ALG},\mathsf{A},\mathsf{E})}{L_t^1(X,\mathsf{A},\mathsf{E})}$,

where the *infimum* and *supremum* of the ratios is taken over all possible combinations of adversarial patterns. They must however allow infinite executions, where any algorithm X will have unbounded completed tasks with time going to infinity, i.e., $\lim_{t\to\infty} \#C_t^s(X,\mathsf{A},\mathsf{E}) = \infty$. For this to be true, it must also be the case that the task arrivals grow infinitely as time goes to infinity, i.e., $\lim_{t\to\infty} \#I_t(\mathsf{A}) = \infty$.

The difference with the classical competitiveness measures, basically lies in the additive term Δ of the formula shown earlier. In the case of long-term competitiveness, this additive term may depend on time. Observe, that the final long-term competitive ratio is taken as the limit of the instantaneous ratio as time goes to infinity.

Chapter 4

NP-hardness

This Chapter, shows that the offline problem of optimally scheduling tasks in order to maximize the completed load or minimize the pending load or latency, is **NP**-hard. In fact, **NP**-hardness is shown for problems with only one single machine. This implies the **NP**-hardness of the problem with more machines as well, since the adversary could crash all but one machine.

The thesis focuses more in the online version of the problem, which will be analyzed with more details in the next chapters. Nonetheless, this analysis justifies even more the approach used in the thesis, using speedup on the machines for the online problems.

4.1. Completed load

Let the Completed Load Problem be defined as follows:

INSTANCE: Let X be a set of tasks, for each task $x \in X$ a size $\pi(x) \in \mathbb{N}^+$, an arrival time $a(x) \in \mathbb{Z}^0$, a sequence of time instants $0 = T_0 < T_1 < T_2 < \cdots < T_k$, $T_i \in \mathbb{N}^0$, so that the machine is crashed and restarted instantaneously at each time T_i, $i \in [1, k]$ (in other words, at each time T_i, any task being processed by the machine is not completed).

QUESTION: is there a schedule of X so that tasks of total load T_k are completed successfully by time T_k by the machine?

Theorem 4.1. *The Completed Load Problem is* **NP***-hard.*

Proof: The proof uses the 3-Partition problem, which is known to be an **NP**-hard problem:

INSTANCE: Let A be a set of $3m$ elements, a bound $B \in \mathbb{N}^+$ and, for each $a \in A$, a size $s(a) \in \mathbb{N}^+$ such that $B/4 < s(a) < B/2$ and $\sum_{a \in A} s(a) = mB$.

QUESTION: can A be partitioned into m disjoint sets $\{A_1, A_2, \ldots, A_m\}$ such that, for each $1 \leq i \leq m$, $\sum_{a \in A_i} s(a) = B$?

A reduction of the 3-Partition problem to the Completed Load Problem, defined for a single machine above, is as follows: Set $X = A$, $\pi() = s()$, $a() = 0$, $k = m$, and $T_i = iB$ for

$i \in [1, k]$. If the answer to 3-Partition is affirmative, then for the Completed Load Problem there is a way to schedule (and complete successfully) the tasks in X in subsets $\{X_1, X_2, \ldots, X_m\} = \{A_1, A_2, \ldots, A_m\}$, so that all the tasks in A_i can be completed by the machine in the interval $[T_{i-1}, T_i]$. Furthermore, since $\sum_{a \in A_i} s(a) = \sum_{x \in X_i} \pi(x) = B$, and $T_i - T_{i-1} = B$, the total length of packets transmitted by time T_k is T_k.

The reverse argument is similar. If there is a way to schedule tasks so that the total completed load by time T_k is T_k, in each interval between two error events of the machine there must be exactly B total load of tasks completed. Then, the tasks can be partitioned into subsets of total length B each, which would imply the partition of A. ■

4.2. Pending load

Consider now $P_SCHED(t, \mathsf{A}, \mathsf{E})$, the problem of scheduling tasks so that the pending load at time t, under adversarial arrival and error patterns A and E, is minimized. Let a decision version of the problem be $DEC_P_SCHED(t, \mathsf{A}, \mathsf{E}, \omega)$, with an additional input parameter ω. An algorithm solving the decision problem outputs a Boolean value $TRUE$ if and only if there is a schedule that achieves pending load no more than ω at time t under adversarial arrival and error patterns A and E. I.e., $DEC_P_SCHED(t, \mathsf{A}, \mathsf{E}, \omega)$ outputs $TRUE$ if and only if $\mathcal{P}_t(\text{OPT}, \mathsf{A}, \mathsf{E}) \leq \omega$.

Theorem 4.2. *The problem $DEC_P_SCHED(t, \mathsf{A}, \mathsf{E}, \omega)$ is* **NP***-hard.*

Proof: The reduction used is from the Partition problem. The input considered is a set of positive numbers $C = \{x_1, x_2, ..., x_k\}$, $k > 1$. The problem is to decide whether then, there is a subset $C' \subset C$ such that $\sum_{x_i \in C'} x_i = \frac{1}{2} \sum_{x_i \in C} x_i$. The Partition problem is know to be **NP**-complete.

Consider any instance I_p of Partition. Construct an instance I_d of $DEC_P_SCHED(t, \mathsf{A}, \mathsf{E}, \omega)$ as follows. The time t is set to $1 + \sum_{x_i \in C} x_i$. The adversarial arrival pattern A injects a set S of k tasks at time 0, so that the ith task has size x_i. The adversarial error pattern E starts the machine at time 0 and crashes it at time $\frac{1}{2} \sum_{x_i \in C} x_i$. Then, it restarts the machine immediately and crashes it again at time $\sum_{x_i \in C} x_i$. The machine does not restart until time t. Finally, the parameter ω is set to 0.

Assume there is an algorithm ALG that solves DEC_P_SCHED. It is shown that ALG can be used to solve the instance I_p of Partition by solving the instance I_d of DEC_P_SCHED obtained as described. If there is a $C' \subset C$ such that $\sum_{x_i \in C'} x_i = \frac{1}{2} \sum_{x_i \in C} x_i$, then there is an algorithm that is able to schedule tasks from S so that the two semi-periods (of length $\frac{1}{2} \sum_{x_i \in C} x_i$ each) the machine is active, it is doing useful work. In that case, the pending load at time t will be $0 = \omega$. If, on the other hand, such subset does not exist, some of the time the machine is active will be wasted, and the load pending at time t has to be larger than ω. ■

4.3. Latency

Now, similar to the pending load case, let $L_SCHED(t, \mathsf{A}, \mathsf{E})$ be the corresponding scheduling problem so that the maximum latency of all pending tasks at time t, under adversarial arrival and error patterns A and E, is minimized. Let a decision version of the problem be $DEC_L_SCHED(t, \mathsf{A}, \mathsf{E}, l)$, with additional input parameter l. An algorithm solving the decision problem outputs a Boolean value $TRUE$ if and only if $\mathcal{L}_t(\text{OPT}, \mathsf{A}, \mathsf{E}) \leq l$.

Theorem 4.3. *The problem $DEC_L_SCHED(t, \mathsf{A}, \mathsf{E}, l)$ is* **NP***-hard.*

Proof: The same reduction used for the pending load is used here; from the Partition problem, which is known to be **NP**-complete. The input considered is a set of positive numbers $C = \{x_1, x_2, ..., x_k\}$, $k > 1$. The problem is to decide whether there is a subset $C' \subset C$ such that $\sum_{x_i \in C'} x_i = \frac{1}{2} \sum_{x_i \in C} x_i$.

Consider any instance I_p of Partition and construct an instance I_d of $DEC_L_SCHED(t, \mathsf{A}, \mathsf{E}, l)$ as follows: The time t is set to $1 + \sum_{x_i \in C} x_i$. The adversarial arrival pattern A injects a set S of k tasks at time 0, so that the ith task has size x_i. The adversarial error pattern E starts the machine at time 0 and crashes it at time $\frac{1}{2} \sum_{x_i \in C} x_i$. Then, it restarts the machine immediately and crashes it again at time $\sum_{x_i \in C} x_i$, after which it does not restart until time t. Finally, the parameter l is set to $\sum_{x_i \in C} x_i$.

Assume there actually is an algorithm ALG that solves DEC_L_SCHED. This means that ALG can be used to solve the instance I_p of Partition by solving the instance I_d of DEC_L_SCHED obtained as described. If there is a $C' \subset C$ such that $\sum_{x_i \in C'} x_i = \frac{1}{2} \sum_{x_i \in C} x_i$, then there is an algorithm that is able to schedule tasks from S so that the two semi-periods (of length $\frac{1}{2} \sum_{x_i \in C} x_i$ each) the machine is active, it is doing useful work. In that case, no task will be pending at time t and hence the latency will be $\sum_{x_i \in C} x_i$, which is equal to l. If, on the other hand, such subset does not exist, some of the time the machine is active will be wasted, and there will be some task pending at time t, which also means that latency will be larger than l. ■

Finally, observe that for the off-line version of minimizing the maximum latency there have been some positive results, however considering different models. In fact, Bender et al. [18] claim that a popular scheduling algorithm solves an off-line version of the problem optimally in polynomial time. However, for that result the authors consider a fixed set of tasks with various arrival and processing times, known from the beginning, and claim that *First In First Out* (FIFO) policy is optimal in the case of a single machine. Although this is true, it is important to observe the difference in the off-line problem definition, which is what justifies the different result.

Chapter 5

Single Machine

This chapter completes a thorough analysis of the basic model of a single machine (i.e., uniprocessor) prone to failures. As described in the model, both task arrivals and machine crashes and restarts are controlled by an adversarial entity, thus giving worst-case scenarios for the analysis.

It starts by showing that with no speedup, i.e., $s = 1$, work-conserving and deterministic scheduling algorithms cannot achieve much. However, it also depends on the number of different task sizes considered; as soon as only two different task sizes are considered, π_{min} and π_{max}, it will be shown that there is an upper bound of $\frac{\bar{\rho}}{\bar{\rho}+\rho} \leq \frac{1}{2}$ for long-term completed-load competitiveness. The aim is therefore to find an *optimal* algorithm; one that can guarantees *that* competitive ratio.

The performance of some of the most widely-used scheduling algorithms is also studied here, for different amounts of speedup, $s \geq 1$, including their advantages and limitations over the rest. Other new algorithms are finally proposed; ones that reach the lower or upper bounds accordingly, of the different efficiency measures shown. The main goal is to reach the best competitive ratios with the least amount of speedup possible; as already described in Chapter 1, the amount of speedup affects the power consumption and thus the goal is to keep it to a minimum.

5.1. Properties of *ALL* work-conserving and deterministic algorithms

This section focuses on the general properties of all work-conserving and deterministic algorithms that tackle the task scheduling problem considered. Obviously, these properties apply to the specific scheduling policies considered in the rest of the Task Scheduling part of the thesis.

The *negative results*; that is, the upper bounds for the completed-load competitiveness and the lower bounds for the pending-load and latency competitiveness, presented using the model of a single machine, also hold for any number of machines, i.e., $m \geq 1$. On the contrary, the *positive results*; that is, the lower bounds for the complete-load competitiveness and the upper bounds for the pending-load and latency competitiveness, presented using the model of a single machine,

Alg.	Model	Completed Load, $\mathcal{C}$	Pending Load, $\mathcal{P}$	Latency, $\mathcal{L}$	Thm.
LIS	$M\langle 1,1,1\rangle$	1	1	1	Prop. 5.1, 5.2
ALG$_W$	$M\langle 1,1,1\rangle$	1	1	–	Prop. 5.1
	$M\langle 1,1,\infty\rangle$	0	∞	∞	5.1, 5.11
	$M\langle 1,1,2\rangle$	$\leq \frac{\bar{\rho}}{\rho+\bar{\rho}}$	∞	∞	5.3, 5.11
	$M\langle 1,s\geq\rho,\infty\rangle$	$[1/\rho,1]$	$[1,\rho]$	–	5.4, 5.5
	$M\langle 1,s\geq 1+\rho,\infty\rangle$	1	1	–	5.6
ALG$_D$	$M\langle 1,1,\infty\rangle$	0	∞	∞	5.2, 5.11
	$M\langle 1,1,2\rangle$	$\leq \frac{\bar{\rho}}{\rho+\bar{\rho}}$	∞	∞	5.3, 5.11
	$M\langle 1,s<\min\{\rho,1+\gamma/\rho\},\infty\rangle$	<1	∞	∞	5.9, 5.10, 5.11

Table 5.1: Metric comparison for all work-conserving (ALG$_W$) or deterministic (ALG$_D$) scheduling algorithms for different ranges of speedup. The last column provides the theorem numbers where the results of the corresponding row can be found. Recall 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. Note that for the latency there are no general results for speedup at least ρ.

may not hold in the general case of, i.e., for $m > 1$. For this reason, additional analysis is done in Chapter 6 to complete the results regarding *any* work-conserving or deterministic algorithm in the case of multiple machines.

Table 5.1 summarizes the results of this section, along with the results of Section 5.3.1. They include the general properties of all deterministic and/or work-conserving scheduling algorithms, running with different ranges of speedup.

5.1.1. Properties of $M\langle 1,1,1\rangle$

The following propositions are true for the case of uniform task sizes.

Proposition 5.1. *Any work-conserving scheduling algorithm ALG$_W$, running on a single machine with no speedup and with all tasks having the same size, i.e., $M\langle 1,1,1\rangle$, has optimal completed and pending load competitiveness, of ratio* 1.

Proof: Consider a work-conserving algorithm ALG$_W$. As long as there are pending tasks available, it schedules them. When the machine is crashed, it simply re-schedules the same task (or schedules another one) as soon as the machine is restarted. Observe that since all tasks have the same size, it doesn't make a difference on the completed load, *which* task will be scheduled. It is then trivial to see that the prior knowledge that an offline optimal algorithm has, does not help in completing more tasks than ALG. Then the pending load is directly optimal as well, since under no other schedule it could have been less. ■

Proposition 5.2. *Scheduling algorithm LIS, running on a single machine with no speedup and with all tasks having the same size, i.e.,* $M\langle 1,1,1\rangle$*, is* 1*-latency competitive.*

Proof: Consider work-conserving algorithm LIS. As long as there are pending tasks available, it schedules the task at the top of the queue. Note that tasks are ordered by their arrival time, with the earliest one being at the top of the queue. When the machine is crashed and later restarted, it simply re-schedules the same task. Since all tasks are of the same size, they take the same time to be completed. Therefore, the only important factor for the latency is the arrival time of the tasks pending at any time instant.

Any prior knowledge that the offline optimal algorithm has (all task arrival times) do not help in completing tasks that would decrease the latency more than LIS. If it does not schedule the task being at the top of the queue, its latency will only increase. Hence, it is not possible to decrease the latency more than LIS, which means that its latency competitiveness is optimal. ■

The above propositions regarding uniform task sizes, justify further the interest and decision to focus on tasks of different task sizes.

5.1.2. Properties of $M\langle 1,1,\infty\rangle$

Now, the analysis of the case of no speedup and an arbitrary number of task sizes follows.

Theorem 5.1. *If tasks can have any size in the range* $[\pi_{min},\pi_{max}]$ *and the machine has no speedup, i.e.,* $M\langle 1,1,\infty\rangle$*, no work-conserving algorithm ALG is competitive with respect to completed load, i.e.,* $\mathcal{C}(ALG_W)=0$ *or pending load, i.e.,* $\mathcal{P}(ALG_W)=\infty$*.*

Proof: Assuming $s=1$, consider the following scenario as a result of adversarial arrival and error patterns A and E respectively. Let some $\varepsilon\in(0,1)$ and $\Delta(k)=(\pi_{max}-\pi_{min})\varepsilon^k$. Then, let τ_k be a task with size $\pi(\tau_k)=\pi_{min}+\Delta(k)$, for all $k=0,1,2,3\ldots$. Observe that $\forall k,\pi(\tau_k)\in(\pi_{min},\pi_{max}]$ and $\pi(\tau_{k+1})<\pi(\tau_k)$. Let now time points t_k, such that $t_0=0$ (the beginning of the execution) and $t_{k+1}=t_k+\pi_{min}+\frac{1+\varepsilon}{2}\Delta(k)$. Let also time points $t'_k=t_{k-1}+\frac{1-\varepsilon}{2}\Delta(k-1)$. The arrival pattern A is such that task $\tau_0=\pi_{max}$ is injected in the system at time instant t_0. Then, for $k=1,2,\ldots$ task τ_k is injected at time t'_k. The error pattern E is such that at every time instant t_k there is an immediate crash and restart.

For the analysis, compare all work-conserving algorithms ALG with any algorithm X. In the execution of ALG, task τ_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 τ_1 and schedules it immediately. When the processor crashes at time t_1 the task τ_0 executed by ALG is interrupted, since $t_1-t_0=\pi_{min}+\frac{1+\varepsilon}{2}\Delta(0)\pi(\tau_0)=\pi_{min}+\Delta(0)$. However, X is able to complete task τ_1 because $t'_1+\pi(\tau_1)=t_0+\frac{1-\varepsilon}{2}\Delta(0)+\pi_{min}+\Delta(1)=t_0+\pi_{min}+\frac{1+\varepsilon}{2}\Delta(0)=t_1$. After the restart at t_1, ALG schedules one of the pending tasks $\{\tau_0,\tau_1\}$, while X waits until t'_2 to schedule the next task to be injected, τ_2.

The general process is as follows. At time instant t_k, ALG schedules one of the pending tasks in $\{\tau_0, \tau_1, \dots \tau_k\}$, while X waits until the next task τ_{k+1} is injected at time t'_{k+1} and schedules it. When the machine crashes at time t_{k+1} the task scheduled by ALG is interrupted, since $t_{k+1} - t_k = \pi_{min} + \frac{1+\varepsilon}{2}\Delta(k) < \pi(\tau_k) = \pi_{min} + \Delta(k)$ and all possible tasks scheduled by ALG are at least $\pi(\tau_k)$ long. However, X is able to complete task τ_{k+1} because $t'_{k+1} + \pi(\tau_{k+1}) = t_k + \frac{1-\varepsilon}{2}\Delta(k) + \pi_{min} + \Delta(k+1) = t_k + \pi_{min} + \frac{1+\varepsilon}{2}\Delta(k) = t_{k+1}$.

Letting this adversarial behavior run to infinity, see that at any point in time t, $C_t(\text{ALG}) = 0$, while X will keep completing the injected tasks. Observe also that the pending load of ALG increases with every phase, while the pending load of X is constant, i.e., the only pending task is the first π_{max}-task injected. This, results to a completed-load competitive ratio $\mathcal{C}(\text{ALG}) = 0$ and a pending-load competitive ratio $\mathcal{P}(\text{ALG}) = \infty$. ■

Theorem 5.2. *If tasks can have any size in the range $[\pi_{min}, \pi_{max}]$ and the machine has no speedup, i.e., $M\langle 1, 1, \infty\rangle$, no deterministic algorithm ALG is competitive with respect to completed load, i.e., $\mathcal{C}(ALG_D) = 0$ or pending load, i.e., $\mathcal{P}(ALG_D) = 0$.*

Proof: Assuming $s = 1$, consider the following scenario as a result of adversarial arrival and error patterns A and E respectively. Fix some $\varepsilon \in (0, 1)$ and let $\Delta(k) = (\pi_{max} - \pi_{min})\varepsilon^k$. Then, let τ_k be a task with size $\pi(\tau_k) = \pi_{min} + \Delta(k)$, for all $k = 0, 1, 2, 3, \dots$. Observe that $\forall k, \pi(\tau_k) \in (\pi_{min}, \pi_{max}]$ and $\pi(\tau_{k+1}) < \pi(\tau_k)$. Let now time points t_k, such that $t_0 = 0$ (the beginning of the execution) and $t_{k+1} = t_k + \pi_{min} + \frac{1+\varepsilon}{2}\Delta(k)$. Let also time points $t'_k = t_{k-1} + \frac{1-\varepsilon}{2}\Delta(k-1)$. The arrival pattern A is such that task $\tau_0 = \pi_{max}$ is injected in the system at time instant t_0. Then, for $k = 1, 2, \dots$ one task τ_k is 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 to be examined:
(a) If ALG schedules a task before t'_{k+1}, then X waits until task τ_{k+1} is injected at time t'_{k+1} and schedules it. Then, crashes the machine right after the τ_{k+1} is completed. Note that X will complete the task at time $t'_{k+1} + \pi(\tau_{k+1}) = t_k + \frac{1-\varepsilon}{2}\Delta(k) + \pi_{min} + \Delta(k+1) = t_k + \pi_{min} + \frac{1+\varepsilon}{2}\Delta(k) = t_{k+1}$. On the other hand, ALG will not be able to complete the task that was scheduled before t'_{k+1}. This is because, $t_{k+1} - t_k = \pi_{min} + \frac{1+\varepsilon}{2}\Delta(k) < \pi(\tau_k)$ and all possible tasks that ALG could have scheduled before t'_{k+1} are of size at least $\pi(\tau_k)$. This would mean, that ALG is not able to complete any task by time t_{k+1}; instead one more task is added in its pending queue, a τ_{k+1}-task. On the other hand, algorithm X was able to complete the new task, maintaining its pending queue as it was at time t_k.
(b) If ALG doesn't schedule any task before t'_{k+1}, then X schedules the task it has pending at time t_k, say τ, which in the worst case is a π_{max}-task, and the machine is not crashed until it is completed. Observe that X will complete the task τ at time $t^* = t_k + \pi(\tau) \leq t_k + \pi_{max}$. On the same time, if ALG schedules any of the available tasks at time t'_{k+1}, say the smallest possible τ_{k+1}, it will only be able to complete it by $t'_{k+1} + \pi(\tau_{k+1}) = t_k + \frac{1-\varepsilon}{2}\Delta(k) + \pi_{min} + \Delta(k+1) = t_k + \pi_{min} + \frac{1+\varepsilon}{2}\Delta(k) = t_{k+1}$, which is bigger than the previously defined t^*. This would mean

that by time t^*, algorithm ALG is not able to complete any task; instead one τ_{k+1}-task is added in its pending queue. On the other hand, algorithm X is able to complete the task it previously had in its queue and only has the new task now, τ_{k+1}.

The fact that algorithm X only has one task pending in each case, is easy to observe, since at every t'_k instant there is only one task injected and in both cases (a) and (b) exactly one task is completed by X. Letting this adversarial behavior run to infinity, one can 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 $\mathcal{C}(\text{ALG}) = 0$, and a pending-load competitive ratio $\mathcal{P}(\text{ALG}) = \infty$ as claimed. ■

5.1.3. Properties of $M\langle 1, 1, 2\rangle$

Having proven the above results, the following question arises: *Would a bounded number of task sizes give any competitiveness?* It is therefore natural, to look at the case where only two different task sizes are considered, which corresponds to the model $M\langle 1, 1, 2\rangle$. For this, a non-zero upper bound on the completed-load competitiveness of any deterministic algorithm is given. Even though this is a negative result, it allows some *hope* for the existence of some possible completed-load competitive scheduling policy, even without the use of speedup.

Let ALG be any deterministic or work-conserving algorithm for the considered task scheduling problem. In order to prove an upper bound on the completed load, ALG will be competing with an off-line algorithm OFF associated with arrival and error patterns A and E respectively, that follow the adversarial strategy described below. Note that the adversary controls the task arrivals and machine crash/restart events, as well as the actions of algorithm OFF.

Description of adversarial strategy

Consider an infinite supply of π_{max}-tasks at all times, and initially assume that there are no π_{min}-tasks pending. The execution is divided in *phases*; the periods between two consecutive crashes of the machine, and two scenarios are distinguished for each phase, given the behavior of the deterministic algorithm and the adversarial arrival and error patterns A and E. The two scenarios can be seen in Figures 5.1 and 5.2, and are as follows:

Scenario 1. The phase starts with ALG scheduling a π_{max}-task. Immediately after ALG starts executing the π_{max}-task, a set of $\hat{\rho}$ π_{min}-tasks arrive, that are scheduled and completed by OFF. After OFF completes these tasks, the machine is crashed, so ALG cannot complete the execution of the π_{max}-task. Here the fact that $\hat{\rho} < \rho$ is used. *Note, that the first phase of the execution belongs to this class.*

Scenario 2. The phase starts with ALG scheduling a π_{min}-task. In this case, OFF executes a π_{max}-task and immediately after its completion, the machine is crashed. Observe that in this

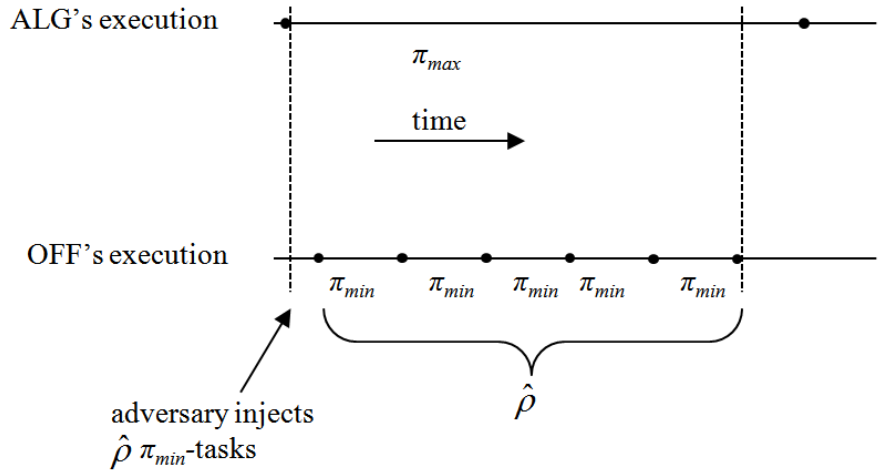

Figure 5.1: Illustration of Scenario 1. It uses the fact that $\hat{\rho} < \rho$.

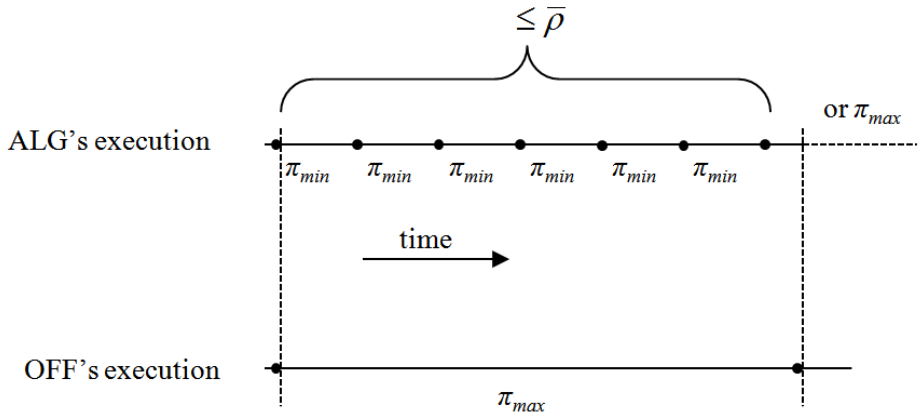

Figure 5.2: Illustration of Scenario 2. It uses the fact that $\bar{\rho} \leq \rho$.

phase ALG will be able to complete successfully several π_{min}-tasks; in particular, up to $\bar{\rho}$ of them.

Analysis of adversarial strategy

For the simplicity of the analysis, consider any time t being the end of a phase in the execution. Let p_1 be the number of phases that belong to scenario 1 executed by time t. Similarly, let $p_2(j)$ be the number of phases belonging to scenario 2 executed by time t, during which ALG completes j π_{min}-tasks, for $j \in [1, \bar{\rho}]$. Then, the completed-load competitive ratio of ALG, at a time instant

t can be computed as follows:

$$\mathcal{C}_t(\text{ALG}, \mathsf{A}, \mathsf{E}) = \frac{\pi_{min} \sum_{j=1}^{\bar{\rho}} j p_2(j)}{\pi_{max} \sum_{j=1}^{\bar{\rho}} p_2(j) + \pi_{min} \hat{\rho} p_1} . \tag{5.1}$$

From the arrival pattern A, the number of π_{min}-tasks injected by time t is exactly $\hat{\rho} p_1$. Hence, $\sum_{j=1}^{\bar{\rho}} j p_2(j) \leq \hat{\rho} p_1$. It can be easily observed from Eq. 5.1 that the completed load increases with the average number of π_{min}-tasks successfully completed in the phases of scenario 2. Hence, it would be maximal if all the π_{min}-tasks are executed in phases of scenario 2 with $\bar{\rho}$-tasks. With the above, the following theorem is obtained.

Theorem 5.3. *The long-term completed-load competitive ratio of any deterministic or work-conserving algorithm ALG, running in model $M\langle 1, 1, 2\rangle$ under adversarial patterns A and E, and up to time t, is at most $\frac{\bar{\rho}}{\rho+\bar{\rho}} \leq \frac{1}{2}$, i.e., $\mathcal{C}(ALG) \leq \frac{\bar{\rho}}{\rho+\bar{\rho}} \leq \frac{1}{2}$. (Observe that the equality holds iff ρ is an integer.)*

Proof: Applying the bound $\sum_{j=1}^{\bar{\rho}} p_2(j) \geq \sum_{j=1}^{\bar{\rho}} \frac{j p_2(j)}{\bar{\rho}}$ in Eq (1),

$$\mathcal{C}_t(\text{ALG}, \mathsf{A}, \mathsf{E}) \leq \frac{\pi_{min} \sum_{j=1}^{\bar{\rho}} j p_2(j)}{\frac{\pi_{max}}{\bar{\rho}} \sum_{j=1}^{\bar{\rho}} j p_2(j) + \pi_{min} \hat{\rho} p_1} ,$$

which is a function that increases with $\sum_{j=1}^{\bar{\rho}} j p_2(j)$. Since $\sum_{j=1}^{\bar{\rho}} j p_2(j) \leq \hat{\rho} p_1$, the competitive ratio can be bounded by

$$\mathcal{C}_t(\text{ALG}, \mathsf{A}, \mathsf{E}) \leq \frac{\pi_{min} \hat{\rho} p_1}{\frac{\pi_{max}}{\bar{\rho}} \hat{\rho} p_1 + \pi_{min} \hat{\rho} p_1} = \frac{\pi_{min} \bar{\rho}}{\pi_{max} + \pi_{min} \bar{\rho}} = \frac{\bar{\rho}}{\rho + \bar{\rho}}.$$

which completes the proof of the claim. ■

From the above upper bound one can show the following corollary.

Corollary 5.1. *For any non-negative value K, there is a time instant t^* in the execution of any deterministic or work-conserving algorithm ALG running on a single machine without speedup, such that for the rest of the execution, its pending load is at least K. I.e., $\forall t > t^*$: $P_t(ALG) \geq K$.*

Proof: For simplicity, let A be used to refer to any algorithm ALG. From the upper bound of long-term completed load competitiveness shown in Theorem 5.3, it means that $\mathcal{C}(A) = \lim_{t\to\infty} \frac{C_t(A)}{C_t(X)} \leq \frac{1}{2}$.

Now, from the definition of limits: $\forall \epsilon > 0, \exists t_0$ such that $\forall t > t_0$:

$$\frac{C_t(A)}{C_t(X)} \leq \frac{1}{2} + \epsilon \tag{5.2}$$

One can choose ϵ such that, $1/2 + \epsilon = \delta < 1$. It is known that the exact pending and completed load of any algorithm and at any point of its execution, equals the total injected load; i.e., $I_t = P_t(A) + C_t(A) = P_t(X) + C_t(X)$. Combining this with Eq. 5.2, it means that $\forall t > t_0$: $I_t - P_t(A) \leq (\frac{1}{2} + \epsilon)(I_t - P_t(X)) = \delta(I_t - P_t(X)) \leq \delta \cdot I_t$ and thus

$$P_t(A) \geq (1 - \delta) \cdot I_t. \tag{5.3}$$

Since infinite executions are being assumed, it is also known that the completed load of X goes to infinity. This also means that the injected load goes to infinity; i.e., $\lim_{t\to\infty} I_t = \infty$. This means, that $\lim_{t\to\infty} \frac{P_t(A)}{1-\delta} = \infty \Rightarrow \lim_{t\to\infty} P_t(A) = \infty$.

Hence, one can define a time instant t^*, such that $\forall t > t^*$, $P_t(A) \geq K$, where K can be set to any positive value, which completes the proof. ■

5.1.4. Properties of $M\langle 1, s, \infty\rangle$

Going back to the case where tasks may have any arbitrary size in $[\pi_{min}, \pi_{max}]$, consider now the addition of resource augmentation, or otherwise, speedup $s \geq 1$. The following theorems can then be proven, giving both negative and positive results.

Theorem 5.4. *Any work-conserving algorithm ALG running on a single machine with any speedup s, i.e., $M\langle 1, s, \infty\rangle$, 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 5.4 follows directly from Lemmas 5.1 and 5.2 below, that show the two bounds of the competitive ratios separately.

Lemma 5.1. *Any work-conserving algorithm ALG running on a single machine with any speedup s, has a completed-load competitive ratio $\mathcal{C}(ALG) \leq 1$; more precisely, in executions where $\mathbf{P}_t(ALG) = \emptyset$ infinitely many times.*

Proof: Consider an adversary that causes the queue of pending tasks of ALG to become empty infinitely many times in an execution. In particular, consider the arrival and error patterns A and E, and let 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 injected, where $\pi \in [\pi_{min}, \pi_{max}]$. Let T_i be the time interval $[t_i, t_{i+1}]$. Observe that an algorithm X (running with $s = 1$) completes the 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. Since ALG has no more tasks to complete, it cannot increase its total completed load, which leads to $\mathcal{C}(\text{ALG}) \leq 1$ as claimed. ■

Lemma 5.2. *Any work-conserving algorithm ALG running on a single machine with any 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: 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. Consider also phases of arbitrarily chosen lengths π, defined as intervals $T_i = [t_k, t_{k+1}]$, where $t_{k+1} = t_k + \pi$ with $t_0 = 0$ and $k = 0, 1, 2 \ldots$ representing time 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 $s\pi$ total load. Assuming that there are no phases of length less than π_{min}, the complementing pending load at a time t_k will therefore be $P_{t_k}(X, \mathsf{A}, \mathsf{E}) \geq I_{t_k}(\mathsf{A}) - t_k$ and $P_{t_k}(\text{ALG}, \mathsf{A}, \mathsf{E}) \geq I_{t_k}(\mathsf{A}) - st_k$. The pending-load competitive ratio becomes $\mathcal{P}(\text{ALG}) \geq \frac{I(\mathsf{A})-st}{I(\mathsf{A})-t}$, which yields to $\mathcal{P}(\text{ALG}) \geq 1$, since $I(\mathsf{A})$ can be made infinitely big. ■

Turn the focus on some general positive results now. It can be shown, that if the speedup used is *large enough*, some competitiveness can be guaranteed for any work-conserving algorithm.

Theorem 5.5. *Any work-conserving algorithm ALG running on a single machine with speedup $s \geq \rho$, has completed-load competitive ratio $\mathcal{C}(ALG) \geq 1/\rho$ and pending-load competitive ratio $\mathcal{P}(ALG) \leq \rho$.*

The proof of Theorem 5.5 follows directly from Lemma 5.3 below.

Lemma 5.3. *No algorithm X, running on a machine 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, $\#C_t(ALG, \mathsf{A}, \mathsf{E}) \geq \#C_t(X, \mathsf{A}, \mathsf{E})$ and hence $\#P_t(ALG, \mathsf{A}, \mathsf{E}) \leq \#P_t(X, \mathsf{A}, \mathsf{E})$.*

Proof: The aim is to prove that $\forall t$, A and E, $\#P_t(\text{ALG}, \mathsf{A}, \mathsf{E}) \leq \#P_t(X, \mathsf{A}, \mathsf{E})$, which implies that $\#C_t(\text{ALG}, \mathsf{A}, \mathsf{E}) \geq \#C_t(X, \mathsf{A}, \mathsf{E})$. (Recall that the notation $\#P$ and $\#C$ represents the number of pending and completed tasks respectively.)

Observe first, that the claim trivially holds for $t = 0$. The proof for the general case follows with induction on t as follows: Consider any time $t > 0$ and corresponding time $t' < t$ such that t' is the latest time instant before t that is either a crash/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, $\#P_{t'}(\text{ALG}) \leq \#P_{t'}(X)$. (For simplification, A and E are omitted in the rest of the proof.)

Since ALG is work-conserving, it is continuously executing tasks in the interval T. Also, ALG needs at most $\pi_{max}/s \leq \pi_{min}$ time to execute any task using speedup $s \geq \rho$, regardless of the task being executed. Then it holds that

$$\#P_t(\text{ALG}) \leq \#P_{t'}(\text{ALG}) + \#I_T - \left\lfloor \frac{t-t'}{\pi_{max}/s} \right\rfloor \leq \#P_{t'}(\text{ALG}) + \#I_T - \left\lfloor \frac{t-t'}{\pi_{min}} \right\rfloor .$$

(Recall that $\#I_T$ is the number of tasks injected in the interval T.)

On the other hand, X can complete at most one task every π_{min} time. Hence, $\#P_t(X) \geq \#P_{t'}(X) + \#I_T - \left\lfloor \frac{t-t'}{\pi_{min}} \right\rfloor$. As a result,

$$\#P_t(X) - \#P_t(\text{ALG}) \geq \#P_{t'}(X) + \#I_T - \left\lfloor \frac{t-t'}{\pi_{min}} \right\rfloor - \#P_{t'}(\text{ALG}) - \#I_T + \left\lfloor \frac{t-t'}{\pi_{min}} \right\rfloor \geq 0.$$

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

Increasing now the amount of speedup even more, it can be shown that both competitive ratios (completed load and pending load) improve.

Theorem 5.6. *Any work-conserving algorithm ALG running on a single machine 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, let time instant $t' < t$ 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, $\mathbf{P}_{t'}(\text{ALG}) = \emptyset$.

It is trivial that $P_0(\text{ALG}, \mathsf{A}, \mathsf{E}) \leq P_0(X, \mathsf{A}, \mathsf{E})$ holds at the beginning of the executions. Now assuming that $P_{t'}(\text{ALG}, \mathsf{A}, \mathsf{E}) \leq P_{t'}(X, \mathsf{A}, \mathsf{E})$ holds at time t', the proof continues with induction showing that $P_t(\text{ALG}, \mathsf{A}, \mathsf{E}) \leq P_t(X, \mathsf{A}, \mathsf{E})$ 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:

- X is not able to complete any task in the interval T.

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

- 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) Time t' is from case (3) of its definition.

Hence, $\mathbf{P}_{t'}(\text{ALG}) = \emptyset$ and $P_t(\text{ALG}, \mathsf{A}, \mathsf{E}) \leq I_T$. 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, $P_t(X, \mathsf{A}, \mathsf{E}) \geq I_T \geq P_t(\text{ALG}, \mathsf{A}, \mathsf{E})$.

(b) Time 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 $P_t(X, \mathsf{A}, \mathsf{E}) = P_{t'}(X, \mathsf{A}, \mathsf{E}) + I_T - \pi$ while $P_t(\text{ALG}, \mathsf{A}, \mathsf{E}) \leq P_{t'}(\text{ALG}, \mathsf{A}, \mathsf{E}) + I_T - (\pi s - \pi_{max})$. Observe that $s \geq 1 + \rho$ implies that $\pi s - \pi_{max} \geq \pi$. Hence, from the induction hypothesis, $P_t(\text{ALG}, \mathsf{A}, \mathsf{E}) \leq P_t(X, \mathsf{A}, \mathsf{E})$.

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. ■

5.2. Competitiveness without speedup

As seen in Section 5.1, little can be achieved without speedup (i.e., $s = 1$), with any deterministic and/or work-conserving algorithm. The only *interesting* performance measure in this case is the completed-load competitiveness.

The first *natural* scheduling policy one could consider for the problem of maximizing the completed-load by scheduling tasks of different sizes on a fault-prone machine, is algorithm *Shortest Processing Time* (SPT), which gives priority to π_{min}-tasks, whenever available. However, even if surprising, this policy is not *that* efficient in the considered setting; it does not reach the upper bound found in Subsection 5.1.3.

It can be shown, that algorithm SPT cannot have completed load competitive ratio larger than $\frac{1}{\rho+1}$, which is strictly less than the upper bound $\frac{\bar{\rho}}{\bar{\rho}+\rho}$. (See also Figure 5.3 for a graphical representation.)

Theorem 5.7. *Algorithm SPT, cannot achieve completed load competitive ratio larger than* $\frac{1}{\rho+1}$ *under adversarial patterns* A *and* E*, even in model* $M\langle 1, 1, 2\rangle$ *and even if there is a schedule that completes all the tasks successfully.*

Proof: The scenario works as follows. At the beginning of an execution, there are two tasks available, one of each size; a π_{max} and a π_{min}. Algorithm SPT schedules first the π_{min}-task, and when completed it schedules the π_{max}-task. Meanwhile, an offline algorithm OFF schedules first the π_{max}-task. After its execution, the adversary causes a machine crash, so SPT does not complete successfully the π_{max}-task. Now, SPT only has one task in its queue (a π_{max}-task). When this scenario is repeated, it will have several π_{max}-tasks but no π_{min} ones. Hence, SPT schedules this task when the machine is restarted, while OFF schedules the π_{min}-task that has in its queue. When OFF completes the π_{min}-task, the adversary crashes the machine again. This complete scenario can be repeated forever. In each instance, OFF completes one π_{max}-task and one π_{min}-task, while SPT only completes one task of size π_{min}. Hence, the completed load competitive ratio achieved in this execution is $\frac{\pi_{min}}{\pi_{max}+\pi_{min}} = \frac{1}{\rho+1}$ and the same holds for the long-term completed load competitiveness. Observe that at the end of each instance of the scenario the

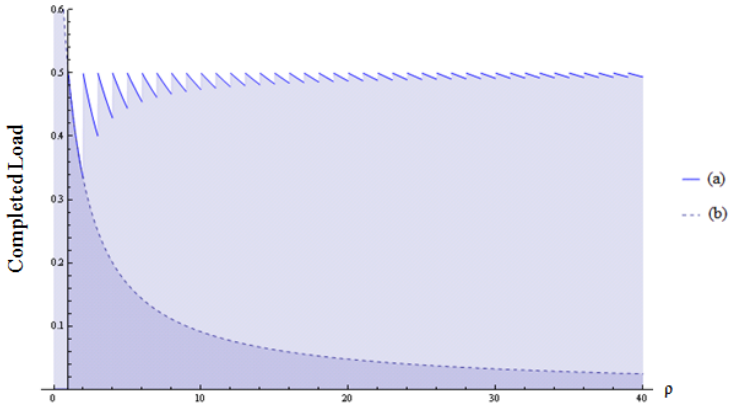

Figure 5.3: Upper bound on the long-term completed-load competitiveness under adversarial task arrivals. (a) For any algorithm ALG, $\mathcal{C}(\text{ALG}) \leq \bar{\rho}/(\rho+\bar{\rho})$ (see analysis in Subsection 5.1.3). (b) For algorithm SPT, $\mathcal{C}(\text{SPT}) \leq 1/(\rho+1)$. Observe that algorithm SPT has a significantly lower bound as ρ increases.

queue of OFF is empty. ■

5.2.1. ρ-SPT-LPT: an *optimal* algorithm for $M\langle 1,1,2\rangle$

The goal is to achieve the upper bound shown in Subsection 5.1.3. Algorithm ρ-SPT-LPT is proposed; one that combines in a graceful and efficient manner two of the widely used policies, SPT and LPT. It is a bit surprising, that their combination actually provides the desired optimal long-term completed-load competitiveness, while none of the two algorithms is sufficiently good when considered on its own.

Algorithm description.

At the beginning of the execution and whenever the machine has recovered from a crash, in other words when restarted, it asks for the queue of pending tasks from the Repository (using the *get* operation) and checks whether there are at least $\bar{\rho}$ tasks of size π_{min} available. If there are, it schedules $\bar{\rho}$ of them — a part of the schedule called the *preamble* — and then, the algorithm continues to schedule tasks using the LPT policy. Otherwise, if there are not "enough" π_{min}-tasks available, it simply schedules tasks following the LPT policy.

Algorithm analysis.

It will be shown that algorithm ρ-SPT-LPT achieves a completed-load competitiveness that matches the upper bound shown in 5.1.3 and hence, it is optimal. Let two types of time intervals for the machine in the executions of algorithm ρ-SPT-LPT be defined as: the *active* and the *inactive* periods. During an active period, the machine is working with no interruptions by crashes

and restarts, and the queue of pending tasks (in ρ-SPT-LPT) does not become empty. An inactive period is a non-active one. In other words, a time interval $T = [t_i, t_{i+1})$ is an active period if it starts with time instant t_i such that (a) it is the time of some task injection after an interval where the queue of ρ-SPT-LPT has been empty, or (b) it is the time right after a restart. Active period T ends with time instant t_{i+1} such that (i) it is the time at which the machine crashes, or (ii) the queue of pending tasks becomes empty for ρ-SPT-LPT (i.e., ρ-SPT-LPT has just completed the last pending task).

Note that in case (a) the corresponding inactive period had started when the queue of ρ-SPT-LPT became empty before time t_i, say at time instant t', and hence covers interval $[t', t_i)$. On the other hand, if cases (b) and (i) hold, the corresponding inactive period will only be the time instant right before t_i, and hence neither ρ-SPT-LPT nor any algorithm X that is designed to solve the same problem, knowing the adversarial patterns a priori, can make any progress in transmitting a packet. Finally, if cases (b) and (ii) hold, the corresponding inactive period will start at time t_{i+1} until new tasks arrive, say at time instant t''. Observe that during the inactive periods it must be the case that the pending queue of X is also empty, otherwise it would contradict the optimality of X. Recall that we consider offline algorithms being work conserving. Algorithm X is also an offline algorithm, since it knows both arrival and error patterns from the beginning.

Hence, looking at the active periods, which are also referred to as *phases*, there are four types of phases that may occur to the above algorithm:

1. Phase starting with π_{min}-task and has length $|T| < \bar{\rho}\pi_{min}$.
2. Phase starting with π_{min}-task and length $|T| \geq \bar{\rho}\pi_{min}$.
3. Phase starting with π_{max}-task and has length $|T| < \pi_{max}$.
4. Phase starting with π_{max}-task and length $|T| \geq \pi_{max}$.

The following notation will be used especially for the analysis of the algorithm: For the execution of ρ-SPT-LPT and within the ith phase, let a_i be the number of successfully completed π_{min}-tasks not in the preambles, b_i the number of successfully completed π_{max}-tasks, and c_i the number of successfully completed π_{min}-tasks in preambles. For the execution of X and within the ith phase, let a_i^* be the total number of successfully completed π_{min}-tasks and b_i^* the total number of successfully completed π_{max}-tasks. Let $C_i(A, j)$ and $C_i(X, j)$ denote the total load successfully completed within a phase i of type j by ρ-SPT-LPT and X, respectively, where A is used to represent the algorithm ρ-SPT-LPT, for the simplicity of presentation.

Analyzing the different types of phases some observations can be made. First, for phases of type 1, ρ-SPT-LPT is not able to complete the $\bar{\rho}\ \pi_{min}$-tasks of the preamble, but X is only able to complete at most as much load, so $C_i(X, 1) \leq C_i(A, 1)$. For phases of type 2, observe that the amount of load completed by X minus the load completed by ρ-SPT-LPT is at most π_{max}, i.e., $C_i(X, 2) - C_i(A, 2) < \pi_{max}$. Therefore, $C_i(A, 2) \geq \frac{\pi_{min}\bar{\rho}}{\pi_{max}+\pi_{min}\bar{\rho}} C_i(X, 2)$; observe that $\frac{\pi_{min}\bar{\rho}}{\pi_{max}+\pi_{min}\bar{\rho}} \leq 1/2$. The same holds for phases of type 4, i.e., $C_i(X, 4) - C_i(A, 4) < \pi_{max}$, and hence in this case $C_i(X, 4) \leq 2C_i(A, 4)$. In the case of phases of type 3, ρ-SPT-LPT is not able to complete any task, and therefore $C_i(A, 3) = 0$, whereas X might complete up to $\hat{\rho}\pi_{min}$-tasks.

There are therefore two cases of executions to be considered separately.

Case 1: The number of phases of type 3 is finite.
In such a case, there is a phase i^* such that $\forall i > i^*$ phase i is not of type 3. Then

$$R_1 = \frac{\sum\limits_{j \leq i^*} C_j(A) + \sum\limits_{j > i^*} C_j(A)}{\sum\limits_{j \leq i^*} C_j(X) + \sum\limits_{j > i^*} C_j(X)} \,. \tag{5.4}$$

It is clear that the total completed load by the end of phase i^* by both algorithms is bounded. Let $\sum\limits_{j \leq i^*} C_j(A) = A$ and $\sum\limits_{j \leq i^*} C_j(X) = X$ and thus,

$$R_1 = \frac{A + \sum\limits_{j > i^*} C_j(A)}{X + \sum\limits_{j > i^*} C_j(X)} \geq \frac{A + \frac{\pi_{min}\bar{\rho}}{\pi_{max}+\pi_{min}\bar{\rho}} \sum\limits_{j > i^*} C_j(X)}{X + \sum\limits_{j > i^*} C_j(X)} \,.$$

Hence, the completed load competitive ratio of ρ-SPT-LPT, looking at the end of each phase, can be computed as $\mathcal{C}(\rho\text{-SPT-LPT}) = \lim\limits_{t \to \infty} R_1$, i.e.,

$$\begin{aligned}
\mathcal{C}(\rho\text{-SPT-LPT}) &= \lim_{j \to \infty} \frac{A + \frac{\pi_{min}\bar{\rho}}{\pi_{max}+\pi_{min}\bar{\rho}} \sum\limits_{j > i^*} C_j(X)}{X + \sum\limits_{j > i^*} C_j(X)} \\
&= \lim_{j \to \infty} \frac{(\pi_{max} + \pi_{min}\bar{\rho})A + (\pi_{min}\bar{\rho}) \sum\limits_{j > i^*} C_j(X)}{(\pi_{max} + \pi_{min}\bar{\rho})(X + \sum\limits_{j > i^*} C_j(X))} \\
&= \lim_{j \to \infty} \left(\frac{\pi_{min}\bar{\rho}}{\pi_{max} + \pi_{min}\bar{\rho}} + \frac{(\pi_{max} + \pi_{min}\bar{\rho})A - (\pi_{min}\bar{\rho})X}{(\pi_{max} + \pi_{min}\bar{\rho})(X + \sum\limits_{j > i^*} C_j(X))} \right) \\
&= \frac{\pi_{min}\bar{\rho}}{\pi_{max} + \pi_{min}\bar{\rho}} = \frac{\bar{\rho}}{\rho + \bar{\rho}}.
\end{aligned}$$

Here it is important to note that the assumption $\lim\limits_{t \to \infty} C_t(X) = \infty$ is used, which corresponds to the expression $\lim\limits_{j \to \infty} \sum\limits_{j > i^*} C_j(X)$ in the above equality.

So far, the analysis has focused on what is the completed load competitive ratio of ρ-SPT-LPT at the end of each phase. It is however, important to guarantee the lower bound at all times within the phases. Consider any time-point t of phase $i > i^*$. Then $R_i(t) = \frac{\sum_{j \in (i^*, i-1]} C_j(A) + Y_t}{\sum_{j \in (i^*, i-1]} C_j(X) + Z_t}$, where Y_t and Z_t is the work completed by ρ-SPT-LPT and X respectively, within phase i up to time t. Using the proof above and the fact that for phases of type 1, 2 and 4 $C(A) \geq \frac{\pi_{min}\bar{\rho}}{\pi_{max}+\pi_{min}\bar{\rho}} C(X)$,

it follows that $Y_t \geq \frac{\pi_{min}\bar{\rho}}{\pi_{max}+\pi_{min}\bar{\rho}} Z_t$ as well. Therefore,

$$\begin{aligned} R_i(t) &\geq \frac{\frac{\pi_{min}\bar{\rho}}{\pi_{max}+\pi_{min}\bar{\rho}} \sum_{j\in(i^*,i-1]} C_j(X) + \frac{\pi_{min}\bar{\rho}}{\pi_{max}+\pi_{min}\bar{\rho}} Z_t}{\sum_{j\in(i^*,i-1]} C_j(X) + Z_t} \\ &= \frac{\pi_{min}\bar{\rho}}{\pi_{max}+\pi_{min}\bar{\rho}} = \frac{\bar{\rho}}{\rho+\bar{\rho}}. \end{aligned}$$

This completes the lower bound of the completed load competitive ratio for Case 1.

Case 2: The number of phases of type 3 is infinite.
In this case the proof looks at how the number of tasks completed, for both sizes π_{min} and π_{max}, is bounded for both ρ-SPT-LPT and X.

Lemma 5.4. *Consider the time point t at the beginning of a phase j of type 3. Then the number of π_{min}-tasks completed by time t in the execution of X is no more than the amount of π_{min}-tasks completed in the execution of ρ-SPT-LPT plus $\bar{\rho}-1$, i.e., $\sum_{i<j} a_i^* \leq \sum_{i<j}(a_i+c_i)+(\bar{\rho}-1)$.*

Proof: Consider the beginning of phase j of type 3. At that point, we know that ρ-SPT-LPT has at most $(\bar{\rho}-1)$ tasks of size π_{min} in its queue by definition of phase type 3. Therefore, the amount of π_{min}-tasks completed by X by the beginning of phase j is no more than the ones completed by ρ-SPT-LPT (including the π_{min}-tasks in preambles) plus $\bar{\rho}-1$. ■

Lemma 5.5. *Considering all kinds of phases and the number of completed π_{max}-tasks, $\sum_{i\leq j} b_i^* \leq \sum_{i\leq j} b_i + \sum_{i\leq j} \frac{c_i}{\bar{\rho}} + 2$, for every j.*

Proof: This claim is proven by induction on phase j. For the *Base Case:* $j=0$ the claim is trivial. Consider the *Induction Hypothesis* stating that

$$\sum_{i\leq j-1} b_i^* \leq \sum_{i\leq j-1} b_i + \sum_{i\leq j-1} \frac{c_i}{\bar{\rho}} + 2\,.$$

For the *Induction Step* one needs to prove it up to the end of phase j. Consider first, the case where during the phase j there is a time when ρ-SPT-LPT has no π_{max}-tasks pending. Let t be the latest such time in the phase. Thus, t is an arrival time of some injected π_{max}-tasks. Let also $b^*(t)$ and $b(t)$ be the number of π_{max}-tasks completed successfully up to time t by X and ρ-SPT-LPT respectively. It is trivial that $b^*(t) \leq b(t)$. Let also $x_j^*(t)$ and $x_j(t)$ be the number of π_{max}-tasks completed by X and ρ-SPT-LPT respectively, after time point t and until the end of the phase j. From the definitions, at time t, algorithm ρ-SPT-LPT is executing a π_{min}-task. Since t is the last time that ρ-SPT-LPT has no π_{max}-task pending in its queue, the worst case is to be at the beginning of the preamble (by inspection of the 4 types of phases). Then, if the phase ends at time t', let period $T=[t,t']$ be such that:

$$|T| < \bar{\rho}\pi_{min} + (x_j(t)+1)\pi_{max} \leq (x_j(t)+2)\pi_{max}\,.$$

The +1 π_{max}-task is because of the machine crash before transmitting completely the last π_{max}-task scheduled in the phase. This means that $x_j^*(t) \leq x_j(t)+2$. Observe now, that X could be executing a π_{max}-task at time t, completed at some point in $[t, t+\pi_{max}]$ and accounted for in $x_j^*(t)$. Therefore,

$$\sum_{i\leq j} b_i^* = b^*(t) + x_j^*(t) \leq b(t) + x_j(t) + 2 = \sum_{i\leq j} b_i + 2.$$

Now consider the case where at all times of a phase j there are π_{max}-tasks in the queue of ρ-SPT-LPT. By inspection of the 4 types of phases, the worst case is when j is of type 2. Since there is always some π_{max}-task pending in ρ-SPT-LPT, after executing the $\bar{\rho}\pi_{min}$-tasks, it will keep scheduling π_{max} ones, until a machine crash interrupts the last one scheduled, or the queue becomes empty. On the same time X is able to successfully complete at most $\left\lfloor \frac{T_j}{\pi_{max}} \right\rfloor \leq b_j + 1$ tasks of size π_{max}, where T_j is the length of the phase. Therefore, in all types of phases, $b_j^* \leq \frac{c_j}{\bar{\rho}} + b_j$. And hence by induction the claim follows; $\sum_{i\leq j} b_i^* \leq \sum_{i\leq j} \frac{c_i}{\bar{\rho}} + \sum_{i\leq j} b_i + 2$. ■

Combining the two lemmas above, Lemma 5.4 and 5.5:

$$\begin{aligned}
R_2 &= \frac{\sum_{i\leq j} C_i(A)}{\sum_{i\leq j} C_j(X)} = \frac{\sum_{i\leq j}[(a_i+c_i)\pi_{min} + b_i\pi_{max}]}{\sum_{i\leq j}[a_i^*\pi_{min} + b_i^*\pi_{max}]} \\
&\geq \frac{\sum_{i\leq j}[(a_i+c_i)\pi_{min} + b_i\pi_{max}]}{\sum_{i\leq j}(a_i+c_i)\pi_{min} + (\bar{\rho}-1)\pi_{min} + \sum_{i\leq j}(b_i+\frac{c_i}{\bar{\rho}})\pi_{max} + 2\pi_{max}} \\
&\geq \frac{\sum_{i\leq j}[(a_i+c_i)\pi_{min} + b_i\pi_{max}]}{\sum_{i\leq j}[(a_i+2c_i)\pi_{min} + b_i\pi_{max}] + 3\pi_{max}} \\
&\geq \frac{\sum_{i\leq j}[(a_i+c_i)\pi_{min} + b_i\pi_{max}] + \frac{3}{2}\pi_{max} - \frac{3}{2}\pi_{max}}{2\sum_{i\leq j}[(a_i+c_i)\pi_{min} + b_i\pi_{max}] + 3\pi_{max}} \\
&\geq \frac{1}{2} - \frac{\frac{3}{2}\pi_{max}}{2\sum_{i\leq j}[(a_i+c_i)\pi_{min} + b_i\pi_{max}] + 3\pi_{max}}\,.
\end{aligned}$$

Note that, due to parameters a_i, b_i and c_i the second ratio tends to zero (the denominator tends to infinity while the nominator is constant). Therefore,

$$\mathcal{C}(\rho\text{-SPT-LPT}) = \lim_{j\to\infty} R_2 \geq \frac{1}{2}. \tag{5.5}$$

Theorem 5.8. *The long-term completed-load competitive ratio of Algorithm ρ-SPT-LPT, running on a single machine without speedup, i.e., $M\langle 1,1,2\rangle$, is $\mathcal{C}(\rho$-SPT-LPT$) \geq \frac{\bar{\rho}}{\rho+\bar{\rho}}$.*

Proof: From the analyses of Cases 1 and 2 and the fact that $\frac{\bar{\rho}}{\rho+\bar{\rho}} \leq \frac{1}{2}$ it is easy to conclude that the long-term completed-load competitive ratio of Algorithm ρ-SPT-LPT is at least $\frac{\bar{\rho}}{\rho+\bar{\rho}}$ as claimed. ■

5.3. Competitiveness with speedup

Having seen the limitations of $s = 1$, the focus is now turned on the case of speeding up the machine, i.e., using speedup $s > 1$.

5.3.1. *Necessary* speedup conditions

In Section 5.1, it was seen that without speedup, i.e., $s = 1$, deterministic and work-conserving algorithms do not have much *hope* to achieve competitiveness, besides *some limited* completed load. This, justifies further the intuition to use speedup in order to achieve competitiveness, especially in the case of arbitrary task sizes. However, in this Section, it will be shown that even with the use of speedup, there is some *threshold* on the value of s *necessary* to achieve competitiveness; in the case of completed load, necessary to achieve *optimal* competitiveness.

For given task sizes π_{min}, π_{max} and speedup s, parameter γ is defined as the smallest number of π_{min}-tasks, in addition to a π_{max}-task, that an algorithm with speedup s can complete in a time interval of length $(\gamma + 1)\pi_{min}$. Observe that an algorithm running with no speedup ($s = 1$) cannot completed more tasks in the same time. The following properties are therefore satisfied:

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

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

It is then not hard to derive that $\gamma = \max\{\lceil \frac{\pi_{max}-s\pi_{min}}{(s-1)\pi_{min}} \rceil, 0\} = \max\{\lceil \frac{\rho-s}{s-1} \rceil, 0\}$ $\big($recall that $\rho = \frac{\pi_{max}}{\pi_{min}}\big)$.

Two conditions are then proposed:

C1: $s < \rho$, and

C2: $s < 1 + \gamma/\rho$

followed by a proof that if *both* of them hold, then *no* deterministic sequential or parallel algorithm is competitive when run with speedup s, regarding *pending-load,* 1-*completed-load* and *latency*, even in a system with a single machine (the corresponding proofs are shown in subsections 5.3.1, 5.3.1 and 5.3.1). In other words, if $s < \min\{\rho, 1 + \gamma/\rho\}$, for any deterministic algorithm ALG the following hold: $\mathcal{P}(\text{ALG}) = \infty$, $\mathcal{C}(\text{ALG}) < 1$ and $\mathcal{L}(\text{ALG}) = \infty$.

Observe that satisfying condition **C2** implies $\gamma > 0$ (since $s \geq 1$), which automatically means that condition **C1** is also satisfied when $\rho > 1$. Observe also, that by the definitions of γ and ρ, and the fact that $s \geq 1$, it follows that $\min\{\rho, 1 + \gamma/\rho\} < 2$.

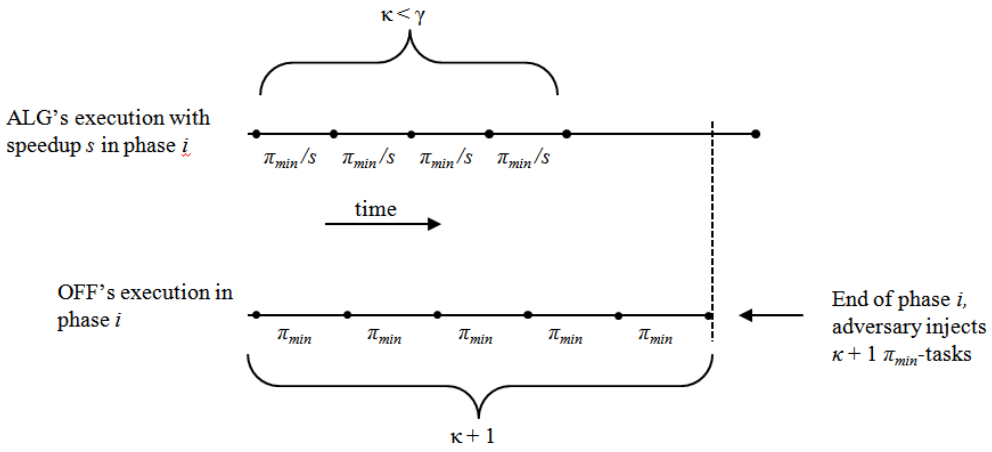

Figure 5.4: Illustration of Scenario 1. It uses the property $(\kappa\pi_{min} + \pi_{max})/s > (\kappa + 1)\pi_{min}$, for any integer $0 \leq \kappa < \gamma$ (Property 2).

Pending Load

Consider a deterministic algorithm ALG and a universal off-line algorithm OFF with associated arrival and error adversarial patterns A and E, to be defined below. It can then be shown, that the pending load of OFF is always bounded while the pending load of ALG is unbounded during the executions under the defined adversarial strategy.

Description of adversarial strategy

In particular, consider an adversary that activates, and later keeps crashing and re-starting one machine. The adversarial patterns and the algorithm OFF are defined recursively in consecutive *phases*, where formally each phase is a closed time interval and every two consecutive phases share an end. The machine is restarted at the beginning and crashed at the end of each phase, while kept continuously alive during the phase. At the beginning of phase 1, there are γ of π_{min}-tasks and one π_{max}-task injected, and the machine is activated.

Suppose that the adversarial patterns, A and E, as well as algorithm OFF until the beginning of phase $i \geq 1$, have already been defined. Suppose also, that in the execution of ALG there are x of π_{min}-tasks and y of π_{max}-tasks pending at the beginning of phase i. The adversary does not inject any tasks until the end of the phase. Under this assumption one could simulate the choices of ALG during the phase i. There are two cases to consider, illustrated in Figures 5.4 and 5.5. First, let parameter Δ be defined as the time elapsed from the beginning of phase i until the time at which ALG starts executing a π_{max}-task, with an intention to complete it (assuming phase i is long enough). Note here, that since ALG is deterministic, the adversary knows the times at which ALG decides to stop any processing cycle and schedule another task. The two possible scenarios are therefore the following:

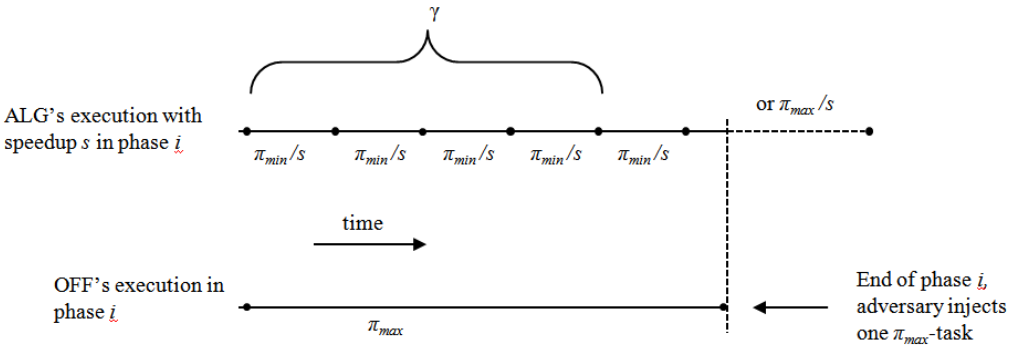

Figure 5.5: Illustration of Scenario 2. It uses the property $(\gamma\pi_{min} + \pi_{max})/s > \pi_{max}$ (condition **C2**).

Scenario 1. When $\Delta < \gamma\pi_{min}/s$, ALG schedules a π_{max}-task sooner than $\gamma\pi_{min}/s$ time from the beginning of the phase. Let $\kappa = \lfloor \Delta/(\pi_{min}/s) \rfloor < \gamma$. The adversary ends the phase $(\kappa + 1)\pi_{min}$ time after the beginning of the phase. From Property 2, $\frac{\kappa\pi_{min}+\pi_{max}}{s} > (\kappa + 1)\pi_{min}$. Therefore, before the end of the phase, OFF has enough time to complete $\kappa + 1$ tasks of size π_{min}, while ALG cannot complete more than κ. Moreover, ALG cannot complete the execution of the π_{max} task. At the end of the phase, the adversary injects $\kappa + 1$ tasks of size π_{min}.

Scenario 2. When $\Delta \geq \gamma\pi_{min}/s$, ALG schedules a π_{max}-task no sooner than $\gamma\pi_{min}/s$ time after the phase starts. On the same time, OFF is able to run a π_{max}-task. The adversary crashes the machine when OFF completes the π_{max}-task, and the phase finishes. At the end of the phase, the adversary injects one π_{max}-task, as OFF has completed one. In this scenario, ALG is at most able to complete γ tasks of size π_{min}.

Analysis of the adversarial pattern

What remains to be shown, is that the definitions of algorithm OFF and the associated adversarial patterns A and E are valid, and that in the execution of OFF the number of pending tasks is bounded, while in the corresponding execution of ALG it is not bounded. Then, since the tasks have bounded size, the same applies to the pending load of both OFF and ALG. First, some useful properties of the considered executions of algorithms ALG and OFF are shown.

Lemma 5.6. *The phases, the adversarial pattern and algorithm OFF are well-defined. Moreover, at the beginning of each phase, there are exactly γ of π_{min}-tasks and one π_{max}-task pending in the execution of OFF.*

Proof: It is argued by induction on the number of phases, that: at the beginning of phase i there are exactly γ of π_{min}-tasks and one π_{max}-task pending in the execution of OFF, and

therefore phase i is well defined. Its specification (including termination time) depends only on whether OFF schedules either up to γ of π_{min}-tasks (in Scenario 1) or one π_{max}-task (in Scenario 2) before the next task arrival at the end of the phase. The invariant holds for phase 1 by definition. By straightforward investigation of both scenarios, the very same configuration of task lengths that has been completed by OFF in its execution during a phase is injected at the end of the phase, and therefore the inductive argument proves the invariant for every consecutive phase. ■

Lemma 5.7. *There is an infinite number of phases.*

Proof: First, by Lemma 5.6, consecutive phases are well-defined. Second, observe that each phase is finite, regardless of whether Scenario 1 or Scenario 2 is applied. It is bounded by the time ALG schedules a π_{max}-task which results to phases of length equal to the time needed by OFF to complete either at most γ of π_{min}-tasks (in Scenario 1) or exactly one π_{max}-task (in Scenario 2). Hence, in an infinite execution the number of phases is infinite. ■

Lemma 5.8. *ALG never completes any π_{max}-task.*

Proof: It follows from the specification of Scenarios 1 and 2, condition **C2** on the speedup s, and from Property 2. Considering a phase, if Scenario 1 is applied for its specification, then ALG could not finish its π_{max}-task scheduled after $\kappa < \gamma$ tasks of sizes π_{min}, because the time needed for completing this sequence of tasks is at least $\frac{\kappa\pi_{min}+\pi_{max}}{s}$, which is larger than the length of this phase $(\kappa+1)\pi_{min}$ (Property 2). If Scenario 2 is applied for the phase specification, then the first π_{max}-task could be finished by ALG no earlier than $\frac{\gamma\pi_{min}+\pi_{max}}{s}$ time after the beginning of the phase, which is again bigger than the length of the phase, π_{max} (by the assumption of condition **C2** on the speedup $s < 1+\gamma/\rho = \frac{\gamma\pi_{min}+\pi_{max}}{\pi_{max}}$). ■

Lemma 5.9. *If Scenario 2 was applied in the specification of a phase i, then the number of pending π_{max}-tasks at the end of the phase in the execution of ALG increases by one comparing with the beginning of the phase. In the execution of OFF on the other hand, the number of pending π_{max}-tasks stays the same.*

Proof: It follows from Lemma 5.8 and from the specification of tasks injections at the end of phase i, by Scenario 2. ■

Putting everything together, Theorem 5.9 can now be proven.

Theorem 5.9. *For any given π_{min}, π_{max} and s, if both conditions* **C1:** *$s < \rho$, and* **C2:** *$s < 1+\gamma/\rho$ are satisfied, then no deterministic algorithm is pending-load competitive when run with speedup $s > 1$ against an adversary that injects tasks with sizes in the range $[\pi_{min}, \pi_{max}]$, even in a system with one single machine.*

Proof: By Lemma 5.6, the adversarial patterns A and E and the corresponding off-line algorithm OFF are well-defined and by Lemma 5.11, the number of phases is infinite. There are

therefore two cases to consider:
(1) If the number of phases for which Scenario 2 was applied is infinite, then by Lemma 5.9 the number of pending π_{max}-tasks increases by one infinitely many times, while by Lemma 5.8 it never decreases. Hence it is unbounded.
(2) Otherwise (i.e., if the number of phases for which Scenario 2 was applied is bounded), after the last Scenario 2 phase in the execution of ALG, there are only phases in which Scenario 1 is applied, and there are infinitely many of them. In each such phase, ALG completes only κ of π_{min}-tasks (where $\kappa < \gamma$) while $\kappa+1$ tasks of size π_{min} will be injected at the end of the phase. Indeed, the length of the phase is $(\kappa + 1)\pi_{min}$, while after completing κ of π_{min}-tasks ALG schedules a π_{max}-task and the machine is crashed before completing it, because $\frac{\kappa\pi_{min}+\pi_{max}}{s} > (\kappa + 1)\pi_{min}$ (cf., Property 2). Therefore, in every such phase of the execution of ALG the number of pending π_{min}-tasks increases by one, and it does not decrease since there are no other kinds of phases (recall that we consider phases with Scenario 1 after the last phase with Scenario 2 finished). Hence the number of π_{min}-tasks grows unboundedly in the execution of ALG.

To conclude, in both cases above, the number of pending tasks in the execution of ALG grows unboundedly in time, while the number of pending tasks in the corresponding execution of OFF (for the same adversarial patterns) is always bounded, by Lemma 5.6.[1] As a consequence, the pending load in the execution of ALG grows unboundedly in time, while the pending load in the corresponding execution of OFF is bounded, to exactly $P_t(\text{OFF}, \mathsf{A}, \mathsf{E}) = \gamma\pi_{min} + \pi_{max}$. ■

1 - Completed Load

For the completed load measure and the same conditions **C1** and **C2**, no deterministic scheduling algorithm can achieve 1-completed-load-competitiveness. To show this, the adversarial strategy used is the same as the one used for the pending task analysis in 5.3.1. The off-line 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 $\gamma\pi_{min} + \pi_{max}$, while the completed load of ALG can only be a fraction of the total size of the injected tasks.

Theorem 5.10. *For any given π_{min},π_{max} and s, if both conditions* **C1:** *$s < \rho$ and* **C2:** *$s < 1+\gamma/\rho$ are satisfied, then no deterministic algorithm ALG is 1-completed-load competitive when run with speedup $s \geq 1$ against an adversary that injects tasks of sizes in the range $[\pi_{min}, \pi_{max}]$, even in a system with one single machine.*

Proof: Before starting the main line of the proof, observe that a third condition, $s \geq \frac{\gamma+\rho}{\gamma+1}$, is implied from conditions **C1** and **C2**. This, comes from the definition of parameter γ and its properties; more precisely Property 1.

Let $f = f_1 + f_2$ be the current number of phases completed so far in the execution, where f_1 is the number of phases of Scenario 1 and f_2 is the analogous number of phases of Scenario 2.

[1] Note that the use of condition **C1** is implicit in our proof.

Recall also, that I is the total injected load of the tasks that have arrived so far (sum of sizes of injected tasks). Note then, that

$$f_2 \cdot \pi_{max} \leq I - (\gamma\pi_{min} + \pi_{max}) \leq f \cdot (\gamma\pi_{min} + \pi_{max}) \ ,$$

since in Scenario 2, where most tasks are completed, there are at least $\gamma\pi_{min}$ and one π_{max}-task pending. Consequently,

$$f \geq \frac{I}{\gamma\pi_{min} + \pi_{max}} - 1$$

Consider now parameter $\beta \in (0, 1)$. There are two cases to be analyzed for the total completed load:

1) If there are at least βf phases of Scenario 2, i.e., $f_2 \geq \beta f$, the completed load of ALG is at most $I - f_2\pi_{max}$; equal to the total size of the tasks that have arrived, I, minus the amount of work OFF was able to complete in the phases of Scenario 2, $f_2\pi_{max}$. It is known that ALG is not able to complete any π_{max}-tasks, but since there were f_2 phases of Scenario 2, at least that many π_{max}-tasks were injected in the system and OFF was able to complete them. This is at most

$$I - \beta f \cdot \pi_{max} \leq I - \frac{\beta I \pi_{max}}{\gamma\pi_{min} + \pi_{max}} + \beta\pi_{max} = I \cdot \left(1 - \frac{\beta\pi_{max}}{\gamma\pi_{min} + \pi_{max}} + \frac{\beta\pi_{max}}{I}\right),$$

by applying the lower bound on f. On the same time, the completed load of OFF is at least

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

by the definition of the adversarial patterns; it completed all injected tasks, except the ones that are pending at the time of observation (at most $\gamma\pi_{min} + \pi_{max}$). The completed-load competitiveness is therefore

$$\mathcal{C}(\text{ALG}) \leq \frac{1 - \frac{\beta\pi_{max}}{\gamma\pi_{min}+\pi_{max}} + \frac{\beta\pi_{max}}{I}}{1 - \frac{\gamma\pi_{min}+\pi_{max}}{I}} \ ,$$

which goes to a constant $\mathcal{C}_1 = 1 - \frac{\beta\pi_{max}}{\gamma\pi_{min}+\pi_{max}} < 1$ as $I \to \infty$.

2) If there are less than βf phases of Scenario 2, i.e., $f_2 < \beta f$, the completed load of ALG is at most $I - (1 - \beta)f\pi_{min} + \beta f\gamma\pi_{min}$. This is because ALG is not able to complete as many π_{min}-tasks as OFF in the phases of Scenario 1 (it completes one less per phase) but is able to complete $\gamma\pi_{min}$ in each phase of Scenario 2. Thus, the completed-load of ALG is at most

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

by applying the lower bound on f. Using again the fact that the completed load of OFF is at least

$I \cdot \left(1 - \frac{\gamma\pi_{min}+\pi_{max}}{I}\right)$, the completed-load competitiveness can be estimated as follows

$$\mathcal{C}(\text{ALG}) \leq \frac{1 - \frac{\pi_{min}(1-\beta-\beta\gamma)}{\gamma\pi_{min}+\pi_{max}} + \frac{\pi_{min}(1-\beta-\beta\gamma)}{I}}{1 - \frac{\gamma\pi_{min}+\pi_{max}}{I}},$$

which goes to a constant $\mathcal{C}_2 = 1 - \frac{\pi_{min}(1-\beta-\beta\gamma)}{\gamma\pi_{min}+\pi_{max}} < 1$ as $I \to \infty$.

This completes the analysis of the two cases. Making the completed-load competitiveness of the two cases equal, i.e., $\mathcal{C}_1 = \mathcal{C}_2$, we get

$$1 - \frac{\beta\pi_{max}}{\gamma\pi_{min} + \pi_{max}} = 1 - \frac{\pi_{min}(1 - \beta - \beta\gamma)}{\gamma\pi_{min} + \pi_{max}},$$

which yields $\beta = \frac{1}{\rho+\gamma+1}$. Thus, for such β the completed-load competitiveness converges to

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

as claimed, regardless of whether the actual phases are of Scenario 1 or Scenario 2. ■

Latency

Following similar ideas, the following adversarial arrival and error patterns A and E are now defined, which as will be shown, they prohibit any deterministic algorithm of achieving any latency competitiveness. Consider any deterministic algorithm ALG running with speedup $s \geq 1$ and let a universal off-line algorithm OFF running with no speedup ($s = 1$), be 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 remained continuously alive during the phase. A *stage* consists of j consecutive phases, during which the adversary allows ALG to complete at most one π_{max}-task. At the beginning of the first phase there are γ tasks of size π_{min} and one of π_{max} injected and the machine is activated.

Assume that the adversarial arrival and error patters are already defined and at the beginning of stage $i \geq 1$ of algorithm ALG there are x of π_{min}-tasks and y of π_{max}-tasks pending. Assume also 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. There are basically four cases to be considered, illustrated in the Figures 5.6, 5.7 and 5.8. Observe that Scenario 1(b)(i) is the same as Scenario 1 in the proof of the pending-load non-competitiveness, shown in Figure 5.4. First, let parameter $\Delta(j_i)$ be the time elapsed from the beginning of phase j in stage i until the time at which ALG starts executing a π_{max}-task (assuming the phase is long enough). Note that since

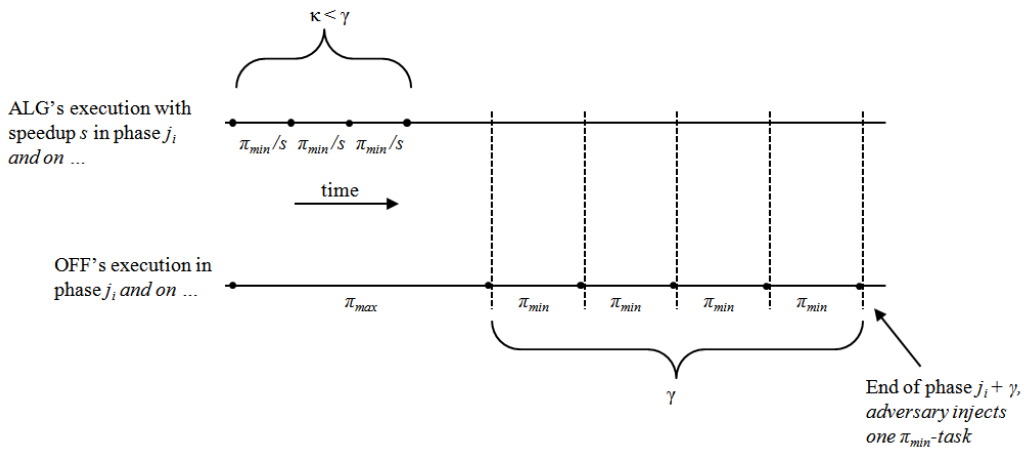

Figure 5.6: Illustration of Scenario 1(a).

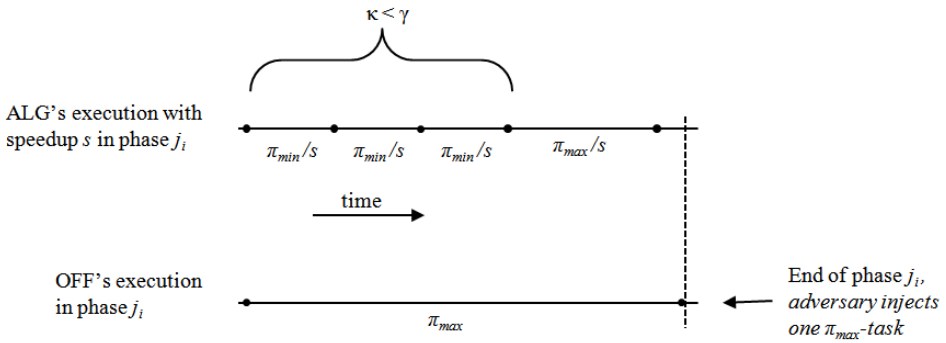

Figure 5.7: Illustration of Scenario 1(b)(ii).

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. There are two scenarios for the stages that may exist:

Scenario 1. When $\forall j_i \in i$, $\Delta(j_i) < \frac{\gamma\pi_{min}}{s}$, ALG schedules a π_{max}-task sooner than $\gamma\pi_{min}/s$ time after the beginning of the phase j_i, for all phases of stage i. Let $\kappa_{j_i} = \lfloor \Delta(j_i)/(\pi_{min}/s) \rfloor < \gamma$ be the number of π_{min}-tasks scheduled before the π_{max} one in phase j_i. These are the types of phases that may occur:
(a) If $\pi_{max} \leq \frac{\kappa_{j_i}\pi_{min}+\pi_{max}}{s}$, then the adversary ends phase j_i after π_{max} time without injecting any additional task. In such phase, OFF has time to complete a π_{max}-task, while ALG is able to complete only up to κ_{j_i} tasks of size π_{min}. Then, exactly γ phases of length π_{min} follow, before the adversary injects a π_{min}-task at the end of the phase. After that, phases of length π_{min}

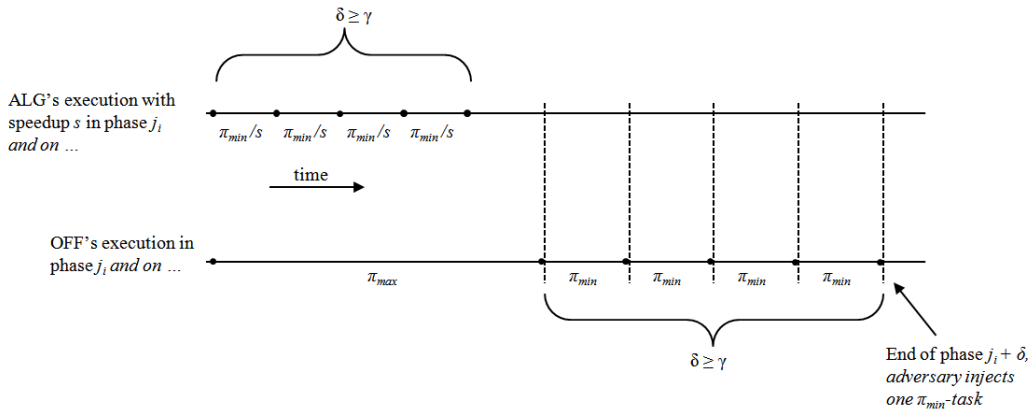

Figure 5.8: Illustration of Scenario 2.

follow, with exactly one π_{min} task injected at the end. By this, it guarantees an infinite execution of latency 0 for OFF (after phase j_i) while the latency of ALG goes to infinity as it is never able to complete the π_{max} tasks pending at the beginning of phase j_i.

(b) If $\pi_{max} > \frac{\kappa_{j_i}\pi_{min}+\pi_{max}}{s}$ and $\pi_{max} > (\kappa_{j_i}+1)\pi_{min}$, then:

(i) if $(j_i-1)\pi_{min} < 2^i\pi_{max} + \gamma\pi_{min}$, then the adversary ends phase j_i after $(\kappa_{j_i}+1)\pi_{min}$ time, at the end of which it injects $\kappa_{j_i}+1$ tasks of size π_{min} (as many as OFF has managed to complete). In such phases, OFF has the time to complete $\kappa_{j_i}+1$ tasks of size π_{min}, 1 more π_{min}-task than the ones completed by ALG before the machine is crashed. What is more, ALG cannot complete the π_{max}-task scheduled after the $\kappa_{j_i}\pi_{min}$-tasks, resulting with one more π_{min}-task pending than the ones it had at the beginning of the phase. At the end of the phase, the latency of both ALG and OFF increases by $(\kappa_{j_i}+1)\pi_{min}$ time.

(ii) if $(j_i-1)\pi_{min} \geq 2^i\pi_{max} + \gamma\pi_{min}$, then the phase ends after π_{max} time, injecting at the end one π_{max}-task. In such phase, OFF completes the pending π_{max}-task, while ALG is able to complete at least κ_{j_i} tasks of size π_{min} and the π_{max}-task as well. However, at the end of such phase the latency of OFF equals the time that the γ pending π_{min}-tasks have been in the system, while the latency of ALG is at least equal to $2^i\pi_{max}$. This, is because the π_{min} packets that are pending in the execution of OFF at the beginning of the phase are of total size at least $2^i\pi_{max} + \gamma\pi_{min}$, and each π_{min}-task is accumulated in intuitively $\approx \gamma\pi_{min}$ 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 2^i times in each stage i.

Scenario 2. When $\exists j_i$ such that, $\Delta(j_i) \geq \frac{\gamma\pi_{min}}{s}$, ALG schedules a π_{max}-task no sooner than $\gamma\pi_{min}/s$ time after the beginning of the j^{th} phase of stage i. The adversary ends the phase with a crash after π_{max} time, so that OFF is able to complete the π_{max}-task pending. No more tasks are introduced and the following phases are of length π_{min}. At the end of the γ^{th} phase and the phases to follow, there is exactly one π_{min}-task injected. OFF is therefore able to complete the remaining

π_{min}-tasks (γ plus all the injected ones) one at a time, while ALG may also complete some π_{min}-tasks, but not the π_{max} that is pending. For the previous $(j-1)$ phases the adversarial behavior followed is the one described in Scenario 1(b)(i); the other kinds of phases assume completing a π_{max}-task, which would put stage i into Scenario 1 directly.

Analysis of the adversarial strategy

In order to analyze the adversarial behavior described above, the following lemmas are first proven, which then lead to the final theorem of the section.

Lemma 5.10. *The phases, the adversarial pattern and algorithm OFF are well-defined. At the beginning of each phase in Scenario 1(b) and the first j_i phases in Scenarios 1(a) and 2, there are exactly γ of π_{min}-tasks and one π_{max}-task pending in the execution of OFF. For the rest of the phases in Scenarios 1(a) and 2, there are less than γ π_{min}-tasks pending in the execution of OFF.*

Proof: Induction is used on the number of phases, to show that at the beginning of phase k in Scenario 1(b) and the first j_i phases in Scenarios 1(a) and 2, there are exactly γ tasks of size π_{min} and one of π_{max} pending in the execution of OFF; therefore phase k is well defined. The specification of the phase depends on the relation between π_{max} and the number of π_{min}-tasks that are scheduled in that phase as well as the number of sequence of the current phase within the stage it is.

The invariant of $\gamma\pi_{min}$ and one π_{max} tasks, holds for the first phase of the execution by definition (initial injection). Looking at the definition of the phases in the two Scenarios, there are three cases:
(1) Phases of scenario 1(b)(i) end after OFF completes $(\kappa+1)\pi_{min}$ tasks, where there are exactly as many injected.
(2) Phases of scenario 1(b)(ii) end after OFF completes the π_{max}-task pending at the beginning, and at the end there is exactly one π_{max} injected.
(3) The first $j-1$ phases of scenarios 1(a) and 2, satisfy the $\Delta(k) < \frac{\gamma\pi_{min}}{s}$ 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 Scenarios 1(a) and 2, there are only some π_{min}-tasks pending in the execution of OFF. Looking first at scenario 1(a), phase j_i will end in π_{max} time, during which OFF will complete the π_{max}-task pending and after which no task will be injected for the next γ phases of length π_{min}. At the beginning of those phases, there will be exactly $\gamma - i < \gamma$ tasks of size π_{min}-pending in the execution of OFF, where $i = 1, 2, \ldots, \gamma$. Then, at the end of the last phase a new π_{min}-task will be injected and every π_{min} time there will be a crash and an injection of an π_{min}-task, causing the rest of the execution to have phases starting with one π_{min}-task pending for OFF. Then, in scenario 2, after the first j_i phases, the idea that follows is the same as in scenario 1(a), with the difference that ALG here is able to complete exactly $\gamma\pi_{min}$ 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. ■

Lemma 5.11. *The number of phases is infinite.*

Proof: First, by Lemma 5.10, consecutive phases are well defined. Moreover, they are finite (each has a finite duration), 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 π_{max}-task (scenarios 1(a),1(b)(ii) and 2), or some π_{min}-tasks (scenarios 1(a), 1(b)(i) and 2). Therefore in an infinite execution there is an infinite amount of phases. ■

Lemma 5.12. *Stages ending with phases of Scenario 1(a) or 2, are the last stages of any execution, causing the infinite latency competitiveness of ALG.*

Proof: First, note that by the definition of scenarios 1(a) and 2, after the completion of the pending π_{max}-task in phase j_i by OFF, there is no other π_{max}-task injected in the system and ALG was not able to complete it. Then, there are γ phases following, each of length π_{min}, with no task injections but the last one. This means that the γ tasks of size π_{min} pending at the beginning of phase j_i are now completed by OFF and its latency becomes zero. On the other hand, even if ALG completes some π_{min}-tasks as well, it will not be able to complete the π_{max} pending task, increasing its latency. After the γ^{th} phase of length π_{min} there are infinite phases of length π_{min} that end with a crash and a single π_{min}-task injection. This results to an infinite latency competitiveness for ALG.

Before scenario 1(a) or 2 occurs in an execution, all previous stages are of Scenario 1(b), either (i) or (ii). As long as Scenario 1(b)(i) happens, there are $\kappa+1$ tasks of size π_{min} completed by OFF that are restored by the injection of another $\kappa+1$ π_{min}-tasks at the end of the phase, while ALG is able to complete only κ of them, causing its pending π_{min}-tasks to increase by one in each such phase. On the other hand, if Scenario 1(b)(ii) happens both OFF and ALG manage to complete their pending π_{max}-task. This gives a latency to OFF equal to the time that the pending π_{min}-tasks are in the system. However, since ALG has accumulated more π_{min} pending tasks up to the current time, its latency remains at least equal to the latency of OFF. ■

Lemma 5.13. *A stage of Scenario 1(b) consists of various phases of Scenario 1(b)(i) and a final phase of Scenario 1(b)(ii). At the end of the phases of Scenario 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 Scenario 1(b)(ii) the latency of ALG is increased by 2^i, while the latency of OFF is bounded by the time the last γ tasks of size π_{min} were waiting in the system.*

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

Then, as soon as $(j_i-1)\pi_{min} \geq 2^i\pi_{max}+\gamma\pi_{min}$, i.e., Scenario 1(b)(ii), the phase has a duration of π_{max} time. This allows for both ALG and OFF to complete their pending

π_{max}-task, causing their latency to depend on their pending π_{min}-tasks. For OFF we know from Lemma 5.10, that it has only γ of them, which cannot have been injected in more than γ phases ago; hence the maximum latency will be $\gamma\pi_{min}$. On the other hand, we know that the current phase was preceded by $j_i - 1$ ones of Scenario 1(b)(i), after each of which ALG has been accumulating one more π_{min}-task. Therefore, at the beginning of the phase it has $j_i - 1$ tasks of size π_{min} more than OFF. Again, from Lemma 5.10 we know that each Scenario 1(b)(i) that preceded had a duration of maximum $\gamma\pi_{min}$ time. That, together with the fact that $(j_i - 1)\pi_{min} \geq 2^i\pi_{max} + \gamma\pi_{min}$, and the fact that within the current phase ALG may complete up to $\gamma\pi_{min}$-tasks, means that its latency will be increased to $2^i\pi_{max}$ (at the end of the previous phase of Scenario 1(b)(ii), it was $2^{i-1}\pi_{max}$). ■

Combining Lemmas 5.10 to 5.13, and the definition of the adversarial arrival and error patterns above, A and E respectively, the following theorem follows.

Theorem 5.11. *For any given π_{min}, π_{max} and s, if both conditions* **C1:** *$s < \rho$ and* **C2:** *$s < 1 + \gamma/\rho$ are satisfied, then no deterministic algorithm ALG is latency competitive when run with speedup $s \geq 1$ against an adversary that injects tasks of sizes in the range $[\pi_{min}, \pi_{max}]$, even in a system with one single machine.*

5.3.2. γ-Burst: an *optimal* algorithm for $M\langle 1, \left[1 + \frac{\gamma}{\rho}, \rho\right), 2\rangle$

Algorithm γ-Burst is proposed in this section, an online scheduling algorithm that achieves both 1-completed-load and latency competitiveness, as soon as condition **C2** does not hold, i.e., $s \geq 1 + \gamma/\rho$. More precisely, by considering only two task sizes, π_{min} and π_{max}, and running with speedup $s \in \left[1 + \frac{\gamma}{\rho}, \rho\right)$, the algorithm becomes competitive for all measures. (To recall the definition of parameter γ, see Section 5.3.1.)

Algorithm Description

Algorithm γ-Burst considers only two task sizes, π_{min} and π_{max}. It separates the pending tasks in two queues according to their size, say $\mathbf{P}_{\pi_{max}}$ and $\mathbf{P}_{\pi_{min}}$, and sorts each of them in ascending order according to their arrival time. This way, the first task in each queue is the one to be scheduled next, if one of *that* size is to be scheduled. Algorithm γ-Burst takes its scheduling decisions at the end of each *stage*, which also indicates 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 were decided to be scheduled at the beginning of the stage. The scheduling decisions are then taken as follows:

1. If $\mathbf{P}_{\pi_{max}} = \emptyset$, then γ-Burst schedules a π_{min}-task; the first task of the queue $\mathbf{P}_{\pi_{min}}$.
2. If $\mathbf{P}_{\pi_{min}} = \emptyset$, then it schedules a π_{max}-task; the first of the queue $\mathbf{P}_{\pi_{max}}$.
3. Else, if $\#P_{\pi_{min}} \geq \gamma$, it schedules γ consecutive π_{min}-tasks, followed by one π_{max}-task after their completion; the first γ tasks from $\mathbf{P}_{\pi_{min}}$ and then the first from $\mathbf{P}_{\pi_{max}}$.

4. Otherwise, if the previous decision stage was other than 4, it schedules one π_{min}-task; again, the first in the queue $\mathbf{P}_{\pi_{min}}$. However, while it remains in consecutive decision stages 4, it schedules tasks from the two queues interchangeably, $\mathbf{P}_{\pi_{min}}$ and $\mathbf{P}_{\pi_{max}}$; always the first task from the corresponding queue. Each such stage ends after a single task is completed.

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 including: a group of γ π_{min}-tasks followed by a π_{max}-task. Therefore, as shown in the detailed analysis that follows, the total size of $\gamma\pi_{min}$ amortizes the size of the π_{max}-task scheduled after each batch, which *may (or not)* be interrupted. What is more, tasks of both queues are getting executed often, hence the latency is not cut down by a task that is never scheduled by the algorithm.

Completed and Pending Load Competitiveness

Focusing first on the completed and pending load competitiveness of algorithm γ-Burst, the following theorem holds.

Theorem 5.12. *For any given π_{min}, π_{max} and speedup satisfying condition* **C1** $\wedge \neg$**C2***, i.e. $s \in [1 + \gamma/\rho, \rho)$, algorithm γ-Burst is 1-completed-load and 1-pending-load competitive. More precisely in $M\langle 1, [1 + \gamma/\rho, \rho), 2\rangle$, its completed load complexity is $\forall t, C_t(\gamma\text{-Burst}) \geq C_t(X) - (\pi_{max} + \pi_{min})$.*

Proof: Consider a time instant t' to be the first point in the execution of γ-Burst where the claim for the total completed load complexity does not hold, i.e. $C_{t'}(\gamma\text{-Burst}) < C_{t'}(X) - (\pi_{max} + \pi_{min})$. Let also a time instant $t^* < t'$ to be 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, $C_{t^*}(\gamma\text{-Burst}) \geq C_{t^*}(X) - (\pi_{max} + \pi_{min})$ holds. However, it could be the case that $C_{t^*}(\gamma\text{-Burst}) < C_{t^*}(X) - \alpha$, where $\alpha < \pi_{max} + \pi_{min}$. Let $T = (t^*, t']$ denote the interval between the two time instants. Then, look at the different cases for the length of the interval for each type of stage:

(a) *Stage of type 1 or 4 when scheduling a π_{min}-task.*
This holds when at time t^*, γ-Burst decides to schedule exactly one π_{min}-task before making its next decision. As it is the beginning of the last stage before t', there are two sub-cases to examine:

1) $|T| \leq \frac{\pi_{min}}{s}$.
In this case, γ-Burst will either be able to complete the π_{min}-task it scheduled at t^*, or not. Hence, it increases its completed load by at most π_{min} by time t'. On the same time, X may complete at most one task, of any size, that was scheduled before t^* (if at t^* there was no machine crash). Define now time instant $t^{**} < t^*$ to be the last time before t^* at which X scheduled this last completed task. This means that new interval $T' = [t^{**}, t')$ will have length $|T'|$ equal to either

π_{min} or π_{max}. The *worst* case is $|T'| = \pi_{max}$. Therefore:

$$\begin{aligned} C_{t'}(\gamma\text{-Burst}) &\geq C_{t^{**}}(\gamma\text{-Burst}) + \pi_{max} - \pi_{min}/s + \pi_{min} \\ &\geq C_{t^{**}}(X) - (\pi_{max} + \pi_{min}) + \pi_{max} - \pi_{min}/s + \pi_{min} \\ &\geq C_{t^{**}}(X) - \pi_{min}/s \geq C_{t'}(X) - \pi_{max} - \pi_{min}/s \\ &\geq C_{t'}(X) - (\pi_{max} + \pi_{min}) \end{aligned}$$

which contradicts the definition of time t'. The first inequality comes form the fact that within interval $[t^{**}, t^*)$ there will be up to $(\pi_{max} - \pi_{min}/s)$ total load completed by γ-Burst, and then in the interval $[t^*, t')$ up to π_{min} more.

2) $|T| > \frac{\pi_{min}}{s}$.

This case directly contradicts the definition of time instant t^*; γ-Burst would be able to take the next scheduling decision after the π_{min}-task was completed, marking the beginning of a new *last* stage before t'.

(b) *Stage of type 2 or 4 when scheduling a π_{max}-task.*

This holds when at time t^*, γ-Burst decides to schedule exactly one π_{max}-task before making its next decision. As it is the beginning of the last stage before t', there are again two sub-cases to examine for period T:

1) $|T| \leq \frac{\pi_{max}}{s}$.

In this case, γ-Burst is either able to complete the π_{max}-task scheduled or not; hence increasing its complete time by at most π_{max} by time t'. On the same time, X may complete at most one task, of any size, that was scheduled before t^*, plus a π_{min}-task within the interval T (if there is enough time left), resulting to a maximum completion time of $\pi_{min} + \pi_{max}$. Then, define time instant $t^{**} < t^*$, being the last time before t^* at which X scheduled this second-to-last completed task. This means that the new interval $T' = [t^{**}, t')$ will have length $|T'|$ equal to at most $\pi_{max} + \pi_{min}$. Therefore,

$$\begin{aligned} C_{t'}(\gamma\text{-Burst}) &\geq C_{t^{**}}(\gamma\text{-Burst}) + \pi_{max} + \pi_{min} - \pi_{max}/s + \pi_{max} \\ &\geq C_{t^{**}}(X) - (\pi_{max} + \pi_{min}) + 2\pi_{max} + \pi_{min} - \pi_{max}/s \\ &\geq C_{t^{**}}(X) + \pi_{max} - \pi_{max}/s \\ &\geq C_{t'}(X) - (\pi_{max} + \pi_{min}) + \pi_{max} - \pi_{max}/s \\ &\geq C_{t'}(X) - \pi_{min} - \pi_{max}/s \\ &\geq C_{t'}(X) - (\pi_{max} + \pi_{min}) \end{aligned}$$

which again contradicts the definition of time t'. The first inequality comes form the fact that within interval $[t^{**}, t^*)$ there will be up to $(\pi_{max} + \pi_{min} - \pi_{max}/s)$ total load completed by γ-Burst, and then in the interval $[t^*, t')$ up to π_{max} more.

2) $|T| > \frac{\pi_{max}}{s}$.

This case again contradicts directly the definition of time t^*; γ-Burst would be able to take the next scheduling decision after the π_{max}-task was completed, marking the beginning of a new *last* stage before t'.

(c) *Stage of type 3.*

This holds, when at time t^*, γ-Burst has enough tasks and decides to schedule γ π_{min}-tasks followed by a π_{max}. In this case there are five cases to examine for the period T:

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

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

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

In this case, γ-Burst is able to complete at most κ tasks of size π_{min}, where $\kappa = \left\lfloor \frac{\pi_{max}/s}{\pi_{min}} \right\rfloor = \left\lfloor \frac{\varrho}{s} \right\rfloor$ and $\kappa \geq 1$. On the same time, X is able to complete tasks of total size up to $\pi_{max} + \pi_{min}$ 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 $\kappa\pi_{min} \leq \frac{\pi_{max}}{s}$, $C_{t'}(\gamma\text{-Burst}) \geq C_{t'}(X) - (\pi_{max} + \pi_{min})$.

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

In this case, γ-Burst can only complete up to γ scheduled π_{min}-tasks; more precisely, $\kappa \leq \gamma$ of them, where $\kappa > 1$. On the same time, X is able to complete tasks of total size up to $\pi_{max} + \frac{\gamma\pi_{min}}{s}$ (one task of any size that was scheduled before t^*, and then up to total time equal to the length of the interval). Let time instant $t^{**} < t^*$ be the last time before t^* at which X scheduled that extra task. This means, that the new interval $T' = [t^{**}, t')$ will have length $|T'| \leq \pi_{max} + \frac{\gamma\pi_{min}}{s}$. Hence,

$$
\begin{aligned}
C_{t'}(\gamma\text{-Burst}) &\geq C_{t^{**}}(\gamma\text{-Burst}) + \pi_{max} + \frac{(\gamma-\kappa)\pi_{min}}{s} + \kappa\pi_{min} \\
&\geq C_{t^{**}}(X) - (\pi_{max} + \pi_{min}) + \pi_{max} + \frac{(\gamma-\kappa)\pi_{min}}{s} + \kappa\pi_{min} \\
&\geq C_{t^{**}}(X) + \frac{(\gamma-\kappa)\pi_{min}}{s} + (\kappa-1)\pi_{min} \\
&\geq C_{t'}(X) - \left(\pi_{max} + \frac{\gamma\pi_{min}}{s}\right) + \frac{(\gamma-\kappa)\pi_{min}}{s} + (\kappa-1)\pi_{min} \\
&\geq C_{t'}(X) - \pi_{max} - \frac{\kappa\pi_{min}}{s} + (\kappa-1)\pi_{min} \\
&\geq C_{t'}(X) - (\pi_{max} + \pi_{min}),
\end{aligned}
$$

which again contradicts the definition of time t'. The first inequality comes form the fact that within interval $[t^{**}, t^*)$ there will be up to $\left(\pi_{max} + \frac{(\gamma-\kappa)\pi_{min}}{s}\right)$ total load completed by γ-Burst, and then in the interval $[t^*, t')$ up to $\kappa\pi_{min}$ more.

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

In this case, γ-Burst completes the γ π_{min}-tasks and it is either able to complete the last π_{max}-task scheduled, or not. On the same time, X is able to complete tasks of total size up to $\pi_{max} + \frac{\gamma\pi_{min}+\pi_{max}}{s}$ (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 time instant $t^{**} < t^*$ be defined as the last time before t^* that X scheduled that extra task, and interval $T' = [t^{**}, t')$ with length $|T'| \leq \pi_{max} + \frac{\gamma\pi_{min}+\pi_{max}}{s}$. Hence,

$$\begin{aligned}
C_{t'}(\gamma\text{-Burst}) &\geq C_{t^{**}}(\gamma\text{-Burst}) + 2\pi_{max} + \gamma\pi_{min} \\
&\geq C_{t**}(X) - (\pi_{max} + \pi_{min}) + 2\pi_{max} + \gamma\pi_{min} \\
&\geq C_{t^{**}}(X) + \pi_{max} + (\gamma - 1)\pi_{min} \\
&\geq C_{t'}(X) - \left(\pi_{max} + \frac{\gamma\pi_{min} + \pi_{max}}{s}\right) + \pi_{max} + (\gamma - 1)\pi_{min} \\
&\geq C_{t'}(X) - \pi_{min} - \frac{\pi_{max}}{s} + \left(1 - \frac{1}{s}\right)\gamma\pi_{min} \\
&\geq C_{t'}(X) - (\pi_{max} + \pi_{min})
\end{aligned}$$

which contradicts the definition of time t'. The first inequality comes form the fact that within interval $[t^{**}, t^*)$ there will be up to $\left(\pi_{max} + \frac{\gamma\pi_{min}+\pi_{max}}{s} - \frac{\gamma\pi_{min}}{s}\right) = 2\pi_{max}$ total load completed by γ-Burst, and then in the interval $[t^*, t')$ up to $\gamma\pi_{min}$ more.

5) $|T| > \frac{\gamma\pi_{min}+\pi_{max}}{s}$.

This case contradicts directly the definition of time t^*; γ-Burst would be able to take the next scheduling decision after the $\gamma\pi_{min}$ and the π_{max}-task were completed, marking the beginning of a new *last* stage before t'.

All possible stages show contradiction of the initial claim. This means that algorithm γ-Burst is 1-completed-time competitive as claimed, with exact completed load $C_t(\gamma\text{-Burst}) \geq C_t(X) - (\pi_{max} + \pi_{min})$ at all time instants t. What is more, since $P_t(\gamma\text{-Burst}) = I_t - C_t(\text{ALG})$ at all time instants t, algorithm γ-Burst is also 1-pending-load competitive. ■

Latency Competitiveness

In order to analyze the latency of algorithm γ-Burst under the same speedup condition, $s \in [1 + \gamma/\rho, \rho)$, a basic invariant I_0 is first defined; it indicates the 1–1 relationship of the latency of π_{max}-tasks between the queues of γ-Burst and X, at any time in an execution. Invariants I_1 and I_2 are also defined, which characterize the general π_{max}-*latency* of γ-Burst with respect to the latency of X and its corresponding π_{min}-*latency*. (The latter concerns the latency of the algorithm's π_{min}-tasks, while the former concerns the latency of its π_{max}-tasks.)

I_0: For any $1 < i \leq z$, where $z = \min\{\#P_{\pi_{max}}(\gamma\text{-Burst}), \#P_{\pi_{max}}(X)\}$, the latency of the i^{th} π_{max}-task in the queue of γ-Burst is not bigger than the latency of the i^{th} π_{max}-task in the pending queue of X.

I_1: $L_t(\gamma\text{-Burst}, \pi_{max}) \leq L_t(X, \pi_{max})$: This means that at time t of the execution, the maximum latency of π_{max}-tasks that are pending in γ-Burst is at most equal to the maximum latency of π_{max}-tasks that are pending in X.

I_2: $L_t(\gamma\text{-Burst}, \pi_{min}) \leq \max\{L_t(X, \pi_{min}), L_t(X, \pi_{max})\}$: This means that the maximum latency of the pending π_{min}-tasks in the execution of γ-Burst is at most equal to the maximum of the latencies of the π_{min}-tasks and π_{max}-tasks that are pending in the execution of X at time t.

Before proving the necessary lemmas for the above invariants, recall that each new task is appended in the queue of pending tasks. The construction of the two queues $\mathbf{P}_{\pi_{max}}(\gamma\text{-Burst})$ and $\mathbf{P}_{\pi_{max}}(X)$ is defined more clearly, as follows: when a new π_{max}-task arrives in the system, it is put at the end of the corresponding queue, indexed as τ_{z+1}, where $z = \min\{\#P_{\pi_{max}}(\gamma\text{-Burst}), \#P_{\pi_{max}}(X)\}$ (as defined in the invariant I_0). At any time instant t, the π_{max}-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 π_{max}-tasks. It is also the one to be scheduled at the next time that γ-Burst, or X respectively, decides to schedule a π_{max}-task. The same holds for the queue of π_{min}-tasks.

Lemma 5.14. *When γ-Burst runs on a machine with speedup $s \in [1 + \gamma/\rho, \rho)$, the invariant I_0 holds at all times of the execution, i.e. $\forall t$, $L_t(\gamma\text{-Burst}, \tau_i) \leq L_t(X, \tau_i)$, where τ_i the i^{th} π_{max}-task in the corresponding queue.*

Proof: Let time instants t_k of an execution of γ-Burst, where $k = 0, 1, 2, \ldots$, be the time instants at which algorithm γ-Burst completed the k^{th} π_{max}-task. Focusing now on all t_k time instants and using the induction that follows, 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, $L_{t_0}(\gamma\text{-Burst}, \tau_i) \leq L_{t_0}(X, \tau_i)$.

Inductive Hypothesis. Assume that at time instant t_{k-1} the invariant holds, i.e., $L_{t_{k-1}}(\gamma\text{-Burst}, \tau_i) \leq L_{t_{k-1}}(X, \tau_i)$.

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

(a) $t_k = t_{k-1} + \frac{\pi_{max}}{s}$. In this case, it is known that a stage of type 2 or 4 has occurred between the two time instants, during which γ-Burst completed exactly one π_{max}-task. During interval $(t_{k-1}, t_k]$ algorithm X could have only completed one π_{max}-task, which was already scheduled before t_{k-1}. Hence, every π_{max}-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 X, they either remain with the same index (if no π_{max}-task is completed) or they move one position on the same way as well. From the induction hypothesis, it follows that $L_{t_k}(\gamma\text{-Burst}, \tau_i) \leq L_{t_k}(X, \tau_i)$.

(b) $t_k = t_{k-1} + \frac{\pi_{max} + \kappa\pi_{min}}{s}$, 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 X was able to complete at most one π_{max}-task as well, that was however scheduled before t_{k-1}. Recall that $s \geq \frac{\gamma\pi_{min} + \pi_{max}}{\pi_{max}}$. This means that $L_{t_k}(\gamma\text{-Burst}, \tau_i) \leq L_{t_k}(X, \tau_i)$, for τ_i begin the i^{th} π_{max}-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 π_{min} followed by a π_{max}-task were completed. On the same time X could only complete at most one π_{max}-task, that was however scheduled before t_{k-1}.

Hence again $L_{t_k}(\gamma\text{-Burst}, \tau_i) \leq L_{t_k}(X, \tau_i)$ as claimed.

3) Otherwise, it means that γ-Burst has been scheduling some π_{min}-tasks before scheduling the next π_{max}-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 π_{max}-tasks pending and hence it was choosing stages of type 1, until time instant $t^* \in (t_{k-1}, t_k)$ where some π_{max}-tasks were injected. Hence, even if X could have completed some π_{max}-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 π_{max}-tasks within interval $(t^*, t_k]$. Observe that $t_k - t^* = \frac{\pi_{max}}{s} < \pi_{max}$. Hence the invariant holds at time t_k as well, even if X completes some π_{max}-tasks. Observe that as long as γ-Burst has no π_{max}-tasks pending, $L_t(\gamma\text{-Burst}, \pi_{max}) = 0$.

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

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 $\mathbf{P}_{\pi_{max}}(\gamma\text{-Burst})$ and $\mathbf{P}_{\pi_{max}}(X)$, and hence all tasks already in the queues will keep their indexes i. Invariant I_0 will continue to hold; $L_{t^*}(\gamma\text{-Burst}, \pi_{max}) \leq L_{t^*}(X, \pi_{max})$. ■

Corollary 5.2. *When γ-Burst runs on a machine with speedup $s \in [1 + \gamma/\rho, \rho)$, the invariant I_1 holds at all times of the execution, i.e. $\forall t$, $L_t(\gamma\text{-Burst}, \pi_{max}) \leq L_t(X, \pi_{max})$.*

Proof: Using Lemma 5.14, invariant I_0 holds. This means that for $z = \min\{\#P_{\pi_{max}}(\gamma\text{-Burst}), \#P_{\pi_{max}}(X)\}$ and $1 < i \leq z$, the latency of the i^{th} π_{max}-task, denoted as τ_i, in the queue of γ-Burst is not bigger than the latency of the corresponding π_{max}-task in the queue of X. Recall that the π_{max}-latency of an algorithm is defined as the maximum latency of all π_{max}-tasks pending in its queue. Hence, at all time instants of an execution,

$$L_t(\gamma\text{-Burst}, \pi_{max}) = L_t(\gamma\text{-Burst}, \tau_1) \leq L_t(X, \tau_1) = L_t(X, \pi_{max},)$$

as claimed. ■

Lemma 5.15. *When γ-Burst runs on a machine with speedup $s \in [1 + \gamma/\rho, \rho)$, the invariant I_2 holds at all times of the execution, i.e. $\forall t$, $L_t(\gamma\text{-Burst}, \pi_{min}) \leq \max\{L_t(X, \pi_{min}), L_t(X, \pi_{max})\}$.*

Proof: Let time instants t_i of an execution of γ-Burst, where $i = 0, 1, 2, 3 \ldots$, represent the end of the i^{th} stage. Focusing on these t_i and using induction, one can prove 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; $L_{t_0}(\gamma\text{-Burst}, \pi_{min}) \leq \max\{L_{t_0}(X, \pi_{min}), L_{t_0}(X, \pi_{max})\}$.

Inductive Hypothesis: Assume that the invariant holds at a time instant t_{i-1}, i.e., $L_{t_{i-1}}(\gamma\text{-Burst}, \pi_{min}) \leq \max\{L_{t_{i-1}}(X, \pi_{min}), L_{t_{i-1}}(X, \pi_{max})\}$.

Inductive Step: To show that it will still hold at time t_i, one needs to consider a few cases regarding the i^{th} stage:

(a) *Its length is equal to* π_{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 π_{min}-task while X is not able to complete any task scheduled within the stage. By the definition of speedup, $s > 1$, and hence X does not have time to complete even a π_{min}-task within at most π_{min}/s time. However, it may complete a task that was scheduled before the beginning of the stage, either a π_{min} or a π_{max}-task. At the beginning of the stage, the π_{min}-latency of γ-Burst was $L_{t_{i-1}}(\gamma\text{-Burst}, \pi_{min}) \leq \max\{L_{t_{i-1}}(X, \pi_{min}), L_{t_{i-1}}(X, \pi_{max})\}$, and by time t_i X may only complete up to one task. Hence, at least one of its partial latencies (π_{min}-latency or π_{max}-latency) will be increased by π_{min}/s and thus the π_{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 π_{min}-task, its π_{min}-latency might decrease or stay the same as at time t_{i-1} depending on the arrival time of the first π_{min}-task that is still pending at time t_i. Therefore, $L_{t_i}(\gamma\text{-Burst}, \pi_{min}) \leq L_{t_{i-1}}(\gamma\text{-Burst}, \pi_{min})$. Combining it with the inductive assumption, the following holds:

$$\begin{aligned} L_{t_i}(\gamma\text{-Burst}, \pi_{min}) &\leq \max\{L_{t_{i-1}}(X, \pi_{min}), L_{t_{i-1}}(X, \pi_{max})\} \\ &< \max\left\{L_{t_{i-1}}(X, \pi_{min}) + \frac{\min}{s}, L_{t_{i-1}}(X, \pi_{max}) + \frac{\pi_{min}}{s}\right\} \\ &\leq \max\{L_{t_i}(X, \pi_{min}), L_{t_i}(X, \pi_{max})\}. \end{aligned}$$

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

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

$$\begin{aligned} L_{t_i}(\gamma\text{-Burst}, \pi_{min}) &= L_{t_{i-1}}(\gamma\text{-Burst}, \pi_{min}) + \pi_{max}/s \\ &\leq L_{t_{i-1}}(X, \pi_{max}) + \pi_{max}/s = L_{t_i}(X, \pi_{max}) \\ &\leq \max\{L_{t_i}(X, \pi_{min}), L_{t_i}(X, \pi_{max})\}. \end{aligned}$$

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

In this case the stage is of type 3 from the algorithm description. During this stage γ-Burst executes γ π_{min}-tasks followed by a π_{max}-task. 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 π_{min}-tasks and *no* π_{max}-task within the stage. The π_{min}-latency of γ-Burst may decrease due to the γ π_{min}-tasks that were completed. However, X's π_{min}-latency may also decrease, since up to the same number of π_{min}-tasks may be completed. Note that even in the case that it does not decrease, X's π_{max}-latency will increase by the length of the period, i.e. $\frac{\gamma\pi_{min}+\pi_{max}}{s}$. This increases the maximum of the two partial latencies of X and hence the following holds:

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

(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 π_{max}-task, X would have been able to complete at most $\lfloor \frac{\rho}{s} \rfloor$ tasks of processing time π_{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 π_{min}-tasks that γ-Burst was able to complete, X's π_{max}-latency will increase by the actual length of the stage. Hence, even if the π_{min}-latency of both γ-Burst and X decreases, the following will hold:

$$L_{t_i}(\gamma\text{-Burst}, \pi_{min}) \leq \max\{L_{t_i}(X, \pi_{min}), L_{t_i}(X, \pi_{max})\}.$$

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

Theorem 5.13. *For any given π_{min}, π_{max} and speedup satisfying condition* **C1** $\wedge \neg$**C2**, *i.e. in $M\langle 1, [1 + \gamma/\rho, \rho), 2\rangle$, algorithm γ-Burst is 1-latency competitive, i.e., $L_t(\gamma\text{-Burst}) \leq L_t(X)$.*

Proof: From Lemmas 5.2 and 5.15, the latency of γ-Burst is

$$\begin{aligned} L_t(\gamma\text{-Burst}) &= \max\{L_t(\gamma\text{-Burst}, \pi_{max}), L_t(\gamma\text{-Burst}, \pi_{min})\} \\ &\leq \max\{L_t(X, \pi_{max}), L_t(X, \pi_{min})\} \\ &= L_t(X) \end{aligned}$$

as claimed, which translated to 1-latency competitiveness. ■

5.3.3. *Sufficient* speedup conditions

In the following parts of this subsection it will be shown, by proposing algorithms, that the condition $s \geq \min\{\rho, 1 + \gamma/\rho\}$ is not only *necessary* for achieving competitiveness, but also

sufficient; as soon as it is TRUE, there exists some algorithm that is competitive, some even *optimal*.

It has already been shown in the previous section, that if the condition is FALSE, it is enough to have NO pending-load or latency competitiveness, and NO 1-completed-load competitiveness. Since this condition on s depends on γ, which implicitly depends on s, this section studies in more detail the bounds for competitiveness and non-competitiveness that relate only s and ρ. Thus, one can name the exact amount of speedup sufficient for competitiveness.

Upper bound on the speedup for non-competitiveness

Recall that the ratio $\rho = \pi_{max}/\pi_{min} \geq 1$. The following properties of ρ guarantee the above condition. From the first part of the condition for non-competitiveness, Condition **C1**, it must hold that $s < \rho$. From the second part, Condition **C2**, we must have

$$s < 1 + \gamma/\rho = 1 + \left\lceil \frac{\rho - s}{s-1} \right\rceil /\rho = 1 + \left(\left\lceil \frac{\rho - 1}{s-1} \right\rceil - 1 \right) /\rho,$$

where the second equality follows from $\lceil \frac{\rho-s}{s-1} \rceil = \lceil \frac{\rho-1}{s-1} \rceil - 1$. This, leads to

$$s < \frac{\lceil \frac{\rho-1}{s-1} \rceil + \rho - 1}{\rho}. \tag{5.6}$$

Let s_1 be the smallest speedup that satisfies Eq. 5.6, then a lower bound on s_1 can be found by removing the ceiling, as

$$s_1 \geq \frac{\frac{\rho-1}{s_1-1} + \rho - 1}{\rho} \implies s_1 \geq 2 - 1/\rho.$$

Summarizing, if $s < \rho$, then the first part of the condition for non-competitiveness (Condition **C1**) holds, and if $s < 2 - 1/\rho$, then the second part of the condition for non-competitiveness (Condition **C2**) holds. It can be shown that $\rho \geq 2 - 1/\rho$ for $\rho \geq 1$. Hence, the following result holds.

Theorem 5.14. *Let $\rho \geq 1$. In order to have non-competitiveness, it is sufficient to set $s < 2 - 1/\rho$.*

Smallest speedup for competitiveness

As mentioned above, in order to have competitiveness, it is sufficient that conditions **C1** and **C2** do not hold simultaneously. This means that at least one of the conditions (¬**C1**) $s \geq \rho$, or (¬**C2**) $s \geq 1 + \gamma/\rho$, must hold, where $\gamma = \max\{\lceil \frac{\rho-s}{s-1} \rceil, 0\}$. To satisfy condition (¬**C1**), the speedup s must satisfy $s \geq \rho = \frac{\pi_{max}}{\pi_{min}}$. Hence, the smallest value of s that guarantees that (¬**C1**) holds is $s_1 = \rho$. In order to satisfy condition (¬**C2**), when condition (¬**C1**) is not satisfied

(observe that when (¬**C1**) holds, $\gamma = 0$),

$$s \geq \frac{\lceil \frac{\rho-1}{s-1} \rceil + \rho - 1}{\rho} . \tag{5.7}$$

Let s_2 be the smallest speedup that satisfies Eq. 5.7; then an upper bound can be obtained by adding one unit to the expression in the ceiling

$$s_2 < \frac{\frac{\rho-1}{s_2-1} + 1 + \rho - 1}{\rho} \implies s_2 < 1 + \sqrt{1 - 1/\rho} .$$

Let $s_2^+ = 1 + \sqrt{1 - 1/\rho}$. Then, in order to guarantee competitiveness, it is enough to choose any $s \geq \min\{s_1, s_2\}$. Since there is no simple form of the expression for s_2, s_2^+ can be used instead, to be safe. Then:

Theorem 5.15. *Let $\rho \geq 1$. In order to have competitiveness, it is sufficient to set $s = s_1 = \rho$ if $\rho \in [1, \varphi]$, and $s = s_2^+ = 1 + \sqrt{1 - 1/\rho}$ if $\rho > \varphi$, where $\varphi = \frac{1+\sqrt{5}}{2}$ is the golden ratio.*

Proof: As mentioned before, a sufficient condition for competitiveness is $s \geq \min\{s_1, s_2^+\}$. Consider $s_1 = s_2^+$, i.e., $\rho = 1 + \sqrt{1 - 1/\rho}$. Solving the equation, $\rho = 1$ or $\rho = \frac{1+\sqrt{5}}{2}$. One can discard the first solution since what is considered is the case of at least two different task sizes. It is easy then to verify that $s_1 = \rho \leq s_2^+$ if $\rho \leq \varphi$. ■

5.4. Competitiveness of popular algorithms

In this section, four of the most widely used algorithms in scheduling are analyzed, in order to compare their fault-tolerant properties under worst-case task arrivals and machine crashes and restarts, as well as their performance for the three efficiency metrics. These algorithms are:
(1) *Longest In System* (LIS): schedules the task the has been waiting the longest; i.e., it follows the FIFO *(First In First Out)* policy,
(2) *Shortest In System* (SIS): schedules the task that has arrived latest; i..e, it follows the LIFO *(Last In First Out)* policy,
(3) *Largest Processing Time* (LPT): schedules the task with the biggest size, and
(4) *Shortest Processing Time* (SPT): schedules the task with the smallest size.

Table 5.2 summarizes the results obtained for the four algorithms. In Section 5.1, and in particular by Theorem 5.6, it has been shown that for speedup at least $1 + \rho$, any work-conserving deterministic scheduling algorithm can achieve optimal competitiveness. However, it is shown here, that for specific cases of speedup less than $1 + \rho$, some of the popular algorithms considered may achieve good competitive ratios; and even reach optimality. Observe, that different algorithms scale differently with respect to the speedup used; with the increase of the machine speed the competitive performance of each algorithm changes in a different way.

Alg.	Model	Completed Load, $\mathcal{C}$	Pending Load, $\mathcal{P}$	Latency, $\mathcal{L}$	Thm.
LIS	$M\langle 1, s < \rho, 2\rangle$	0	∞	∞	5.16, 5.23
	$M\langle 1, s \in [\rho, 1+1/\rho), \infty\rangle$	$[1/\rho, \frac{1}{2}+\frac{1}{2\rho}]$	$[\frac{1+\rho}{2}, \rho]$	$(0, 1]$	5.6, 5.19, 5.24
	$M\langle 1, s \in [\max\{\rho, 1+\frac{1}{\rho}\}, 2), \infty\rangle$	$[1/\rho, s/2]$	$[\frac{s}{2(s-1)}, \rho]$	$(0, 1]$	5.6, 5.19, 5.24
	$M\langle 1, s \geq \max\{\rho, 2\}, \infty\rangle$	1	1	$(0, 1]$	5.6, 5.19, 5.24
SIS	$M\langle 1, s < \rho, 2\rangle$	0	∞	∞	5.16, 5.22
	$M\langle 1, s \in [\rho, 1+1/\rho), \infty\rangle$	$\frac{1}{\rho}$	ρ	∞	5.6, 5.20, 5.22
	$M\langle 1, s \in [1+1/\rho, 1+\rho), \infty\rangle$	$[1/\rho, s/(1+\rho)]$	$[\frac{1}{s}+\frac{\rho}{1+\rho}, \rho]$	∞	5.6, 5.20, 5.22
	$M\langle 1, s \geq 1+\rho, \infty\rangle$	1	1	∞	5.6, 5.20, 5.22
LPT	$M\langle 1, s < \rho, 2\rangle$	0	∞	∞	5.16, 5.22
	$M\langle 1, s \geq \rho, \infty\rangle$	1	1	∞	5.21, 5.22
SPT	$M\langle 1, s < \rho, 2\rangle$	$[\frac{1}{2+\rho}, \frac{\lfloor(s-1)\rho\rfloor+1}{\lfloor(s-1)\rho\rfloor+1+\rho}]$	∞	∞	5.17, 5.18, 5.22
	$M\langle 1, s \geq \rho, \infty\rangle$	1	1	∞	5.21, 5.22

Table 5.2: Detailed metric comparison of the four widely-used scheduling algorithms, studied in detail for different ranges of speedup. The last column provides the theorem numbers where the results of the corresponding row can be found. Recall 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.

Nonetheless, it is difficult to say that one of the algorithms is overall better than the rest. It will be shown, that with the exception of algorithm SPT, no algorithm is competitive in any of the three metrics considered when $s < \rho$; in particular, it is competitive in terms of completed-load when tasks have only two possible sizes. Recall that SPT is even completed-load competitive without speedup in this case. It will also be shown, that in terms of latency, only algorithm LIS is competitive, when $s \geq \rho$, which may not be too surprising since it gives priority to the tasks that have been waiting the longest in the system. An interesting observation is that algorithms LPT and SPT become 1-completed and 1-pending load competitive as soon as $s \geq \rho$, whereas LIS and SIS require greater speedup. In some sense, this demonstrates the differences between two classes of polices: ones that give priority to tasks based on the *arrival time* and ones that give priority based on the required *task size*.

5.4.1. Completed and pending load competitiveness

A detailed analysis of the four algorithms with respect to the completed and pending load metrics, is presented here. The results are separated in two groups; the ones of speedup $s < \rho$, and the ones of speedup $s \geq \rho$.

Speedup $s < \rho$

First, some negative results are given, whose proofs involve specifying the combinations of arrival and error patterns that force the claimed *bad* performances of the algorithms. And then, some positive results for SPT, the only algorithm that can achieve a non-zero completed-load competitiveness under some circumstances.

Lemma 5.16. *When algorithms LIS and LPT run on a single machine 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 the same combination of algorithm X, arrival and error patterns A and E, be used to prove the non-competitiveness of both algorithms. Consider an infinite arrival pattern which injects one π_{max}-task at the beginning of the execution, $t = 0$, and after that it keeps injecting one π_{min}-task every π_{min} time. Consider also an infinite error pattern that sets instantaneous machine failure points (a crash immediately followed by a restart) at time instants $t_i = i \cdot \pi_{min}$, where $i = 1, 2, \ldots$.

It can be easily seen, that an algorithm X running with no speedup ($s = 1$), will be able to complete the π_{min}-tasks injected, while neither LIS nor LPT will manage to complete any task, running with speedup $s < \rho$, since they will both insist on scheduling the π_{max}-task injected at the beginning. In an interval of length π_{min}, algorithm X is able to complete a π_{min}-task but neither LIS nor LPT can complete the π_{max}-task since it needs time $\frac{\pi_{max}}{s} > \pi_{min}$. This means that the number of pending tasks in the queues of both LIS and LPT will be continuously increasing with time, 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 π_{max} and one π_{min} tasks. As for the total size of completed tasks, $C(\text{LIS}, \mathsf{A}, \mathsf{E}) = C(\text{LPT}, \mathsf{A}, \mathsf{E}) = 0$ 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 5.17. *When algorithm SIS runs on a single machine 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: The proof is divided in two parts giving different combinations of arrival and error patterns for the completed load and the pending load respectively.

First, consider a combination of arrival and error patterns A and E that behave as follows: let 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 π_{min}-task followed by a π_{max}-task. This creates *alive* intervals $[t_i, t_{i+1})$ of length π_{min}.

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

Now, consider another combination of arrival and error patterns A' and E' respectively, as well as an algorithm X'. Let 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 π_{min}-tasks injected followed by a π_{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 π_{min} between $t_{k'}$ and $t_{k'+1}$.

The arrival pattern A' causes algorithm SIS to schedule the last π_{max}-task injected right after time instant $t_{k'}$. However, since all alive intervals are of length π_{min} and $s < \rho$, created by the error pattern E', algorithm SIS can never complete the π_{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 π_{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 π_{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 5.16 and 5.17 the following theorem holds.

Theorem 5.16. *NONE of the three algorithms LIS, LPT and SIS is competitive when run on a single machine with speedup $s < \rho$, with respect to completed or pending load, even in the case of only two task sizes (i.e., π_{min} and π_{max}).*

As mentioned, for algorithm SPT one is able to prove a non-zero completed-load competitiveness when $s < \rho$. This will be later justified by the fact that for two task sizes SPT guarantees a positive completed-load competitiveness (see Theorem 5.18). Nonetheless, the following upper bound restriction for the completed-load of algorithm SPT is proven now.

Theorem 5.17. *For a single machine run with 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 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 one can fix $\gamma = \lfloor(s-1)\rho\rfloor + 1$.

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

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

Looking at the pending load at such points, it is easily observed that SPT's is constantly increasing, while X is able to have pending load zero; $P_{t_k}(\text{SPT}, \mathsf{A}, \mathsf{E}) = k\pi_{max}$ but $P_{t_k}(X, \mathsf{A}, \mathsf{E}) = 0$. As a result, a maximum completed-load competitive ratio $\mathcal{C}(\text{SPT}) \leq \frac{\gamma}{\gamma+\rho} = \frac{\lfloor(s-1)\rho\rfloor+1}{\lfloor(s-1)\rho\rfloor+1+\rho}$ is achieved, and no pending load competitive ratio; $\mathcal{P}(\text{SPT}) = \infty$. ■

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

Theorem 5.18. *If tasks can be of only two sizes (π_{min} and π_{max}), algorithm SPT can achieve a completed-load competitive ratio $\mathcal{C}(SPT) \geq \frac{1}{2+\rho}$, for a machine run with 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: Assume fixed arrival and error patterns A and E respectively, as well as an algorithm X, and look at any time t in the execution of SPT. Let τ be a task completed by X by time t (i.e., $\tau \in \mathbf{C}_t(X)$), where t_τ is the time τ was scheduled and $f(\tau) \leq t$ the time it completed its execution. Task τ is associated with the following tasks in set $\mathbf{C}_t(\text{SPT})$: (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 $\mathbf{C}_t(X)$ is associated to some task in $\mathbf{C}_t(\text{SPT})$, but it will be shown now that most tasks are. In fact, it is shown that the aggregate sizes of the tasks in $\mathbf{C}_t(X)$ that are not associated with any task in $\mathbf{C}_t(\text{SPT})$ is at most π_{max}. More specifically, there is only one task execution of a π_{max}-task by SPT, namely w, such that the π_{min}-tasks scheduled and completed by X concurrently with the execution of w fall in this class.

Consider the generic task $\tau \in \mathbf{C}_t(X)$ from above, and look at the following cases:

- If $\tau \in \mathbf{C}_t(\text{SPT})$, then task τ is associated at least with itself in the execution of SPT, regardless of τ's size.
- If $\tau \notin \mathbf{C}_t(\text{SPT})$, 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 $\mathbf{C}_t(\text{SPT})$. 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 $\tau \notin \mathbf{C}_t(\text{SPT})$, it means that τ is not completed by SPT in the interval $[t^*, t]$ either. Hence, SPT never schedules a π_{max}-task in the interval $[t^*, t]$, and the case that a task from $\mathbf{C}_t(X)$ is not associated to any task in $\mathbf{C}_t(\text{SPT})$ cannot occur again in that interval.

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

To evaluate the sizes of the tasks in $\mathbf{C}_t(X)$ associated to a task in $w \in \mathbf{C}_t(\text{SPT})$, consider any task w successfully completed by SPT at a time $f(w) \leq t$. Task w can be associated at most with itself and all the tasks that X scheduled within the interval $T_w = [f(w) - \pi(w), f(w)]$. The latter set can include tasks whose aggregate size is at most $\pi(w) + \pi_{max}$, since the first such task starts its execution no earlier than $f(w) - \pi(w)$ and in the extreme case a π_{max}-task could have been scheduled at the end of T_w and completed at $t_w + \pi_{max}$. Hence, if task w is a π_{min}-task, it will be associated with tasks completed by X that have total size at most $2\pi_{min} + \pi_{max}$, and if w is a π_{max}-task, it will be associated with tasks completed by X that have a total size of at most $3\pi_{max}$. Observe that $\frac{\pi_{min}}{2\pi_{min}+\pi_{max}} < \frac{\pi_{max}}{3\pi_{max}}$. As a result, $C_t(\text{SPT}, \mathsf{A}, \mathsf{E}) \geq \frac{\pi_{min}}{2\pi_{min}+\pi_{max}} C_t(X, \mathsf{A}, \mathsf{E}) - \pi_{max} = \frac{1}{2+\rho} C_t(X) - \pi_{max}$, and thus the completed-load competitive ratio $\mathcal{C}(\text{SPT}) \geq \frac{1}{2+\rho}$ claimed. ■

Conjecture 5.1. *The above lower bound on completed load, still holds in the case of any bounded number of task sizes in the range* $[\pi_{min}, \pi_{max}]$.

Speedup $s \geq \rho$

First, recall that in Section 5.1, and in particular by Theorem 5.5, it is 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 these ratios can be improved. Starting from some negative results, the focus is turned first on the two policies that schedule tasks according to their arrival time, algorithms LIS and SIS.

Lemma 5.18. *When algorithm LIS runs on a machine with speedup* $s \in [\rho, 1 + 1/\rho)$*, it has a completed-load competitive ratio* $\mathcal{C}(\text{LIS}) \leq \frac{1}{2} + \frac{1}{2\rho}$ *and a pending-load competitive ratio*

$\mathcal{P}(LIS) \geq \frac{1+\rho}{2}$.

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

This behavior results to the following execution. There are only active phases of size π_{max}, during which an algorithm X can successfully execute the π_{max} task injected, while LIS is forced to schedule the tasks in the order they arrive. Observe that, since $s < 1 + 1/\rho = (\pi_{min} + \pi_{max})/\pi_{max}$, LIS is able to complete only one task in each phase, either a π_{min}-task or a π_{max}-task. Observe also, that after k phases, where k is a multiple of 2, there will be exactly k tasks of size π_{min} pending in the queue of X, while LIS will have pending half of the tasks injected, half of which are of size π_{min} and the other half π_{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 5.19. *When algorithm LIS runs of a machine 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 $s \in [1 + 1/\rho, 2)$ and a combination of arrival and error patterns A and E, as well as algorithm X. Consider also tasks of sizes π_{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 $\pi = \varepsilon\frac{\pi_{min}}{s-1}$, where $\varepsilon \in (0, 1)$.

Patterns A and E behave as follows: let 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 π_{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 π_{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 π_{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 π_{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 $P_{t_k}(\text{LIS}, \mathsf{A}, \mathsf{E}) = \frac{k(\pi_{min}+\pi)}{2}$ while $P_{t_k}(X, \mathsf{A}, \mathsf{E}) = k\pi_{min}$. 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. Assume otherwise for the completed load competitiveness, 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, which completes the proof. ■

Lemma 5.20. *When algorithm LIS runs on a machine 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 focus 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 $C_{t'}(\text{LIS}) \geq C_{t'}(X) - \pi_{max}$.

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 $C_i(X)$ and $C_i(\text{LIS})$ denote the load completed in period i by X and LIS respectively. The periods in which $C_i(\text{LIS}) = 0$ can be safely discarded, since $C_i(X) = 0$ will hold as well (recall that $s \geq \rho$). After discarding these periods, renumber the rest in sequence from 1 to k'.

To prove the theorem, one needs to be 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.,

$$C_j(X) > C_j(\text{LIS}), \tag{5.8}$$

then the following holds,

$$\frac{C_j(\text{LIS}) + \pi(\tau_{j+1})}{s} > C_j(X) \Rightarrow s \cdot C_j(X) - \pi(\tau_{j+1}) < C_j(\text{LIS}), \tag{5.9}$$

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 $\forall j \in [2, k']$,

$$C_j(\text{LIS}) \geq \pi(\tau_j). \tag{5.10}$$

From equations 5.8 and 5.9, it holds that $C_j(X) > C_j(\text{LIS}) > s \cdot C_j(X) - \pi(\tau_{j+1})$. Since $s \geq 2$,

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

What is more, from equations 5.8 and 5.10, $C_j(X) > \pi(\tau_j)$ holds, and hence

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

Combining this with equation 5.9:

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

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

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

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 the following can be claimed:

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

which completes the proof for both completed-load and pending-load competitive ratios being optimal for algorithm LIS when speedup $s \in [2, 1+\rho)$, i.e., $\mathcal{C}(\text{LIS}) \geq 1$ and $\mathcal{P}(\text{LIS}) \leq 1$. ■

Combining Lemmas 5.18, 5.19, 5.20 and Theorem 5.6 the following theorem holds.

Theorem 5.19. *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 5.21. *When algorithm SIS runs on a machine 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 $s \in [\rho, 1 + 1/\rho)$ and a combination of arrival and error patterns A and E, as well as algorithm X as follows: At the beginning of the execution there is a π_{max}-task injected, followed by a π_{min}-task. After every period of π_{max} time there is a crash and restart of the machine, followed by the same injection sequence (a π_{max}-task and then a π_{min}-task).

This behavior results to the following execution. There are only active phases of size π_{max}, during which an algorithm X can successfully execute the π_{max} 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 π_{min}-task injected. Observe also, that after k phases, there will be exactly k tasks of size π_{min} pending in the queue of X, while SIS will have pending k tasks of size π_{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 5.22. *When algorithm SIS runs on a machine 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 $s \in [1+1/\rho, 1+\rho)$ and a combination of arrival and error patterns A and E, as well as algorithm X. Consider tasks of sizes π_{max}, π_{min} and π, where $\pi \in (\pi_{min}, \pi_{max})$, such that $\pi < \frac{\pi_{min}+\pi_{max}}{s}$. Note that, such a value π always exists since $s \in [1 + 1/\rho, 1 + \rho)$. More specifically, let $\pi = \varepsilon\frac{\pi_{min}+\pi_{max}}{s}$, where $\varepsilon \in (0, 1)$.

Patterns A and E behave as follows: Let time instants $t_k = t_{k-1}+\pi$, where k is an increasing positive integer ($k = 0, 1, 2, \dots$), 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 π_{max}-task, followed by one π_{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 π_{min}-task followed by the latest π_{max}-task, and hence being able to complete only the π_{min}-task. Therefore, at the end of each alive interval, $C_{t_k}(\text{SIS}, \mathsf{A}, \mathsf{E}) = k\pi_{min}$, $C_{t_k}(X, \mathsf{A}, \mathsf{E}) = k\pi$, $P_{t_k}(\text{SIS}, \mathsf{A}, \mathsf{E}) = k(\pi + \pi_{max})$ and $P_{t_k}(X, \mathsf{A}, \mathsf{E}) = k(\pi_{min} + \pi_{max})$. 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

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

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

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 5.21, 5.22 and Theorem 5.6, the following theorem holds.

Theorem 5.20. *Algorithm SIS has a completed-load competitive ratio*

$$\mathcal{C}(SIS) \leq \begin{cases} 1/\rho & s \in [\rho, 1 + 1/\rho) \\ \frac{s}{1+\rho} & s \in [1 + 1/\rho, 1 + \rho) \end{cases}, \text{ and } \mathcal{C}(SIS) \geq 1 \text{ when } s \geq 1 + \rho.$$

It also has a pending-load competitive ratio

$$\mathcal{P}(SIS) \geq \begin{cases} \rho & s \in [\rho, 1 + 1/\rho) \\ \frac{1}{s} + \frac{\rho}{1+\rho} & s \in [1 + 1/\rho, 1 + \rho) \end{cases}, \text{ and } \mathcal{P}(SIS) \leq 1 \text{ when } s \geq 1 + \rho.$$

In contrast with these negative results, some positive results for algorithms LPT and SPT can be shown. 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 5.23. *When algorithm LPT runs on a machine 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 5.3 in Section 5.1, the number of completed tasks of any work conserving algorithm under any combination of arrival and error patterns A and E, and speedup $s \geq \rho$, is never smaller than the number of completed tasks of X. The same holds for algorithm LPT, $\#C_t(\text{LPT}) \geq \#C_t(X)$.

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, $C_t(\text{LPT}, \mathsf{A}, \mathsf{E}) \geq C_t(X, \mathsf{A}, \mathsf{E})$ 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, recall 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, which is equal to the total injected load; i.e., $\forall t, A, E, X$, $C_t(\text{ALG}) + P_t(\text{ALG}) = C_t(X) + P_t(X) = I_t(\mathsf{A})$. This holds for LPT as well, hence $C_t(\text{LPT}) + P_t(\text{LPT}) = I_t(\mathsf{A})$. It has already been shown that $C_t(\text{LPT}) \geq C_t(X)$. Hence replacing with the corresponding expressions for the pending load, $I_t(\mathsf{A}) - P_t(\text{LPT}) \geq I_t(\mathsf{A}) - P_t(X)$ which leads to $P_t(\text{LPT}) \leq P_t(X)$ and $\mathcal{P}_t(\text{LPT}) \leq 1$ as claimed. ■

Lemma 5.24. *When algorithm SPT runs on a machine 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: Consider any execution of algorithm SPT running speedup $s \geq \rho$ under any arrival and error patterns A and E respectively. It will be shown, that at all times in the execution, the completed load of SPT is more than that of an algorithm X, i.e., $\forall t, C_t(\text{SPT}) \geq C_t(X)$.

By contradiction, 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^-, $C_{t^-}(\text{SPT}) \geq C_{t^-}(X)$.

First, consider the case where X has completed a π_{min}-task. This means that during the interval $(t - \pi_{min}, t)$ no machine failure has occurred and hence algorithm SPT was also able to complete some tasks. Let t^* be the last time in $(t - \pi_{min}, t)$ that SPT completes a task. 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 π_{min}. Hence

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

which contradicts the initial assumption.

Then, consider the case where X has completed a π_{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 π_{max}. Hence

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

which again contradicts the initial assumption.

This, shows 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 5.23 and 5.24 the following theorem holds.

Theorem 5.21. *When algorithms LPT and SPT run on a machine 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 $\mathcal{P}(SPT) \leq 1$.*

5.4.2. 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 5.22. *NONE of the algorithms LPT, SIS or SPT can be competitive with respect to the latency for any speedup $s \geq 1$. That is, $\mathcal{L}(LPT) = \mathcal{L}(SIS) = \mathcal{L}(SPT) = \infty$.*

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

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

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

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

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

then $\pi x^k + \pi(w) > s\pi x^{k-1}$ and hence $\frac{\pi x^k + \pi(w)}{s} > \pi x^{k-1}$. Then, ALG again will not be able to complete w in the interval.

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

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

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

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

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 $\pi(x^{R-1} + x^R) \geq \pi(w)$.

Since this contradicts the assumption, $\mathcal{L}(\text{ALG}) = \infty$, as claimed. ■

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

Proof: Let us consider a combination of arrival and error patterns A and E, and algorithm X. Pattern A is an infinite arrival pattern that injects a π_{min}-task at the beginning of the execution, followed by a π_{max}-task (after an infinitesimally small time ε). After that, it injects only π_{min}-tasks, one every π_{min} time. Pattern E sets the first crash/restart instant at $\pi_{max} + \varepsilon$ time from the beginning and then every π_{min} period of time, creating a *phase* (time period between a restart and the next crash) of length π_{max} followed by infinite phases of length π_{min}. These patterns

allow an algorithm X to execute successfully the π_{max}-task injected at the beginning on the first phase, while algorithm LIS's policy to schedule the one that was injected earlier in the system forces it to schedule the π_{min}-task. Even though it will also be executed, the π_{max}-task scheduled next will never be completed in any of the following phases since they are all of size π_{min} and $\frac{\pi_{max}}{s} > \pi_{min}$. This means that algorithm's LIS latency will increase to infinity with time, while X's latency will remain bounded (each task is completed at most $\pi_{max} + \pi_{min}$ time after its injection).

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

Theorem 5.24. *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 and 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 5.3, we know that the there are never more pending tasks in the queue of LIS than that of X and hence $\#P_t(\text{LIS}) = x + 1 \leq \#P_t(X)$. 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. ■

Chapter 6

Multiple Machines

This chapter studies the general model of multiple machines prone to failures ($m > 1$), sharing the repository exactly as described in the model (Chapter 3). Observe, that all the negative results shown in Chapter 5 still hold for this setting; an adversary can crash (and keep crashed) all the machines except one, thus creating the same scenarios used to prove the corresponding lower or upper bounds of the single machine case. The main goal of this chapter is therefore to propose new algorithms and make those bounds tight.

Parallel algorithms are used here; run by each machine in order to schedule tasks from the repository, independently. One of the main challenges of these algorithms, and a main difference with the single machine scheduling, is that since there is no communication between the machines, a *redundancy-avoidance* mechanism is needed, in order for them to schedule and complete different tasks.

Table 6.1 summarizes the results of this chapter, along with the main negative results of Chapter 5 that still hold for the case of multiple machines. More precisely, by ALG it denotes all work-conserving or deterministic scheduling algorithms for the case of a single machine. Observe, that this chapter focuses mainly on the completed and pending load competitiveness measures, and in the case of k-Amortized and Gk-Amortized it looks at the long-term completed load competitiveness. The chapter starts by showing some properties for a group of parallel algorithms, named GroupLIS(β), that guarantee non-redundant scheduling under some circumstances. Then these properties are used to design *optimal* algorithms for both cases of using speedup or not. Recall, that in the case that speedup is considered, all machines have the same fixed speedup s from the beginning of executions.

6.1. Scheduling with *redundancy avoidance*

This section presents some definitions and properties for all parallel scheduling algorithms that have a specific *redundancy avoidance* mechanism. These preliminary results will help the better understanding of the rest of the chapter and the analysis of the algorithms studied. Let the

Alg.	Model	Completed Load, $\mathcal{C}$	Pending Load, $\mathcal{P}$	Thm.
ALG	$M\langle 1,1,\infty\rangle$	0	∞	5.2
	$M\langle 1,1,2\rangle$	$\leq \frac{\bar{\rho}}{\rho+\bar{\rho}} \approx \frac{1}{2}$	∞	5.3,Cor. 5.1
	$M\langle 1, s < \min\{\rho, 1+\gamma/\rho\}, \infty\rangle$	< 1	∞	5.9, 5.10
GroupLIS(β)	$M\langle m,1,1\rangle$	1	1	6.1
	$M\langle m, s \geq \rho, \infty\rangle$	$[1/\rho, 1]$	$[1,\rho]$	5.4, 6.2
	$M\langle m, s \geq 1+\rho, \infty\rangle$	1	1	6.3
k**-Amortized**	$M\langle m,1,k\rangle$, pairwise divisible	$1/2$	∞	6.4,Cor. 5.1
Gk**-Amortized**	$M\langle m,1,k\rangle$, general	$1/2$	∞	6.5,Cor. 5.1
(m,β)**-LIS**	$M\langle m, s \geq \rho, \infty\rangle$	$[1/\rho, 1]$	$[1,\rho]$	5.4, 6.6
γ**m-Burst**	$M\langle m, s \in [1+\frac{\gamma}{\rho}, \rho), 2\rangle$	1	1	6.8
(m,β)**-LAF**	$M\langle m, 7/2, k\rangle$	1	1	6.9

Table 6.1: Detailed metric comparison of the different algorithms proposed for the multiple machine setting, as well as negative results from Chapter 5 that still hold, for different ranges of speedup. The last column provides the theorem numbers where the results of the corresponding row can be found. Recall that by definition, 0-completed-load competitiveness ratio equals to non-competitiveness, as opposed to the pending load, where non-competitiveness corresponds to an ∞ competitiveness ratio. Note, that ALG is used for any work-conserving or deterministic algorithm, and GroupLIS is a type of parallel algorithms introduced here. Note also, that parameter β is a constant that characterizes the corresponding algorithms.

following definitions be used.

Definition 6.1. *An **absolute task execution** of a task τ, is the interval $\alpha = [t, t']$ in which a machine p schedules τ at time t and reports its completion to the repository at t', without stopping its execution within α (either due to a crash or a decision to stop the execution of task τ).*

Definition 6.2. *A scheduling algorithm is of type **GroupLIS**(β), $\beta \in \mathbb{N}^+$, if all the following hold:*

- *It classifies the pending tasks into* classes *where each class contains tasks of the same size.*
- *It sorts the tasks in each class in increasing order with respect to their arrival time.*
- *If a class contains at least $\beta \cdot m^2$ pending tasks and a machine p schedules a task from that class, then it schedules the $(p \cdot \beta m)$th task in the class.*

The next lemmas show some of the useful properties of all the distributed algorithms of type GroupLIS.

Lemma 6.1. *Consider an algorithm ALG of type GroupLIS(β) running with speedup $s \geq 1$ and a time interval T in which set $\mathbf{P}_\pi$ has at least $\beta \cdot m^2$ pending tasks. Then, any two absolute task executions fully contained in T, of tasks $\tau_1, \tau_2 \in \mathbf{P}_\pi$ by machines p_1 and p_2 respectively, must have $\tau_1 \neq \tau_2$.*

Proof: Suppose by contradiction, that two machines p_1 and p_2 schedule the same π-task, say $\tau \in \mathbf{P}_\pi$, to be executed during the interval T. Assume times t_1 and t_2, where $t_1, t_2 \in T$ and $t_1 \leq t_2$, to be the times when each of the machines scheduled the task, correspondingly. Since any π-task takes time $\frac{\pi}{s}$ to be completed, then p_2 must schedule the task before time $t_1 + \frac{\pi}{s}$, or else it would contradict the property of the Repository stating that each reported task is immediately removed from the set of pending tasks.
Since algorithm ALG is of type GroupLIS(β), at time t_1, when p_1 schedules τ, the task's position on the queue $\mathbf{P}_\pi$ is $p_1 \cdot \beta n$. In order for machine p_2 to schedule τ at time t_2, it must be at position $p_2 \cdot \beta m$. There are two cases to be considered:
(1) If $p_1 < p_2$, then during the interval $[t_1, t_2]$, task τ must increase its position in the queue $\mathbf{P}_\pi$ from $p_1 \cdot \beta m$ to $p_2 \cdot \beta m$, i.e., by at least βm positions. This can happen only in the case when new tasks are injected and are placed before τ. This however, is not possible, since new π-tasks are appended at the end of the queue. (Recall that in algorithms of type GroupLIS, the tasks in the different queues $\mathbf{P}_\pi$ (classes) are sorted in an increasing order with respect to their arrival times.)
(2) If $p_1 > p_2$, then during the interval $[t_1, t_2]$, task τ must decrease its position in the list by at least βm places. This may happen only in the case where at least βm tasks ordered before τ in $\mathbf{P}_\pi$ at time t_1, are completed and reported by time t_2. Since all tasks in the queue have the same size π, and the considered interval has length $\frac{\pi}{s}$, each machine may complete at most one task during that time. Hence, at most $n-1$ tasks of size π may be completed, which are not enough to change τ's position from $p_1 \cdot \beta m$ to $p_2 \cdot \beta m$, even when $\beta = 1$ (observe that $\beta \geq 1$), by time t_2.
The two cases above contradict the initial assumption and hence the claim of the lemma follows.
■

Lemma 6.2. *Let $\mathbf{C}_T$ be a set of tasks reported as completed by an algorithm ALG of type GroupLIS(β) in a time interval T, and $\#C_T = |\mathbf{C}_T| > m$. Then at least $\#C_T - m$ such tasks have their absolute task execution fully contained in T.*

Proof: A task τ which is reported in T by machine p and its absolute task execution $\alpha \not\subseteq T$, has $\alpha = [t, t']$ where $t \notin T$ and $t' \in T$. Since p does not stop executing τ in α, only one such task may occur for p. Then, there can not be more than m such reports overall and the lemma follows.
■

6.2. Properties of *some* work-conserving and deterministic algorithms

Some properties for some parallel work-conserving and deterministic scheduling algorithms, are presented first, more precisely for algorithms of type GroupLIS(β). As already mentioned, the positive results of completed-load competitiveness shown in Chapter 5 do not necessarily hold for the case of m machines. It is shown here, that for no speedup and some specific amounts of it,

some positive results are preserved in the multiple machine setting, provided that the algorithms follow the non-redundancy characteristics of type GroupLIS(β) described earlier.

6.2.1. Properties of $M\langle m, 1, 1\rangle$

Fist, the following proposition is true for all algorithms of type GroupLIS(β) in the case of uniform task sizes.

Theorem 6.1. *Any distributed work-conserving scheduling algorithm of type GroupLIS(β), running on a multiple machine setting with no speedup and with all tasks having the same size, i.e., $M\langle m, 1, 1\rangle$, has optimal completed and pending load competitiveness, of ratio* 1.

Proof: Consider distributed algorithm ALG of type GroupLIS(β). From the definition of GroupLIS(β), and the fact that all tasks have the same size, there will only be one class; which means one queue of pending tasks.

Look at any time instant t of the execution. In case there are less than $\beta \cdot m^2$ tasks pending, then $C_t(\text{ALG}) \geq C_t(X) - \beta \cdot m^2\pi$ holds directly, where π is the size of any task. If however, there are at least $\beta \cdot m^2$ tasks pending, consider a time instant $t' < t$, such that it is the earliest time before t where there were at least $\beta \cdot m^2$ tasks in the queue of the repository. Observe then, that within $T = [t', t]$, Lemmas 6.1 and 6.2 can be applied, which means that no task has more than one absolute executions within T and there are $\#C_T - m$ such tasks. In other words, there are at most m tasks reported as completed within T that might have been executed redundantly. What is more, within T algorithm X can complete at most m more tasks than ALG, and hence for all the time instances within T, $C_T(\text{ALG}) \geq C_T(X) - 2m\pi$ holds, which gives a 1-completed-load competitive ratio and thus 1-pending-load competitive ratio as well. ■

6.2.2. Properties of $M\langle m, s, \infty\rangle$

Consider now any parallel work-conserving algorithm ALG_W of type GroupLIS(β), running with speedup s. This means, that while there are at least βm^2 tasks pending in its execution, it will guarantee non redundant task executions (see Lemma 6.1). The following lemmas and theorems also hold for ALG_W.

Lemma 6.3. *Consider T, a time interval during which the queue of pending tasks of ALG_W, running with speedup $s \geq \rho$, is always non-empty. Then the total number of tasks reported by ALG_W in T is not smaller than the total number of tasks reported by X in the same interval, minus m (counting redundancy).*

Proof: For each machine in the execution of X in the considered interval T, exclude the first reported task to eliminate the tasks that might have been scheduled before the beginning of the interval. There are at most m such tasks reported by X (see Lemma 6.2). Then, the speedup $s \geq \rho$, implies that during the interval T when a machine p executes a task τ in the execution

of X, the same machine reports at least one task to the repository in the execution of ALG_W (executing any task in the execution of X takes at least time π_{min}, while executing any task in the execution of ALG_W takes no more than $\pi_{max}/s \leq \pi_{min}$). Observe, that some of the tasks reported, and included in the counting, may be redundant. Recall, that no active machine in the execution of ALG_W is ever idle (the queue of pending tasks is never empty). Hence, a $1-1$ association can be defined, from the tasks completed by X (the ones started and reported in T) to the family of different tasks reported by ALG_W in the same interval. This, completes the proof of the lemma. ■

Lemma 6.4. *At all time instants t of an execution of algorithm ALG_W, running with $s \geq \rho$, the following holds with respect to the number of pending tasks: $\#P_t(ALG_W) \leq \#P_t(X) + \beta \cdot m^2 + 3m$, where $\beta \geq \rho$.*

Proof: Assume by contradiction a time instant t', such that $\#P_{t'}(\text{ALG}_W) > \#P_{t'}(X) + \beta \cdot m^2 + 3m$. Let also a time instant $t^* < t'$ to be the smallest time such that $\#P_t(\text{ALG}_W) > \#P_t(X) + B$ holds for $t \in [t^*, t')$. Observe, that at time t^*, $\#P_{t^*}(\text{ALG}_W) \leq \#P_{t^*}(X) + \beta \cdot m^2 + m$ holds, because no more than m tasks could have been reported as completed at t^* by X, while just before that instant the difference of pending task between ALG_W and X was at most $\beta \cdot m^2$.

What is more, in the interval $(t^*, t^* + \pi_{min}]$, algorithm X can report at most m completed tasks, as each machine may finish at most one task. For each time $t \in (t^*, t^* + \pi_{min}]$, let $T = (t^*, t]$ and $\#I_T$ the number of injected tasks within interval T. Then, since $\#P_t(\text{ALG}_W) \leq \#P_{t^*}(\text{ALG}_W) + \#I_T$ and $\#P_t(X) \geq \#P_{t^*}(X) + \#I_T - m$, it follows that $\#P_t(\text{ALG}_W) \leq \#P_{t^*}(X) + \beta \cdot m^2 + 2m$.

Look now at the rest of the interval $(t^* + \pi_{min}, t']$. From Lemma 6.1, no task is completed and reported twice by ALG_W in the interval, given that $\beta \geq \rho$.

Now assume that ALG_W has completed x tasks in the interval $(t^*, t']$. By lemma 6.3, algorithm X has completed at most $x + m$ in the same interval. What is more, since $\#P(\text{ALG}_W) \geq \beta \cdot m^2$ during the interval, then the x tasks reported are distinct. Combining with the fact that ALG_W guarantees non-redundant executions in the same interval, $\#P_{t'}(\text{ALG}_W) \leq \#P_{t^*+\pi_{min}}(\text{ALG}_W) - x$. Thus,

$$\begin{aligned} \#P_{t'}(\text{ALG}_W) &\leq \#P_{t^*+\pi_{min}}(X) + \beta \cdot m^2 + 2m - x \\ &\leq \#P_{t'}(X) + \beta \cdot m^2 + 3m, \end{aligned}$$

which contradicts the definition of time instant t', and completes the proof. ■

Theorem 6.2. *Any distributed work-conserving algorithm ALG_W, that guarantees non redundant executions while there are at least B tasks pending, running on a system with m machines and speedup $s \geq \rho$, has a completed-load competitive ratio $\mathcal{C}(ALG_W) \geq 1/\rho$ and a pending load competitiveness ratio $\mathcal{P}(ALG_W) \leq \rho$.*

Proof: From the above lemma 6.4, and the fact that the number of completed tasks of an execution is complementary to the number of pending tasks, we have that $\#C_t(\text{ALG}_W) \geq \#C_t(X) - \beta \cdot m^2 - 3m$, for all time instants t, where $\beta \geq \rho$. Since the size of any task completed by X is at most $\frac{\pi_{max}}{\pi_{min}} = \rho$ times bigger than the size of any task completed by ALG_W, the completed load follows directly as $C_t(\text{ALG}_W) \geq \frac{1}{\rho}(\#C_t(X) - \beta \cdot m^2 - 3m)$ for all time instants of the execution. From the same argument, the pending load follows as $P_t(\text{ALG}_W) \leq \rho(\#P_t(X) + \beta \cdot m^2 + 3m)$. Thus the completed and pending load competitiveness is $\mathcal{C}(\text{ALG}_W) \geq 1/\rho$ and $\mathcal{P}(\text{ALG}_W) \leq \rho$ respectively, as claimed. ■

Lemma 6.5. *Consider T, a time interval during which the queue of pending tasks of ALG_W, running with speedup $s \geq \rho + 1$, is always non-empty. Then the total load reported as completed by ALG_W in T is not smaller than the total load reported by X in the same interval, minus $m\pi_{max}$ (counting redundant task executions).*

Proof: For each machine in the execution of X in the considered interval T, exclude the first reported task to eliminate the tasks that might have been scheduled before the beginning of the interval. There are at most m such tasks reported by X, with a maximum size π_{max} (see Lemma 6.2). Then, the speedup $s \geq \rho + 1$, implies that during the interval of length $|T|$ when a machine p completes load $|T|$ in the execution of X, the same machine reports at least $|T|s - \pi_{max} \geq |T|$ load in the execution of ALG_W. Observe, that some of the tasks reported, and included in the counting, may even be redundant. Recall, that no active machine in the execution of ALG_W is ever idle (the queue of pending tasks is never empty). Hence, one can associate the load completed by X (including only the tasks started and reported in T) to the load completed by ALG_W (the family of different tasks reported by ALG_W in the same interval). This, completes the proof of the lemma. ■

Theorem 6.3. *The completed and pending load competitiveness of algorithm ALG_W, running on m machines with $s \geq \rho + 1$, is $\mathcal{C}(\text{ALG}_W) \geq 1$ and $\mathcal{P}(\text{ALG}_W) \leq 1$ respectively. More precisely, at all time instants t of an execution of ALG_W, the following holds with respect to its completed load: $C_t(\text{ALG}_W) \geq C_t(X) - \beta \cdot m^2\pi_{min} - m(2\pi_{max} + \pi_{min})$, where $\beta \geq \rho$.*

Proof: Assume by contradiction a time instant t', such that $C_{t'}(\text{ALG}_W) < C_{t'}(X) - \beta \cdot m^2\pi_{min} - m(2\pi_{max} + \pi_{min})$. Let also time instant $t^* < t'$ to be the smallest time such that $C_t(\text{ALG}_W) < C_t(X) - \beta \cdot m^2\pi_{min}$ holds for $t \in [t^*, t')$. Observe, that at time t^*, $C_{t^*}(\text{ALG}_W) \geq C_{t^*}(X) - \beta \cdot m^2\pi_{min} - m\pi_{max}$ holds, because no more than m tasks could have been reported as completed at t^* by X, with a maximum size π_{max}, while just before that instant the difference of completed load between ALG_W and X was at most $\beta \cdot m^2\pi_{min}$.

What is more, in the interval $(t^*, t^* + \pi_{min}]$, algorithm X can report at most m completed tasks of size π_{min}, as each machine may finish at most one task. For each time $t \in (t^*, t^* + \pi_{min}]$, let $T = (t^*, t]$ and I_T the injected load within interval T. Then, since $C_t(\text{ALG}_W) \geq C_{t^*}(\text{ALG}_W) - I_T$ and $C_t(X) \leq C_{t^*}(X) - I_T + m\pi_{min}$, it follows that $C_t(\text{ALG}_W) \geq C_{t^*}(X) - \beta \cdot m^2\pi_{min} - m(\pi_{max} + \pi_{min})$.

Now look at the rest of the interval $(t^* + \pi_{min}, t']$. From Lemma 6.1, no task is completed and reported twice by ALG_W in the interval, given that $\beta \geq \rho$.

Now assume that ALG_W has completed load x in the interval $(t^*, t']$. By lemma 6.3, algorithm X has completed at most $x + m\pi_{max}$ in the same interval. What is more, since $\#P(\text{ALG}_W) \geq \beta \cdot m^2$ during the interval, then the tasks reported are distinct, and are of at least size π_{min}. Combining with the previous claim of non-redundant task executions by ALG_W in the same interval, we have that $C_{t'}(\text{ALG}_W) \geq C_{t^*+\pi_{min}}(\text{ALG}_W) - x$. Thus,

$$\begin{aligned} C_{t'}(\text{ALG}_W) &\geq C_{t^*+\pi_{min}}(X) - \beta \cdot m^2\pi_{min} - m(\pi_{max} + \pi_{min}) - x \\ &\geq C_{t'}(X) - \beta \cdot m^2\pi_{min} - m(2\pi_{max} + \pi_{min}), \end{aligned}$$

which contradicts the definition of time instant t', and completes the proof. ■

6.3. Competitiveness without speedup

This section focuses on the case when machines have no speedup, i.e., $s = 1$. The aim is to show that the upper bound of long-term completed-load competitiveness shown in Chapter 5 can be achieved by some online algorithms in the distributed setting of m machines. In particular, two scheduling algorithms are proposed, k-Amortized and Gk-Amortized, both inspired by the work of Jurdzinski et al. [60]. They are adapted to fit the multiple machine model considered, and some important properties are proved in order to analyze their performance under worst-case arrival and error patterns A and E. The upper bound of long-term completed-load competitiveness is guaranteed, and therefore it is tight.

6.3.1. k-Amortized: an *optimal* algorithm for $M\langle m, 1, k\rangle$ – pairwise divisible

Here, algorithm k-Amortized is proposed, adapting algorithm Greedy of Jurdzinski et al. [60] to the case of multiple machines. See the pseudo-code in Alg. 1 and 2. In order to explain the cases in which this algorithm is useful, let $\pi_{min} = \pi_1 < \pi_2 < \ldots < \pi_k = \pi_{max}$. It is assumed that each ratio $\rho_{i,j} = \pi_i/\pi_j$ is an integer for any $1 \leq j < i \leq k$, a property of the task sizes called *pairwise divisibility*.

The algorithm follows the *Shortest Processing Time* (SPT) first policy, with some additional balancing constraints. The basic idea is to schedule tasks in batches (or groups) that will balance the length of the next larger task. What is more, it considers redundancy avoidance, requiring *enough* tasks available before scheduling.

Before looking at the details of the algorithm, some necessary notation and terminology is introduced. First, parameter $\rho_{i,j} = \pi_i/\pi_j$, where $1 \leq j < i \leq k$, as already mentioned at the beginning of the subsection, represents the ratio between two task sizes and is considered to be an integer for this algorithm. A special case of this ratio used in the algorithm, is $\rho_i = \frac{\pi_i}{\pi_{i-1}}$, where $i \in [2, k]$; it represents the ratio between two *consecutive* task sizes.

Algorithm 1: k**-Amortized** (for machine p)

Parameters: $m, \{\pi_1, \pi_2, \ldots, \pi_k\}$
Upon awaking or restart
  Repeat
    Get $\mathbf{P}_1$ to $\mathbf{P}_k$ from the Repository;
    While $\pi_k \left\lfloor \frac{\#P_k}{m^2} \right\rfloor + \sum_{i=1}^{k-1} \pi_i \left\lfloor \frac{\#P_i}{m^2 + m\rho_{i+1}} \right\rfloor < \pi_k$ **Do**
      execute task π at position $(p \cdot m) \mod \#P$ in $\mathbf{P}$;
      Inform the Repository for the task completion;
    $Schedule_Group(k)$;

Algorithm 2: $Schedule_Group(j)$

Parameters: $m, j, \{\pi_1, \pi_2, \ldots, \pi_k\}, \{\mathbf{P}_1, \mathbf{P}_2, \ldots, \mathbf{P}_j\}$
  If $\sum_{i=1}^{j-1} \pi_i \left\lfloor \frac{\#P_i}{m^2 + m\rho_{i+1}} \right\rfloor \geq \pi_j$ **then**
    For $\alpha = 1$ to ρ_j **Do**
      Schedule_Group(j-1);
  Else
    execute task π_j at position $p \cdot m$ in $\mathbf{P}_j$;
    Inform the Repository for the task completion;
  Return

Definition 6.3. *Let the* adequate sizes *of pending tasks, be the task sizes whose pending queues have* $\gtrsim m^2$ *tasks. More precisely, for size* π_k *to be adequate there must be at least* m^2 *tasks pending in* $\mathbf{P}_k$*, while for any other* π_j *size, where* $j \in [1, k-1]$*, the corresponding pending queue* $\mathbf{P}_j$ *must have at least* $m^2 + m\rho_{j+1}$ *tasks. (In Lemma 6.6, it is shown that this is the necessary number of tasks in order to guarantee the non-redundancy property of the algorithm.)*

Definition 6.4. *Let an* i-group of tasks *be the tasks completed in the execution of a machine under the recursive call to the function* Schedule_Group(i)*, where* $i \in [1, k]$*. Note, that an* i*-group has a total size of* π_i*, but may be the result of the completion of several smaller tasks.*

Definition 6.5. *A machine is considered to be in an* n-busy *interval, say* $T = [t_1, t_2]$*, where* $t_1 < t_2$*, satisfying the following properties:*
(1) The machine is busy *at each time* $t \in T$ *(it is either executing some task, it has just completed one, or it is crashed).*
(2) The machine does not schedule tasks of size π_i *for* $i > n$ *during the interval* T*.*
(3) At the beginning of interval T*, i.e., time instant* t_1*, algorithm* k*-Amortized has at least as many tasks of size* π_i *pending as* X*, for each* $i \leq n$*. Hence,* $\#P_{t_1}(A, \pi_i) \geq \#P_{t_1}(X, \pi_i)$ *for each* $i \in [1, n]$*.*

Finally, note that in the pseudo-code the queue of pending tasks $\mathbf{P}$ is mentioned. In line with its definition in Chapter 3, here it is considered to be the unified set of all $\mathbf{P}_i$, having the tasks sorted in an ascending order according to their arrival times and breaking ties with the task sizes.

Algorithm description

After awakening or restart, a machine schedules the task at position $p \cdot m$ of the pending queue $\mathbf{P}$, until the sum of the *adequate sizes* of the pending tasks is at least π_k. Following this strategy, the algorithm guarantees the ability to cover a time interval of length π_k, with non-redundant task executions, if it is not interrupted by a machine crash. Otherwise, being work-conserving, it schedules the task in the position already mentioned, but with no guarantees for non-redundancy. Then, it calls the recursive function *Schedule_Group*(j) (starting with $j = k$) which checks whether the sum of *adequate sizes* of the pending tasks smaller than π_j is at least equal to π_j (resp., π_k). If the condition is true, the algorithm makes $\rho_j = \frac{\pi_j}{\pi_{j-1}}$ calls to function *Schedule_Group*$(j-1)$ – resp., *Schedule_Group*$(k-1)$ – in order to cover the corresponding time interval π_j with ρ_j groups of π_{j-1} aggregate size; in other words, ρ_j $(j-1)$-groups. In the following recursion levels more recursive calls may occur, if there are again enough pending tasks, thus covering the corresponding time intervals by tasks of smaller size each time. Otherwise, when there are not enough tasks pending in a recursion level, a task of the current size, π_j, is scheduled by the machine and when completed, returns to the previous recursion level.

Algorithm Analysis

Some important properties of the algorithm will now be proved, needed for the complete proof that its completed-load competitiveness is indeed optimal, i.e., $\mathcal{C}(k\text{-Amortized}) \gtrsim 1/2$. First, from Corollary 5.1 the following holds:

Corollary 6.1. *There is a time instant t^* in the execution of k-Amortized, such that for the rest of the execution, tasks are scheduled only by function $Schedule_Group$.*

Proof: To see that the corollary is true, one needs to set K of Corollary 5.1 equal to $\pi_k m^2 + \sum_{i=1}^{k-1} \pi_i (m^2 + m\rho_{i+1})$. This means that one can define a time instant t^*, such that $\forall t > t^*$: $\pi_k \left\lfloor \frac{\#P_k}{m^2} \right\rfloor + \sum_{i=1}^{k-1} \pi_i \left\lfloor \frac{\#P_i}{m^2 + m\rho_{i+1}} \right\rfloor \geq \pi_k$, and hence function *Schedule_Group* will always be called after t^*, which completes the proof. ■

The following two lemmas lead to the *non-redundancy* property of the algorithm, safely omitting the case when lines 6 and 7 of Algorithm 1 are executed, due to the above corollary. In other words, for the case of tasks being executed within the *Schedule_Group* function. (Note also, that in the case of lines 6 and 7 of Algorithm 1 are executed, the pending load of the algorithm is bounded, so it does not affect the completed-load competitiveness in the long run.)

Lemma 6.6. *Algorithm k-Amortized schedules a π_j-task (in line 6 of Alg. 2), only when there are at least m^2 tasks in the corresponding queue of pending tasks, $\mathbf{P}_j$.*

Proof: First, look at the algorithm description and its pseudo-code. The first call to schedule some tasks – calling function *Schedule_Group*(k) – is done only if *enough* tasks are pending to

cover the π_k-time without redundancy. A task size is accounted for only when it is *adequate*; only when there are $\gtrsim m^2$ tasks of that size pending (recall Definition 6.3).

Then, within the *Schedule_Group*(j) function, starting by $j = k$, the algorithm checks whether it can be covered non-redundantly by tasks of smaller size. If it does, it makes a recursive call to the function with parameter $j-1$, which corresponds to the next smaller task size, π_{j-1}. A task π_j cannot be covered non-redundantly by smaller tasks when $\sum_{i=1}^{j-1} \pi_i \left\lfloor \frac{\#P_i}{m^2+m\rho_{i+1}} \right\rfloor < \pi_j$ (see condition of line 2 in Algorithm 2). However, a function call with task size π_j means that it was either called by algorithm k-Amortized directly (and $j = k$), in which case the condition in line 8 of Alg. 1 does not hold, or it was called by the previous recursion level; by function handling the next bigger task size, π_{j+1}, in which case the condition in line 2 holds. It will now be shown that in either case, there are *enough* tasks of size π_j to be scheduled by the system's machines without executing any of them redundantly.

Consider the function call *Schedule_Group*(j), for which the *if* condition in line 2 does not hold. This implies

$$\sum_{i=1}^{j-1} \pi_i \left\lfloor \frac{\#P_i}{m^2 + m\rho_{i+1}} \right\rfloor < \pi_j. \tag{6.1}$$

Then, consider the two cases mentioned above separately:

Case 1: A previous function call, *Schedule_Group*$(j+1)$, in which the *if* condition in line 2 holds, i.e., $\sum_{i=1}^{j} \pi_i \left\lfloor \frac{\#P_i}{m^2+m\rho_{i+1}} \right\rfloor \geq \pi_{j+1}$, implies that

$$\pi_j \left\lfloor \frac{\#P_j}{m^2 + m\rho_{j+1}} \right\rfloor + \sum_{i=1}^{j-1} \pi_i \left\lfloor \frac{\#P_i}{m^2 + m\rho_{i+1}} \right\rfloor \geq \pi_{j+1} \tag{6.2}$$

Combining the two equations 8.2 and 8.3 the following holds:

$$\pi_j \left\lfloor \frac{\#P_j}{m^2 + m\rho_{j+1}} \right\rfloor \geq \pi_{j+1} - \sum_{i=1}^{j-1} \pi_i \left\lfloor \frac{\#P_i}{m^2 + m\rho_{i+1}} \right\rfloor > \pi_{j+1} - \pi_j > 0,$$

which means that $|L_j| \geq m^2 + m\rho_{j+1}$.

Case 2: The function call *Schedule_Group*(j) was actually *Schedule_Group*(k) and came directly from line 8 of algorithm k-Amortized. Hence, $\pi_k \left\lfloor \frac{\#P_k}{m^2} \right\rfloor + \sum_{i=1}^{k-1} \pi_i \left\lfloor \frac{\#P_i}{m^2+m\rho_{i+1}} \right\rfloor \geq \pi_k$ holds, which implies that

$$\pi_k \left\lfloor \frac{\#P_k}{m^2} \right\rfloor \geq \pi_k - \sum_{i=1}^{k-1} \pi_i \left\lfloor \frac{\#P_i}{m^2 + m\rho_{i+1}} \right\rfloor \tag{6.3}$$

Replacing $j = k$ in equation 8.2, $\sum_{i=1}^{k-1} \pi_i \left\lfloor \frac{\#P_i}{m^2+m\rho_{i+1}} \right\rfloor < \pi_k$, which combined with equa-

tion 6.3 one can easily see that

$$\pi_k \left\lfloor \frac{\#P_k}{m^2} \right\rfloor > 0 \Rightarrow \left\lfloor \frac{\#P_k}{m^2} \right\rfloor > 0,$$

which in its turn means that $\#P_k \geq m^2$.

As shown, there are at least m^2 tasks of size π_j or π_k in each case respectively. However, in case 1 above, there will be ρ_{j+1} iterations of the recursive function call *Schedule_Group*(j) of line 4. It must therefore be made clear that there are at least m^2 available tasks for all iterations.

Consider for example, the case in which at a time t all m machines are in a $(j+1)$-group execution; following the *Schedule_Group*$(j+1)$ function, and having condition of line 2 TRUE. Then, they all start the ρ_{j+1} iterations of scheduling j-groups of tasks, calling the recursive function *Schedule_Group*(j). Consider now, that in all corresponding conditions of line 2, are FALSE. This means, that all m machines will simultaneously execute one π_j-task in every iteration. Therefore, having $m^2 + m\rho_{j+1}$ pending tasks of size π_j at the beginning of iterations, will guarantee that in every iteration there are still at least m^2 tasks pending in queue $\mathbf{P}_j$. This completes the proof of the lemma. ■

Observe now, that algorithm k-Amortized is of type GroupLIS; it separates the pending tasks into classes depending on their size (has them sorted with respect to their arrival time, as assumed by the model), and if a class contains at least m^2 pending tasks, a machine p schedules the task at position $(p \cdot m)$. Hence, Lemma 6.1 from Section 6.1 also holds for it. Combining the two Lemmas 6.6 and 6.1, the following property for algorithm k-Amortized follows.

Observation 6.1. *When Algorithm k-Amortized, schedules tasks through its function Schedule_Group (Alg. 2), it never completes the same task more than once. In particular, the same task cannot be simultaneously executed in more than one machines of the system.*

Lemma 6.7. *When a task of size π_j is scheduled by k-Amortized, through its function Schedule_Group, say at time instant t, the total size of smaller pending tasks is $P_t(A, < \pi_j) \leq \sum_{i=1}^{j-1}(\pi_j + \pi_i)(m^2 + m\rho_{i+1})$.*

Proof: When a task of size π_j is scheduled by k-Amortized in line 6 of Alg. 2, as we have seen also from Lemma 6.6, the following inequality must hold: $\sum_{i=1}^{j-1} \pi_i \left\lfloor \frac{\#P_i}{m^2+m\rho_{i+1}} \right\rfloor < \pi_j$. This also means that $\forall i \in [1, j-1]$, the following is true:

$$\begin{aligned} \pi_i \left\lfloor \frac{\#P_i}{m^2 + m\rho_{i+1}} \right\rfloor < \pi_j \quad &\Rightarrow \quad \pi_i \left(\frac{\#P_i}{m^2 + m\rho_{i+1}} - 1 \right) < \pi_j \\ &\Rightarrow \quad \pi_i \#P_i < (\pi_j + \pi_i)(m^2 + m\rho_{i+1}). \end{aligned}$$

Therefore, the sum of all pending tasks smaller than π_j is $P_t(A, < \pi_j) = \sum_{i=1}^{j-1} \pi_i \#P_i \leq$

$\sum_{i=1}^{j-1}(\pi_j + \pi_i)(m^2 + m\rho_{i+1})$ as claimed. ■

By Observation 6.1, no task is executed more than once by algorithm k-Amortized when scheduled by its function *Schedule_Group*. Hence, one can safely separate the analysis of each machine individually, ignoring any task execution by line 6 of Alg. 1. Focusing on one machine, say p, and then generalizing for all m machines to give the final result, the proof of Theorem 1 in the work of Jurdzinski et al. [60], holds per machine. Here, only the statement of the necessary lemma is given, which is adapted for algorithm k-Amortized.

Lemma 6.8 ([60]). *Let f_n be a function such that $f_1 = \pi_k$ and $f_{i+1} = f_i + \sum_{j=1}^{i}(\pi_{i+1}+\pi_j)(m^2 + m\rho_{j+1}) + \pi_{i+1} + 2\pi_k$. For a machine p that is n-busy at a time interval T (recall Definition 6.5), where $n \leq k$,*

$$2C_T^p(A) \geq C_T^p(X) - f_n.$$

Theorem 6.4. *Algorithm k-Amortized has an optimal long-term completed load competitiveness of 1/2, provided that $\pi_i/\pi_{i-1} \in \mathbb{N}$ for each $i \in [2, k]$.*

Proof: First, by Observation 6.1, each task completion within function *Schedule_Group*, occurs only once. Then, looking at Lemma 6.8, it implies that the completed-load competitiveness of algorithm k-Amortized gets arbitrarily close to 1/2 on sufficiently long periods of time. On the other hand, k-Amortized cannot guarantee non redundancy when its queue contains few tasks, i.e., when $\pi_k \left\lfloor \frac{\#P_k}{m^2} \right\rfloor + \sum_{i=1}^{k-1} \pi_i \left\lfloor \frac{\#P_i}{m^2+m\rho_{i+1}} \right\rfloor < \pi_k$. However, from Corollary 6.1, it is safe to ignore these cases. This means, that as time goes to infinity, the completed load competitiveness goes to 1/2 as claimed.

In Chapter 5 it has been shown that the long-term completed load of any online algorithm for two different task sizes is at most $\frac{\bar{\rho}}{\bar{\rho}+\rho}$, which is equal to $1/2$ when $\rho \in \mathbb{N}$. Since an adversary can decide to schedule merely two different task sizes among the available k ones, it means that the completed load competitiveness shown is in fact optimal. ■

6.3.2. Gk-Amortized: an *optimal* algorithm for $M\langle m, 1, k\rangle$ – general

This subsection looks at the case when $\rho_{i,j} = \pi_i/\pi_j \notin \mathbb{N}$. Jurdzinski et al. [60] show that the completed load competitive ratio of any scheduling algorithm running without speedup, is at most $\min_{i \leq j < i \leq k} \left\{ \frac{\overline{\rho_{i,j}}}{\overline{\rho_{i,j}}+\rho_{i,j}} \right\}$ (in Theorem 2). This upper bound also holds in the case of multiple machines, since the adversary can force only one of the machines to be alive at all times, while keeping the rest crashed.

Algorithm Gk-Amortized presented here, is an adaptation of the algorithm MGreedy used by Jurdzinski et al. [60], to the case of multiple machines. See the pseudo-code in Alg. 3 and 4.

Algorithm 3: Gk**-Amortized** (for machine p)

Parameters: $m, \{\pi_1, \pi_2, \ldots, \pi_k\}$
Upon awaking or restart
  Repeat
    Get all pending queues from the Repository, $\mathbf{P}_i$;
    $\mathsf{C} \leftarrow \left\{ i \middle| \pi_i \left\lfloor \frac{\#P_i}{m^2} \right\rfloor \geq ck\pi_k \right\}$;
    While $\left\{ i \middle| \pi_i \left\lfloor \frac{\#P_i}{m^2} \right\rfloor \geq ck\pi_k \right\} = \emptyset$ **Do**
      execute task π at position $(p \cdot m) \mod \#P$ in $\mathbf{P}$;
      Inform the Repository for the task completion;
    $\mathsf{C} \leftarrow \left\{ i \middle| \pi_i \left\lfloor \frac{\#P_i}{m^2} \right\rfloor \geq ck\pi_k \right\}$;
    $i^* \leftarrow \min(\mathsf{C})$;
    For $a = 1$ to ck **Do**
      $\pi' \leftarrow Schedule_Group(k)$;

Algorithm 4: $Schedule_Group(j)$

Parameters: $m, j, \{\pi_1, \pi_2, \ldots, \pi_k\}, \{\mathbf{P}_1, \mathbf{P}_2, \ldots, \mathbf{P}_j\}$
  $\pi \leftarrow 0$;
  While $\pi \leq \pi_j - \pi_{i^*}$ **Do**
    If $j > i^*$ **then**
      $\pi' \leftarrow Schedule_Group(j-1)$;
      $\pi \leftarrow \pi + \pi'$;
    Else
      execute task π_j at position $p \cdot m$ in $\mathbf{P}_j$;
      If task π_j completed successfully **then**
        Inform the Repository for the task completion;
        $\pi \leftarrow \pi_j$;
      $\mathsf{C} \leftarrow \mathsf{C} \cup \left\{ i \middle| \pi_i \left\lfloor \frac{\#P_i}{m^2} \right\rfloor \geq ck\pi_k \right\}$;
      $i^* \leftarrow \min(\mathsf{C})$;
  Return π

Algorithm description

Algorithm Gk-Amortized completes tasks of the same size as long as possible and changes only when it is necessary. For that, the execution is split into *stages* of total length $ck\pi_k$, where $c \in \mathbb{N}$ is a fixed large constant. At the beginning of a stage, a set of *candidate* task sizes is established as $\mathsf{C} = \left\{ i \middle| \pi_i \lfloor \frac{|L_i|}{m^2} \rfloor \geq ck\pi_k \right\}$. Then, the *appropriate size* π_{i^*} is set as the minimum size in the set of candidate sizes, i.e., $i^* = \min(\mathsf{C})$. This is the size of tasks that the algorithm will start executing. The appropriate size is updated after every task completion, checking first whether there has been some change in the set of candidate tasks – due to new task injections.

Algorithm analysis

As a first observation, following algorithm Gk-Amortized only tasks of size π_{i^*} are scheduled, unless there are not enough tasks to guarantee non-redundancy (when set $\mathsf{C} = \emptyset$), in which case, a

task π at position $p \cdot m$ of the whole set of pending tasks $\mathbf{P}$ is scheduled. To see this clearly, look at lines 6-12 in Alg. 3 and lines 4,7 and 8 in Alg. 4. It is also important to note that parameter i^* can only decrease in each stage. This is because at the beginning of the stage there are enough pending π_i-tasks for each candidate task size $i \in \mathsf{C}$ to cover a time interval of length $ck\pi_k$ (line 9 of Alg. 3). Similar to the case of algorithm k-Amortized, the following corollary holds:

Corollary 6.2. *There is a time instant t^* in the execution of k-Amortized, such that for the rest of the execution, tasks are scheduled only by function $Schedule_Group$.*

Proof: To see that the corollary is true, one needs to set K of Proposition 5.1 equal to $ck\pi_k$, for which $\exists i$, such that $\pi_i m^2 \geq ck\pi_k$. This means that one can define a time instant t^*, such that $\forall t > t^*$: $\exists i$ for which $\pi_i m^2 \geq ck\pi_k$ holds, and hence function *Schedule_Group* will always be called after t^*, which completes the proof. ■

Also, like algorithm k-Amortized, the modified algorithm Gk-Amortized belongs to the GroupLIS algorithms and has the property of *non redundancy* when enough tasks are pending, which is shown in the following lemma.

Lemma 6.9. *Algorithm Gk-Amortized never completes the same task more than once within Schedule_Group (Alg. 4).*

Proof: First, observe that the algorithm schedules a π_j-task in function *Schedule_Group* only when there are at least m^2 tasks in the corresponding pending queue $\mathbf{P}_j$. Looking at the pseudo-code, a task π_j is scheduled in line 8 of Alg. 4, only when the condition in line 4 does not hold, and hence $j = i^*$. From lines 9 and 10 of Alg. 3 and lines 12 and 13 of Alg. 4, it means that i^* belongs to the set of *candidate* task sizes, for which every task size has at least as many tasks pending as necessary to "cover" $ck\pi_k$ time, i.e., ck calls to the *Schedule_Group(j)* function. This number of tasks is:

$$\pi_j \left\lfloor \frac{\#P_j}{m^2} \right\rfloor \geq ck\pi_k \Rightarrow \left\lfloor \frac{\#P_j}{m^2} \right\rfloor \geq ck\lfloor \rho_{k,i} \rfloor \Rightarrow \#P_i \geq \big(ck\lfloor \rho_{k,i} \rfloor + 1\big)m^2.$$

What is more, since $\#P_i \geq m^2$, Lemma 6.1 holds for algorithm Gk-Amortized as well (it belongs to the $GroupLIS$ algorithms), and hence combining the two, the claim of the lemma holds. ■

Theorem 6.5. *Algorithm Gk-Amortized can reach the optimal completed-load competitiveness of* $\min\limits_{1 \leq j < i \leq k} \left\{ \frac{\overline{\rho_{i,j}}}{\overline{\rho_{i,j}} + \rho_{i,j}} \right\}$.

Proof: First, by Lemma 6.9, each task completion within function *Schedule_Group*, occurs only once. Then, combining it with the fact that Gk-Amortized is a GroupLIS algorithm, the analysis of each machine can be considered separately while *Schedule_Group* function is run. Thus, the proof of Theorem 3 in [60], will hold for each machine individually. The theorem states

Algorithm 5: (m, β)**-LIS** (for machine p)

```
1 Parameters: m, β
2 Repeat                                                    //Upon awaking or restart
3     Get from the Repository the set of pending tasks P;
4     Sort tasks in P by arrival time and break ties with their ids;
5     If #P ≥ 1 then
6         execute task with rank p · βm mod #P;
7     Inform the Repository of the task executed.
```

that the greedy algorithm used has an optimal long-term completed load competitiveness equal to $\min_{1\leq j<i\leq k}\left\{\frac{\overline{\rho_{i,j}}}{\overline{\rho_{i,j}}+\rho_{i,j}}\right\}$.

What is more, from Corollary 6.2, it is safe to ignore the cases where lines 7 and 8 of Alg. 3 are executed, for which non redundant task executions cannot be guaranteed. This means, that as time goes to infinity, the completed load competitiveness becomes $\min_{1\leq j<i\leq k}\left\{\frac{\overline{\rho_{i,j}}}{\overline{\rho_{i,j}}+\rho_{i,j}}\right\}$ as claimed. ■

6.4. Competitiveness with speedup

This section turns its focus on the case that machines run under speedup, i.e., $s > 1$. Recall that all machines run under the same fixed speedup from the beginning of an execution. Recall also that in Subsection 5.3.1 it has been shown that when $s < \min\{\rho, 1+\gamma/\rho\}$ then no algorithm can be competitive in terms of pending load, 1-completed-load or latency, even in the case of 1 machine. One must therefore look into ranges of speedup that don't satisfy both conditions **C1** and **C2**.

The section starts by looking at the case of $s \geq \rho$. However, since the aim is to keep speedup as small as possible, the case of $s \in [1+\gamma/\rho, \rho)$ is also explored.

6.4.1. Algorithm (m, β)-LIS

First, Algorithm (m, β)-LIS is presented, for the case of $s \geq \rho$. It follows the Longest-In-System (LIS) scheduling policy, while trying to avoid redundant task executions. More precisely, the machine running the algorithm aims to schedule the task that has been waiting the longest and which will not cause any redundant executions of the task. See the algorithm's pseudo-code (Algorithm 8) for details and observe that when $s \geq \rho$, Algorithm (m, β)-LIS is able to complete one task for each task completed by the off-line algorithm. The redundancy avoidance is actually achieved if the number of pending tasks is *sufficiently large*; if there are at least $\beta \cdot m^2$ tasks pending, where $\beta = \rho$, no two machines schedule the same task.

It will be shown that algorithm (m, β)-LIS is ρ-pending-load competitive for speedup $s \geq \rho$ when $\beta \geq \rho$, but first, a high-level idea of the proof is given. Note, that some properties similar to the ones used for type GroupLIS algorithms are used here, though (m, β)-LIS does not belong to that type of algorithms.

Overview of the proof: The proof focuses on the number of pending tasks, by which the result on the pending load follows. It is assumed by contradiction, that $\#P((m,\beta)\text{-LIS}, \mathsf{A}, \mathsf{E}) \geq \#P(X, \mathsf{A}, \mathsf{E}) + \beta m^2 + 3m$, for $\beta \geq \rho$ and $s \geq \rho$. An execution witnessing this fact is considered with fixed adversarial patterns A and E associated with it, together with the "solution" X.

Then, time instant t^* is defined as the time when $\#P_{t^*}((m,\beta)\text{-LIS}, \mathsf{A}, \mathsf{E}) > \#P_{t^*}(X, \mathsf{A}, \mathsf{E}) + \beta m^2 + 3m$ holds. Let $t_* \leq t^*$ be the smallest time such that $\forall t \in [t_*, t^*)$, $\#P_t((m,\beta)\text{-LIS}, \mathsf{A}, \mathsf{E}) > \#P_t(X, \mathsf{A}, \mathsf{E}) + \beta m^2$. Note that the selection of the smallest time instant satisfying some properties defined by the computation, is possible due to the fact that the computation is split into discrete processing cycles. Also, observe that $\#P_{t_*}((m,\beta)\text{-LIS}, \mathsf{A}, \mathsf{E}) \leq \#P_{t_*}(X, \mathsf{A}, \mathsf{E}) + \beta m^2 + m$, because at time t_* no more than m tasks could be reported to the repository by X, while just before t_* the difference between (m,β)-LIS and X was at most βm^2.

The above definitions are used to prove the following lemmas that will lead to the contradiction of the initial assumption and yield the proof of the claimed result of Theorem 6.6.

Lemma 6.10. *Let $t_* < t^* - \pi_{min}$. For every $t \in [t_*, t_* + \pi_{min}]$ the following holds with respect to the number of pending tasks: $\#P_t((m,\beta)\text{-LIS}, \mathsf{A}, \mathsf{E}) \leq \#P_t(X, \mathsf{A}, \mathsf{E}) + \beta m^2 + 2m$.*

Proof: The case of $t = t_*$ has already been discussed. In the interval $(t_*, t_* + \pi_{min}]$, X can notify the repository about at most m completed tasks, as each of m machines may finish at most one task. Consider any $t \in (t_*, t_* + \pi_{min}]$ and let T be fixed to $(t_*, t]$. It holds that $\#P_t((m,\beta)\text{-LIS}, \mathsf{A}, \mathsf{E}) \leq \#P_{t_*}((m,\beta)\text{-LIS}, \mathsf{A}, \mathsf{E}) + \#I_T$ and $\#P_t(X, \mathsf{A}, \mathsf{E}) \geq \#P_{t_*}(X, \mathsf{A}, \mathsf{E}) + \#I_T - m$, where $\#I_T$ is the number of tasks injected within T. It follows that

$$\begin{aligned}
\#P_t((m,\beta)\text{-LIS}, \mathsf{A}, \mathsf{E}) &\leq \#P_{t_*}((m,\beta)\text{-LIS}, \mathsf{A}, \mathsf{E}) + \#I_T \\
&\leq \big(\#P_{t_*}(X, \mathsf{A}, \mathsf{E}) + \beta m^2 + m\big) + \big(\#P_t(X, \mathsf{A}, \mathsf{E}) - \#P_{t_*}(X, \mathsf{A}, \mathsf{E}) + m\big) \\
&\leq \#P_t(X, \mathsf{A}, \mathsf{E}) + \beta m^2 + 2m.
\end{aligned}$$

It also follows that any such t must be smaller than t^*, by definition of t^*. ■

Lemma 6.11. *Consider a time interval T during which the queue of pending tasks in (m,β)-LIS is always non-empty. Then the total number of tasks reported by X in the period T is not bigger than the total number of tasks reported by (m,β)-LIS in the same period plus m (counting possible redundancy).*

Proof: For each machine in the execution of X, under the adversarial patterns A and E, in the considered period, exclude the first reported task; this is to eliminate from further analysis tasks that might have been started before time interval T. There are at most m such tasks reported by X.

It is left to be shown that the number of remaining tasks reported to the repository by X is not bigger than those reported in the execution of (m,β)-LIS in the considered period T. It follows

from the property that $s \geq \rho$, which implies that during the time period when a machine p executes a task τ in the execution of X, the same machine reports at least one task to the repository in the execution of (m, β)-LIS. This is because executing any task by a machine in the execution of X takes at least time π_{min}, while executing any task in the execution of (m, β)-LIS takes no more than $\frac{\pi_{max}}{s} \leq \pi_{min}$ (recall that $s \geq \rho = \frac{\pi_{max}}{\pi_{min}}$), and also because no active machine in the execution of (m, β)-LIS is ever idle (non-emptiness of the pending task queue in the interval). Hence a 1-1 function can be defined, from the considered tasks completed by X (i.e., tasks which are started and reported in time interval T) to the family of different tasks reported by (m, β)-LIS in the period T, which completes the proof. ■

Lemma 6.12. *In the interval* $(t_* + \pi_{min}, t^*]$ *no task is reported twice to the repository by* (m, β)*-LIS.*

Proof: The proof is by contradiction. Suppose that task τ is reported twice in the considered time interval of the execution of (m, β)-LIS, under adversarial patterns A and E. Consider the first two such reports, by machines p_1 and p_2; w.l.o.g. it may be assumed that p_1 reported τ at time t_1, not later than p_2 reported τ at time t_2. Let π_τ denote the cost of task τ. The considered reports have to occur within time period shorter than the cost of task τ, in particular, shorter than $\pi_{max}/s \leq \pi_{min}$; otherwise it would mean that the machine which reported second would have started executing this task not earlier than the previous report to the repository, which contradicts the property of the repository that each reported task is immediately removed from the list of pending tasks. It also implies that $p_1 \neq p_2$.

From the algorithm description, the queue of pending tasks $\mathbf{P}$ at time $t_1 - \pi_\tau/s$ had task τ at position $p_1\beta n$, while at time $t_2 - \pi_\tau/s$ it had task τ at position $p_2\beta m$. Note that interval $[t_1 - \pi_\tau/s, t_2 - \pi_\tau/s]$ is included in $[t_*, t^*]$, and thus, by the definition of t_*, at any time of this interval there are at least βm^2 tasks in the queue $\mathbf{P}$.

There are two cases to consider. First, if $p_1 < p_2$, then because new tasks in the queue $\mathbf{P}$ are appended to its end, it will never happen that a task with rank $p_1\beta m$ would increase its rank in time, in particular, not to $p_2\beta m$. Second, if $p_1 > p_2$, then during time interval $[t_1 - \pi_\tau/s, t_2 - \pi_\tau/s]$ task τ has to decrease its rank from $p_1\beta m$ to $p_2\beta m$, i.e., by at least βm positions. It may happen only if at least βm tasks ranked before τ on the queue $\mathbf{P}$ at time $t_1 - \pi_\tau/s$ become reported in the considered time interval. Since all of them are of cost at least π_{min}, and the considered time interval has length smaller than π_{max}/s, each machine may report at most $\frac{\pi_{max}/s}{\pi_{min}/s} \leq \beta$ tasks (this is the part of analysis requiring $\beta \geq \rho = \frac{\pi_{max}}{\pi_{min}}$). Since machine p_2 can report at most $\beta - 1$ tasks different than τ, the total number of tasks different from τ reported to the repository is at most $\beta m - 1$, and hence it is not possible to reduce the rank of τ from $p_1\beta m$ to $p_2\beta m$ within the considered time interval. This contradicts the assumption that p_2 reports τ to the repository at time t_2. ■

Theorem 6.6. *For speedup* $s \geq \rho$*, parameter* $\beta \geq \rho$*, and under adversarial patterns* A *and* E*, algorithm* (m, β)*-LIS has completed and pending load competitiveness* $\mathcal{C}((m, \beta)\text{-LIS}) \geq 1/\rho$

and $\mathcal{P}((m,\beta)$-LIS$) \leq \rho$, respectively. More precisely, the following holds for any time t in the execution of (m,β)-LIS:

$$P_t((m,\beta)\text{-}\mathit{LIS}, \mathsf{A}, \mathsf{E}) \leq \rho \cdot \left(\#P_t(X, \mathsf{A}, \mathsf{E}) + \beta m^2 + 3m\right).$$

$$C_t((m,\beta)\text{-}\mathit{LIS}, \mathsf{A}, \mathsf{E}) \geq \frac{1}{\rho} \cdot \left(\#C_t(X, \mathsf{A}, \mathsf{E}) - \beta m^2 - 3m\right)$$

Proof: From Lemma 6.10, $\#P_{t_*+\pi_{min}}((m,\beta)\text{-LIS}, \mathsf{A}, \mathsf{E}) \leq \#P_{t_*+\pi_{min}}(X, \mathsf{A}, \mathsf{E}) + \beta m^2 + 2m$. Now let y be the total number of tasks reported by (m,β)-LIS in $(t_* + \pi_{min}, t^*]$. By Lemma 6.11 and definitions t_* and t^*, X reports no more than $y + n$ tasks in $(t_* + \pi_{min}, t^*]$. Therefore,

$$\#P_{t^*}(X, \mathsf{A}, \mathsf{E}) \geq \#P_{t_*+\pi_{min}}(X, \mathsf{A}, \mathsf{E}) - (y + m).$$

By Lemma 6.12, in the interval $(t_* + \pi_{min}, t^*]$, no redundant work is reported by (m,β)-LIS. Thus,

$$\#P_{t^*}((m,\beta)\text{-LIS}, \mathsf{A}, \mathsf{E}) \leq \#P_{t_*+\pi_{min}}((m,\beta)\text{-LIS}, \mathsf{A}, \mathsf{E}) - y.$$

Consequently,

$$\begin{aligned}
\#P_{t^*}((m,\beta)\text{-LIS}, \mathsf{A}, \mathsf{E}) &\leq \#P_{t_*+\pi_{min}}((m,\beta)\text{-LIS}, \mathsf{A}, \mathsf{E}) - y \\
&\leq \left(\#P_{t_*+\pi_{min}}(X, \mathsf{A}, \mathsf{E}) + \beta m^2 + 2m\right) - y \\
&\leq \#P_{t^*}(X, \mathsf{A}, \mathsf{E}) + (\beta m^2 + 2m) + m \\
&\leq \#P_{t^*}(X, \mathsf{A}, \mathsf{E}) + \beta m^2 + 3m
\end{aligned}$$

as desired. This contradicts the initial definition of time t^* in the overview of the proof, and hence $\#P_{t^*}((m,\beta)\text{-LIS}, \mathsf{A}, \mathsf{E}) \leq \#P_{t^*}(X, \mathsf{A}, \mathsf{E}) + \beta m^2 + 3m$, from which the result of pending load competitiveness is a direct consequence, as the size of any pending task in (m,β)-LIS is at most $\frac{\pi_{max}}{\pi_{min}} = \rho$ times bigger than the size of any pending task in X. Therefore, $P_t((m,\beta)\text{-LIS}, \mathsf{A}, \mathsf{E}) \leq \rho \cdot \left(\#P_t(X, \mathsf{A}, \mathsf{E}) + \beta m^2 + 3m\right)$ as claimed.

For the completed load, since the number of completed tasks is complementary to the number of pending ones, it means that for all time instants t, $\#C_t((m,\beta)\text{-LIS}, \mathsf{A}, \mathsf{E}) \leq \#C_t(X, \mathsf{A}, \mathsf{E}) - \beta m^2 - 3m$ and with the same argument as for the pending load, the corresponding completed load will be $C_t((m,\beta)\text{-LIS}, \mathsf{A}, \mathsf{E}) \leq \rho \cdot (\#C(X, \mathsf{A}, \mathsf{E}) - \beta m^2 - 3m)$, as also claimed. ■

Observe that algorithm (m,β)-LIS uses the parameter β explicitly, which is critical for the proof of Lemma 6.12. According to its value, and if there are enough tasks in the queue, it achieves complete redundancy avoidance. More precisely, redundancy is avoided when β is no smaller than ρ and there are more than βm^2 tasks pending. If (some upper bound on) ρ is not available to the algorithm, then an inaccurate estimate of the value of β does not guarantee complete redundancy avoidance, causing the above claimed competitiveness (and its proof) not to hold.

6.4.2. γm-Burst: an *optimal* algorithm for $M\langle m, \left[1+\frac{\gamma}{\rho}, \rho\right), 2\rangle$

Consider an adversarial strategy that at the beginning of the execution injects only one π_{max}-task and then continues only with π_{min}-task injections. If algorithm (m, β)-LIS runs in a system with one machine under such adversary while condition **C1** holds, i.e., speedup $s < \rho$, it will have unbounded pending-load competitiveness, i.e., $\mathcal{C}((m, \beta)\text{-LIS}) = \infty$. This is true due to the algorithm's nature to insist on scheduling the same task over and over again when stopped by a crash. An optimal algorithm on the other hand, would execute the task with the appropriate size in each alive interval of the machine. What is more, this can be generalized for m machines, and it is also the case for algorithms that use other scheduling policies, e.g. scheduling first the larger tasks. This suggests that when condition **C1** holds, a scheduling policy that alternates executions of smaller and larger tasks should be devised.

It is shown here, that if the speedup satisfies condition **C1** $\wedge \neg$**C2**, which implies $1 + \gamma/\rho \leq s < \rho$, and the tasks can have only two different sizes, π_{min} and π_{max}, then there is an algorithm, called γm-Burst, that achieves 1-pending-load competitiveness and 1-completed-load competitiveness in a system with m machines. It is the generalization of algorithm γ-Burst presented earlier, in the case of one machine, and it will be shown that it keeps its property of competitiveness while guaranteeing non redundant task executions when "enough" tasks are pending. See the algorithm's pseudo-code (Algorithm 6) for details and the overview of the main idea behind the algorithm presented first.

Algorithm description

Each machine gets the ordered sets of pending tasks, $\mathbf{P}_{\pi_{min}}$ and $\mathbf{P}_{\pi_{max}}$, corresponding to the tasks of size π_{min} and π_{max} respectively. Recall that the tasks are ordered by their arrival times. Following the same idea of (m, β)-LIS and all GroupLIS algorithms, γm-Burst avoids redundancy when "enough" tasks are pending. Furthermore, the algorithm needs to take into consideration parameter γ and the bounds on speedup s. In particular, in the case that there exist enough π_{min}- and π_{max}-tasks (more than m^2 to be exact), each machine completes no more than γ consecutive π_{min}-tasks and then a π_{max}-task. This is equal to the time it takes for the same machine to complete a π_{max}-task in X. To this respect, a counting variable c is used to keep track of the number of consecutive π_{min}-tasks completed, which *resets* when a π_{max}-task is completed. Special care needs to be taken for all other cases, e.g., when there are more than m^2 tasks of size π_{max} pending but less than m^2 tasks of size π_{min}, etc.

Algorithm analysis

Recall that in Section 6.1 the class of algorithms GroupLISwas defined, to which algorithm γm-Burst belongs. It is therefore known, that if enough tasks are pending there will be no redundant task executions. Observe that γm-Burst attempts to alternate the execution of γ π_{min}-tasks with one π_{max}-task, and $s \geq 1 + \gamma/\rho = \frac{\gamma\pi_{min}+\pi_{max}}{\pi_{max}}$. Then, if there are enough pending π_{max}-

Algorithm 6: **γm-Burst** (for machine p)

Parameters: $m, s, \pi_{min}, \pi_{max}$
Calculate $\gamma \leftarrow \lceil \frac{\pi_{max} - s\pi_{min}}{(s-1)\pi_{min}} \rceil$
Repeat //Upon awaking or restart
  $c \leftarrow 0$; //Reset counter
  Get from the Repository the sets of pending tasks $\mathbf{P}_{\pi_{min}}$ and $\mathbf{P}_{\pi_{max}}$;
  Case 1: $\#P_{\pi_{min}} < m^2$ and $\#P_{\pi_{max}} < m^2$
    If previously executed task was a π_{min} **then**
      execute task $(p \cdot m) \mod nP_{\pi_{max}}$ in $\mathbf{P}_{\pi_{max}}$;
      $c \leftarrow 0$; //Reset counter
    else execute task $(p \cdot m) \mod \#P_{\pi_{min}}$ in $\mathbf{P}_{\pi_{min}}$;
      $c \leftarrow \min(c+1, \gamma)$;
  Case 2: $\#P_{\pi_{min}} \geq m^2$ and $\#P_{\pi_{max}} < m^2$
    execute the task at position $p \cdot m$ in $\mathbf{P}_{\pi_{min}}$;
    $c \leftarrow \min(c+1, \gamma)$;
  Case 3: $\#P_{\pi_{min}} < m^2$ and $\#P_{\pi_{max}} \geq m^2$
    execute the task at position $p \cdot m$ in $\mathbf{P}_{\pi_{max}}$;
    $c \leftarrow 0$; //Reset counter
  Case 4: $\#P_{\pi_{min}} \geq m^2$ and $\#P_{\pi_{max}} \geq m^2$
    If $c = \gamma$ **then**
      execute task at position $p \cdot m$ in $\mathbf{P}_{\pi_{max}}$;
      $c \leftarrow 0$; //Reset counter
    else execute task at position $p \cdot m$ in $\mathbf{P}_{\pi_{min}}$;
      $c \leftarrow \min(c+1, \gamma)$;
  Inform the Repository of the task executed.

tasks, γm-Burst completes at least roughly the same number as any algorithm X. Similarly, if there are enough pending π_{min}-tasks, γm-Burst completes again at least roughly the same number as any algorithm X. Combining these results the completed and pending load competitiveness bounds are derived.

Consider two types of intervals, for which the observation that follows holds due to Lemma 6.1:

T^+: an interval where $\#P_t(\gamma\text{m-Burst}, \pi_{max}) \geq m^2, \forall t \in T^+$

T^-: an interval where $\#P_t(\gamma\text{m-Burst}, \pi_{min}) \geq m^2, \forall t \in T^-$

Observation 6.2. *All absolute task executions of π_{max}-tasks in Algorithm γm-Burst within any interval T^+ appear exactly once. In a similar way, all absolute task executions of π_{min}-tasks in Algorithm γm-Burst within any interval T^- appear exactly once too.*

The above leads to the following upper bound on the difference in the number of pending π_{max}-tasks.

Lemma 6.13. *The number of pending π_{max}-tasks in any execution of γm-Burst, under any adversarial patterns* A *and* E*, running with speedup $s \geq 1 + \gamma/\rho$, is never larger than the number of pending π_{max}-tasks in the execution of X plus $m^2 + 2m$.*

Proof: Fix a combination of adversarial patterns A and E, and consider for contradiction, interval $T^+ = (t_*, t^*]$, where t^* is the first time when $\#P_{t^*}(\gamma\text{m-Burst}, \pi_{max}) > \#P_{t^*}(X, \pi_{max}) + m^2 + 2m$, and t_* is the largest time before t^* such that $\#P_{t_*}(\gamma\text{m-Burst}, \pi_{max}) < m^2$.

Claim: The number of absolute task executions of π_{max}-tasks $\alpha \subset T^+$, by X, is no bigger than the number of π_{max}-task reports by γm-Burst in interval T^+.

Proof of claim: Since $s \geq 1 + \gamma/\rho = \frac{\gamma\pi_{min}+\pi_{max}}{\pi_{max}}$, while machine p is running a π_{max}-task in the execution of X, the same machine has time to execute $\gamma\pi_{min} + \pi_{max}$ tasks in the execution of γm-Burst. But, by definition, within the interval T^+ there are at least m^2 π_{max}-task pending at all times, which implies the execution of Case 3 or Case 4 of the γm-Burst algorithm. This means that no machine may run $\gamma + 1$ consecutive π_{min}-tasks, as a π_{max}-task is guaranteed to be executed by one of the cases. Hence, as claimed, the number of absolute task executions of π_{max}-tasks by X in the interval T^+ is no bigger than the number of π_{max}-task reports by γm-Burst in the same interval.

Now let κ be the number of π_{max}-tasks reported by X. From Lemma 6.2, at least $\kappa - m$ such tasks have absolute task executions in interval T^+. From the above claim, for every absolute task execution of π_{max}-tasks in the interval T^+ by X, there is at least a completion of a π_{max}-task by γm-Burst which gives a 1-1 correspondence, so γm-Burst has at least $\kappa - m$ reported π_{max}-tasks in T^+. Also, from Lemma 6.2, it may be concluded that there are at least $\kappa - 2m$ absolute task executions of π_{max}-tasks in the interval. Then from Lemma 6.1, γm-Burst reports at least $\kappa - 2m$ different tasks, while X reports at most κ.

Now let $\mathbf{I}_{T^+}(\mathsf{A}, \pi_{max})$ be the set of π_{max}-tasks injected during the interval T^+, under adversarial arrival pattern A. Then, for the number of π_{max}-tasks pending at time t^*, it holds that $\#P_{t^*}(\gamma\text{m-Burst}, \pi_{max}) < m^2 + \#I_{I^+}(\mathsf{A}, \pi_{max}) - (\kappa - 2m)$, and since $\#P_{t^*}(X, \pi_{max}) \geq \#I_{T^+}(\mathsf{A}, \pi_{max}) - \kappa$ it leads to a contradiction, which completes the proof. ■

Theorem 6.7. *$\#P_t(\gamma$m-Burst$, \mathsf{A}, \mathsf{E}) \leq \#P_t(X, \mathsf{A}, \mathsf{E}) + 2m^2 + (3 + \lceil\rho/s\rceil)m$, for any time t and adversarial patterns* A *and* E.

Proof: Consider any combination of adversarial patterns A and E, and for contradiction the interval $T^- = (t_*, t^*]$ as defined above, where t^* is the first time when $\#P_{t^*}(\gamma\text{m-Burst}, \mathsf{A}, \mathsf{E}) > \#P_{t^*}(X, \mathsf{A}, \mathsf{E}) + 2m^2 + (3 + \lceil\rho/s\rceil)m$ and t_* being the largest time before t^* such that $\#P_{t_*}(\gamma\text{m-Burst}, \pi_{max}) < m^2$. Notice that t_* is well defined in Lemma 6.13, i.e., such time t_* exists and it is smaller than t^*.

Consider each machine individually and break the interval T^- into sub-intervals $[t, t']$ such that times t and t' are instances in which the counter c is reset to 0; this can be either due to a simple reset in the algorithm or due to a crash and restart of a machine. More concretely, the boundaries of such sub-intervals are as follows. An interval can start either when a reset of the counter occurs or when the machine (re)starts. On its side, an interval can finish due to either a reset of the counter or a machine crash. Hence, these sub-intervals can be grouped into two types, depending on how they end: Type (a) which includes the ones that end by a crash and

Type (b) which includes the ones that end by a reset from the algorithm. Note that in all cases γm-Burst starts the sub-interval scheduling a new task to the machine at time t, and that the machine is never idle in the interval. Hence, all tasks reported by γm-Burst have their absolute task executions completely into the sub-interval. The goal is to show that the number of absolute task executions in each such sub-interval with γm-Burst is no less than the number of reported tasks by X.

First, consider a sub-interval $[t, t']$ of Type (b), that is, such that the counter c is set to 0 by the algorithm (in a line $c = 0$) at time t'. This may happen in algorithm γm-Burst in Cases 1, 3 or 4. However, observe that the counter cannot be reset in Cases 1 and 3 at time $t' \in T^-$ since, by definition, there are at least m^2 π_{min}-tasks pending during the whole interval I^-. Case 4 implies that there are also at least m^2 π_{max}-tasks pending in γm-Burst. This means that in the interval $[t, t']$ there have been κ π_{min} and one π_{max} absolute task executions, $\kappa \geq \gamma$. Then, the sub-interval $[t, t']$ has length $\frac{\pi_{max}+\kappa\pi_{min}}{s}$, and X can report at most $\kappa+1$ task completions during the sub-interval. This latter property follows from $\frac{\pi_{max}+\kappa\pi_{min}}{s} = \frac{\pi_{max}+\gamma\pi_{min}}{s} + \frac{(\kappa-\gamma)\pi_{min}}{s} \leq (\gamma+1)\pi_{min} + (\kappa-\gamma)\pi_{min} \leq (\kappa+1)\pi_{min}$, where the first inequality follows from the definition of γ and the fact that $s > 1$. Now consider a sub-interval $[t, t']$ of Type (a) which means that at time t' there was a crash. This means that no π_{max}-task was completed in the sub-interval, but it can be safely assumed that there were κ complete executions of π_{min}-tasks in γm-Burst. It is now shown, that X cannot report more than κ task completions. In the case where $\kappa \geq \gamma$, then the length of the sub-interval $[t, t']$ satisfies

$$t' - t < \frac{\kappa\pi_{min} + \pi_{max}}{s} \quad \leq \quad (\kappa+1)\pi_{min}.$$

In the case where $\kappa < \gamma$ then the length of the sub-interval $[t, t']$ satisfies

$$t' - t < \frac{(\kappa+1)\pi_{min}}{s} \quad \leq \quad (\kappa+1)\pi_{min}.$$

Then in none of the two cases X can report more than κ tasks in sub-interval $[t, t']$.

After splitting T^- into the above sub-intervals, the whole interval is of the form $(t_*, t_1][t_1, t_2] \ldots [t_l, t^*]$. All the intervals $[t_i, t_{i+1}]$ where $t = 1, 2, \ldots, l$, are included in the sub-interval types already analyzed. There are therefore two remaining sub-intervals to consider now. The analysis of sub-interval $[t_l, t^*]$ is verbatim to that of an interval of Type (a). Hence, the number of absolute task executions in that sub-interval with γm-Burst is no less than the number of reported tasks by X.

Consider now the sub-interval $(t_*, t_1]$. Assume with γm-Burst there are κ absolute task executions fully contained in the sub-interval. Also observe that at most one π_{max}-task can be reported in the sub-interval (since then the counter is reset and the sub-interval ends). Then, the length of the sub-interval is bounded as

$$t_1 - t_* < \frac{(\kappa+1)\pi_{min} + \pi_{max}}{s}$$

(assuming the worst case that a π_{min}-task was just started at t_* and that the machine crashed at t_1 when a π_{max}-task was about to finish). The number of tasks that X can report in the sub-interval is hence bounded by

$$\left\lceil \frac{(\kappa+1)\pi_{min}+\pi_{max}}{s\pi_{min}} \right\rceil = \left\lceil \frac{(\kappa+1)+\rho}{s} \right\rceil < \kappa+1+\left\lceil \frac{\rho}{s} \right\rceil .$$

This means that for every machine, the number of reported tasks by X might be at most the number of absolute task executions by γm-Burst fully contained in T^- plus $1+\lceil \rho/s \rceil$. From this and Observation 6.2, it follows that in interval T^- the difference in the number of pending tasks between γm-Burst and X has grown by at most $(1+\lceil \rho/s \rceil)m$. Observe that at time t_* the difference between the number of pending tasks satisfied

$$\#P_{t_*}(\gamma\text{m-Burst}, \mathsf{A}, \mathsf{E}) - \#P_{t_*}(X, \mathsf{A}, \mathsf{E}) < 2m^2 + 2m,$$

This follows from Lemma 6.13, which bounds the difference in the number of π_{max}-tasks to m^2+2m, and the assumption that $\#P_{t_*}(\gamma\text{m-Burst}, \pi_{max}) < m^2$. Then, it follows that $\#P_{t^*}(\gamma\text{m-Burst}, \mathsf{A}, \mathsf{E}) - \#P_{t_*}(X, \mathsf{A}, \mathsf{E}) < 2m^2+2m+(1+\lceil \rho/s \rceil)m = m^2+(3+\lceil \rho/s \rceil)m$, which is a contradiction. Hence, $\#P_t(\gamma\text{m-Burst}, \mathsf{A}, \mathsf{E}) \leq \#P_t(X, \mathsf{A}, \mathsf{E}) + 2m^2 + (3+\lceil \rho/s \rceil)m$, for any time t and adversarial patterns A and E, as claimed. ■

The difference in the number of π_{max}-tasks between ALG and X can be bounded by m^2+2m (see Lemma 6.13). This, and Theorem 6.7, yield the following bound on the pending load of γm-Burst, which implies that it is 1-pending-load competitive.

Theorem 6.8. *$P_t(\gamma$m-Burst$, \mathsf{A}, \mathsf{E}) \leq P_t(X, \mathsf{A}, \mathsf{E}) + \pi_{max}(m^2+2m) + \pi_{min}(m^2+(1+\lceil \rho/s \rceil)m)$, for any time t and adversarial patterns A and E, i.e., $\mathcal{P}(\gamma$m-Burst$) \leq 1$. It is trivial that $\mathcal{C}(\gamma$m-Burst$) \geq 1$ holds too.*

Unlike Algorithm (m, β)-LIS, even though γm-Burst uses the two task sizes (see Algorithm 6), there is a simple way for it to work without having that knowledge. Algorithm γm-Burst works only for the case that there are two different task sizes, maintaining two queues for the tasks according to their size. Even if the values of π_{min} and π_{max} are not given, it can follow the following strategy and still be 1-pending-load competitive.

As long as the pending tasks are all of the same size, no specific scheduling policy is necessary and therefore the simplest strategy to be followed is the *Longest In System*. As soon as two tasks of different size arrive in the system, the machines can distinguish them by looking at their specifications. The algorithm will extract the two values, π_{max} and π_{min}, calculate the value of γ and create the corresponding queues as seen in the pseudo-code (Algorithm 6). Hence, the following observation.

Observation 6.3. *The analysis of Algorithm γm-Burst holds even in the case that the values of π_{min} and π_{max} are unknown to the algorithm.*

Algorithm 7: (m, β)**-LAF** (for machine p)

```
1  Parameters: C, m, β
2  total ← 0                                          //Upon awaking or restart
3  Repeat
4     Get from the Repository the sets of pending tasks P_x, ∀x ∈ C;
5     If {x ∈ C : x ≤ total and #P_x ≥ βm^2} ≠ ∅ then
6        x_max ← arg max{x ∈ C : x ≤ total ∧ #P_x ≥ βm^2};
7        execute task p · βm in P_{x_max};
8        total ← total + x_max;
9     else
10       execute a random task w in P;
11       total ← total + π(w);
12    Inform the Repository of the task executed;
```

6.4.3. (m, β)-LAF: an *optimal* algorithm for $M\langle m, 7/2, k\rangle$

In the case of only two different sizes, a competitive solution for speedup that matches the lower bound from Theorem 5.9 can be obtained. More precisely, for given two different size values, π_{min} and π_{max}, one can compute the minimum speedup s^* satisfying condition C2 from Theorem 5.9 for these two sizes, and choose (m, β)-LIS with speedup ρ in case $\rho \leq s^*$ and γm-Burst with speedup s^* otherwise[1]. However, in the case of more than two different task sizes algorithm γm-Burst cannot be used, and so far one could only rely on (m, β)-LIS with speedup ρ, which may be large.

Here,a "substitute" for algorithm γm-Burst is designed, working for any *finite* set C of different task sizes in the interval $[\pi_{min}, \pi_{max}]$, that is competitive for some fixed speedup ($s \geq 7/2$ to be exact). Note that $s \geq 2$ is enough to guarantee that condition **C2** does not hold. This algorithm can therefore be used when ρ is large.

The new algorithm is called *Largest_Amortized_Fit* (LAF for short). It is parametrized by m and $\beta \geq \rho$ and is more "geared" towards pending and completed load efficiency. In particular, each machine keeps the variable $total$, storing the total load of tasks reported by machine p, since the last restart (recall that upon a restart machines have no recollection of the past). Each machine schedules a task from the queue of pending tasks which has the largest size that is not bigger than $total$ and is such that there are at least βm^2 tasks of that size pending, for $\beta \geq \rho$. Recall the assumption that all sets of pending tasks $\mathbf{P}_x$ have the tasks sorted using the Longest-in-System (LIS). If there is no size meeting these requirements, the machine schedules an arbitrary pending task. See the algorithm's pseudo-code (Algorithm 7) for details.

This algorithm, together with algorithm (m, β)-LIS, guarantee competitiveness for speedup $s \geq \min\{\rho, 7/2\}$. In more detail, one could apply (m, β)-LIS with speedup ρ when $\rho \leq 7/2$ and (m, β)-LAF with speedup $7/2$ otherwise.

In the following theorem, it is proved that in order for the algorithm to be competitive, the number of different sizes of injected tasks, i.e. k, must be finite in the range $[\pi_{min}, \pi_{max}]$. Oth-

[1] Note that s^* is upper bounded by 2, as explained in Section 5.3.3.

erwise, the number of tasks of the same size might never be larger than βm^2, which is necessary to assure redundancy avoidance. Whenever this redundancy avoidance is possible, the algorithm behaves in a conservative way in the sense that it schedules a large task, but not larger than the total load already completed. This implies that in every alive interval of a machine (the continuous period between a restart and a crash of the machine) only a constant fraction of this period could be wasted (with respect to the total completed task load covered by X in the same period). Based on this observation, a non-trivial argument shows that a constant speedup suffices for obtaining 1-pending and thus 1-completed load competitiveness.

Theorem 6.9. *Algorithm (m, β)-LAF is 1-pending-load and 1-completed-load competitive, for speedup $s \geq 7/2$, provided the number of different sizes of tasks in the execution is finite.*

Proof: Note that algorithm (m, β)-LAF belongs to the class of GroupLIS(β) algorithms, for $\beta \geq \rho$. Therefore Lemma 6.1 applies, and together with the algorithm specification it guarantees no redundancy in absolute task executions in case that one of the queues has at least βm^2 tasks pending.

Consider any combination of adversarial arrival and error patterns A and E. It is shown now that

$$P_t^*((m, \beta)\text{-LAF}, \geq x) \leq P_t^*(X, \geq x) + 2k\beta m^2 \pi_{max} + 2m\pi_{max} + 3m\frac{\pi_{max}}{s}$$

for every size x at any time t and for speedup s, where $P_t^*(\text{ALG}, \geq x)$ denotes the sum of sizes of pending tasks of size at least x, and such that the number of pending tasks of such size is at least βm^2 in (m, β)-LAF at time t of its execution, under adversarial patterns A and E; k is the number of the possible different task sizes that are injected under A. Note that this implies the statement of the theorem, since if x is taken equal to the smallest possible size, and an upper bound $k\beta m^2 \pi_{max}$ is added on the size of tasks on pending queues of (m, β)-LAF of size smaller than βm^2, the upper bound on the amount of pending load of (m, β)-LAF is obtained, for any combination of adversarial patterns A and E.

Fix some size x, and adversarial patterns A and E. Assume now, by contradiction, that the sought property does not hold, and let t^* be the first time t when $P_t^*((m, \beta)\text{-LAF}, \geq x) > P_t^*(X, \geq x) + 2k\beta m^2 \pi_{max} + 2m\pi_{max} + 3m\frac{\pi_{max}}{s}$ for task size x. Denote by t_* the largest time before t^* such that for every $t \in (t_*, t^*]$, the following holds:

$$P_t^*((m, \beta)\text{-LAF}, \geq x) \geq P_t^*(X, \geq x) + k\beta m^2 \pi_{max}.$$

Observe that t_* is well-defined, and moreover, $t_* \leq t^* - (\pi_{max} + 3\frac{\pi_{max}}{s})$: it follows from the definition of t^* and from the fact that within a time interval $(t, t^*]$ of length smaller than $\pi_{max} + 3\frac{\pi_{max}}{s}$, X can report tasks of total load at most $2m\pi_{max} + 3n\frac{\pi_{max}}{s}$, plus additional load of at most $k\beta m^2 \pi_{max}$ that can be caused by other queues growing beyond the threshold βm^2, and thus starting to contribute to the pending load P^*.

Consider now, interval $(t_*, t^*]$. By the specification of t_*, at any time of the interval there is

at least one queue of pending tasks of size at least x that has length at least βm^2. Consider an alive interval of a machine p that starts in the considered time period; restrict this alive interval up to time t^*, and let π be the actual length of this interval. Let $z > 0$ be the total load of tasks, when counted only those of size at least x, reported by machine p in the execution of X in the considered alive interval. It is argued, that in the same time interval, the total size of tasks (when counting only those of size at least x) reported by p in the execution of (m, β)-LAF, is at least z. Observe that once machine p in (m, β)-LAF schedules a task of size at least x for the first time in the considered period, it continues scheduling a task of size at least x until the end of the considered period. Therefore, with respect to the corresponding execution of X, machine p could only waste its time (from perspective of executing a task of size smaller than x or executing a task not reported in the considered period) in the first less than $(2x)/s$ time of the period or the last less than $(\pi/2)/s$ time of the period. Therefore, in the remaining period of length bigger than $\pi - (\pi/2 + 2x)/s$, machine p is able to complete and report tasks, each of size at least x, of total load larger than

$$s\pi - (\pi/2 + 2x) \geq \pi(s - 1/2 - 2) \geq \pi \geq z\,;$$

here in the first inequality we used the fact that $\pi \geq x$, which follows from the definition of $z > 0$, and in the second inequality the property $s - 1/2 - 2 \geq 1$ for $s \geq 7/2$ is used. Applying Lemma 6.12, justifying no redundancy in absolute tasks executions of (m, β)-LAF in the considered time periods, one can conclude that alive intervals as considered do not contribute to the growth of the difference between $P^*((m, \beta)\text{-LAF}, \geq x)$ and $P^*(X, \geq x)$.

Therefore, only alive intervals that start before t_* can contribute to the difference in loads. However, if their intersections with the time interval $(t_*, t^*]$ is of length π at least $(2x + \pi_{max})/s$, that is, enough for a machine running (m, β)-LAF to report at least one task of length at least x, the same argument as in the previous paragraph yields that the total load of tasks of size at least x reported by a machine in the execution of (m, β)-LAF is at least as large as in the execution of X, minus the size of the very first task reported by each machine in (m, β)-LAF (which may not be an absolute task execution and thus there may be redundancy on them) — i.e., minus at most $m\pi_{max}$ in total. In the remaining case, i.e., when the intersection of the life period with $(t_*, t^*]$ is smaller than $(2x + \pi_{max})/s$, the machine may not report any task of length x when running (m, β)-LAF, but when executing X the total size of all reported tasks is smaller than $(2x + \pi_{max})/s \leq 3\pi_{max}/s$. Therefore, the difference in sizes on tasks of size at least x between X and (m, β)-LAF could grow by at most $m\pi_{max} + 3m\pi_{max}/s$ in the alive intervals considered in this paragraph. Hence, $P^*_{t^*}((m, \beta)\text{-LAF}, \geq x) - P^*_{t^*}(X, \geq x) \leq P^*_{t_*}((m, \beta)\text{-LAF}, \geq x) - P^*_{t_*}(X, \geq x) + m\pi_{max} + 3m\frac{\pi_{max}}{s} \leq \pi_{max} k\beta m^2 + m\pi_{max} + 3m\frac{\pi_{max}}{s}$, which violates the initial contradictory assumption. This proved the 1-pending-load competitiveness of algorithm (m, β)-LAF and thus the 1-completed-load competitiveness follows. ■

Observe that since Algorithm (m, β)-LAF uses parameter β in a similar manner as Algorithm (m, β)-LIS, its claimed competitiveness depends on the knowledge of (an upper bound on) ρ.

Redundancy of task executions is avoided when β is no smaller than ρ and there are more than βm^2 tasks pending. If some upper bound on ρ is not available, then an inaccurate estimate of the value of β might lead to redundant task executions that will affect the actual competitiveness.

Part II

Packet Scheduling

Chapter 7

The Impact of Adversarial Jamming

This chapter, studies the first problem of online packet scheduling over an unreliable wireless communication link considered in the thesis. In particular, it investigates the *impact of adversarial jamming* on a wireless channel between two network nodes. The problem of interest is to achieve efficient packet transmissions over the link with worst-case occurrence of errors. The problem and the exact model is described with more details in the first section, along with a small summary of the focus of the study.

7.1. Problem definition and model specifications

Consider a single communication channel between two nodes, a sender and a receiver, where packets of different lengths are being transmitted. The sender needs to make scheduling decisions without the knowledge of future packets, and thus the online nature of the problem. One can easily assume that the channel may be jammed, thus corrupting the packet that is being transmitted at the time. The scope of the thesis for this problem is to do worst-case analysis, thus the channel jams are modeled by an adaptive adversary like in the case of task scheduling.

Packet scheduling performance is often evaluated using throughput, measured in absolute terms (e.g., in bits per second) or normalized with respect to the bandwidth (maximum transmission capacity) of the link. This throughput metric makes sense for a link without errors or with random errors, where the full capacity of the link can be achieved under certain conditions. However, if adversarial bit errors can occur during the transmission of packets, the full capacity is usually not achievable by any protocol, unless restrictions are imposed on the adversary [12, 83]. Moreover, since a bit error renders a whole packet unusable (unless costly techniques like Partial Packet Recovery (PPR) [57] are used), a throughput equal to the capacity minus the bits with errors is not achievable either. As a consequence, in a link with adversarial bit errors, a fair comparison should compare the throughput of a specific algorithm to the maximum achievable amount of traffic that any protocol could send across the link. This introduces the challenge of identifying an appropriate metric to measure the throughput of a protocol over a link with adversarial errors.

The *asymptotic throughput* metric is therefore introduced. This performance metric corresponds to the long-term competitive ratio with regard to the throughput of the channel. Observe here, the connection with the long-term completed-load competitiveness measure defined in the task scheduling part. Due to this connection, the results from the single machine analysis without speedup, can be directly translated to this setting. Hence, the chapter focuses mostly on stochastic packet arrivals and completes a stochastic analysis comparing the two results.

The chapter also studies the impact of *deferred* and *instantaneous* feedback mechanisms for the notification of the sender for the packet jams. The usual mechanism [65], is the deferred feedback; it detects and notifies the sender that a packet has suffered an error after the whole packet has been received by the receiver. For this, it is easily shown (see Section 7.2) that even when the packet arrivals are stochastic and packets have the same length, no online scheduling algorithm can be competitive with respect to the offline one. Hence, the second mechanism, the *instantaneous feedback*, is of more interest. It detects and notifies the sender of an error, at the moment this error occurs. This mechanism can be thought of as an abstraction of the emerging Continuous Error Detection (CED) framework [78] that uses arithmetic coding to provide continuous error detection. The difference between deferred and instantaneous feedback is drastic, since for the instantaneous feedback mechanism, and for packets of the same length, it is easy to obtain optimal asymptotic throughput of 1, even in the case of adversarial arrivals (see Proposition 5.1 presented earlier for the corresponding completed load competitiveness in the task scheduling problem). However, the problem becomes substantially more challenging in the case of non-uniform packet lengths.

7.1.1. Network setting

Consider a sending station transmitting packets over a communication link, or channel. Packets arrive at the sending station continuously and may have different lengths; each of them associated with its arrival time (based on the station's local clock) and its length. Similar to the part of task scheduling, π_{min} and π_{max} denote the smallest and largest possible packet lengths respectively, and the same notation $\rho = \pi_{max}/\pi_{min}$, $\bar{\rho} = \lfloor \rho \rfloor$ and $\hat{\rho} = \lceil \rho \rceil - 1$ is used for the ratio between the packet sizes.

An important assumption made, is that all packets are transmitted at the same bit rate through the link, hence the transmission time is proportional to the packet's length. However, the link is unreliable, that is, transmitted packets might be erroneous; corrupted by bit errors.

7.1.2. Arrival models

For the packet arrivals there are two models of interest:

- *Adversarial:* The packets' arrival time and length are governed by an adversary. An adversarial arrival pattern is defined as a collection of packet arrivals caused by the adversary. In this case A is the adversarial pattern.

- *Stochastic:* In this case, consider a probabilistic distribution D_a, under which packets arrive at the sending station and a probabilistic distribution D_s, for the length of the packets. In particular, assume packets arriving according to a Poisson process with parameter $\lambda > 0$. When considering two packet lengths, π_{min} and π_{max}, each packet that arrives is assigned one of the two lengths independently, with probabilities $p > 0$ and $q > 0$ respectively, where $p + q = 1$. A_S denotes the stochastic arrival pattern.

7.1.3. Packet bit errors / Channel jams

For the channel jams, consider an adversary that controls the bit errors of the packets transmitted over the link. An adversarial error pattern E is defined as a collection of error events (or jams) on the communication link caused by the adversary. More precisely, an error event at time t specifies that an instantaneous jam occurs on the link at time t, and the packet that happens to be on the link at that time is corrupted with bit errors.

What is more, a corrupted packet transmission is considered to be unsuccessful, therefore the packet needs to be retransmitted in full. The main focus of this chapter is on the *instantaneous feedback* mechanism for the notification of the sender about the error; the sending station is notified about the bit error as soon as it happens. However, in the case of *deferred feedback* the sending station is only notified about the error when the packet is fully received by the receiving end of the channel. As mentioned earlier, the usual mechanism is the deferred feedback. The detection and notification of the sender for a corrupted packet happens after the whole packet has been received by the receiver; i.e., if the jam ocurs at time t, then the sender will only be notified after $t_\pi + \pi$, where t_π the time instant when the packet has started being transmitted. On the other hand, the instantaneous feedback mechanism detects and notifies the sender of an error, at the moment this error occurs; i.e., at time instant t.

7.1.4. The power of the adversary

Adversarial models are typically used to argue about the algorithm's behavior in worst-case scenarios. In this work an *adaptive offline* adversary is assumed: one that knows the algorithm and the history of the execution up to the current point in time. In the case of stochastic packet arrivals, this includes all stochastic packet arrivals up to this point, and the length of the packets that have arrived. However it only knows the distribution, but neither the exact timing nor the length of the packets arriving beyond the current time.

Note that, in the model of adversarial arrivals the adversary has full knowledge of the computation, as it controls both packet arrivals and errors, and can simulate the behavior of the algorithm in the future (there are no random bits involved in the computation). This is not the case in the model with stochastic arrivals, where the adversary does not control the timing of future packet arrivals, but knows only about the distributions of the arrival times and lengths.

7.1.5. Efficiency metric: asymptotic throughput

Due to dynamic packet arrivals and adversarial errors, the real link capacity may vary throughout the execution. Therefore, the problem of packet scheduling in this setting is an online problem, for which long-term competitive analysis is pursued. Specifically, let A be an arrival pattern and E an error pattern. For a given deterministic algorithm ALG, let $C_t(\text{ALG}, \mathsf{A}, \mathsf{E})$ be the total length of all the successfully transmitted (i.e., non-corrupted) packets by time t under patterns A and E. Observe that it corresponds to the long-term *completed load* of the Task Scheduling Part. Now let X be an algorithm that knows the exact arrival and error patterns before the start of the execution and thus may devise an optimal schedule that maximizes at each time t the sum of lengths of the successfully transmitted packets, $C_t(X, \mathsf{A}, \mathsf{E})$.

Observe that in the case of stochastic arrivals, the worst-case adversarial error pattern may depend on the stochastic injections. Therefore, E is considered a function of a stochastic arrival pattern A_S and time t. In particular, for an arrival pattern A_S consider a function $\mathsf{E} = E(\mathsf{A}_S, t)$ that defines errors up to time t based on the behavior of a given algorithm ALG under the arrival pattern A_S up to time t and the values of function $E(\mathsf{A}_S, t')$ for $t' < t$.

Let $\mathcal{A}$ denote the considered arrival model, i.e., a set of arrival patterns in case of adversarial, or a distribution of packet arrival patterns in case of stochastic, and let $\mathcal{E}$ denote the corresponding adversarial error model, i.e., a set of error patterns derived by the adversary, or a set of functions defining the error event times in response to the arrivals that already took place in case of stochastic arrivals. What is more, any pair of arrival and error patterns $\mathsf{A} \in \mathcal{A}$ and $\mathsf{E} \in \mathcal{E}$ must allow non-trivial communication, that is, the total length of transmitted packets is unbounded with t going to infinity; $\lim_{t\to\infty} C_t(X, \mathsf{A}, \mathsf{E}) = \infty$, for any algorithm X.

For arrival pattern A, adversarial error pattern E and time t, we define the *throughput* $T_t(\mathit{ALG}, \mathsf{A}, \mathsf{E})$ *of a deterministic algorithm ALG by time* t as:

$$T_t(\text{ALG}, \mathsf{A}, \mathsf{E}) = \frac{C_t(\text{ALG}, \mathsf{A}, \mathsf{E})}{C_t(C, \mathsf{A}, \mathsf{E})}.$$

For completeness, $T_t(\text{ALG}, \mathsf{A}, \mathsf{E})$ equals 1 if $C_t(\text{ALG}, \mathsf{A}, \mathsf{E}) = C_t(C, \mathsf{A}, \mathsf{E}) = 0$. Then, the *asymptotic throughput* of algorithm ALG in the adversarial arrival model is defined as:

$$\mathcal{T}(\text{ALG}) = \lim_{t\to\infty} \inf_{\mathsf{A}\in\mathcal{A},\mathsf{E}\in\mathcal{E}} T_t(\text{ALG}, \mathsf{A}, \mathsf{E}) \ ,$$

while in the stochastic arrival model it needs to take into account the random distribution of arrival patterns in $\mathcal{A}$, and is defined as follows:

$$\mathcal{T}(\text{ALG}) = \lim_{t\to\infty} \inf_{\mathsf{E}\in\mathcal{E}} \mathbb{E}_{\mathsf{A}\in\mathcal{A}}[T_t(\text{ALG}, \mathsf{A}, \mathsf{E})] \ .$$

Note that the asymptotic throughput is different from the classical competitiveness ratio. In the classical competitive analysis, an algorithm ALG would be x-competitive if $C_t(\text{ALG}, \mathsf{A}, \mathsf{E}) \leq$

$x \cdot C_t(X, \mathsf{A}, \mathsf{E}) + \Delta$, for any t, X and patterns A and E, from which Δ is independent. The difference with the efficiency measure described above, basically lies in the additive term Δ of the competitiveness formula which in this case may depend on time, and the fact that the final ratio is taken as the limit of the instantaneous ratio as time goes to infinity. (A detailed explanation is already included in the Task Scheduling Part, in Chapter 3.)

To prove lower bounds on the asymptotic throughput, one must compare the performance of a given algorithm with that of any algorithm X that solves the problem. When deriving upper bounds, one compares the performance of some carefully chosen offline algorithm OFF, that co-operates with the adversary, with the performance of *any* online algorithm ALG. As demonstrated later, this approach leads to accurate upper bound results.

Finally, *work conserving* online scheduling algorithms are considered. This means, that as long as there are pending packets, the sender does not cease to schedule them. Note that it does not make any difference whether one assumes that offline algorithms are work-conserving or not, since their throughput is the same in both cases (a work conserving offline algorithm always transmits, but stops the ongoing transmission as soon as an error occurs and then continues with the next packet). Hence for simplicity, do not assume offline algorithms to be work conserving.

7.2. Asymptotic throughput competitiveness

Considering first the deferred feedback mechanism it can be shown that in both cases of arrivals, adversarial and stochastic, the upper bound on the asymptotic throughput is 0.

Theorem 7.1. *No packet scheduling algorithm ALG can achieve an asymptotic throughput larger than 0 under adversarial arrivals in the deferred feedback model, even with one packet length.*

Proof: Consider the case at which packets arrive frequently enough so that there are always some packets ready at the sender station, when it is about to make a decision. The algorithm will then greedily send a train of packets, while the adversary injects bit errors at a distance of exactly π so that each error hits a different packet. Hence, the ALG cannot successfully complete any transmission (that is, it cannot transmit non-corrupted packets). At the same time, an offline algorithm OFF is able to send packets in each interval of length π without errors, which results to an asymptotic throughput equal to 0 as claimed. ■

Theorem 7.2. *No packet scheduling algorithm ALG can achieve an asymptotic throughput larger than* 0 *under stochastic arrivals in the deferred feedback model, even with one packet length.*

Proof: As described in the model specifications, packets arrive at a rate λ. Here we assume that all packets have the same length π. Observe, that if $\lambda\pi < 1$ there are many times when there is no packet ready to be sent and the link will be idle. In any case, the adversary can inject errors following the next rule: inject an error in the middle point of each packet sent by ALG. Applying this rule, no packet sent by ALG is received without errors. However, between two errors there is

at least π space (even if packets are contiguous) and the offline algorithm OFF can send a packet. The conclusion is that OFF is able to successfully send at least one packet between two attempts of ALG, while ALG cannot complete successfully any transmission, which completes the proof. ■

The rest of the section is therefore focused on analyzing the *immediate* feedback mechanism. What is more, one can see directly the connection between the adversarial arrival case and the task scheduling on a single machine when considering the completed-load competitiveness. The upper and lower bound results presented in 5.1.3 and 5.2, for a single machine and no speedup, also hold in this setting.

The focus is therefore turned on analyzing the *stochastic packet arrivals* and as one might expect, better asymptotic throughput can be obtained in some cases. A graphical representation of the upper bound results is also given, so that the reader can have a direct comparison and appreciate the improvement. For that, see Figure 7.1. For the lower bound, an algorithm called C-ρ-SPT-LPT is proposed and analyzed, showing that it achieves optimal asymptotic throughput. C-ρ-SPT-LPT schedules packets according to algorithm ρ-SPT-LPT presented earlier for the task scheduling problem, giving preference to short packets depending on the parameters of the stochastic distribution of packet arrivals[1]. The performance of algorithm C-ρ-SPT-LPT is optimal for a wide range of parameters of stochastic distributions of packets arrivals, and this is shown by proving the matching upper bound of the asymptotic throughput for any algorithm in this setting.

7.2.1. Upper bounds

In order to find the upper bound of the asymptotic throughput, consider an arbitrary work conserving algorithm ALG. Recall that $\lambda p > 0$ and $\lambda q > 0$, which implies that there are in fact injections of packets of both lengths π_{min} and π_{max} (see the definitions of λ, p and q from Section 7.1). The following adversarial error model $\mathcal{E}$ is defined.

1. When ALG starts a phase by transmitting a π_{max} packet then,

   a) If OFF has π_{min} packets pending, then the adversary extends the phase so that OFF can transmit successfully as many π_{min} packets as possible, up to $\hat{\rho}$. Then, it ends the phase so that ALG does not complete the transmission of the π_{max} packet (since $\hat{\rho}\pi_{min} < \pi_{max}$).

   b) If OFF does not have any π_{min} packets pending, then the adversary inserts a link error immediately (say after infinitesimally small time ϵ).

2. When ALG starts a phase by transmitting a π_{min} packet then,

   a) IF OFF has a packet of length π_{max} pending, then the adversary extends the phase so OFF can transmit a π_{max} packet. By the time this packet is successfully transmitted, the

[1] If the distribution is not known, then one needs to use the algorithm developed for the case of adversarial arrivals that needs no knowledge a priori.

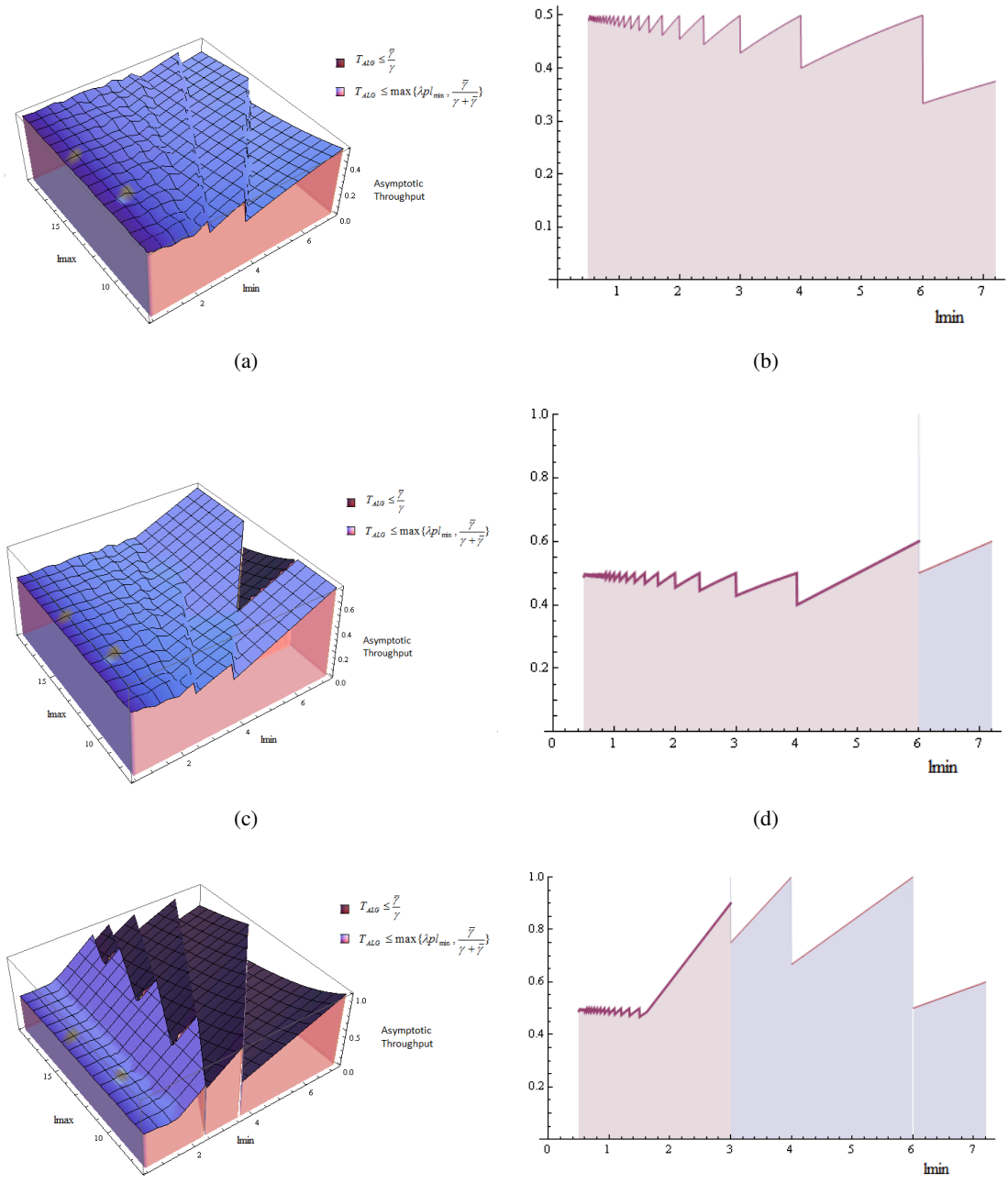

Figure 7.1: Upper bounds on the asymptotic throughput under stochastic packet arrivals with π_{min}-packet arrival probabilities as follows: (a) and (b) $p = 0.01$, (c) and (d) $p = 0.1$ and (e) and (f) $p = 0.3$. On the left column we give 3D representations for a range of π_{min} and π_{max} values, while on the right we give 2D representations of the same graph, under arbitrarily fixed π_{max}.

adversary inserts an error and finishes the phase. Observe that in this case ALG was able to successfully transmit up to $\bar{\rho}$ packets π_{min}.

b) If OFF has no π_{max} packets pending, then the adversary inserts an error immediately and ends the phase.

Observe that in phases of type 1*b* and 2*b*, neither OFF nor ALG are able to transmit any packet. These phases are just used by the adversary to wait for the conditions required by phases of type 1*a* and 2*a* to hold. In these latter types some packets are successfully transmitted (at least by OFF). They are called *productive* phases. Analyzing a possible execution, in addition to the concept of phase that have already be used, let *rounds* be defined. There is a round associated with each productive phase. The round ends when its corresponding productive phase ends, and starts at the end of the prior round (or at the start of the execution if no prior round exists). Depending on the type of productive phase they contain, rounds can be classified as type 1*a* or 2*a*.

Let some (large) time instant t be fixed. Denote by $r_{1a}^{(j)}$ the number of rounds of type 1*a* in which $j \leq \hat{\rho}$ packets of length π_{min} are sent by OFF completed by time t. The value $r_{2a}^{(j)}$ with $j \leq \bar{\rho}$ packets of length π_{min} sent by ALG, is defined similarly for rounds of type 2*a*. (Here rounding effects do not have any significant impact, since they will be compensated by the assumption that t is large.) Assume that t is a time when a round finishes. Let us denote by r the total number or rounds completed by time t, i.e., $\sum_{j=1}^{\bar{\rho}} r_{2a}^{(j)} + \sum_{j=1}^{\hat{\rho}} r_{1a}^{(j)} = r$.

The throughput by time t can be computed as

$$T_t(\mathsf{ALG}, \mathsf{A}, \mathsf{E}) = \frac{\pi_{min} \sum_{j=1}^{\bar{\rho}} j \cdot r_{2a}^{(j)}}{\pi_{max} \sum_{j=1}^{\bar{\rho}} r_{2a}^{(j)} + \pi_{min} \sum_{j=1}^{\hat{\rho}} j \cdot r_{1a}^{(j)}} \,. \tag{7.1}$$

From this expression, the following result can be shown.

Theorem 7.3. *No algorithm ALG has asymptotic throughput larger than* $\frac{\bar{\rho}}{\rho}$*, i.e.,* $\mathcal{T}(ALG) \leq \frac{\bar{\rho}}{\rho}$.

Proof: It can be observed in Eq. 7.1 that, for a fixed r, the lower the value of $r_{1a}^{(j)}$ the higher the asymptotic throughput. Regarding the values $r_{2a}^{(j)}$, the throughput increases when there are more rounds in the larger values of j. E.g., under the same conditions, a configuration with $r_{2a}^{(j)} = k_1$ and $r_{2a}^{(j+1)} = k_2$, has lower throughput than one with $r_{2a}^{(j)} = k_1 - 1$ and $r_{2a}^{(j+1)} = k_2 + 1$. Then, the throughput is maximized when $r_{2a}^{(\bar{\rho})} = r$ and the rest of values $r_{1a}^{(j)}$ and $r_{2a}^{(j)}$ are 0, which yields the bound. ■

To provide tighter bounds for some special cases, the following lemma is proven.

Lemma 7.1. *Consider any two constants* η, η' *such that* $0 < \eta < \lambda < \eta'$*. Then:*

(a) there is a constant $c > 0$*, dependent only on* λ, p, η*, such that for any time* $t \geq \pi_{min}$*, the number of packets of length* π_{min} *(resp.,* π_{max}*) injected by time* t *is* at least $t\eta p$ *(resp.,* $t\eta q$*) with probability at least* $1 - e^{-ct}$*;*

(b) there is a constant $c' > 0$, dependent only on λ, p, η', such that for any time $t \geq \pi_{min}$, the number of packets of length π_{min} (resp., π_{max}) injected by time t is at most *$t\eta' p$ (resp., $t\eta' q$) with probability at least $1 - e^{-c't}$.*

Proof: Statement 1(a) is proven first. The Poisson process governing arrival times of packets of length π_{min} has parameter λp. By the definition of a Poisson process, the distribution of packets of length π_{min} arriving to the system in the period $[0, t]$ is the Poisson distribution with parameter λpt. Consequently, by Chernoff bound for Poisson random variables (with parameter λpt), cf., [70], the probability that at least ηpt packets arrive to the system in the period $[0, t]$ is at least

$$1 - e^{-\lambda pt} \frac{(e\lambda pt)^{\eta pt}}{(\eta pt)^{\eta pt}} = 1 - e^{-tp(\lambda - \eta \ln(e\lambda/\eta))} \geq 1 - e^{-ct} ,$$

for some constant $c > 0$ dependent on λ, η, p. In the above, the argument behind the last inequality is as follows. It is a well-known fact that $x > 1 + \ln x$ holds for any $x > 1$; in particular, for $x = \lambda/\eta > 1$. This implies that $x - \ln(ex)$ is a positive constant for $x = \lambda/\eta > 1$, and after multiplying it by $\eta > 0$ another positive constant is obtained, equal to $\lambda - \eta \ln(e\lambda/\eta)$ that depends only on λ and η. Finally, multiplying this constant by $p > 0$ the final constant $c > 0$ is obtained, dependent only on λ, η, p.

The same result for packets of length π_{max} can be proved by replacing p by $q = 1 - p$ in the above analysis.

Statement 1(b) is proved analogously to the first one, by replacing η by η'. This is possible because the Chernoff bound for Poisson process has the same form regardless whether the upper or the lower bound on the Poisson value is considered, cf., [70]. ■

The following result can now be shown.

Theorem 7.4. *Let $p < q$. Then, the asymptotic throughput of any algorithm ALG is at most* $\min \left\{ \max \left\{ \lambda p \pi_{min}, \frac{\bar{\rho}}{\rho + \bar{\rho}} \right\}, \frac{\bar{\rho}}{\rho} \right\}$.

Proof: The claim has two cases. In the first case, $\lambda p \pi_{min} \geq \frac{\bar{\rho}}{\rho}$. In this case, the upper bound of $\frac{\bar{\rho}}{\rho}$ is provided by Theorem 7.3. In the second case $\lambda p \pi_{min} < \frac{\bar{\rho}}{\rho}$. For this case, define two constants η, η' such that $0 < \eta < \lambda < \eta'$ and $\eta' p < \eta q$. Observe that such constants always exist. Then, the asymptotic throughput of any algorithm ALG in this case is proved to be at most $\max \left\{ \eta' p \pi_{min}, \frac{\bar{\rho}}{\rho + \bar{\rho}} \right\}$.

Let some notation be introduced. Let $a_t^{\min}$ and $a_t^{\max}$ denote the number of π_{min} and π_{max} packets, respectively, injected up to time t. Let r_t^{off} and s_t^{off} be the number of π_{max} and π_{min} packets respectively, successfully transmitted by OFF by time t. Similarly, let s_t^{alg} be the number of π_{min} packets transmitted by algorithm ALG by time t. Observe that $s_t^{alg} \geq r_t^{off} \geq \lfloor \frac{s_t^{alg}}{\bar{\rho}} \rfloor$.

Consider a given execution and the time instants at which the queue of OFF is empty of π_{min} packets in the execution. There are two cases to be considered.

Case 1: For each time t, there is a time $t' > t$ at which OFF has the queue empty of π_{min} packets. Fix a value $\delta > 0$ and let time instants $t_0, t_1, \ldots$ be defined as follows: t_0 is the first time instant not smaller than π_{min} at which OFF has no π_{min} packet and such that $a_{t_0}^{\min} > \pi_{max}$. Then, for $i > 0$, t_i is the first time instant no smaller than $t_{i-1} + \delta$ at which OFF has no π_{min} packets. The asymptotic throughput at time t_i can be bounded as

$$\begin{aligned} T_{\text{ALG}}(A, E, t_i) &\leq \frac{s_{t_i}^{alg}\pi_{min}}{r_{t_i}^{off}\pi_{max} + a_{t_i}^{\min}\pi_{min}} \leq \frac{s_{t_i}^{alg}\pi_{min}}{\lfloor \frac{s_{t_i}^{alg}}{\bar{\rho}} \rfloor \pi_{max} + a_{t_i}^{\min}\pi_{min}} \\ &\leq \frac{s_{t_i}^{alg}\pi_{min}}{(\frac{s_{t_i}^{alg}}{\bar{\rho}} - 1)\pi_{max} + a_{t_i}^{\min}\pi_{min}} . \end{aligned}$$

This bound grows with $s_{t_i}^{alg}$ when $a_{t_i}^{\min} > \pi_{max}$, which leads to a bound on the asymptotic throughput as follows.

$$T_{\text{ALG}}(A, E, t_i) \leq \frac{a_{t_i}^{\min}\pi_{min}}{a_{t_i}^{\min}(\frac{\pi_{max}}{\bar{\rho}} + \pi_{min}) - \pi_{max}} = \frac{a_{t_i}^{\min}\bar{\rho}}{a_{t_i}^{\min}(\rho + \bar{\rho}) - \rho\bar{\rho}}.$$

Which as i goes to infinity yields a bound of $\frac{\bar{\rho}}{\rho+\bar{\rho}}$.

Case 2: There is a time t_* after which OFF never has the queue empty of π_{min} packets. Recall that for any $t \geq \pi_{min}$, from Lemma 7.1, the number of π_{min} packets injected by time t satisfy $a_t^{\min} > \eta' pt$ with probability at most $e^{-c't}$ and the injected $\max$ packets satisfy $a_t^{\max} < \eta qt$ with probability at most e^{-ct}. By the assumption of the theorem and the definition of η and η', $\eta' p < \eta q$. Let $t^* = 1/(\eta q - \eta' p)$. Then, for all $t \geq t^*$ it holds that $a_t^{\max} \geq a_t^{\min} + 1$, with probability at least $1 - e^{-c't} - e^{-ct}$. If this holds, it implies that OFF will always have π_{max} packets in the queue.

Fix a value $\delta > 0$ and let $t_0 = \max(t_*, t^*)$ along with the sequence of instants $t_i = t_0 + i\delta$, for $i = 0, 1, 2, \ldots$. By the definition of t_0, at all times $t > t_0$ OFF is successfully transmitting packets. Using Lemma 7.1, it can also be claimed that in the interval $(t_0, t_i]$ the probability that more than $\eta' pi\delta$ packets π_{min} are injected is no more than $e^{-c''i\delta}$.

With the above, the asymptotic throughput at any time t_i for $i \geq 0$ can be bounded as

$$T_{\text{ALG}}(A, E, t_i) \leq \frac{(a_{t_0}^{\min} + \eta' p \cdot i\delta)\pi_{min}}{r_{t_0}^{off}\pi_{max} + s_{t_0}^{off}\pi_{min} + i\delta} ,$$

with probability at least $1 - e^{-ct_i} - e^{-c't_i} - e^{-c''t_i}$. Observe that as i goes to infinity the above bound converges to $\eta' p\pi_{min}$, while the probability converges exponentially fast to 1. ■

In the Task Scheduling Part, and in particular in Section 5.2 a natural algorithm called SPT was shown to have a long-term completed-load competitive ratio at most $\frac{1}{\rho+1}$. Algorithm SPT could be considered for maximizing the completed-load in the task scheduling problem, and thus

the throughput it this case, assuming adversarial arrival patterns. It is now shown that algorithm SPT cannot have asymptotic throughput larger than $\frac{1}{\rho+1}$ under stochastic arrivals either, with specific arrival rates. This motivates the need for devising a new online algorithm for packet scheduling in these cases.

Theorem 7.5. *$\forall \varepsilon > 0$, $\exists \lambda, p, q$ such that algorithm SPT cannot achieve an asymptotic throughput larger than $\frac{1}{(1-\varepsilon)\rho+1} + \varepsilon$.*

Proof: Consider an execution of the SPT algorithm. Let intervals $I_1, I_2, \ldots, I_i$ be defined as follows. The first such interval, I_1, starts with the arrival of the first π_{min} packet. Then, I_i starts as soon as a π_{min} packet is in the queue of SPT after the end of interval I_{i-1}. The length of each interval depends on whether OFF has a π_{max} packet in its queue at the start of the interval or not. If it has a π_{max} packet, the length of the interval is $|I_i| = \pi_{min} + \pi_{max}$, and it is considered a *long* interval. If it does not, the length is $|I_i| = \pi_{min}$ and the interval is called *short.*

Between intervals the adversary injects frequent errors, so SPT cannot transmit any packet. In every interval I_i, SPT starts by scheduling a π_{min} packet. In a short interval, OFF sends a π_{min} packet, followed by an error injected by the adversary. Hence, in a short interval both SPT and OFF successfully transmit one π_{min} packet. In a long interval, OFF sends a π_{max} packet, after which the adversary injects an error. (Up to that point SPT has been able to complete the transmission of one or more π_{min} packets, but no π_{max} packet.) After the error, OFF sends a π_{min} packet (which is available since beginning of the interval) after which continuous errors will be injected by the adversary until the next interval. Hence, in a long interval OFF successfully transmits one π_{min} packet and one π_{max} packet, while SPT transmits only π_{min} packets. This implies that in both types of intervals OFF is transmitting useful packets during the whole interval.

Let s_k be the total length of the intervals $I_1, I_2, \ldots, I_k$, i.e., $s_k = \sum_{i=1}^{k} |I_i|$. Observe that the total number of π_{min} packets that arrive up to the end of interval I_k is bounded by k (that accounts for the π_{min} packet in the queue of SPT at the start of each interval) plus the packets that arrive in the intervals. From Lemma 7.1, there is a constant $\eta' > \lambda$ and a constant $c' > 0$ which depends only on η', λ and p, such that the number of π_{min} packets that arrive in the intervals is at most $\eta' p s_k$ with probability at least $1 - e^{-c' s_k}$.

Let T_k be the throughput of SPT at the end of interval I_k. From the above, T_k is bounded as

$$T_k \leq \frac{\pi_{min}(k + \eta' p s_k)}{s_k} = \frac{\pi_{min} k}{s_k} + \pi_{min} \eta' p \,,$$

with probability at least $\pi_1(k) = 1 - e^{-c' s_k}$. Observe that in the above expression it is assumed that all π_{min} packets that arrive by the end of I_k are successfully transmitted by SPT.

Claim: Let us consider the first $x + 1$ intervals I_i, for $x > 1$. The number of long intervals is at least $(1-\delta)(1 - e^{-\lambda q \pi_{min}})x$ with probability at least $1 - e^{-\delta^2 (1 - e^{-\lambda q \pi_{min}})x/2}$, for any $\delta \in (0, 1)$. *Proof of claim:* Observe that if a π_{max} packet arrives during interval I_i then the next interval I_{i+1}

is long. Consider now the first x intervals. Since each of these intervals has length at least π_{min}, some π_{max} packet arrives in the interval with probability at least $1 - e^{-\lambda q \pi_{min}}$ (independently of what happens in other intervals). Hence, using a Chernoff bound, the probability of having less than $(1-\delta)(1-e^{-\lambda q \pi_{min}})x$ intervals among the x first intervals in which π_{max} packets arrive is at most $e^{-\delta^2(1-e^{-\lambda q \pi_{min}})x/2}$. This completes the proof of the claim. □

From the claim, it follows that there are at least $(1-\delta)(1-e^{-\lambda q \pi_{min}})(k-1)$ long intervals among the first k intervals, with high probability. Hence, the value of s_k is bounded as

$$\begin{aligned} s_k &\geq (1-\delta)(1-e^{-\lambda q \pi_{min}})(k-1)(\pi_{max}+\pi_{min}) \\ &\quad +(k-(1-\delta)(1-e^{-\lambda q \pi_{min}})(k-1))\pi_{min} \\ &= (1-\delta)(1-e^{-\lambda q \pi_{min}})(k-1)\pi_{max} + k\pi_{min} \end{aligned}$$

with probability at least $\pi_2(k) = 1 - e^{-\delta^2(1-e^{-\lambda q \pi_{min}})(k-1)/2}$. Note that T_K cannot be larger than 1. Hence, the expected value of T_k can be bounded as follows.

$$\begin{aligned} \mathbb{E}[T_k] &\leq (1-\pi_1(k)\pi_2(k)) + \pi_1(k)\pi_2(k) \cdot \\ &\quad \cdot \left(\frac{\pi_{min} k}{(1-\delta)(1-e^{-\lambda q \pi_{min}})(k-1)\pi_{max} + k\pi_{min}} + \pi_{min}\eta' p \right) . \end{aligned}$$

Since $\pi_1(k)$ and $\pi_2(k)$ tend to one as k tends to infinity, we have that

$$\begin{aligned} \lim_{k\to\infty} \mathbb{E}[T_k] &\leq \frac{\pi_{min}}{(1-\delta)(1-e^{-\lambda q \pi_{min}})\pi_{max} + \pi_{min}} + \pi_{min}\eta' p \\ &= \frac{1}{(1-\delta)(1-e^{-\lambda q \pi_{min}})\gamma + 1} + \pi_{min}\eta' p \, . \end{aligned}$$

Hence, choosing η', p, q, and δ appropriately, the claim of the theorem follows. (E.g., they must satisfy $\pi_{min}\eta' p \leq \varepsilon$ and $(1-\delta)(1-e^{-\lambda q \pi_{min}}) \geq (1-\varepsilon)$.) ■

7.2.2. Lower bound and algorithm C-ρ-SPT-LPT

In this section algorithm C-ρ-SPT-LPT is proposed (stands for Conditional ρ-SPT-LPT). It builds on algorithm ρ-SPT-LPT presented in 5.2.1, in order to solve the packet scheduling problem in the setting of stochastic packet arrivals. The algorithm, depending on the arrival distribution, either follows the SPT policy (giving priority to π_{min} packets) or algorithm ρ-SPT-LPT. More precisely, algorithm C-ρ-SPT-LPT acts as follows:

If $\lambda p \pi_{min} > \frac{\bar{\rho}}{2\rho}$ then algorithm SPT is run,
otherwise algorithm ρ-SPT-LPT is executed.

Then, the following theorem can be shown:

Theorem 7.6. *The asymptotic throughput of algorithm C-ρ-SPT-LPT is not smaller than $\frac{\bar{\rho}}{\rho+\bar{\rho}}$ for $\lambda p \pi_{min} \leq \frac{\bar{\rho}}{2\rho}$, and not smaller than $\min\left\{\lambda p \pi_{min}, \frac{\bar{\rho}}{\rho}\right\}$ otherwise.*

Proof: There are three complementary cases to be considered.

Case $\lambda p \pi_{min} \leq \frac{\bar{\rho}}{2\rho}$.

In this case algorithm C-ρ-SPT-LPT runs algorithm ρ-SPT-LPT, achieving, per Theorem 5.8, asymptotic throughput of at least $\frac{\bar{\rho}}{\rho+\bar{\rho}}$ under *any* error pattern.

Case $\frac{\bar{\rho}}{2\rho} < \lambda p \pi_{min} \leq 1$.

The goal is to prove that the asymptotic throughput is not smaller than $\min\left\{\eta p \pi_{min}, \frac{\bar{\rho}}{\rho}\right\}$, for any $\eta = \delta\lambda$, with $\delta < 1$. Considering such an η, Lemma 7.1 can be used with respect to λ, η, p. The asymptotic throughput compares the behavior of algorithm C-ρ-SPT-LPT, which is simply SPT in this case, with any algorithm X for each execution. Hence, for the purpose of the analysis, the following modification is introduced in every execution: all periods in which X is not transmitting any packet are removed. By "removing", it is understood that time counts after removing the X-unproductive periods and "gluing" the remaining periods so that they form one time line. Observe that any time instant t in the modified time line, say $t = t_m$, cannot be larger than the corresponding time t in the global time line, say t_g (i.e., $t_m \leq t_g$). In the remainder of the analysis of this case consider these modified executions with modified time lines and whenever there is a need to refer to the "original" time line, the notion of *global time* is used. For any positive integer i, let time points $t_i = i \cdot \pi_{max}$. Consider events S_i, for positive integers i, defined as follows: the number of packets arrived by time t_i (on the modified time line of the considered execution) is at least $t_i \eta p$. By Lemma 7.1 and the fact that $t_m \leq t_g$, there is a constant c dependent only on λ, η, p such that for any i: the event S_i holds with probability at least $1 - e^{-ct_i}$.

Consider an integer $j > 1$ being a square of another integer. It is proved, that by time t_j the asymptotic throughput is at least

$$\min\left\{\eta p \pi_{min} - \frac{\bar{\rho}\pi_{min}}{t_j}, (1 - 1/\sqrt{j}) \cdot \frac{\bar{\rho}}{\rho}\right\},$$

with probability at least $1 - c' e^{-ct\sqrt{j}}$, for some constant $c' > 1$ dependent only on λ, η, p. To show this, consider two complementary scenarios that may happen at time t_j: there are at least $\bar{\rho}$ pending packets of length π_{min}, or otherwise. It is sufficient to show the sought property separately in each of these two scenarios.

Consider the first scenario, when there are at least $\bar{\rho}$ pending packets of length π_{min} at time t_j. With probability at least $1 - c' e^{-ct\sqrt{j}}$, for every $\sqrt{j} \leq i \leq j$ at least $t_i \eta p$ packets arrive by time t_i. This is because of the union bound of the corresponding events S_i and the fact that $\sum_{i \geq \sqrt{j}} e^{-ct_i} \leq c' \cdot e^{-ct\sqrt{j}}$ for some constant $c' > 1$ dependent on λ, η, p (note here that although c' seems to depend also on c, c' is still dependent only on λ, η, p because c is a function of these three parameters as well). Consider executions in $\bigcup_{i=\sqrt{j}}^{j} S_i$; executions at which all S_i events happen, for $\sqrt{j} \leq i \leq j$. Using induction on i, the following claim is proved:

Claim: At least $t_i \eta p - \bar{\rho}$ packets of length π_{min} have been successfully transmitted by time t_i, or at least $\bar{\rho}$ packets of length π_{min} are successfully transmitted in the interval $[t_i, t_{i+1}]$.

Proof of Claim: First, recall that algorithm C-ρ-SPT-LPT runs the SPT policy, since $\lambda p \pi_{min} > \frac{\bar{\rho}}{2\rho}$. Hence, as long as there are π_{min} packets pending, it will schedule them for transmission. Recall also, that times t_i represent time instants such that $t_i = i \cdot \pi_{max}$ in the modified time line.

Base Case: By time $t_{\sqrt{j}}$ and with probability at least $1 - e^{ct_{\sqrt{j}}}$, there will be at least $\eta p t_{\sqrt{j}}$ packets of length π_{min} arriving. Now since $t_{\sqrt{j}+1} = t_{\sqrt{j}} + \pi_{max}$, if there are at least $\bar{\rho}$ pending packets of length π_{min} at time $t_{\sqrt{j}}$, they will be successfully transmitted during the interval $[t_{\sqrt{j}}, t_{\sqrt{j}+1}]$, which guarantees the invariant. Otherwise, there are at least $\eta p t_{\sqrt{j}} - \bar{\rho}$ pending packets of length π_{min} at time $t_{\sqrt{j}}$ (as many as the ones that arrived minus the completed ones since the beginning of the execution, in a duration of $\sqrt{j} \cdot \pi_{max}$ time).

Induction Hypothesis: For $\sqrt{j} < k < j$, the invariant holds.

Induction Step: It must be shown that the invariant holds for $k+1$. Since only executions in the union $\bigcup_{i=\sqrt{j}}^{j} S_i$ are considered, by time $t_{k+1} = (k+1) \cdot \pi_{max}$ there are at least $\eta p t_{k+1}$ packets of length π_{min} arriving, with probability at least $1 - c' e^{-ct_{\sqrt{j}}}$. Now, since $t_{k+2} = t_{k+1} + \pi_{max}$, if there are at least $\bar{\rho}$ pending packets of length π_{min} at time t_{k+1}, they will be successfully transmitted during the interval $[t_{k+1}, t_{k+2}]$. Otherwise, there are at least $\eta p t_{k+1} - \bar{\rho}$ pending packets of length π_{min} at time t_{k+1} (as many as the ones that arrived minus the completed ones since the beginning of the execution, in a duration of $(k+1) \cdot \pi_{max}$ time). This guarantees the invariant and hence completes the proof of the claim. □

The inductive proof of this invariant follows directly from the specification of algorithm C-ρ-SPT-LPT (recall that it simply runs algorithm SPT in the currently considered case) and from the definition of the modified execution and time line. Let i^* denote the largest $i \in [\sqrt{j}, j]$ satisfying the following condition: there are less than $\bar{\rho}$ packets of length π_{min} pending in time t_i; if such an i does not exist, set $i^* = -1$. Consider two sub-cases:

Sub-case $i^* = -1$ (i^* does not exist) . Note that, by definition of i^*, at every time $t_i \in [\sqrt{j}, j]$, there are at least $\bar{\rho}$ pending packets of length π_{min} pending. Consequently, by the specification of the algorithm C-ρ-SPT-LPT, in each interval $[t_i, t_{i+1}]$, for $\sqrt{j} \leq i < j$, at least $\bar{\rho}$ packets of length π_{min} finish their transmission successfully. Therefore, by time t_j the total length of π_{min}-packets successfully transmitted by algorithm C-ρ-SPT-LPT is at least

$$\frac{t_j - t_{\sqrt{j}}}{\pi_{max}} \cdot \bar{\rho} \pi_{min} \ ,$$

while the total length of successfully transmitted packets by OPT is at most t_j (by the definition of the modified execution and time line). Hence, the asymptotic throughput is at least

$$\frac{\frac{t_j - t_{\sqrt{j}}}{\pi_{max}} \cdot \bar{\rho} \pi_{min}}{t_j} = (1 - 1/\sqrt{j}) \cdot \frac{\bar{\rho}}{\rho} \ ,$$

which converges to $\frac{\bar{\rho}}{\rho}$ with j going to infinity.

Sub-case $i^* \in [\sqrt{j}, j]$. It follows from the invariant and the definition of i^*, that by time t_{i^*} there are at least $t_{i^*} \eta p - \bar{\rho}$ successfully transmitted packets of length π_{min}, and in each interval

$[t_i, t_{i+1}]$, for $i^* \leq i < j$, at least $\bar{\rho}$ packets of length π_{min} finish their transmission successfully. Therefore, by time t_j the total length of π_{min}-packets successfully transmitted by algorithm C-ρ-SPT-LPT is at least

$$(t_{i^*}\eta p - \bar{\rho})\pi_{min} + \frac{t_j - t_{i^*}}{\pi_{max}} \cdot \bar{\rho}\pi_{min} \ ,$$

while the total length of successfully transmitted packets by X is at most t_j (by the definition of the modified execution and time line). Therefore the asymptotic throughput is at least

$$\frac{(t_{i^*}\eta p - \bar{\rho})\pi_{min} + \frac{t_j - t_{i^*}}{\pi_{max}} \cdot \bar{\rho}\pi_{min}}{t_j} \geq \min\left\{\frac{(t_j\eta p - \bar{\rho})\pi_{min}}{t_j}, \frac{\frac{t_j - t_{\sqrt{j}}}{\pi_{max}} \cdot \bar{\rho}\pi_{min}}{t_j}\right\} \quad (7.2)$$

$$= \min\left\{\eta p\pi_{min} - \frac{\bar{\rho}\pi_{min}}{t_j}, (1 - 1/\sqrt{j}) \cdot \frac{\bar{\rho}}{\rho}\right\}, \quad (7.3)$$

which converges to $\min\left\{\eta p\pi_{min}, \frac{\bar{\rho}}{\rho}\right\}$ with j going to infinity. For the proof of the inequality in the expression above, please refer to the Lemmas 7.2 and 7.3 that follow.

Finally, it is important to notice that the final converge of the ratio, with j going to infinity, in both sub-cases gives a valid bound on the asymptotic throughput, since the subsequent ratios hold with probabilities approaching 1 exponentially fast (in j), i.e., with probabilities at least $1 - c'e^{-ct\sqrt{j}}$, where c and c' are positive constants dependent only on λ, η, p. The minimum of the two asymptotic throughputs, coming from the sub-cases, is $\min\left\{\eta p\pi_{min}, \frac{\bar{\rho}}{\rho}\right\}$, as desired and therefore the asymptotic throughput is at least $\min\left\{\delta\lambda p\pi_{min}, \frac{\bar{\rho}}{\rho}\right\}$ in this case.

Case $\lambda p\pi_{min} > 1$**.** In this case we simply observe that one gets at least the same asymptotic throughput as in case $\lambda p\pi_{min} = 1$, because executions are saturated with packets of length π_{min} with probability converging to 1 exponentially fast. (Recall that the same algorithm SPT in the specification of C-ρ-SPT-LPT is used, both for $\lambda p\pi_{min} = 1$ and for $\lambda p\pi_{min} > 1$.) Consequently, the asymptotic throughput in this case is at least $\min\left\{\eta p\pi_{min}, \frac{\bar{\rho}}{\rho}\right\}$, for any $\lambda/2 < \eta < \lambda$, and therefore it is at least $\min\left\{\lambda p\pi_{min}, \frac{\bar{\rho}}{\rho}\right\} \geq \min\left\{1, \frac{\bar{\rho}}{\rho}\right\} = \frac{\bar{\rho}}{\rho}$. ■

Lemma 7.2. *When* $\eta p\pi_{min} \leq \frac{\bar{\rho}}{\rho}$ *it holds that* $\frac{(t_{i^*}\eta p - \bar{\rho})\pi_{min} + \frac{t_j - t_{i^*}}{\pi_{max}}\bar{\rho}\pi_{min}}{t_j} \geq \frac{(t_j\eta p - \bar{\rho})\pi_{min}}{t_j}$.

Proof: Assume first the case of $i^* < j$. This means that:

$$\eta p\pi_{min} \leq \frac{\bar{\rho}}{\rho} = \frac{(j - i^*)\bar{\rho}}{(j - i^*)\rho} = \frac{(j - i^*)\bar{\rho}\pi_{min}}{(j - i^*)\pi_{max}}$$

$$\Rightarrow \eta p\pi_{min}(i^* - j)\pi_{max} + (j - i^*)\bar{\rho}\pi_{min} \geq 0$$

$$\Rightarrow i^*\pi_{max}\eta p\pi_{min} + (j - i^*)\bar{\rho}\pi_{min} \geq j\pi_{max}\eta p\pi_{min}$$

$$\Rightarrow t_{i^*}\eta p\pi_{min} + \frac{t_j - t_{i^*}}{\pi_{max}}\bar{\rho}\pi_{min} \geq t_j\eta p\pi_{min}$$

$$\Rightarrow (t_{i^*}\eta p - \bar{\rho})\pi_{min} + \frac{t_j - t_{i^*}}{\pi_{max}}\bar{\rho}\pi_{min} \geq (t_j\eta p - \bar{\rho})\pi_{min} \ .$$

What is more, for the case when $i^* = j$:

$$(t_{i^*}\eta p - \bar{\rho})\pi_{min} = (t_j \eta p - \bar{\rho})\pi_{min}$$

$$\Rightarrow (t_{i^*}\eta p - \bar{\rho})\pi_{min} + \frac{t_j - t_{i^*}}{\pi_{max}}\bar{\rho}\pi_{min} \geq (t_j \eta p - \bar{\rho})\pi_{min} \, .$$

Both cases conclude to the same, which proves the lemma. ■

Lemma 7.3. *When* $\eta p \pi_{min} > \frac{\bar{\rho}}{\rho}$ *it holds that* $\frac{(t_{i^*}\eta p - \bar{\rho})\pi_{min} + \frac{t_j - t_{i^*}}{\pi_{max}}\bar{\rho}\pi_{min}}{t_j} \geq \frac{\frac{(t_j - t_{\sqrt{j}})}{\pi_{max}}\bar{\rho}\pi_{min}}{t_j}$.

Proof: When $\eta p \pi_{min} > \frac{\bar{\rho}}{\rho}$, the following is also true:

$$\eta p \pi_{min} \geq \frac{\bar{\rho}}{\rho} + \frac{(1 - \sqrt{j})\bar{\rho}}{i^*\rho} \, .$$

This means that:

$$\eta p \pi_{min} \geq \frac{(1 + i^* - \sqrt{j})\bar{\rho}\pi_{min}}{i^*\pi_{max}}$$

$$\Rightarrow \eta p \pi_{min} i^* \pi_{max} + \bar{\rho}\pi_{min}(j - i^* - 1 - j + \sqrt{j}) \geq 0$$

$$\Rightarrow i^*\pi_{max}\eta p \pi_{min} - \bar{\rho}\pi_{min} + (j - i^*)\bar{\rho}\pi_{min} \geq (j - \sqrt{j})\bar{\rho}\pi_{min}$$

$$\Rightarrow (t_{i^*}\eta p - \bar{\rho})\pi_{min} + \frac{t_j - t_{i^*}}{\pi_{max}}\bar{\rho}\pi_{min} \geq \frac{(t_j - t_{\sqrt{j}})}{\pi_{max}}\bar{\rho}\pi_{min} \, .$$

■

Observe, that if the upper bound on asymptotic throughput is compared with the lower bound of the above theorem, then one may conclude that in the case where ρ is an integer, algorithm C-ρ-SPT-LPT is optimal, with respect to asymptotic throughput. In the case where ρ is not an integer, there is a small gap between the upper and lower bound results.

Chapter 8

Constrained Jamming

This chapter, studies the second and last problem of online packet scheduling over an unreliable communication link considered in the thesis. More precisely, it studies the impact of *constrained jamming* on a wireless channel between two network nodes; a similar setting as the one considered in the previous chapter, but with some restrictions on the adversarial jamming entity. The problem is described in detail in the first section, followed by the model specifications and the main results, clarifying their importance and difference with the previous chapter.

8.1. Problem definition and model specifications

For this problem, the wireless communication setting of Chapter 7 is kept; a single channel between a sender and a receiver is considered, being watched and disrupted by a malicious, adversarial jammer. The goal of the sender is to fully transmit as much data possible despite the jams, in the most efficient way. It is assumed that the sender has an infinite amount of data to be sent and needs to decide on the packet sizes. Note that if a packet is jammed it needs to be re-transmitted; the data is sent as the payload of the packets, and becomes useless if the packet is jammed.

However, after seeing the impact of *unconstrained* adversarial jamming in Chapter 7, the power of the malicious entity is now bounded, in order to see what is the possible performance and whether it can be improved. For this, parameters ρ and σ are used (we define them in detail in the model section). They represent the rate at which the jammer can corrupt a packet and the largest size of bursts of jams that can be caused, respectively. This constraint corresponds to the translation of Adversarial Queuing Theory (AQT) constraints, typically defined for packet arrivals, to channel jamming. In practice, this adversarial model can represent an entity with limited sources of rechargeable energy, e.g., malicious mobile devices [1, 2] or battery-operated military drones [37, 56].

In fact, drones or *Unmanned Aerial Vehicles (UAV)* are at the peak of their development. As an upcoming technology that is rapidly improving, it has already attracted the colossi of industry,

like Google or Amazon, to invest in UAV research and development, creating even commercial models. There have already been a few research works but the area is still being studied; the work in [37] focuses on UAV's risk analysis and the work in [56] focuses in analyzing cellular network coverage using UAV's and software defined radio. Regarding mobile jammers, in the recent years, many companies have made available battery-operated 3G/4G, WiFi or GPS mobile jammers (e.g., [1, 2]); this market can only increase, as wireless communication is becoming the dominating communication technology.

This chapter, proposes deterministic algorithms that decide the lengths of the packets to be sent. Their aim is to minimize the transmission time, and thus maximize the *goodput ratio*; the amount of useful payload successfully transmitted over time. First, the same packet length (uniform) is assumed, characterizing the corresponding "optimal" length. Then, by adapting the packet length depending on the jams, it is shown that the transmission time can be improved. To do this, a *static* version of the problem must be studied (its details are given below in Subsection 8.1.2); that is, a model used as an important *building block* for the solution of the main model following the AQT jamming approach, called *dynamic*.

8.1.1. Dynamic model

The *dynamic* model is formalized first; the main model considered in this chapter.

Network setting

Consider a setting of a sending station (sender) that transmits packets to a receiving station (receiver) over an unreliable wireless channel. Assume that the sender has enough data to transmit, covering any interval length T, and follows some *online scheduling* [76,77] in order to decide the lengths of the packets to be sent in the transmission. The decisions need to be made during the course of the execution, taking into consideration (or not) the channel jams. Each packet π consists of a *header* of a fixed predefined size h and a *payload* of length l chosen by the algorithm. The payload represents the useful data to be sent across the channel and is to be chosen by the sender. The total length of the packet is then denoted by $p = h + l$. For simplicity and without loss of generality $h = 1$ is used throughout the analysis, and hence $p = l + 1$. (Note that l needs not be an integer.) Furthermore, constant bit rate is considered for the channel, which means that the transmission time of each packet is proportional to its length (i.e., a packet of size $l + 1$ takes $l + 1$ time units to be transmitted in full).

Packet failures

The unavailability of the channel is modeled as being controlled by the omniscient and adaptive adversary (σ, ρ)-$\mathcal{A}$, which is defined by its two "restrictive" parameters, $\rho \in [0, 1]$ and $\sigma \geq 1$ as follows. The adversary has a token bucket of size σ where it stores "error tokens" and is initially full. From the beginning of the execution and up to a time t, within interval $T = [0, t]$, there

will be $\lfloor \rho T \rfloor$ such error tokens created, where ρ is the rate at which they become available to the adversary. In other words, a new error token becomes available at times $1/\rho, 2/\rho, \ldots$. Note that the values of the parameters are given to the adversary (they are not chosen by it) and it can only use them in a "smart" way in order to control the packet jams in the channel. If there is at least one token in the bucket, the adversary can introduce an error in the channel and jam the packet being transmitted, consuming one token. If the token bucket is full (i.e., there are already σ error tokens in the bucket) and a new token arrives, then one token is lost. (This, models, for example, the fact that a fully charged battery cannot be further charged.) As a worst case analysis, consider that the adversary jams some bit in the header of the packets in order to ensure their destruction. Therefore, adversary (σ, ρ)-$\mathcal{A}$ defines the *error pattern* E as a collection of jamming events on the channel, jamming the packet that is being transmitted in that instant. Finally, it is assumed that parameters ρ and σ are known to the scheduling algorithm.

Efficiency measures

For the efficiency of a scheduling algorithm, the *goodput rate*, G is considered; the ratio of the total amount of payload successfully transmitted over time, despite the jams in the channel.

To be more precise, let the amount of payload successfully transmitted be defined as *useful payload*. The useful payload of an algorithm ALG in a time interval T, under error pattern E, is denoted by $\text{UP}_T(\text{ALG}, \mathsf{E})$, and it is calculated as the sum of payloads of the packets successfully transmitted in the interval. Since worst-case analysis is considered, the *worst* useful payload of a fixed algorithm A is actually calculated, as $\text{UP}_T(A) = \min_{\mathsf{E} \in \mathcal{E}(\rho,\sigma)} \text{UP}_T(A, \mathsf{E})$, where $\mathcal{E}(\rho, \sigma)$ is the set of all possible error patterns with parameters ρ and σ. We also define the optimal useful payload as $\text{UP}_T^* = \max_A \text{UP}_T(A)$. Now, when examining a period T in the execution of an algorithm ALG, under error pattern E, its goodput rate is defined as $G_T(\text{ALG}, \mathsf{E}) = \text{UP}_T(\text{ALG})/T$ and the optimal goodput as $G^* = \text{UP}_T^*/T$.

For simplicity, the shorter notations UP and G are used, when the algorithm used or the time interval considered, respectively, are implied. The notation T is also overloaded to refer both to the interval and to its length. Finally, note that in most of the analysis *floors* and *ceilings* are avoided, in order to keep the readability of the results as simple as possible for the reader. Nonetheless, this does not affect the correctness of the results, since when being applied on large enough time intervals and data, the "losses" become negligible.

Feedback mechanism

As a feedback mechanism, following the analysis of the previous Chapter 7, assume that the sender receives instantaneous feedback for a packet successfully received. Assume also, that the notification packets cannot be jammed by the errors in the channel because of their relatively small size. In particular, consider notification/acknowledgement messages sent for every packet that is received successfully. If such a message is not received by the sender, then it considers the

packet to be jammed.

8.1.2. Static model

Here, the differences of the static model to the dynamic one are highlighted.

Packet failures

Assume that the channel jams are orchestrated by an omniscient and adaptive adversary, (T, f)-$\mathcal{A}$. However, it has a constrained number of jams it can cause in a given period. Specifically, for a time interval of length T, $T \geq 1$, it can cause up to f packet jams. Thus, given a parameter T, the adversary defines the *error pattern* E as a set of up to f jamming events on the channel over that period, each given by a time instant in the period. As in the dynamic model, for a worst case analysis assume that the adversary jams some bit in the header of the packets in order to ensure their destruction. The special error pattern $\mathsf{E} = \emptyset$ that corresponds to the case in which the adversary causes no jamming, will sometimes be used. For a given T, it is assumed that f is known to the scheduling algorithm.

Efficiency measures

The same performance measure as in the dynamic model is considered; *goodput rate*, and the *useful payload* is used in order to calculate the exact amount of data successfully transmitted; this time for interval of length T and f error tokens.

More formally, similar to the dynamic model, let $\mathrm{UP}_{(T,f)}(\mathrm{ALG}, \mathsf{E})$ be the useful payload (payload successfully received) when using scheduling algorithm ALG in an interval of length T against an adversary of power f that uses error pattern E. Then, for a fixed algorithm A, its useful payload is $\mathrm{UP}_{(T,f)}(A, \mathsf{E}) = \min_{\mathsf{E} \in \mathcal{E}(f)} \mathrm{UP}_{(T,f)}(A, \mathsf{E})$, where $\mathcal{E}(f)$ is the set of all possible error patterns with at most f jams. From this, let the optimal useful payload be defined as $\mathrm{UP}^*_{(T,f)} = \max_A \mathrm{UP}_{(T,f)}(A)$. For simplicity, the shorter notation UP is used. This is done when the algorithm used and the number of possible errors in the interval are implied.

The goodput rate is defined similarly, by simply dividing the useful payload by the length of the interval. More precisely, when using scheduling algorithm ALG in an interval of length T against an adversary of power f that uses error pattern E, its goodput rate is $G_{(T,f)}(\mathrm{ALG}, \mathsf{E}) = \mathrm{UP}_{(T,f)}(\mathrm{ALG}, \mathsf{E})/T$, and the optimal goodput is $G^*_{(T,f)} = \mathrm{UP}^*_{(T,f)}/T$.

Feedback mechanism

As in the dynamic model, instantaneous feedback is assumed. Nonetheless, observe that if $T \leq f$, then the adversary can jam all packets sent in the interval and no useful data will be received. Hence, only time periods that are initially of length $T > f$ are of interest.

An absolute bound on the error rate with respect to the maximum packet length can be proved.

Observation 8.1. *Let π be the smallest packet size used by an algorithm (i.e., $\forall p, p_{.len} \geq \pi$). For any error rate $\rho \geq 1/\pi$, no goodput larger than zero can be achieved.*

Proof: If the error rate is $\rho \geq 1/\pi$, a new error token arrives during the transmission of any packet (recall that packets are of size at least π). Hence, there are error tokens in the bucket at all times for the adversary to corrupt all packets being transmitted. Using an error token every π time, is sufficient to keep the goodput at zero. ■

From this observation, it can be derived that algorithms that only use packets of length $p_{.len} \geq 1/\rho$ are not interesting.

8.1.3. Moving from the static to the dynamic model

The approach followed in the rest of the chapter, is to first analyze the static model, and then explore the way its solutions can be applied in the dynamic model. In particular, the executions of the continuous (dynamic) version of the problem can be divided into successive intervals of length $1/\rho$, and let σ error tokens be assumed available at the beginning of each interval. Then these intervals will become instances of the static model, where $T = 1/\rho$ and $f = \sigma$.

An algorithm ALG_D is therefore proposed. It is an algorithm that uses the optimal solution of the static model, say algorithm A, to solve the problem in the dynamic model, with parameters $1/\rho$ and σ.

Algorithm ALG_D Description:
For every time interval $T_i = \left[\frac{i}{\rho}, \frac{i+1}{\rho}\right)$, where $i = 0, 1, \ldots$, run $A(1/\rho, \sigma)$.

Observation 8.2. *Observe that, if the goodput of algorithm A is $G(A)$, then the goodput of ALG_D will also be $G(ALG_D) = G(A)$. This is because the goodput per-interval will be repeated throughout the whole execution.*

8.2. Uniform packets for the static model

This section studies the static model, for the case when the algorithm used is restricted in sending packets of equal/uniform length. This could be due to limitations in the communication protocol or the sender's specification. The aim is to define a quasi-optimal algorithm, named S-UNI, that schedules uniform packets taking into account the parameters of the adversary. For that, the quasi-optimal necessary packet length p^* is computed, which maximizes the minimum useful payload considering time interval T and maximum number of errors f.
Note that the approximations below are due to floors and ceilings; these approximations get closer to equality as Tf grows.

Theorem 8.1. *Let S-UNI use only uniform packets of length p. In an interval of length T and maximum number of errors f, the optimal packet length for these algorithms, p^*, gives a useful payload, $UP_{(T,f)}(\text{S-UNI}_{p^*})$, equal to*

$$\max\left\{\frac{1}{\lfloor\sqrt{Tf}\rfloor}\big(\lfloor\sqrt{Tf}\rfloor-f\big)\big(T-\lfloor\sqrt{Tf}\rfloor\big),\frac{1}{\lceil\sqrt{Tf}\rceil}\big(\lceil\sqrt{Tf}\rceil-f\big)\big(T-\lceil\sqrt{Tf}\rceil\big)\right\}$$

and thus a corresponding goodput rate, $G_{(T,f)}(\text{S-UNI}_{p^*})$, equal to

$$\max\left\{\frac{1}{T\lfloor\sqrt{Tf}\rfloor}\big(\lfloor\sqrt{Tf}\rfloor-f\big)\big(T-\lfloor\sqrt{Tf}\rfloor\big),\frac{1}{T\lceil\sqrt{Tf}\rceil}\big(\lceil\sqrt{Tf}\rceil-f\big)\big(T-\lceil\sqrt{Tf}\rceil\big)\right\}.$$

In fact, $UP(\text{S-UNI}_{p^*})\approx T+f-2\sqrt{Tf}$ and $G(\text{S-UNI}_{p^*})\approx\left(1-\sqrt{f/T}\right)^2$.

Proof: Let n denote the number of uniform packets of length $p=\frac{T}{n}$ sent in an interval of length T when the adversary has f error tokens available. In the worst case, the adversary will use its error tokens to jam f packets in the interval, and hence there will be at least $n-f$ successfully received packets by the receiver by the end of the interval.

Let S-UNI$_n$ and S-UNI$_p$ denote the same algorithm, that uses n uniform packets of length p. Recall that each packet consists of the payload and a unit-size header. Its useful payload will then be $\text{UP}_{(T,f)}(\text{S-UNI}_n)=(n-f)\left(\frac{T}{n}-1\right)$. Deriving this expression with respect to n, it gives

$$\frac{\partial\text{UP}_{(T,f)}(\text{S-UNI}_n)}{\partial n}=\frac{fT}{n^2}-1,$$

which implies that $\text{UP}_{(T,f)}(\text{S-UNI}_n)$ is maximized for $n=\sqrt{Tf}$. What is more, the derivative is positive for $n<\sqrt{Tf}$ and negative for $n>\sqrt{Tf}$. This means, that the useful payload is strictly increasing on the left of $n=\sqrt{Tf}$ and strictly decreasing on the right. From this, we get that (1) there is no other n that maximizes the useful payload, and (2) since the number of packets has to be an integer value, the only two candidates for the optimal number of packets n^* are $\lfloor\sqrt{Tf}\rfloor$ and $\lceil\sqrt{Tf}\rceil$. Hence the value of these two that maximizes the useful payload is the optimal number n^*.

Thus, the optimal useful payload is $\text{UP}_{(T,f)}(\text{S-UNI}_{n^*})=(n^*-f)\big(\frac{T}{n^*}-1\big)=\max\left\{\frac{1}{\lfloor\sqrt{Tf}\rfloor}\big(\lfloor\sqrt{Tf}\rfloor-f\big)\big(T-\lfloor\sqrt{Tf}\rfloor\big),\frac{1}{\lceil\sqrt{Tf}\rceil}\big(\lceil\sqrt{Tf}\rceil-f\big)\big(T-\lceil\sqrt{Tf}\rceil\big)\right\}$ and the corresponding goodput rate $G_{(T,f)}(\text{S-UNI}_{n^*})=\frac{\text{UP}_{(T,f)}(\text{S-UNI}_{n^*})}{T}=\max\left\{\frac{1}{T\lfloor\sqrt{Tf}\rfloor}\big(\lfloor\sqrt{Tf}\rfloor-f\big)\big(T-\lfloor\sqrt{Tf}\rfloor\big),\frac{1}{T\lceil\sqrt{Tf}\rceil}\big(\lceil\sqrt{Tf}\rceil-f\big)\big(T-\lceil\sqrt{Tf}\rceil\big)\right\}$, as claimed.

From the optimal number n^*, and the fact that $p^*=\frac{T}{n^*}$, it holds that $p^*\approx\sqrt{T/f}$. Then, the optimal achievable useful payload becomes $\text{UP}_{(T,f)}(\text{S-UNI}_{p^*})\approx T+f-2\sqrt{Tf}$ and the corresponding optimal goodput rate, $G_{(T,f)}(\text{S-UNI}_{p^*})\approx\left(1-\sqrt{f/T}\right)^2$, as also claimed. ■

8.3. Adaptive algorithms for static model with $f = 1$

Remaining in the static model, the attention is turned to some adaptive algorithms, in order to see whether the goodput found above can be improved. These, algorithms change the packet sizes according to the jams they have observed so far. Starting from the case of $f = 1$, the following algorithms proposed achieve a goodput rate greater than $G^*_{(T,1)}(\text{S-UNI}) \approx (1 - \sqrt{1/T})^2$.

8.3.1. Algorithm S-DEC

The first adaptive algorithm proposed is called S-DEC and it is shown here that for time intervals T large enough, for $T > \frac{2}{7-3\sqrt{5}}$ to be exact, it achieves goodput rate greater than $G_{(T,1)}(\text{S-UNI}) \approx (1 - \sqrt{1/T})^2$.

Algorithm S-DEC Description:
Each period starts by scheduling packets of decreasing length $p_i = Z - i$ for $i = 0, 1, 2, 3 \ldots$. If a packet π_j is jammed during the period, this transmission sequence is stopped, and after π_j, a single more packet is scheduled by the algorithm whose length spans the rest of the period.

Theorem 8.2. *Adaptive algorithm S-DEC, with* $Z = \frac{1}{2}\left(\sqrt{1+8T} - 1\right)$*, achieves goodput* $G(\text{S-DEC}) = 1 - \frac{1}{2T}\left(1 + \sqrt{1+8T}\right)$*. This value is larger than the upper bound for the uniform case, if* $T > \frac{2}{7-3\sqrt{5}} \approx 6.8541$.

Proof: There are two cases to be considered in a period:

(a) If the adversary jams a packet π_j, the useless data sent in the period adds to $Z + 1$. This number comes from the j headers of the packets sent before π_j, plus the length $p_j = Z - j$ of the packet jammed, plus the header of the last packet sent in the period (which cannot be jammed). Hence, in this case, the useful payload of the period is $T - (Z + 1)$.

Otherwise, (b) if no packet is jammed, the useless data sent in the period correspond only to the headers of the packets sent. Then, if the last packet sent in the interval is π_k, the useless data is $k + 1$, and the corresponding useful payload is $T - (k + 1)$. The value Z is chosen so that the total length of the packets sent in this case is equal the length of the interval. From this property, $\sum_{i=0}^{k} p_i = T$, the value of Z must satisfy $Z(k + 1) - \frac{k(k+1)}{2} = T$ and hence

$$Z = \frac{k}{2} + \frac{T}{k+1}. \tag{8.1}$$

In a given period the choice of whether case (a) or (b) occurs is up to the adversary, since it can decide which packet to jam, if any. This means that the useful payload achieved will be the minimum of the two cases, UP $= \min\{T-(Z+1), T-(k+1)\}$. Observe from this Eq. 8.1 that the length Z of the initial packet increases if the number of packets k decreases. Additionally, it must hold that $Z \geq k$ and therefore UP is maximized when when $Z = k$. Hence, the optimal k is the suitable solution of the equation $k = \frac{k}{2} + \frac{T}{k+1}$, which is $k = \frac{1}{2}\left(\sqrt{1+8T} - 1\right) = Z$.

The useful payload achieved is then UP(S-DEC) $= T - \left(\frac{1}{2}\sqrt{1+8T} - \frac{1}{2} + 1\right) = T - \frac{1}{2}\left(\sqrt{1+8T}+1\right)$, which is more than UP*(S-UNI) $= T \cdot G_{(T,1)}$(S-UNI) $= T\left(1-\sqrt{1/T}\right)$. The corresponding goodput is therefore G(S-DEC) $= \frac{\text{UP}}{T} = 1 - \frac{1}{2T}\left(\sqrt{1+8T}+1\right)$. ■

8.3.2. Algorithm S-OPT$(T,1)$: optimal for $f = 1$

Since the performance of algorithm S-DEC is only better than the uniform packet scheduling approach for a limited range of intervals, i.e., $T > \frac{2}{7-3\sqrt{5}}$, it is only natural to wonder whether the result given by S-DEC in the previous subsection can be improved, and see whether a goodput rate that surpasses G(S-UNI) exists for time intervals $T < \frac{2}{7-3\sqrt{5}}$. The following adaptive algorithm is developed, named S-OPT$(T,1)$, which is proved to be optimal for the static model, for $f = 1$. (See the algorithm's pseudo-code in Alg. 8.) By proving this, the reader will hopefully get an intuition to on *how* the optimal algorithm for any number of error tokens will work.

Algorithm 8: S-OPT$(T,1)$

If $T \in [1,2)$ **then**
  Send packet π with length $p = T$
else
  Let i be the integer such that $T \in \left[\frac{(i-1)i}{2}+1, \frac{i(i+1)}{2}+1\right)$
  Let $\alpha = i-2$, and $\beta = \frac{(i-1)i}{2} - 1$
  Send packet π with length $p = \frac{T+\beta}{\alpha+2} = \frac{T-1}{i} + \frac{i-1}{2}$
  If packet π is jammed **then**
    Send packet with length $p' = T - p$
  else
    Call S-OPT$(T-p,1)$

Algorithm S-OPT$(T,1)$ is used in a time recursive fashion, with respect to the length of the interval of interest, T. Its scheduling policy is as follows: It chooses the length p of the first packet to be transmitted as a function of T. If the packet is jammed then it transmits a second packet of length $T-p$ which is guaranteed not to be jammed. If the first packet goes through, then the algorithm is invoked recursively as S-OPT$(T-p,1)$.

A detailed pseudo-code for the algorithm is given as Algorithm 8. Fix the interval length $T \geq 1$, and let i be the integer such that $T \in \left[\frac{(i-1)i}{2}+1, \frac{i(i+1)}{2}+1\right)$, as described in the above pseudo-code. Let also parameters $\alpha = i-2$ and $\beta = \frac{(i-1)i}{2} - 1$, packet length $p = \frac{T+\beta}{\alpha+2}$, and interval length $T' = T - p$. The following two lemmas are used to show the optimality of Algorithm S-OPT$(T,1)$ in the static model.

Lemma 8.1. *Interval length $T' = T-p$ is such that $T' \in \left[\frac{(j-1)j}{2}+1, \frac{j(j+1)}{2}+1\right)$ for $j = i-1$, where i is an integer such that $i \geq 1$.*

Proof: Replacing the values of α and β in the calculation of $T' = T - p$,

$$T' = \frac{(\alpha+1)T - \beta}{\alpha+2} = \frac{(i-2+1)T - \left(\frac{(i-1)i}{2} - 1\right)}{i-2+2} = \frac{(i-1)T - \frac{(i-1)i}{2} + 1}{i}.$$

Now, using the fact that $T \geq \frac{(i-1)i}{2} + 1$,

$$T' \geq \frac{(i-1)\left(1 + \frac{(i-1)i}{2}\right) - \frac{(i-1)i}{2} + 1}{i} = \cdots = \frac{(i-1)(i-2)}{2} + 1.$$

Similarly, using the fact that $T < \frac{i(i+1)}{2} + 1$,

$$T' < \frac{(i-1)\left(1 + \frac{i(i+1)}{2}\right) - \frac{(i-1)i}{2} + 1}{i} = \cdots = \frac{(i-1)i}{2} + 1.$$

Setting $j = i - 1$ in both cases, $T' \in \left[\frac{(j-1)j}{2} + 1, \frac{j(j+1)}{2} + 1\right)$ as claimed. ■

Lemma 8.2. *Let $T \geq 2$ and assume that $UP_{(T',1)}(S\text{-}OPT) = \frac{\alpha T' - \beta}{\alpha+1}$, where $T' = T - p$. Then, Algorithm S-OPT$(T, 1)$ achieves useful payload $UP_{(T,1)}(S\text{-}OPT) = \frac{(\alpha+1)T - (\beta+\alpha+2)}{\alpha+2}$.*

Proof: Since $T \geq 2$, that Algorithm S-OPT$(T, 1)$ schedules first a packet π with length $p = \frac{T+\beta}{\alpha+2}$. If π is jammed, then a packet of length equal to the rest of the interval, i.e., $T' = T - p$, can be sent successfully, and hence the useful payload will be $\text{UP}_{(T,1)}(\text{S-OPT}) = T - \frac{T+\beta}{\alpha+2} - 1 = \frac{(\alpha+1)T - (\beta+\alpha+2)}{\alpha+2}$.

Otherwise, if π is not jammed, the useful payload is obtained as $\text{UP}_{(T,1)}(\text{S-OPT}) = p - 1 + \text{UP}_{(T',1)}(\text{S-OPT}) = p - 1 + \frac{\alpha T' - \beta}{\alpha+1} = p - 1 + \frac{\alpha(T-p) - \beta}{\alpha+1} = \frac{(\alpha+1)T - (\beta+\alpha+2)}{\alpha+2}$. In both cases, the useful payload is as claimed, which completes the proof. ■

Theorem 8.3. *Given an interval of length $T \geq 1$, Algorithm S-OPT$(T, 1)$ achieves optimal useful payload $UP^*_{(T,1)} = \frac{i-1}{i}T - \frac{i+1}{2} + \frac{1}{i}$, where i is the integer such that $T \in \left[\frac{(i-1)i}{2} + 1, \frac{i(i+1)}{2} + 1\right)$.*

Proof: The proof is by induction on T. The base case is when $T \in [1, 2)$, which implies that $i = 1$. In this case only one packet is sent by S-OPT$(T, 1)$, which spans the whole interval and can be jammed by the adversary. Observe that in this case at most one packet can in fact be sent in the interval. This matches the claim that S-OPT$(T, 1)$ achieves optimal useful payload $\text{UP}^*_{(T,1)} = 0$ in this case.

Consider now any interval length $T \geq 2$, which implies $i \geq 2$. Then, from Lemma 8.1, interval length $T' = T - p \in \left[\frac{(j-1)j}{2} + 1, \frac{j(j+1)}{2} + 1\right)$ for $j = i - 1$. By induction hypothesis, $\text{UP}_{(T',1)}(\text{S-OPT}) = \text{UP}^*_{(T',1)} = \frac{j-1}{j}T - \frac{j+1}{2} + \frac{1}{j} = \frac{\alpha T' - \beta}{\alpha+1}$, and from Lemma 8.2, $\text{UP}_{(T,1)}(\text{S-OPT}) = \frac{(\alpha+1)T - (\beta+\alpha+2)}{\alpha+2} = \frac{i-1}{i}T - \frac{i+1}{2} + \frac{1}{i}$.

To show that the useful payload achieved by S-OPT is optimal for this case $T \geq 2$, consider an algorithm A that follows one of the following approaches:

(a) First sends a packet π' of length $p' > \frac{T+\beta}{\alpha+2}$. Assume then that the adversary jams π'. The length of the rest of the interval is $T - p' < T - \frac{T+\beta}{\alpha+2}$. Hence the useful payload will be

$$\text{UP}_{(T,1)}(A) < T - \frac{T+\beta}{\alpha+2} - 1 = \frac{(\alpha+1)T - (\beta+\alpha+2)}{\alpha+2} = \text{UP}_{(T,1)}(\text{S-OPT}).$$

(b) First sends a packet π' of length $p' < \frac{T+\beta}{\alpha+2}$, $p' \geq 1$. Then the adversary does not jam π'. The rest of the interval has length $T - p' = T' + (p - p') > T'$. Consider two cases (from Lemma 8.1 no other case is possible):
Case (b).1: $T - p' = T' + (p - p') \in \left[\frac{(j-1)j}{2} + 1, \frac{j(j+1)}{2} + 1\right)$ for $j = i - 1$. Then, by induction hypothesis, $\text{UP}^*_{(T'+(p-p'),1)} = \frac{j-1}{j}(T' + (p - p')) - \frac{j+1}{2} + \frac{1}{j} < \frac{j-1}{j}T' - \frac{j+1}{2} + \frac{1}{j} + (p - p') = \text{UP}^*_{(T',1)} + (p - p')$. Hence,

$$\begin{aligned} \text{UP}_{(T,1)}(A) &\leq p' - 1 + \text{UP}^*_{(T'+(p-p'),1)} < p' - 1 + \text{UP}^*_{(T',1)} + (p - p') \\ &= p - 1 + \text{UP}^*_{(T',1)} = \text{UP}_{(T,1)}(\text{S-OPT}). \end{aligned}$$

Case (b).2: $T - p' = T' + (p - p') \in \left[\frac{(i-1)i}{2} + 1, \frac{i(i+1)}{2} + 1\right)$. In this case,

$$\begin{aligned} \text{UP}_{(T,1)}(A) &\leq p' - 1 + \text{UP}^*_{(T-p',1)} = p' - 1 + \frac{i-1}{i}(T - p') - \frac{i+1}{2} + \frac{1}{i} \\ &< \frac{i-1}{i}T - \frac{i+1}{2} + \frac{1}{i} = \text{UP}_{(T,1)}(\text{S-OPT}), \end{aligned}$$

where the first equality follows from induction hypothesis, and the second inequality follows from the fact that $p' < i$ (derived from $p' < \frac{T+\beta}{\alpha+2}$, the definition of α and β, and the fact that $T < \frac{i(i+1)}{2} + 1$).

Hence, in none of the two cases, neither (a) nor (b), Algorithm A was able to achieve a higher useful payload than S-OPT, which implies that the latter achieves optimality. ■

8.4. Algorithm S-OPT(T, f): optimal for any $f > 1$ in the static model

This section turns its focus on the case of any number of error tokens available to the adversary, for an interval of length T, i.e., $s > 1$. The general adaptive algorithm S-OPT(T, f) is presented, for $f > 1$ as Algorithm 9, and its optimality in the static model is proved. The pseudo-code of S-OPT(T, f) for $f > 1$ is similar to that of S-OPT$(T, 1)$, with a couple of differences. First, in this case it is not possible to explicitly give the length p of the first packet π sent (values of α, β, and γ) when $T \geq f + 1$ (see Theorem 8.4). Second, if π is jammed, the adversary still has some error tokens that it can use. Hence, instead of sending a packet that spans the rest of the interval, S-OPT(T, f) makes the call S-OPT$(T - p, f - 1)$, which could be recursive if $f > 2$, or a call to the algorithm S-OPT$(T - p, 1)$ (see Algorithm 8), if $f = 2$. It will not be surprising then

that the proof of optimality of the algorithm S-OPT(T, f) will use induction on f.

Algorithm 9: S-OPT(T, f), for $f > 1$

```
1 If T < f + 1 then
2     Send packet π with length p = T
3 else
4     Send packet π with length p = (αT+β)/γ      // α, β and γ depend on T; see Theorem 8.4
5     If packet π is jammed then
6         Call S-OPT(T − p, f − 1)
7     else
8         Call S-OPT(T − p, f)
```

Some observations that hold for any optimal algorithm OPT, are proven first, to be used later in the analysis of Algorithm S-OPT(T, f).

Observation 8.3. *The useful payload of an optimal algorithm OPT, follows a non-decreasing function with respect to the length of the interval of interest, T, when there are $f \geq 0$ available errors, i.e., $UP^*_{(T,f)} \leq UP^*_{(T+\delta,f)}$, for $\delta > 0$.*

Proof: Consider an optimal algorithm OPT that achieves optimal useful payload $\text{UP}^*_{(T,f)} = \alpha$, for an interval of length T and f error tokens available within the interval. Now construct an algorithm A, that for interval length $T + \delta$ initially uses the exact same approach as OPT for T; choosing the same packet lengths OPT does during the initial T time of the interval. This means that it has at least the same useful payload as OPT for T, i.e., $\text{UP}_{(T+\delta,f)}(A) \geq \alpha$. Since OPT is the optimal algorithm, it must achieve at least the same useful payload as A for the interval of length $T + \delta$, i.e., $\text{UP}^*_{(T+\delta,f)} \geq \text{UP}_{(T+\delta,f)}(A)$. Hence, $\text{UP}^*_{(T,f)} \leq \text{UP}^*_{(T+\delta,f)}$ as claimed. ■

Observation 8.4. *The useful payload of an optimal algorithm OPT, follows a non-increasing function with respect to the number of available errors in an interval of length T, i.e., $UP^*_{(T,f)} \leq UP^*_{(T,f-1)}$, where $f \geq 1$.*

Proof: Consider an optimal algorithm OPT, with a useful payload $\text{UP}^*_{(T,f)} = \beta$ for an interval length T with f errors available. Then, construct an algorithm A that for $f - 1$ error tokens during the same interval length T, uses the exact approach as OPT for f errors; choosing the same packet lengths until $f-1$ error tokens are used by the adversary. Then, it schedules one packet equal to the size of the remaining interval. This means that it has at least the same useful payload as OPT does for f errors, $\text{UP}_{(T,f-1)}(A) \geq \beta$. And since OPT is the optimal algorithm, it must achieve at least the same useful payload for the same interval and $f - 1$ errors, i.e., $\text{UP}^*_{(T,f-1)} \geq \text{UP}_{(T,f-1)}(A)$. Hence, $\text{UP}^*_{(T,f)} \leq \text{UP}^*_{(T,f-1)}$ as claimed. ■

Lemma 8.3. *There is an optimal algorithm OPT that is work-conserving, i.e., for each T and for each f, there is an optimal work-conserving strategy deciding the packet lengths.*

Proof: Assume by contradiction that there is some combination of interval and number of error tokens (T, f), for which no work-conserving scheduling strategy is optimal. Choose the

smallest such T and consider the following:
(1) There is an optimal strategy for this pair of T and f that does not send any packet during the interval. Hence the optimal useful payload is zero, $\text{UP}^*_{(T,f)} = 0$. In this case, sending one packet that spans the whole interval will lead to the same payload.
(2) There is a strategy that waits for Δ time at the beginning of the interval before sending a packet of length p. This packet can be jammed. Therefore,

$$\begin{aligned}\text{UP}^*_{(T,f)} &= \min\{\text{UP}^*_{(T-\Delta-p,f-1)}, p-1+\text{UP}^*_{(T-\Delta-p,f)}\} \\ &\leq \min\{\text{UP}^*_{(T-p,f-1)}, p-1+\text{UP}^*_{(T-p,f)}\}.\end{aligned}$$

Where the inequality follows from Observation 8.3. The right side of the inequality is the useful payload obtained by the strategy that does not wait the Δ period, but instead schedules the packet of length p at the beginning of the interval (which is work-conserving). Since both cases lead to a contradiction, the claim follows. ■

Lemma 8.4. *The optimal useful payload is a continuous function with respect to the length of the interval, T, when there are $f \geq 1$ errors available.*

Proof: Assume by contradiction that the optimal useful payload is not a continuous function. This means that there is an interval length T for which the following holds: $\lim_{\epsilon \to 0} \text{UP}^*_{(T-\epsilon,f)} < \text{UP}^*_{(T,f)}$. Fix parameter $\epsilon > 0$, and observe the behavior of a work-conserving optimal algorithm OPT for interval lengths T and $T-\epsilon$ (such an algorithm exists by Lemma 8.3). Then, denote by p_O and p_ϵ the lengths of the first packet scheduled by OPT in each case respectively. These packets can be jammed or not. Observe that

$$\text{UP}^*_{(T-\epsilon,f)} = \min\{\text{UP}^*_{(T-\epsilon-p_\epsilon,f-1)}, p_\epsilon - 1 + \text{UP}^*_{(T-\epsilon-p_\epsilon,f)}\} \tag{8.2}$$

$$\text{UP}^*_{(T,f)} = \min\{\text{UP}^*_{(T-p_O,f-1)}, p_O - 1 + \text{UP}^*_{(T-p_O,f)}\} \tag{8.3}$$

However, if an alternative algorithm A is constructed, one that chooses a packet of length $p'' = p_O - \epsilon$ in the case of interval of length $T-\epsilon$, and works as OPT for smaller interval lengths, then

$$\text{UP}_{(T-\epsilon,f)}(A) = \min\{\text{UP}^*_{(T-p_O,f-1)}, p_O - \epsilon - 1 + \text{UP}^*_{(T-p_O,f)}\} \geq \text{UP}^*_{(T,f)} - \epsilon.$$

Since $\text{UP}^*_{(T-\epsilon,f)} \geq \text{UP}_{(T-\epsilon,f)}(A)$, it is then trivial to conclude that $\lim_{\epsilon \to 0} \text{UP}^*_{(T-\epsilon,f)} = \text{UP}^*_{(T,f)}$, which is a contradiction. Hence the optimal useful payload is a continuous function with respect to the length of the interval, as claimed. ■

It will now be shown, how Algorithm S-OPT(T,f) computes the packet length p of the packet π sent when $T \geq f+1$. The computation assumes that it is possible to recursively call S-OPT(T',j) for any $T' < T$ and $j \leq f$, and that the useful payload of each of these recursive calls is the optimal value $\text{UP}^*(T',j)$. Then, S-OPT(T,f) chooses as length of packet π the

smallest value $p \in [1, T]$ that satisfies the equality $\text{UP}^*_{(T-p,f-1)} = p - 1 + \text{UP}^*_{(T-p,f)}$. Table 8.1 shows the values of p chosen for some interval lengths T when $f = 2$. It also shows the useful payload achieved by the algorithm using these values of p.

T	$[1, 3)$	$[3, 9/2)$	$[9/2, 17/3)$	$[17/3, 19/3)$	$[19/3, 70/9)$	$[70/9, 308/36)$
p	T	$\frac{T}{3}$	$\frac{T+6}{7}$	$\frac{3T+3}{12}$	$\frac{5T+16}{26}$	$\frac{6T+42}{42}$
$\text{UP}^*_{(T,2)}$	0	$\frac{T-3}{3}$	$\frac{3T-10}{7}$	$\frac{6T-22}{12}$	$\frac{14T-54}{26}$	$\frac{24T-98}{42}$

Table 8.1: Values of packet length p and optimal useful payload $\text{UP}^*_{(T,2)}$ achieved with Algorithm S-OPT$(T, 2)$.

The following theorem proves that the described process to make the choice leads to optimality.

Theorem 8.4. *Given an interval of length $T \geq f + 1$, Algorithm S-OPT(T, f) achieves optimal useful payload by choosing the smallest value $p \in [1, T]$ that satisfies the equality*

$$UP^*_{(T-p,f-1)} = p - 1 + UP^*_{(T-p,f)}.$$

Moreover, there are constants α_l, β_l, γ_l, α_k, β_k, and γ_k such that $UP^*_{(T-p,f)} = \frac{\alpha_l(T-p)-\beta_l}{\gamma_l}$ and $UP^*_{(T-p,f-1)} = \frac{\alpha_k(T-p)-\beta_k}{\gamma_k}$, and hence

$$p = \frac{(\alpha_k\gamma_l - \gamma_k\alpha_l)T + \gamma_k\gamma_l + \gamma_k\beta_l - \beta_k\gamma_l}{\gamma_k\gamma_l + \alpha_k\gamma_l - \gamma_k\alpha_l}.$$

(Observe that the parameters used in Algorithm 9 are hence $\alpha = \alpha_k\gamma_l - \gamma_k\alpha_l$, $\beta = \gamma_k\gamma_l + \gamma_k\beta_l - \beta_k\gamma_l$, and $\gamma = \gamma_k\gamma_l + \alpha_k\gamma_l - \gamma_k\alpha_l$.) The optimal useful payload obtained is then

$$UP^*_{(T,f)} = \frac{\alpha_k\gamma_l T - (\alpha_k\gamma_l + \alpha_k\beta_l + \beta_k\gamma_l - \beta_k\alpha_l)}{\gamma_k\gamma_l + \alpha_k\gamma_l - \gamma_k\alpha_l}.$$

Proof: Using a double induction on the number of error tokens f and the length of the interval T, it can be proven that the approach followed by Algorithm S-OPT(T, f) gives the optimal useful payload.

Base Cases. As base case of the induction on the number of error tokens: (1) when $f = 0$ the optimal strategy is to send a single packet of length T that spans the whole interval, leading to $\text{UP}^*_{(T,0)} = T - 1$, and (2) the algorithm S-OPT$(T, 1)$ presented before is optimal for any T, which covers the case $f = 1$.

For a given $f > 1$, induction in the length of the interval T is used. In this case the base case is when $T < f + 1$, which has optimal payload $\text{UP}^*_{(T,f)} = 0$, since the adversary can jam each of the up to f packets that can be sent.

Induction Hypotheses. Assume that S-OPT(T, j) is optimal for any number of tokens $j < f$

available to the adversary at the beginning of the interval and any interval length $T > j$. In particular, for any $j < f$ and any $T > j$, there is a known range $R_{ij} = [a_{ij}, b_{ij})$ such that $T \in R_{ij}$, and the optimal useful payload is known to be $\text{UP}^*_{(T,j)} = \frac{\alpha_{ij}T-\beta_{ij}}{\gamma_{ij}}$. Parameters α_{ij}, β_{ij} and γ_{ij} are known positive integers, such that $\beta_{ij} > \gamma_{ij} > \alpha_{ij}$.

Assume also, that for f error tokens, there are m known ranges $R_{if} = [c_{if}, d_{if})$ for $i = 1, 2, \ldots, m$, such that $\bigcup_{i=1}^{m} R_{if} = [1, d_{mf})$. Also, for any interval length T such that $T < d_{mf}$ and $T \in R_{if} = [c_{if}, d_{if})$, the optimal useful payload is known to be $\text{UP}^*_{(T,f)} = \frac{\alpha_{if}T-\beta_{if}}{\gamma_{if}}$. Parameters α_{if}, β_{if} and γ_{if} are known positive integers such that (1) $\beta_{if} > \gamma_{if} > \alpha_{if}$, and for $l \leq r \leq m$ it holds that (2) $\frac{\beta_{rf}}{\gamma_{rf}} \geq \frac{\beta_{lf}}{\gamma_{lf}}$ and (3) $\frac{\alpha_{rf}}{\gamma_{rf}} \geq \frac{\alpha_{lf}}{\gamma_{lf}}$.

Inductive Step. For interval length $T \in [d_{mf}, d_{mf} + 1)$, the algorithm S-OPT$(T, f)$ chooses the smallest packet length $p \in [1, T]$ that satisfies the following condition

$$\text{UP}^*_{(T-p,f-1)} = p - 1 + \text{UP}^*_{(T-p,f)}. \tag{8.4}$$

Claim 8.1. *There is at least one packet length $p \in [1, T]$ that satisfies Eq. 8.4.*

Proof: Observe that, when $p = 1$, from Observation 8.4 it holds that $\text{UP}^*_{(T-p,f-1)} \geq p-1+\text{UP}^*_{(T-p,f)}$. On the other hand, when $p = T$, $\text{UP}^*_{(T-p,f-1)} = 0 \leq p-1+\text{UP}^*_{(T-p,f)} = T-1$. Hence, taking into consideration the continuity of the useful payload function of both $f - 1$ and f error tokens (Lemma 8.4) and the Mean Value Theorem, there always exists a packet size $p \in [1, T]$ such that $\text{UP}^*_{(T-p,f-1)} = p - 1 + \text{UP}^*_{(T-p,f)}$. ■

Now, let p be the packet length chosen, and assume that $T - p \in R_{kj}$ and $T - p \in R_{lf}$. Then, by induction hypothesis $\text{UP}^*_{(T-p,f)} = \frac{\alpha_{lf}(T-p)-\beta_{lf}}{\gamma_{lf}}$ and $\text{UP}^*_{(T-p,f-1)} = \frac{\alpha_{kj}(T-p)-\beta_{kj}}{\gamma_{kj}}$. Then, solving Eq. 8.4 for p, the packet length is

$$p = \frac{(\alpha_{kj}\gamma_{lf} - \gamma_{kj}\alpha_{lf})T + \gamma_{kj}\gamma_{lf} + \gamma_{kj}\beta_{lf} - \beta_{kj}\gamma_{lf}}{\gamma_{kj}\gamma_{lf} + \alpha_{kj}\gamma_{lf} - \gamma_{kj}\alpha_{lf}},$$

and the useful payload obtained is

$$\begin{aligned} \text{UP}_{(T,f)}(\text{S-OPT}) &= \text{UP}^*_{(T-p,f-1)} = p - 1 + \text{UP}^*_{(T-p,f)} = \frac{\alpha_{kj}(T-p) - \beta_{kj}}{\gamma_{kj}} \\ &= \frac{\alpha_{kj}\gamma_{lf}T - (\alpha_{kj}\gamma_{lf} + \alpha_{kj}\beta_{lf} + \beta_{kj}\gamma_{lf} - \beta_{kj}\alpha_{lf})}{\gamma_{kj}\gamma_{lf} + \alpha_{kj}\gamma_{lf} - \gamma_{kj}\alpha_{lf}}, \end{aligned}$$

as claimed. To complete the induction step, let $\alpha = \alpha_{kj}\gamma_{lf}$, $\beta = \alpha_{kj}\gamma_{lf}+\alpha_{kj}\beta_{lf}+\beta_{kj}\gamma_{lf}-\beta_{kj}\alpha_{lf}$ and $\gamma = \gamma_{kj}\gamma_{lf}+\alpha_{kj}\gamma_{lf}-\gamma_{kj}\alpha_{lf}$. Then, the following three properties are shown: (1) $\beta > \gamma > \alpha$, (2) $\frac{\beta}{\gamma} \geq \frac{\beta_{lf}}{\gamma_{lf}}$, and (3) $\frac{\alpha}{\gamma} \geq \frac{\alpha_{lf}}{\gamma_{lf}}$, as follows.

Property 8.1. *For the new parameters $\alpha = \alpha_{kj}\gamma_{lf}$, $\beta = \alpha_{kj}\gamma_{lf} + \alpha_{kj}\beta_{lf} + \beta_{kj}\gamma_{lf} - \beta_{kj}\alpha_{lf}$ and $\gamma = \gamma_{kj}\gamma_{lf} + \alpha_{kj}\gamma_{lf} - \gamma_{kj}\alpha_{lf}$, it holds that $\beta > \gamma > \alpha$.*

Proof: First, from the *induction hypotheses*, recall the definition of parameters α_{ij},β_{ij} and γ_{ij},

being known positive integers such that $\beta_{ij} > \gamma_{ij} > \alpha_{ij}$. Looking now at the current parameters α, β and γ individually, the following holds:
(a) $\alpha = \alpha_{kj}\gamma_{lf}$.
(b) $\beta = \alpha_{kj}\gamma_{lf} + \alpha_{kj}\beta_{lf} + \beta_{kj}\gamma_{lf} - \beta_{kj}\alpha_{lf} = \alpha_{kj}(\gamma_{lf} + \beta_{lf}) + \beta_{kj}(\gamma_{lf} - \alpha_{lf})$.
(c) $\gamma = \gamma_{kj}\gamma_{lf} + \alpha_{kj}\gamma_{lf} - \gamma_{kj}\alpha_{lf} = \gamma_{kj}(\gamma_{lf} - \alpha_{lf}) + \alpha_{kj}\gamma_{lf}$.

Observe that $\gamma_{kj}(\gamma_{lf} - \alpha_{lf}) + \alpha_{kj}\gamma_{lf} > \alpha_{kj}\gamma_{lf}$, since $\gamma_{kj} > 0$ and $\gamma_{lf} - \alpha_{lf} > 0$ by induction hypothesis. Hence, from (a) and (c) $\gamma > \alpha$. Also, $\alpha_{kj}(\gamma_{lf} + \beta_{lf}) + \beta_{kj}(\gamma_{lf} - \alpha_{lf}) > \gamma_{kj}(\gamma_{lf} - \alpha_{lf}) + \alpha_{kj}\gamma_{lf}$, since by induction hypothesis $\beta_{kj} > \gamma_{kj}$, $\gamma_{lf} - \alpha_{lf} > 0$, and all parameters are positive. Hence, from (b) and (c) $\beta > \gamma$ holds as well. This completes the proof of the claim. ■

Property 8.2. *For the new parameters $\beta = \alpha_{kj}\gamma_{lf} + \alpha_{kj}\beta_{lf} + \beta_{kj}\gamma_{lf} - \beta_{kj}\alpha_{lf}$ and $\gamma = \gamma_{kj}\gamma_{lf} + \alpha_{kj}\gamma_{lf} - \gamma_{kj}\alpha_{lf}$, it holds that $\frac{\beta}{\gamma} > \frac{\beta_{lf}}{\gamma_{lf}}$.*

Proof: For this proof observe first, that since $\beta > \gamma$ (as shown in Property 8.1), the fact that $\frac{\beta}{\gamma} > \frac{\beta - c}{\gamma - c}$, where c is positive, can safely be used. Also by induction hypothesis, $\gamma_{lf} - \alpha_{lf} > 0$ and $\beta_{kj} - \gamma_{kj} > 0$. Some fraction inequality properties are therefore used as follows:

$$\begin{aligned}
\frac{\beta}{\gamma} &= \frac{\alpha_{kj}\gamma_{lf} + \alpha_{kj}\beta_{lf} + \beta_{kj}\gamma_{lf} - \beta_{kj}\alpha_{lf}}{\gamma_{kj}\gamma_{lf} + \alpha_{kj}\gamma_{lf} - \gamma_{kj}\alpha_{lf}} = \frac{\alpha_{kj}(\gamma_{lf} + \beta_{lf}) + \beta_{kj}(\gamma_{lf} - \alpha_{lf})}{\gamma_{kj}(\gamma_{lf} - \alpha_{lf}) + \alpha_{kj}\gamma_{lf}} \\
&> \frac{\alpha_{kj}(\gamma_{lf} + \beta_{lf}) + (\beta_{kj} - \gamma_{kj})(\gamma_{lf} - \alpha_{lf})}{\alpha_{kj}\gamma_{lf}} > \frac{\alpha_{kj}\gamma_{lf} + \alpha_{kj}\beta_{lf}}{\alpha_{kj}\gamma_{lf}} = 1 + \frac{\beta_{lf}}{\gamma_{lf}} \\
&> \frac{\beta_{lf}}{\gamma_{lf}},
\end{aligned}$$

which completes the proof. ■

Property 8.3. *For the new parameters $\alpha = \alpha_{kj}\gamma_{lf}$ and $\gamma = \gamma_{kj}\gamma_{lf} + \alpha_{kj}\gamma_{lf} - \gamma_{kj}\alpha_{lf}$, it holds that $\frac{\alpha}{\gamma} > \frac{\alpha_{lf}}{\gamma_{lf}}$.*

Proof: For this proof observe first, that since $\gamma > \alpha$ (as shown in Property 8.1), the fact that $\frac{\alpha}{\gamma} > \frac{\beta + c}{\gamma + c}$, where c is positive, can safely be used. Also by induction hypothesis, $\gamma_{lf} - \alpha_{lf} > 0$ holds. Some fraction inequality properties are therefore used as follows:

$$\begin{aligned}
\frac{\alpha}{\gamma} &= \frac{\alpha_{kj}\gamma_{lf}}{\gamma_{kj}\gamma_{lf} + \alpha_{kj}\gamma_{lf} - \gamma_{kj}\alpha_{lf}} = \frac{\alpha_{kj}\gamma_{lf} + \gamma_{kj}\alpha_{lf}}{\alpha_{kj}\gamma_{lf} + \gamma_{kj}\gamma_{lf}} \\
&= \frac{\alpha_{kj}\alpha_{lf} + \alpha_{kj}(\gamma_{lf} - \alpha_{lf}) + \gamma_{kj}\alpha_{lf}}{\gamma_{lf}(\alpha_{kj} + \gamma_{kj})} = \frac{\alpha_{lf}(\alpha_{kj} + \gamma_{kj})}{\gamma_{lf}(\alpha_{kj} + \gamma_{kj})} + \frac{\alpha_{kj}(\gamma_{lf} - \alpha_{lf})}{\gamma_{lf}(\alpha_{kj} + \gamma_{kj})} \\
&> \frac{\alpha_{lf}}{\gamma_{lf}},
\end{aligned}$$

which completes the proof. ■

It must now be shown, that this useful payload is in fact optimal in the static model. Assume by contradiction that an algorithm A is able to achieve a larger useful payload for the pair (T, f)

by sending first a different packet length $p' \neq p$. Consider the following cases:
(a) Algorithm A chooses a packet π' of length $p' > p$. Then, assume that the adversary will jam the packet π'. Hence, the useful payload achieved by A will be upper bounded as $\text{UP}_{(T,f)}(A) \leq \text{UP}^*_{(T-p',f-1)}$ which by Observation 8.3 is smaller than $\text{UP}^*_{(T-p,f-1)} = \text{UP}_{(T,f)}(\text{S-OPT})$, since $T - p' < T - p$.
(b) Algorithm A chooses a packet π' of length $p' < p$. Observe that p' does not satisfy Eq. 8.4, since p is the smallest length that does. Then the adversary does not jam π'. Then, $\text{UP}_{(T,f)}(A) \leq p' - 1 + \text{UP}^*_{(T-p',f)}$. It is shown now that this value is no larger than $p - 1 + \text{UP}^*_{(T-p,f)} = \text{UP}_{(T,f)}(\text{S-OPT})$. Let $T - p' \in R_{rf}$, where $r \geq l$. Then, $\text{UP}^*_{(T-p',f)} = \frac{\alpha_{rf}(T-p')-\beta_{rf}}{\gamma_{rf}} \leq \frac{\alpha_{rf}}{\gamma_{rf}}(T - p') - \frac{\beta_{lf}}{\gamma_{lf}}$, since $\frac{\beta_{rf}}{\gamma_{rf}} \geq \frac{\beta_{lf}}{\gamma_{lf}}$ as shown by Property 8.2. Similarly, $\text{UP}^*_{(T-p,f)} = \frac{\alpha_{lf}(T-p)-\beta_{lf}}{\gamma_{lf}} \geq \frac{\alpha_{rf}}{\gamma_{rf}}(T - p) - \frac{\beta_{lf}}{\gamma_{lf}}$, since $\frac{\alpha_{rf}}{\gamma_{rf}} \geq \frac{\alpha_{lf}}{\gamma_{lf}}$ as shown by Property 8.3. Finally, combining these bounds and the fact that $\frac{\alpha_{rf}}{\gamma_{rf}} < 1$ (see Property 8.1),

$$\begin{aligned}
\text{UP}_{(T,f)}(A) &\leq p' - 1 + \text{UP}^*_{(T-p',f)} \leq p' - 1 + \frac{\alpha_{rf}}{\gamma_{rf}}(T - p') - \frac{\beta_{lf}}{\gamma_{lf}} \\
&\leq p' - 1 + \frac{\alpha_{rf}}{\gamma_{rf}}(T - p') - \frac{\beta_{lf}}{\gamma_{lf}} + (p - p') - \frac{\alpha_{rf}}{\gamma_{rf}}(p - p') \\
&= p - 1 + \frac{\alpha_{rf}}{\gamma_{rf}}(T - p) - \frac{\beta_{lf}}{\gamma_{lf}} \leq \text{UP}_{(T,f)}(\text{S-OPT}).
\end{aligned}$$

In all cases the resulting useful payload is smaller than the one achieved by choosing the smallest packet size p such that $\text{UP}^*_{(T-p,f-1)} = p - 1 + \text{UP}^*_{(T-p,f)}$. Hence the packet size calculated by S-OPT(T, f) is optimal. ■

8.5. Uniform packets for the dynamic model

The main goal for the algorithms in the dynamic model, is to maximize the data successfully transmitted to the receiver in any interval T. This, corresponds to minimizing the transmission time needed to successfully transmit a total amount of data P to the receiver, considering a value P that will eventually grow to infinity. As a consequence, this would also maximize the goodput rate, which is our main efficiency measure for the two models. Knowing both adversarial parameters, ρ and σ, consider algorithm D-UNI and uniform packets of size $p_{.len} = l + 1 < 1/\rho$. The quasi optimal value for the length of the payload l in each packet, can then be found, and minimize the transmission time. For simplicity, assume that the total length of the data to be transmitted, P, is a multiple of the payload length l. (For large values of P the error introduced by this assumption is negligible.) Then, the objective is that P/l packets arrive successfully at the receiver.

A *lower bound* on the transmission time that can be achieved using uniform packets can then be derived. Denote with $Tr(l)$ the transmission time with packets of uniform payload l. Let r be

the number of packets jammed and re-transmitted by the sender. Then,

$$Tr(l) = (P/l + r)(l + 1). \tag{8.5}$$

Observe that the last packet transmitted was correctly received, since otherwise the data would have been completely transmitted by time $Tr(l) - (l+1)$, which contradicts the fact that $Tr(l)$ is the transmission time. Hence, the number of packets jammed and re-transmitted is upper bounded as

$$r \leq \lceil (Tr(l) - (l + 1))\rho \rceil - 1 + \sigma, \tag{8.6}$$

where the fact that the last error used by the adversary must have been available before time $Tr(l) - (l+1)$, is used. It is claimed, that the number of packets jammed by the adversary and re-transmitted is in fact equal to the bound of Eq. 8.6. Otherwise, the adversary could have jammed the last packet sent (at time $Tr(l) - (l + 1)$), achieving a longer transmission time. Hence,

$$r = \lceil (Tr(l) - (l + 1))\rho \rceil - 1 + \sigma. \tag{8.7}$$

Moreover, since the adversary could not jam the last packet sent, it must also hold that $r+1 \geq Tr(l)\rho + \sigma = (P/l + r)(l + 1)\rho + \sigma$, from which the value of r can be bounded as

$$r \geq \frac{P\rho(l + 1) + (\sigma - 1)l}{l - l\rho(l + 1)}. \tag{8.8}$$

Let the lower bound of the transmission time when packets of uniform payload l are used, be defined as function $LB(l)$. Then,

Lemma 8.5. *Using algorithm D-UNI with uniform packets of payload l, the lower bound of the transmission time is*

$$Tr(l) \geq LB(l) = \frac{P + (\sigma - 1)l}{l(1 - \rho(l + 1))}(l + 1).$$

Proof: Replacing the lower bound of r (Eq. 8.8) in Eq. 8.5,

$$Tr(l) \geq \left(\frac{P}{l} + \frac{P\rho(l + 1) + (\sigma - 1)l}{l - l\rho(l + 1)} \right) (l + 1) = \frac{P + (\sigma - 1)l}{l(1 - \rho(l + 1))}(l + 1),$$

which when combined with the definition of $LB(l)$, completes the proof. ■

Using Calculus, the payload length l^* that minimizes $LB(l)$, can be found, and yield to the following theorem.

Theorem 8.5. *Using uniform packets the transmission time is lower bounded as*

$$Tr \geq LB(l^*) = \frac{P + (\sigma - 1)l^*}{l^*(1 - \rho(l^* + 1))}(l^* + 1)$$

and the goodput rate is upper bounded as

$$G(\textit{D-UNI}) \leq \frac{P}{LB(l^*)} = \frac{Pl^*(1-\rho(l^*+1))}{(P+(\sigma-1)l^*)(l^*+1)},$$

where

$$l^* = \frac{\sqrt{P(P\rho+(\sigma-1)(1-\rho))}-P\rho}{P\rho+\sigma-1}.$$

Obviously, when P tends to ∞, so does the transmission time Tr. However, in this case, an upper bound on the goodput can be derived as follows.

Corollary 8.1. *Using algorithm D-UNI with uniform packets, the goodput rate is upper bounded as* $G(\textit{D-UNI}) \leq (1-\sqrt{\rho})^2$*, and in the limit as the value of* P *grows,*

$$G^* = \lim_{P\to\infty} G(\textit{D-UNI}) = (1-\sqrt{\rho})^2$$

Proof: Using Calculus it can be shown that the upper bound of G(D-UNI) obtained in Theorem 8.5 grows with P. Observe that $\lim_{P\to\infty} G(\text{D-UNI}) = l^*(1-\rho(l^*+1))/(l^*+1)$ and $\lim_{P\to\infty} l^* = (\sqrt{\rho}-\rho)/\rho = 1/\sqrt{\rho}-1$. Replacing the latter in the former the claims follow. ■

A corresponding *upper bound* on the transmission time will now be shown. Start by combining Eqs. 8.7 and 8.5 as follows:

$$\begin{aligned} r &= \lceil (Tr(l)-(l+1))\rho \rceil - 1 + \sigma < (Tr(l)-(l+1))\rho + \sigma \\ &= ((P/l+r)(l+1)-(l+1))\rho + \sigma \\ &= (P/l+r)(l+1)\rho + \sigma - (l+1)\rho. \end{aligned}$$

This allows to find an upper bound of r as

$$r < \frac{P\rho(l+1)+(\sigma-(l+1)\rho)l}{l-l\rho(l+1)}. \tag{8.9}$$

Let the upper bound of the transmission time when packets of payload l are used, be defined as function $UB(l)$. Then,

Lemma 8.6. *Using algorithm D-UNI with uniform packets of payload* l*, the upper bound of the transmission time is*

$$Tr(l) < UB(l) = \frac{P+(\sigma-(l+1)\rho)l}{l(1-\rho(l+1))}(l+1).$$

Proof: Replacing the upper bound of r (Eq. 8.9) in Eq. 8.5,

$$Tr(l) < \left(\frac{P}{l} + \frac{P\rho(l+1)+(\sigma-(l+1)\rho)l}{l-l\rho(l+1)}\right)(l+1) = \frac{P+(\sigma-(l+1)\rho)l}{l(1-\rho(l+1))}(l+1),$$

which when combined with the definition of $UB(l)$, completes the proof. ■

From Observation 8.1, $\rho < 1/(l+1)$ must hold. Then, $(l+1)\rho < 1$ and the bound obtained in the above lemma is strictly bigger than the lower bound presented in Lemma 8.5, as expected. In fact, the gap between bounds can be obtained as shown in the following lemma.

Lemma 8.7. *Using uniform packets of payload l, the transmission time satisfies $Tr(l) \in [LB(l), LB(l)+l+1)$.*

Proof: Recall that the lower bound $LB(l)$ is obtained in Lemma 8.5. Subtracting this expression from the upper bound $UB(l)$ presented in Lemma 8.6,

$$\begin{aligned} UB(l) - LB(l) &= \frac{P+(\sigma-(l+1)\rho)l}{l(1-\rho(l+1))}(l+1) - \frac{P+(\sigma-1)l}{l(1-\rho(l+1))}(l+1) \\ &= \frac{l(1-\rho(l+1))}{l(1-\rho(l+1))}(l+1) = l+1. \end{aligned}$$

From the above and the fact that $Tr(l) < UB(l)$ the claim follows. ■

Corollary 8.2. *Using uniform packets of payload l, $Tr(l)$ is the only multiple of $l+1$ that falls in the interval $[LB(l), LB(l)+l+1)$.*

Finally, combining Lemma 8.7 with Theorem 8.5 the following theorem is derived.

Theorem 8.6. *Consider l^* as defined in Theorem 8.5. Then*

- *the transmission time $Tr(l^*)$ observed is less that l^*+1 (one packet) longer that the optimal. I.e., $Tr(l^*) < Tr + l^* + 1$.*

- *the goodput $G(l^*)$ converges to the optimal goodput G(D-UNI) as P grows. Additionally, when P goes to infinity the goodput matches the optimal G^*, i.e. $\lim_{P\to\infty} G(l^*) = \lim_{P\to\infty} G$(D-UNI$) = (1-\sqrt{\rho})^2$.*

Proof: The first claim follow directly from Lemma 8.7, since the value of l^* is the one that minimizes $LB(l)$. For the second, recall that $G(l^*) = \frac{P}{Tr(l^*)}$. Hence, observing again Lemma 8.7,

$$G(l^*) > \frac{P}{LB(l^*)+l^*+1} = \frac{1}{\frac{LB(l^*)}{P} + \frac{l^*+1}{P}}.$$

As P grows $\frac{l^*+1}{P}$ tends to 0, making $G(l^*)$ converge to $P/LB(l^*)$ which is an upper bound on the optimal goodput. Finally, as shown in Corollary 8.1, when P tends to infinity, $P/LB(l^*)$ tends to $(1-\sqrt{\rho})^2$, which completes the proof. ■

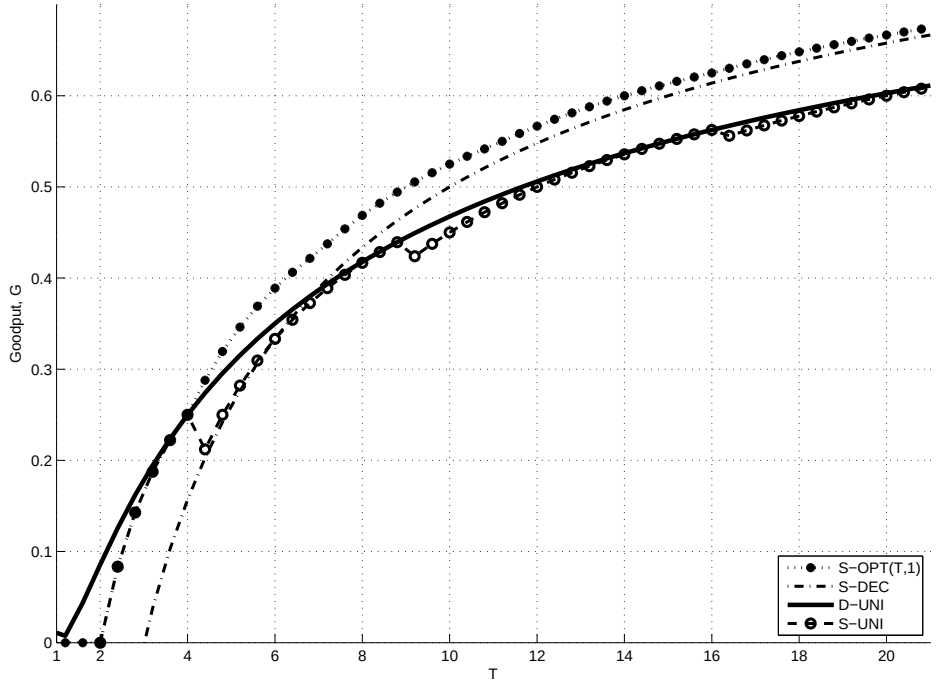

Figure 8.1: The goodput rate of algorithms S-OPT$(T, 1)$, S-DEC and the uniform packet scheduling for both static and dynamic models, with $\sigma = f = 1$ in a time interval $1/\rho = T = 1 \ldots 22$.

8.6. Discussion

In this chapter, an Adversarial Queuing Theory (AQT) approach has been applied to model the constrained adversarial jamming considered. AQT is a well known theoretical modeling tool, used for the first time to restrict the channel jams on wireless networks, leading to the *dynamic model* studied. The constrained adversarial entity chosen, considered a bounded error-token capacity σ and an error-token availability rate ρ. This model could be applied in various battery-operated malicious devices, such as drones or mobile jammers.

A *static* model has also been studied, for which new parameters were considered; for an interval of time T the adversary is able to create at most f jams, having all f error-tokens available at the beginning of the interval. This model is used as a building block in the attempt to find a solution to the problem of the dynamic model.

First, an upper bound on the goodput rate of the static model is shown, proposing algorithm S-UNI. Then, focusing on $f = 1$, it is shown that adaptive algorithms that change the packet length based on feedback received for jammed packets, can actually achieve better goodput rates, thus the uniform packet scheduling is not the best approach. What might seem surprising is that even for the "simple" case of $f = 1$, the analysis of the adaptive algorithms is nontrivial, and imposes constraints also on T.

In Figure 8.1, the reader can see a graphical representation of the improvement in the goodput

rate by the different algorithms developed, for the case of $\sigma = f = 1$ and $T = 1/\rho$. Unfortunately, as also shown by the analysis, algorithm S-DEC is better than S-UNI only for $T > \frac{2}{7-3\sqrt{5}}$. Nonetheless, this has given a positive intuition for the fact that other adaptive algorithms may exist with better goodput rate, as well as for the smaller time intervals. Then, algorithm S-OPT$(T, 1)$ is proposed, which exceeds the performance of S-UNI for $T > 4$, and is in fact optimal. Finally, the goodput rate of the uniform packet scheduling algorithm D-UNI, developed for the dynamic model, is shown to be better than all proposed algorithms for intervals $T < 4$. This is probably due to the fact that D-UNI is not restricted to fit the packet length in the intervals $1/\rho$, and further investigation is necessary to see whether there exists any other adaptive packet scheduling algorithm that exceeds that goodput rate.

In Section 8.1.3 a recursive algorithm ALG$_D$ is proposed, using the optimal solution of the static model to solve the problem in the dynamic model. It divides the executions into consecutive intervals of length $1/\rho$, and assumes σ error tokens available at the beginning of each one. Then these intervals can be seen as instances of the static model, where $T = 1/\rho$ and $f = \sigma$. However, this algorithm may not be the best possible, as it makes the pessimistic assumption that at the beginning of each interval, the adversary will have all σ error tokens available to use; this is true for the first interval, but in successive intervals this might not be the case (with the exception of the case $\sigma = 1$, which is discussed further below).

Based on the dynamic model, a new error token will be arriving at the beginning of each interval. If there are already σ tokens, then a token is lost (σ represents, for example, the capacity of the battery of a jamming device – this cannot be exceeded). If in this interval, the adversary performs, say three packet jams, then at the beginning of the next interval it will have $\sigma - 2$ available tokens. If the scheduling algorithm keeps track of this, then in this interval it should use S-OPT$(1/\rho, \sigma - 2)$ instead of S-OPT$(1/\rho, \sigma)$. So, in order to produce more efficient solutions, the scheduling algorithm needs to keep track (using the feedback mechanism) how many jams took place in the previous interval, and using its knowledge of $1/\rho$, run the appropriate version of S-OPT$()$. Although there are other subtle issues that also need to be considered, the proposed approach can be used as the basis for obtaining an optimal solution to the continuous version of the problem.

Regarding the case of $f = \sigma = 1$, as demonstrated in Figure 8.1 above, algorithm S-OPT$(1/\rho, 1)$ obtains better results than Algorithm S-DEC. Since in the case of $\sigma = 1$ it is best for the adversary to use the error token (otherwise it will lose it), the improved goodput demonstrates the promise of the aforementioned approach. Nonetheless, note that the uniform packet scheduling algorithm D-UNI still achieves better goodput rate for some small values of T. Apart from whether *that* can be exceeded, an intriguing open question is whether it is still possible to obtain better efficiency than the uniform packet lengths "policy", with adaptive algorithms for $\sigma > 1$. Considering for example $\sigma = 2$ seems to already be a challenging task.

Chapter 9

Conclusions

This thesis explores two families of *online scheduling problems* in computer science: Task Scheduling in computational systems prone to failures, and Packet Scheduling in unreliable wireless communication links. This final chapter discusses the main results of the thesis focusing on the most important contributions of each part, and showing the connection between the two families of problems. Nonetheless, all the results are of importance for the community of online scheduling; they give answers to some challenging problems of the area, contributing mainly with upper and lower bounds of the performance measures that have been taken into account. It is surprising that some of the simplest versions of the models considered in the thesis have not been previously studied. Open problems and possible future research lines that arise from the two areas explored are also discussed.

9.1. Task scheduling

The first and main problem of interest is the *task scheduling* in computing systems, presented in Part I. As seen, these systems present several challenges; there are continuous and dynamic task arriving, of different sizes (processing times) and the machines are prone to unpredictable crashes and restarts. Apart from that, such computing systems spend large amounts of energy, hence preserving their power consumption is another important aspect that should be considered.

In Chapter 3 the general model of these systems is described in detail, exploiting the three parameters that affect the performance of the scheduling algorithms; i.e., the number of machines m, the number of different task sizes k, and the amount of machine speedup s (which affects the overall power consumption). In Chapters 4 to 6, worst-case analysis is pursued for work-conserving and deterministic algorithms, by means of competitiveness with respect to three equally important performance measures; the completed-load, the pending load, and the latency. Long-term competitiveness is also considered when appropriate, looking at the performance as time goes to infinity.

Starting from the simplest model of one machine and no speedup, it is shown that little can be

achieved by *any* scheduling algorithm. Only limited completed-load competitiveness is possible, which also depends on the number of different task sizes taken into account. It is shown that if there is an arbitrary number of task sizes, no competitiveness can be achieved neither for the completed nor for the pending load measures. Since the performance of the system is limited in the simplest model, where $m = s = 1$, a form of resource augmentation is essential. The thesis exploits it by increasing the speedup of the machines, $s \geq 1$, and shows a threshold for the necessary and sufficient amount of speedup, in order to achieve some competitiveness (in the case of the completed-load, optimal competitiveness). More precisely, if $s < \min\{\rho, 1 + \gamma/\rho\}$, where $\rho = \pi_{max}/\pi_{min}$ (the ratio of the largest and smallest task size) and $\gamma = \max\{\lceil \frac{\rho - s}{s-1} \rceil, 0\}$, then no deterministic algorithm can be competitive neither with respect to the pending load nor with respect to latency, or be 1-completed-load competitive. Once the speedup surpasses this threshold, it is shown that competitiveness can be achieved. In particular, any work-conserving algorithm ALG running with speedup $s \geq \rho$ can achieve completed load competitiveness $\mathcal{C}(\text{ALG}) \geq 1/\rho$ and pending load competitiveness $\mathcal{P}(\text{ALG}) \leq \rho$, while as soon as $s \geq 1 + \rho$, both completed and pending load measures become 1. For the latency competitiveness, no general result is shown, and I believe it cannot, because it seems to be more dependent to the scheduling decisions of each particular algorithm than the other two measures. However, since the amount of resource augmentation might be large, depending on the value of ρ, one of the most important results is the proposal of algorithm γ-Burst, which has optimal competitiveness in all performance measures when run with $s \in [1 + \gamma/\rho, \rho)$. Recall that the more speedup, the larger the power consumption of such systems. One of the goals of the thesis has therefore been to optimize the performance of algorithms, while using the least amount of speedup possible. The only "handicap" of algorithm γ-Burst is that it considers only two different task sizes. It would be of great value to explore the idea behind its scheduling policy for the case of arbitrary task sizes in the future.

The analysis also includes some of the most widely used scheduling algorithms; i.e., LIS, SIS, LPT, SPT, each of which scales differently with the amount of speedup used. It is quite surprising that no previous research has been found for these popular algorithms considering such a model, not even for the case of single machine. In general, it is difficult to distinguish a *best* algorithm overall. Each one provides a better trade-off between competitiveness and speedup for a different performance metric. For example, with the exception of SPT, for the completed load measure and only when two task sizes are possible, no other algorithm (from these four popular ones) is competitive in any of the measures when $s < \rho$. Then, in terms of latency only algorithm LIS is competitive, and only when $s \geq \rho$. These two results may seem intuitive and not very surprising to some, but they now provide formal upper and lower bounds for the performance of these popular algorithms in the setting considered. It is interesting that even LIS, which basically follows the FIFO policy becomes competitive only when sufficient amount of speedup is provided. This shows some of the difficulty of proving any general result for all work-conserving or deterministic algorithms regarding latency competitiveness, as done for the completed and pending load measures. Another interesting observation, is the fact that algorithms LPT and SPT–

the ones that schedule tasks according to their size – become 1-completed and 1-pending load load competitive as soon as $s \geq \rho$, while algorithms LIS and SIS– the ones that schedule tasks according to their arrival time – require a greater amount of speedup. This means that for *these* measures the sizes of the pending tasks are more critical than the arrival times of the tasks. I believe it would be very interesting to formally define some classification of scheduling algorithms and prove the scaling trends of their performance with the increase of speedup. This would give a better understanding of the strengths of each class of algorithms and their relation with each performance metric.

Moving to the case of multiple machines, the research focuses on two of the performance measures: completed and pending load competitiveness. The algorithms proposed in this model use the parallel nature of the system in order to guarantee non redundant task executions. For that, *enough* tasks need to be pending in the repository. An important part of Chapter 6, is the definition and analysis of algorithms that are of type GroupLIS(β). The definition of this type of algorithms is a way to classify parallel scheduling algorithms for which some important results are shown. They are algorithms that classify the pending tasks according to their sizes, sort them according to their arrival times and when there are at least $\beta \cdot m^2$ tasks pending, a machine p schedules the $(p \cdot \beta m^2)$th task from the desired class. Any algorithm of type GroupLIS(β) is 1-completed and 1-pending load competitive in the case of no speedup and uniform task sizes, exactly as in the case of work-conserving algorithms in the single machine setting. The negative results of the single machine setting still hold though, regarding all work-conserving and deterministic algorithms; no algorithm can be competitive with respect to the performance measures in the case of no speedup and arbitrary number of task sizes, no algorithm can have long-term completed load competitive ratio more than $\frac{\bar{\rho}}{\rho+\bar{\rho}}$ without speedup but only two task sizes, and no algorithm can be pending load competitive or 1-completed-load competitive in the case of arbitrary task sizes if $s < \min\{\rho, 1 + \gamma/\rho\}$. However, if speedup is $s \geq \rho$ these GroupLIS(β) algorithms are at least $1/\rho$-completed-load competitive and ρ-pending-load competitive, whereas if speedup is $s \geq 1 + \rho$ they become 1-completed and 1-pending load competitive. Observe that these results are the analogous positive general results of the single machine case, though only for this particular group of work-conserving algorithms that guarantee non redundant task executions when "enough" tasks are pending. This classification of parallel algorithms gives a first insight on the necessary characteristics of such scheduling algorithms in order to achieve some competitiveness, and can be used in future research to derive more general results.

Furthermore, some specific algorithms have been proposed and analyzed for different amounts of speedup, with the objective to achieve optimal competitiveness while minimizing the speedup used. Apart from the redundancy avoidance mechanism suggested by GroupLIS(β) algorithms, which must be present in these algorithms, amortization techniques seem to be essential for this part as well, especially for the case of more than two task sizes.

It is important to note that the latency measure is more complicated to analyze in this case of multiple machines. Nonetheless, it is very important especially for the cases where starvation of

pending tasks cannot be accepted. I find it a challenging and very significant research line for the future, and I hope to extend the results presented in this thesis showing its relation to the other two measures as well.

In general, Part I completes a thorough analysis of the three possible directions for the whole framework of the online task scheduling problem – increasing the number of machines, looking at different number of task sizes, and considering different ranges of speedup – while exploring three different performance measures. The work done has combined several aspects of task scheduling problems that were only studied in different settings or individually through the years of research on the area, tackling overall the importance of energy efficiency in such systems. It shows the value of resource augmentation in the form of processing speedup, giving the necessary and sufficient amount of speedup in order to guarantee competitiveness. Then, depending on the performance measure of interest and the exact model considered, one can choose among the algorithms presented in order to guarantee optimal executions.

Finally, apart from all the answers and insights given with this thesis, some interesting questions are created as well. A challenging research line for one to consider, would be to look at the problem from a more practical point of view and compare the upper and lower bounds of worst-case scenarios found, with actual real life experiments; i.e., in data centers. Note however, that these experiments would provide insight on the average case scenarios, which would give a broader image of the task scheduling problems in fault-prone settings. Furthermore, considering practical setting, machines could be heterogeneous, having different processing powers. This could be translated as having different speedups for the machines, and thus the system would behave differently, probably requiring new algorithms in order to achieve optimal performance. Adding communication between the machines, thus changing the model from being parallel to a more distributed one, would be yet another extension of further challenge; different analysis methods might be necessary and further parameters should be taken into consideration. One could also consider the case of unreliable communication between the machines, thus combining in some sense the two families of problems studied in the thesis. As mentioned earlier, there are some unanswered questions in the lines of the research done in the thesis. They mostly concern the latency competitiveness, especially in the case of multiple machines setting, and *that* is something I would like to look at in the near future. Considering randomized algorithms and pursuing the corresponding analysis is yet another possible extension of this work, which would broaden the scope and knowledge for the problem. Using randomization makes some different assumptions that one should take into account but it usually helps to improve the competitiveness bounds, thus I believe it is worth studying.

9.2. Packet scheduling

The second family of problems studied in the thesis is *packet scheduling* over an unreliable wireless link. This is one of the fundamental problems of computer networks, and although the

connection has not been considered before, it is directly related to the task scheduling problem; packets arrive to the sending station dynamically, may have different lengths, and the communication channel may suffer from interference, noise, or even be jammed, thus disrupting the transmissions. There are two problems defined in this part, sharing the core of their model; one unreliable wireless channel between two nodes with the sender trying to transmit packets successfully through the link.

First, in Chapter 7, the unconstrained adversarial jamming model is defined and analyzed, explaining its connection with the results of the task scheduling problem in the case of a single machine and no use of speedup. The *asymptotic throughput* measure introduced for this model is the corresponding long-term completed-load competitive ratio used for the online task scheduling. This is a new way of measuring the performance of scheduling algorithms in packet scheduling settings, focusing on the competitiveness in the long-run. The main contribution of this chapter is the study of the case of stochastic packet arrivals and the comparison with the completely adversarial case. It is not surprising that the upper bound proved for this case is close to the bound of the adversarial arrivals case, though it becomes bigger depending on the exact ratio between the task sizes. Another important conclusion of this part is the fact that *deferred feedback* for the error detection and feedback to the sender, makes any algorithm useless in front of adversarial channel jams. Since immediate feedback is assumed after that, this analysis will also hold for the task scheduling problem in the case of $m = s = 1$ and stochastic arrivals of tasks.

Then, in Chapter 8, the constrained adversarial jamming model is defined, assuming an infinite amount of data available at the sending station, with the objective of the algorithm being to decide on the appropriate packet length in order to maximize the goodput rate; the rate of successfully transmitted load over the link. To model the power of the adversary, an AQT approach has been considered for the first time, bounding the packet jams in the following way: The adversarial entity has a maximum of σ error tokens to be used on jamming different packets, and a new token becomes available at a rate ρ. This model represents adversaries with limited sources of rechargeable energy, which is in fact realistic for communication jams; think of mobile devices that can be used maliciously to interfere with a communication, or battery-operated drones that could be used in the same way. This work in general, gives an inside in a realistic constraint jamming model, using AQT for the first time to characterize jams (instead of packet arrivals as usually done) adding in the state-of-art of channel jams some non-trivial results.

Optimizing even the case of $\sigma = 1$ is not trivial. An idea for using a static version of the model as a building block to design an optimal scheduling algorithm is developed. However, as soon as $\sigma > 1$ the considered approach is not easily adapted, and is something that I would like to study further in the near future. As also discussed in Section 8.6, it appears that for big values of rate ρ, following a uniform packet scheduling approach gives better results than adapting the packet lengths. *This* makes the question whether the performance of the uniform packet scheduling approach can actually be exceeded in that case, even more intriguing. Finally, the fact that even the simplest case is not easily analyzed, creates more questions: How would the

analysis be if one or both parameters ρ and σ are not known? And how would the performance scale if additional channel errors due to congestion, noise and transmission rate were considered? In the first question, one will probably need to monitor the history of the observed jams in an attempt to estimate these parameters, assuming that they are fixed from the beginning. Note that if they can change with time, the adversary will try to "hide" the true value of these parameters, yielding an interesting gameplay between the adversary and an algorithm.

References

[1] http://alljammer.com/ (last accessed: May 20, 2016).

[2] http://www.jammer-store.com/ (last accessed: May 20, 2016).

[3] Enhanced intel speedstep technology for the Intel Pentium M Processor. Intel White Paper 301170-001, 2004.

[4] Miklós Ajtai, James Aspnes, Cynthia Dwork, and Orli Waarts. A theory of competitive analysis for distributed algorithms. In *35th Annual Symposium on Foundations of Computer Science, FOCS 1994, Santa Fe, New Mexico, USA, 20-22 November 1994*, pages 401–411, 1994.

[5] Susanne Albers and Antonios Antoniadis. Race to idle: New algorithms for speed scaling with a sleep state. *ACM Trans. Algorithms*, 10(2):9, 2014.

[6] Susanne Albers, Antonios Antoniadis, and Gero Greiner. On multi-processor speed scaling with migration. *J. Comput. Syst. Sci.*, 81(7):1194–1209, 2015.

[7] Dan Alistarh, Michael A. Bender, Seth Gilbert, and Rachid Guerraoui. How to allocate tasks asynchronously. In *53rd Annual IEEE Symposium on Foundations of Computer Science, FOCS 2012, New Brunswick, NJ, USA, October 20-23, 2012*, pages 331–340, 2012.

[8] S. Anand, Naveen Garg, and Nicole Megow. Meeting deadlines: How much speed suffices? In *Automata, Languages and Programming - 38th International Colloquium, ICALP 2011, Zurich, Switzerland, July 4-8, 2011, Proceedings, Part I*, pages 232–243, 2011.

[9] Lakshmi Anantharamu, Bogdan S. Chlebus, Dariusz R. Kowalski, and Mariusz A. Rokicki. Medium access control for adversarial channels with jamming. In *Structural Information and Communication Complexity - 18th International Colloquium, SIROCCO 2011, Gdansk, Poland, June 26-29, 2011. Proceedings*, pages 89–100. 2011.

[10] Richard J. Anderson and Heather Woll. Algorithms for the certified write-all problem. *SIAM J. Comput.*, 26(5):1277–1283, 1997.

[11] Matthew Andrews, Baruch Awerbuch, Antonio Fernández, Frank Thomson Leighton, Zhiyong Liu, and Jon M. Kleinberg. Universal-stability results and performance bounds for greedy contention-resolution protocols. *J. ACM*, 48(1):39–69, 2001.

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

[13] Jordi Arjona Aroca, Angelos Chatzipapas, Antonio Fernández Anta, and Vincenzo Mancuso. A measurement-based characterization of the energy consumption in data center servers. *IEEE Journal on Selected Areas in Communications*, 33(12):2863–2877, 2015.

[14] Hagit Attiya and Jennifer Welch. *Distributed computing: fundamentals, simulations, and advanced topics*, volume 19. John Wiley & Sons, 2004.

[15] Baruch Awerbuch, Shay Kutten, and David Peleg. Competitive distributed job scheduling (extended abstract). In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing, STOC 1992, Victoria, British Columbia, Canada, May 4-6, 1992*, pages 571–580, 1992.

[16] Baruch Awerbuch, Andréa W. Richa, and Christian Scheideler. A jamming-resistant MAC protocol for single-hop wireless networks. In *Proceedings of the Twenty-Seventh Annual ACM Symposium on Principles of Distributed Computing, PODC 2008, Toronto, Canada, August 18-21, 2008*, pages 45–54, 2008.

[17] Nikhil Bansal, Ho-Leung Chan, and Kirk Pruhs. Speed scaling with an arbitrary power function. *ACM Trans. Algorithms*, 9(2):18, 2013.

[18] Michael A. Bender, Soumen Chakrabarti, and S. Muthukrishnan. Flow and stretch metrics for scheduling continuous job streams. In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 1998, San Francisco, California, 25-27 January 1998*, pages 270–279, 1998.

[19] Pravin Bhagwat, Partha P. Bhattacharya, Arvind Krishna, and Satish K. Tripathi. Enhancing throughput over wireless lans using channel state dependent packet scheduling. In *Proceedings IEEE INFOCOM '96, The Conference on Computer Communications, Fifteenth Annual Joint Conference of the IEEE Computer and Communications Societies, Networking the Next Generation, San Francisco, CA, USA, March 24-28, 1996*, pages 1133–1140, 1996.

[20] Allan Borodin and Ran El-Yaniv. *Online computation and competitive analysis*. Cambridge University Press, 1998.

[21] Allan Borodin, Jon M. Kleinberg, Prabhakar Raghavan, Madhu Sudan, and David P. Williamson. Adversarial queuing theory. *J. ACM*, 48(1):13–38, 2001.

[22] Joan Boyar and Faith Ellen. Bounds for scheduling jobs on grid processors. In *Space-Efficient Data Structures, Streams, and Algorithms - Papers in Honor of J. Ian Munro on the Occasion of His 66th Birthday*, pages 12–26, 2013.

[23] Ho-Leung Chan, Joseph Wun-Tat Chan, Tak Wah Lam, Lap-Kei Lee, Kin-Sum Mak, and Prudence W. H. Wong. Optimizing throughput and energy in online deadline scheduling. *ACM Trans. Algorithms*, 6(1), 2009.

[24] Ho-Leung Chan, Jeff Edmonds, and Kirk Pruhs. Speed scaling of processes with arbitrary speedup curves on a multiprocessor. *Theory Comput. Syst.*, 49(4):817–833, 2011.

[25] Joseph Wun-Tat Chan, Prudence W. H. Wong, and Fencol C. C. Yung. On dynamic bin packing: An improved lower bound and resource augmentation analysis. *Algorithmica*, 53(2):172–206, 2009.

[26] Bogdan S Chlebus, Vicent Cholvi, and Dariusz R Kowalski. Universal routing in multi hop radio network. In *Proceedings of the 10th ACM international workshop on Foundations of mobile computing*, pages 19–28. ACM, 2014.

[27] Bogdan S. Chlebus, Vicent Cholvi, and Dariusz R. Kowalski. Stability of adversarial routing with feedback. *Networks*, 66(2):88–97, 2015.

[28] Bogdan S. Chlebus, Leszek Gasieniec, Dariusz R. Kowalski, and Alexander A. Shvartsman. A robust randomized algorithm to perform independent tasks. *J. Discrete Algorithms*, 6(4):651–665, 2008.

[29] Bogdan S. Chlebus, Leszek Gasieniec, Dariusz R. Kowalski, and Alexander A. Shvartsman. Doing-it-all with bounded work and communication. *CoRR*, abs/1409.4711, 2014.

[30] Bogdan S. Chlebus, Dariusz R. Kowalski, and Mariusz A. Rokicki. Stability of the multiple-access channel under maximum broadcast loads. In *Stabilization, Safety, and Security of Distributed Systems, 9th International Symposium, SSS 2007, Paris, France, November 14-16, 2007, Proceedings*, pages 124–138, 2007.

[31] Bogdan S. Chlebus, Dariusz R. Kowalski, and Mariusz A. Rokicki. Adversarial queuing on the multiple access channel. *ACM Trans. Algorithms*, 8(1):5, 2012.

[32] Bogdan S. Chlebus, Roberto De Prisco, and Alexander A. Shvartsman. Performing tasks on synchronous restartable message-passing processors. *Distributed Computing*, 14(1):49–64, 2001.

[33] Gennaro Cordasco, Grzegorz Malewicz, and Arnold L. Rosenberg. Advances in IC-Scheduling Theory: Scheduling Expansive and Reductive Dags and Scheduling Dags via Duality. *IEEE Trans. Parallel Distrib. Syst.*, 18(11):1607–1617, 2007.

[34] Shlomi Dolev, Seth Gilbert, Rachid Guerraoui, Dariusz R Kowalski, Calvin Newport, Fabian Kuhn, and Nancy Lynch. Reliable distributed computing on unreliable radio channels. In *Proceedings of the 2009 MobiHoc S 3 workshop on MobiHoc S 3*, pages 1–4. ACM, 2009.

[35] Yuval Emek, Magnús M. Halldórsson, Yishay Mansour, Boaz Patt-Shamir, Jaikumar Radhakrishnan, and Dror Rawitz. Online set packing and competitive scheduling of multi-part tasks. In *Proceedings of the 29th Annual ACM Symposium on Principles of Distributed Computing, PODC 2010, Zurich, Switzerland, July 25-28, 2010*, pages 440–449, 2010.

[36] Leah Epstein and Rob van Stee. Online bin packing with resource augmentation. *Discrete Optimization*, 4(3-4):322–333, 2007.

[37] Michelle S Faughnan, Brian J Hourican, G Collins MacDonald, Megha Srivastava, John-Patrick A Wright, Yacov Y Haimes, Eva Andrijcic, Zhenyu Guo, and James C White. Risk analysis of unmanned aerial vehicle hijacking and methods of its detection. In *Systems and Information Engineering Design Symposium (SIEDS), 2013 IEEE*, pages 145–150. IEEE, 2013.

[38] Antonio Fernández Anta, Chryssis Georgiou, Dariusz R. Kowalski, Joerg Widmer, and Elli Zavou. Measuring the impact of adversarial errors on packet scheduling strategies. In *Structural Information and Communication Complexity - 20th International Colloquium, SIROCCO 2013, Ischia, Italy, July 1-3, 2013, Revised Selected Papers*, pages 261–273, 2013.

[39] 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.

[40] Antonio Fernández Anta, Chryssis Georgiou, Dariusz R. Kowalski, and Elli Zavou. Online parallel scheduling of non-uniform tasks: Trading failures for energy. In *Fundamentals of Computation Theory - 19th International Symposium, FCT 2013, Liverpool, UK, August 19-21, 2013. Proceedings*, pages 145–158, 2013.

[41] 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 - Second International Workshop, ARMS-CC 2015, Held in Conjunction with ACM Symposium on Principles of Distributed Computing, PODC 2015, Donostia-San Sebastián, Spain, July 20, 2015, Revised Selected Papers*, pages 1–16, 2015.

[42] Antonio Fernández Anta, Chryssis Georgiou, Dariusz R. Kowalski, and Elli Zavou. Online parallel scheduling of non-uniform tasks. *Theor. Comput. Sci.*, 590(C):129–146, July 2015.

[43] Antonio Fernández Anta, Chryssis Georgiou, Dariusz R. Kowalski, and Elli Zavou. Competitive analysis of fundamental scheduling algorithms on a fault-prone machine and the impact of resource augmentation. *Future Generation Computer Systems*, 2016.

[44] Antonio Fernández Anta, Chryssis Georgiou, and Elli Zavou. Adaptive scheduling over a wireless channel under constrained jamming. In *Combinatorial Optimization and Applications - 9th International Conference, COCOA 2015, Houston, TX, USA, December 18-20, 2015, Proceedings*, pages 261–278, 2015.

[45] Antonio Fernández Anta, Chryssis Georgiou, and Elli Zavou. Packet Scheduling over a Wireless Channel: AQT-Based Constrained Jamming. In *Networked Systems - Third International Conference, NETYS 2015, Agadir, Morocco, May 13-15, 2015, Revised Selected Papers*, pages 230–245, 2015.

[46] Enabling Grids for E-sciencE (EGEE). `http://www.eu-egee.org`.

[47] Chryssis Georgiou and Dariusz R. Kowalski. On the competitiveness of scheduling dynamically injected tasks on processes prone to crashes and restarts. *J. Parallel Distrib. Comput.*, 84:94–107, 2015.

[48] Chryssis Georgiou, Alexander Russell, and Alexander A. Shvartsman. The complexity of synchronous iterative do-all with crashes. *Distributed Computing*, 17(1):47–63, 2004.

[49] Chryssis Georgiou and Alexander A. Shvartsman. *Do-All Computing in Distributed Systems: Cooperation in the Presence of Adversity*. Springer, 2008.

[50] Chryssis Georgiou and Alexander A. Shvartsman. *Cooperative Task-Oriented Computing: Algorithms and Complexity*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2011.

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

[52] Seth Gilbert, Rachid Guerraoui, and Calvin C. Newport. Of malicious motes and suspicious sensors: On the efficiency of malicious interference in wireless networks. *Theor. Comput. Sci.*, 410(6-7):546–569, 2009.

[53] Gero Greiner, Tim Nonner, and Alexander Souza. The bell is ringing in speed-scaled multiprocessor scheduling. *Theory Comput. Syst.*, 54(1):24–44, 2014.

[54] Ramakrishna Gummadi, David Wetherall, Ben Greenstein, and Srinivasan Seshan. Understanding and mitigating the impact of RF interference on 802.11 networks. In *Proceedings of the ACM SIGCOMM 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Kyoto, Japan, August 27-31, 2007*, pages 385–396, 2007.

[55] Kwang Soo Hong and Joseph Y.-T. Leung. On-line scheduling of real-time tasks. *IEEE Trans. Computers*, 41(10):1326–1331, 1992.

[56] Michal Jakubiak. Cellular network coverage analysis using UAV and SDR. Master's thesis, Tampere University of Technology, 2014.

[57] Kyle Jamieson and Hari Balakrishnan. PPR: partial packet recovery for wireless networks. In *Proceedings of the ACM SIGCOMM 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Kyoto, Japan, August 27-31, 2007*, pages 409–420, 2007.

[58] Kevin Jeffay, Donald F. Stanat, and Charles U. Martel. On non-preemptive scheduling of period and sporadic tasks. In *Proceedings of the Real-Time Systems Symposium - 1991, San Antonio, Texas, USA, December 1991*, pages 129–139, 1991.

[59] David S. Johnson, Alan J. Demers, Jeffrey D. Ullman, M. R. Garey, and Ronald L. Graham. Worst-case performance bounds for simple one-dimensional packing algorithms. *SIAM J. Comput.*, 3(4):299–325, 1974.

[60] Tomasz Jurdzinski, Dariusz R. Kowalski, and Krzysztof Lorys. Online packet scheduling under adversarial jamming. In *Approximation and Online Algorithms - 12th International Workshop, WAOA 2014, Wrocław, Poland, September 11-12, 2014, Revised Selected Papers*, pages 193–206, 2014.

[61] Paris Christos Kanellakis and Alex Allister Shvartsman. *Fault-tolerant parallel computation*, volume 401. Springer Science & Business Media, 2013.

[62] Thomas Kesselheim. Dynamic packet scheduling in wireless networks. In *ACM Symposium on Principles of Distributed Computing, PODC '12, Funchal, Madeira, Portugal, July 16-18, 2012*, pages 281–290, 2012.

[63] Eric Korpela, Dan Werthimer, David Anderson, Jeff Cobb, and Matt Lebofsky. SETI@ HOME-massively distributed computing for SETI. *Computing in science & engineering*, 3(1):78–83, 2001.

[64] Fei Li, Jay Sethuraman, and Clifford Stein. An optimal online algorithm for packet scheduling with agreeable deadlines. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2005, Vancouver, British Columbia, Canada, January 23-25, 2005*, pages 801–802, 2005.

[65] Shu Lin and Daniel J Costello. *Error control coding*. Pearson Education India, 2004.

[66] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[67] Zhoujia Mao, Can Emre Koksal, and Ness B. Shroff. Online packet scheduling with hard deadlines in multihop communication networks. In *Proceedings of the IEEE INFOCOM 2013, Turin, Italy, April 14-19, 2013*, pages 2463–2471, 2013.

[68] Chad R. Meiners and Eric Torng. Mixed criteria packet scheduling. In *Algorithmic Aspects in Information and Management, Third International Conference, AAIM 2007, Portland, OR, USA, June 6-8, 2007, Proceedings*, pages 120–133, 2007.

[69] Sparsh Mittal. Power management techniques for data centers: A survey. *CoRR*, abs/1404.6681, 2014.

[70] Michael Mitzenmacher and Eli Upfal. *Probability and computing - randomized algorithms and probabilistic analysis*. Cambridge University Press, 2005.

[71] Chandra Nair, Abbas El Gamal, Balaji Prabhakar, Elif Uysal-Biyikoglu, and Sina Zahedi. Energy-efficient scheduling of packet transmissions over wireless networks. In *Proceedings IEEE INFOCOM 2002, The 21st Annual Joint Conference of the IEEE Computer and Communications Societies, New York, USA, June 23-27, 2002*, 2002.

[72] Konstantinos Pelechrinis, Marios Iliofotou, and Srikanth V Krishnamurthy. Denial of service attacks in wireless networks: The case of jammers. *Communications Surveys & Tutorials, IEEE*, 13(2):245–257, 2011.

[73] Cynthia A. Phillips, Clifford Stein, Eric Torng, and Joel Wein. Optimal time-critical scheduling via resource augmentation. *Algorithmica*, 32(2):163–200, 2002.

[74] Michael Pinedo. *Scheduling: theory, algorithms and systems, 1995*. Prentice-Hall, Englewood Cliffs, NJ.

[75] Balaji Prabhakar, Elif Uysal-Biyikoglu, and Abbas El Gamal. Energy-efficient transmission over a wireless link via lazy packet scheduling. In *Proceedings IEEE INFOCOM 2001, The Conference on Computer Communications, Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies, Twenty years into the communications odyssey, Anchorage, Alaska, USA, April 22-26, 2001*, pages 386–394, 2001.

[76] Kirk Pruhs. Competitive online scheduling for server systems. *SIGMETRICS Performance Evaluation Review*, 34(4):52–58, 2007.

[77] Kirk Pruhs, Jirí Sgall, and Eric Torng. Online scheduling. In *Handbook of Scheduling - Algorithms, Models, and Performance Analysis.* 2004.

[78] Anand Raghavan, Kannan Ramchandran, and Igor Kozintsev. Continuous error detection (CED) for reliable communication. *IEEE Trans. Communications*, 49(9):1540–1549, 2001.

[79] M. Raynal. *Algorithms for mutual exclusion*. The MIT Press,Cambridge, MA, Jan 1986.

[80] Andrea Richa, Christian Scheideler, Stefan Schmid, and Jin Zhang. Towards jamming-resistant and competitive medium access in the sinr model. In *Proceedings of the 3rd ACM workshop on Wireless of the students, by the students, for the students*, pages 33–36. ACM, 2011.

[81] Andréa W. Richa, Christian Scheideler, Stefan Schmid, and Jin Zhang. Competitive and fair medium access despite reactive jamming. In *2011 International Conference on Distributed Computing Systems, ICDCS 2011, Minneapolis, Minnesota, USA, June 20-24, 2011*, pages 507–516, 2011.

[82] Andréa W. Richa, Christian Scheideler, Stefan Schmid, and Jin Zhang. Competitive and fair throughput for co-existing networks under adversarial interference. In *ACM Symposium on Principles of Distributed Computing, PODC '12, Funchal, Madeira, Portugal, July 16-18, 2012*, pages 291–300, 2012.

[83] Andréa W. Richa, Christian Scheideler, Stefan Schmid, and Jin Zhang. Competitive throughput in multi-hop wireless networks despite adaptive jamming. *Distributed Computing*, 26(3):159–171, 2013.

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

[85] Karsten Schwan and Hongyi Zhou. Dynamic scheduling of hard real-time tasks and real-time threads. *IEEE Trans. Software Eng.*, 18(8):736–748, 1992.

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

[87] David Thuente and Mithun Acharya. Intelligent jamming in wireless networks with applications to 802.11 b and other networks. In *Proc. of MILCOM*, volume 6, 2006.

[88] Vagelis Tsibonis, Leonidas Georgiadis, and Leandros Tassiulas. Exploiting wireless channel state information for throughput maximization. *IEEE Trans. Information Theory*, 50(11):2566–2582, 2004.

[89] Elif Uysal-Biyikoglu, Balaji Prabhakar, and Abbas El Gamal. Energy-efficient packet transmission over a wireless link. *IEEE/ACM Transactions on Networking (TON)*, 10(4):487–499, 2002.

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

[91] Adam Wierman, Lachlan L. H. Andrew, and Ao Tang. Power-aware speed scaling in processor sharing systems. In *INFOCOM 2009. 28th IEEE International Conference on Computer*

Communications, Joint Conference of the IEEE Computer and Communications Societies, 19-25 April 2009, Rio de Janeiro, Brazil, pages 2007–2015, 2009.

[92] F. Frances Yao, Alan J. Demers, and Scott Shenker. A scheduling model for reduced CPU energy. In *36th Annual Symposium on Foundations of Computer Science, Milwaukee, FOCS 1995, Wisconsin, 23-25 October 1995*, pages 374–382, 1995.

[93] Mitsuo Yokokawa, Fumiyoshi Shoji, Atsuya Uno, Motoyoshi Kurokawa, and Tadashi Watanabe. *The K computer*: Japanese next-generation supercomputer development project. In *Proceedings of the 2011 International Symposium on Low Power Electronics and Design, 2011, Fukuoka, Japan, August 1-3, 2011*, pages 371–372, 2011.

Appendix A

Summary of Publications

The contributions of this thesis have been published and presented in three journal articles [39, 42, 43], five conference papers [38, 40, 41, 44, 45], and four workshop talks without printed publication. Apart from the comments included below, on three more journal articles that are under revision, one new conference paper is also under submission. Note, that the order of the authors is alphabetical in all cases, as commonly done in theoretical computer science.

Journal articles:

- A. Fernández Anta, C. Georgiou, D. R. Kowalski, **E. Zavou**. "Online Parallel Scheduling of Non-uniform Tasks: Trading Failures for Energy", In: *Theoretical Computer Science (TCS 2015)*.
- A. Fernández Anta, C. Georgiou, D. R. Kowalski, J. Widmer, **E. Zavou**. "Measuring the Impact of Adversarial Errors on Packet Scheduling Strategies", In: *Journal of Scheduling (JOSH 2015)*.
- A. Fernández Anta, C. Georgiou, D. R. Kowalski, **E. Zavou**. "Competitive Analysis of Task Scheduling Algorithms on a Fault-Prone Machine and the Impact of Resource Augmentation", In: *Future Generation Computer Systems (FGCS 2016)* (In Press).

These journal articles are enriched versions of the works of my first conference papers, published in FCT 2013 and SIROCCO 2013 respectively, as well as the journal version of the invited paper in ARMS-CC 2015. All three works were invited to special issues of the journals.

The first article, initiates the task scheduling work, that is the main part of this thesis, formally introducing the model and the problem it presents when cooping with adversarial machine crashes and restarts. It assumes a parallel system of multiple machines, and dynamic task arrivals of different processing times. Both the task arrivals and the machines' crashes and restarts are controlled by an adversary, in order to analyze the worst-case scenarios, and pending-load is studied in terms of competitive analysis.

The second on the other hand, initiates the packet scheduling work, that has also been further developed, and is the second part in the thesis. It introduces the asymptotic throughput measure,

a long-term competitive ratio, and uses it to analyze the performance of online packet scheduling algorithms through an unreliable channel between two nodes. Adaptive and stochastic packet arrivals are taken into consideration, as well as two error feedback mechanisms.

The latter, continues the work on task scheduling, making an emphasis on the missing results in the single machine model and comparing the three efficiency measures analyzed in the thesis. It includes both the completed-load and the latency metric in the research, along with the pending-load, and analyzes four of the most widely-used scheduling algorithms, which surprisingly have not been studied under this model.

Conference papers:

- A. Fernández Anta, C. Georgiou, D. R. Kowalski, **E. Zavou**. "Online Parallel Scheduling of Non-uniform Tasks: Trading Failures for Energy", In: *The 19th International Symposium on Fundamentals of Computation Theory (FCT 2013), 19 - 21 August 2013, Liverpool, England, UK*.

- A. Fernández Anta, C. Georgiou, D. R. Kowalski, J. Widmer, **E. Zavou**. "Measuring the Impact of Adversarial Errors on Packet Scheduling Strategies", In: *The 20th International Colloquium on Structural Information and Communication Complexity (SIROCCO 2013), 1 - 3 July 2013, Ischia, Italy*.

- A. Fernández Anta, C. Georgiou, **E. Zavou**. "Packet Scheduling over a Wireless Channel: AQT-based Constrained Jamming" In: *The International Conference on NETworked sYStems (NETYS 2015), 13-15 May 2015, Agadir, Morocco*.

- A. Fernández Anta, C. Georgiou, D. R. Kowalski, **E. Zavou**. "Competitive Analysis of Task Scheduling Algorithms on a Fault-Prone Machine and the Impact of Resource Augmentation", In: *Workshop on Adaptive Resource Management and Scheduling for Cloud Computing (ARMS-CC 2015), 20 July 2015, San Sebastian, Spain*.

- A. Fernández Anta, C. Georgiou, **E. Zavou**. "Adaptive Scheduling over a Wireless Channel under Constrained Jamming", In: *The 9th Annual International Conference on Combinatorial Optimization and Applications (COCOA 2015), 18 - 20 December, 2015, Houston, Texas*.

This work has also been invited to a Special Issue of the Journal Algorithmica, and combined with the work published in NETYS 2015 that complement each other, have been submitted and are under revision.

The first two papers in the list above, as well as the forth one, are the conference versions of the articles mentioned earlier. The other two works are on packet scheduling, and constrain the power of the adversary. In the paper initiating the work, it was seen that one cannot hope for better results with a completely adaptive adversary, even if (s)he considers randomized algorithms. The aim was therefore slightly modified to study whether an adversary with constrained power could be affronted in a better way.

Workshops:

- A. Fernández Anta, C. Georgiou, D. R. Kowalski J. Widmer, **E. Zavou**. "Relative Throughput - Measuring the Impact of Adversarial Errors on Packet Scheduling Strategies", In: *The 9th International Workshop on Foundations of Mobile Computing (FOMC 2013), 17 - 18 October 2013, Jerusalem, Israel*.

- **E. Zavou**. "Asymptotic Competitive Analysis of Task Scheduling Algorithms on a Fault-Prone Machine", In: *The 1st Young Researcher Workshop on Automata, Languages and Programming (YR-ICALP 2014), 7 July 2014, Copenhagen, Denmark*
This talk has also been accompanied by a poster in *The 41st International Colloquium on Automata, Languages and Programming (ICALP 2014), 7 - 11 July 2014, Copenhagen, Denmark*.

- D. R. Kowalski, P. W.H. Wong, **E. Zavou**. "Fault Tolerant Scheduling of Non-uniform Tasks under Resource Augmentation", In: *The 12th Workshop on Models and Algorithms for Planning and Scheduling Problems (MAPSP 2015), 8 - 12 June, 2015, La Roche-en-Ardenne, Belgium*.

- A. Fernández Anta, C. Georgiou, **E. Zavou**. "Adaptive Scheduling over a Wireless Channel under Constrained Jamming", In: *Doctoral Consortium en Tecnologías Informáticas, XXIII Jornadas de Concurrencia y Sistemas Distribuidos (JCSD 2015), 10-12 June 2015, Málaga, Spain*.

These works have been presented in workshops, but they either belong to more extended works and are published already, or they are under submission for publication. In particular, the first work presented in FOMC'13 has an extended version published in SIROCCO 2013 and a journal article in JOSH 2015. The second work, presented in ICALP 2014 is part of the invited work in ARMS-CC 2015. The third is under revision in the Journal of Scheduling and the fourth is part of the COCOA 2015 paper.